

COMBINATIONAL LOGIC

INTRODUCTION

- Combinational logic circuits are circuits in which the output at any time depends upon the combination of input signals present at that instant only, and does not depend on any past conditions.
- The block diagram of a combinational circuit with m inputs and n outputs is shown in Fig. 4.0.

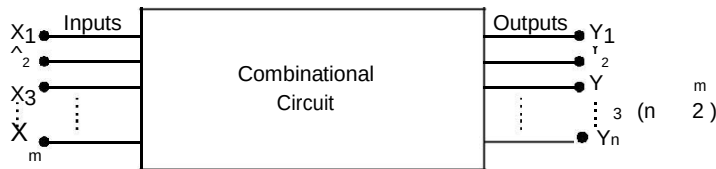


Fig. 4.0 Block diagram of combinational logic circuit

- In particular, the output of particular circuit does not depend upon any past inputs or outputs i.e. the output signals of combinational circuits are not feedback to the input of the circuit.
- Moreover, in a combinational circuit, for a change in the input, the output appears immediately, except for the propagation delay through circuit gates.
- The combinational circuit block can be considered as a network of logic gates that accept signals from inputs and generate signals to outputs.
- For m input variables, there are 2^m possible combinations of binary input values.
- Each input combination to the combinational circuit exhibits a distinct (unique) output.
- Thus a combinational circuit can be described by n boolean functions, one for each input combination, in terms of m input variables with n is always less than or equal to 2^m . [$n \leq 2^m$].
- Thus, a combinational circuit performs a specific information processing operation which is specified by Boolean functions.
- $n < 2^m$ represent the condition, that in a particular application there are some unused input combinations.
- For example, when we are using NBCD codes, the six combinations (1010, 1011, 1100, 1101, 1110 and 1111) are never used. So with four input variables ($m = 4$) we are using only 10 i/p combinations i.e., 10 O/p's instead of $2^4 = 16$.
- The digital systems perform a member of information processing tasks.
- The basic arithmetic operations used by digital computers and calculators are implemented by combi-national circuits using logic gets.
- We proceed with the implementation of these basic functions by first looking the simple design procedure.

Combinational circuit Design Procedure

It involves following steps :

Step 1 : From the word description of the problem, identify the inputs and outputs and draw a block diagram.

Step 2 : Make a truth table based on problem statement which completely describes the operations of circuit for different combinations of inputs.

Step 3 : Simplified output functions are obtained by algebraic manipulation, k-map method or tabular method.

Step 4 : Implement the simplified expression using logic gates.

To explain the procedure, let us take an example that we have already been used in chapter 2.

Example: A TV is connected through three switches. The TV becomes 'on' when atleast two switches are in 'ON' position; In all other conditions, TV is 'OFF'.

Solution. Step I : The TV is connected with 3 switches; thus there are three inputs to TV, represented by variables say A, B and C. The o/p of TV is represented by variable say, F.

The block diagram is shown in Fig. 4.1 :

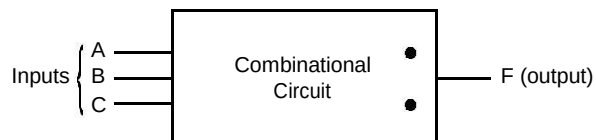


Fig. 4.1

Step 2.

Truth Tables

TV switches ← INPUTS			OUTPUTS	
A	B	C		F
0	0	0		0
0	0	1		0
0	1	0		0
0	1	1		1
1	0	0		0
1	0	1		1
1	1	0		1
1	1	1		1

0 → switch off

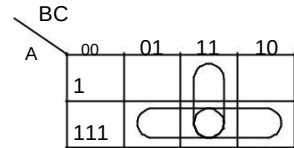
1 → switch on

It means for the input combinations in which there are two or more 1's, the output $F = 1$ (TV is ON) and for rest combinations, output $F = 0$ (TV is OFF).

Step 3 : In general, in simplifying boolean functions upto four variables, the best method is K- map technique. Thus, using a 3 variable K- map, we can simplify the function obtained in step II.

We get

$$F = AB + AC + BC$$



We can observe that if the value of any two variables is equal to 1, the output is equal to 1.

Step IV. For implementation we need three 'AND' gates and one 'OR' gate as shown in Fig. 4.2.

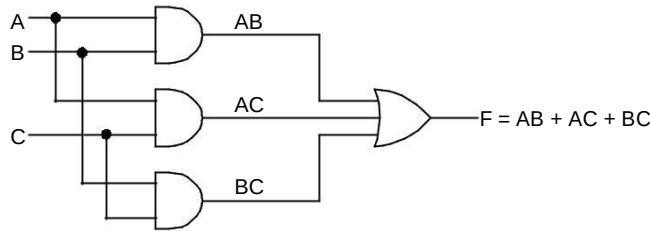


Fig. 4.2

ARITHMATIC CIRCUITS

- The logic circuits which are used for performing the digital arithmetic operations such as addition, subtraction, multiplication and division are called 'arithmetic circuits'.

Adders

- The most common arithmetic operation in digital systems is the addition of two binary digits. The combinational circuit that performs this operation is called a half-adder.

Half Adder

Fig. 4.3 shows a half adder (HA).

It has two inputs A and B, that are two 1-bit members, and two output sum (S) and carry (C) produced by addition of two bits.

2. Truth Table :

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

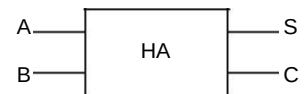
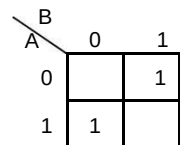


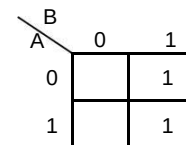
Fig. 4.3 Half adder

The sum output is 1 when any of inputs (A and B) is 1 and the carry output is 1 when both the inputs are 1.

3. Using a two variable k-map, separately for both outputs S and C.



$$\begin{aligned} \text{For 'S'} \\ S &= AB' + A'B \\ &= A \oplus B. \end{aligned}$$



$$\begin{aligned} \text{For 'C'} \\ C &= AB \end{aligned}$$

Logical Implementation.

(i) Using Basic gates (as shown in Fig. 4.4(a)).

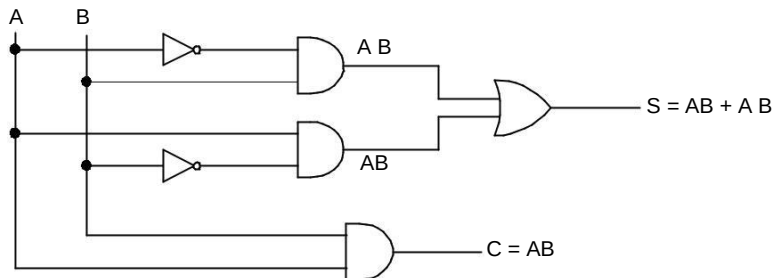


Fig. 4.4 (a)

(ii) Using XOR gate as shown in Fig. 4.4 (b).

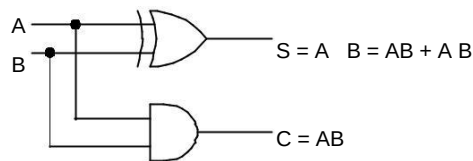


Fig. 4.4 (b)

Implementation using only NAND or only NOR gates is left as an exercise.

Full Adder

- Full adder is a combinational circuit that performs the addition of three binary digits.
- Fig. 4.5 shows a full adder (FA).
- It has three inputs A, B and C and two outputs S and C_0 produced by addition of three input bits.
- Carry output is designated C_0 just to avoid confusion between with i/p variable C.



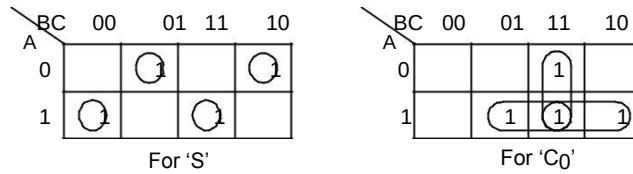
Fig. 4.5 Full adder

Truth Table :

- The eight possible combinations of three input variables with their respective outputs is shown. We observe that when all the three inputs are 1, the sum and carry both outputs, are 1.

Inputs			Output	
A	B	C	S	C_0
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

Using a three variable map for both outputs.



$$S = ABC + AB'C' + A'BC' + A'B'C \text{ and } C_0 = AB + AC + BC.$$

Logical Implementation. (i) Using basic gates as shown in Fig. 4.6.

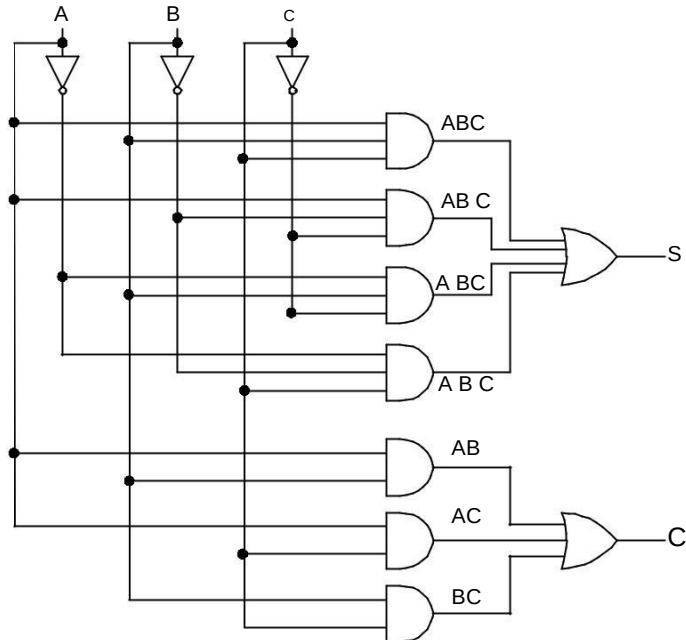


Fig. 4.6

(ii) A 'Full Adder' can also be implemented using two half adders and an 'OR' Gate as shown in Fig. 4.7

The Sum

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

and the carry

$$\begin{aligned} C_0 &= AB + AC + BC \\ &= AB + C(A + B) \\ &= AB + C(A + B)(A + A')(B + B') \\ &= AB + C[AB + AB' + A'B] \\ &= AB + ABC + C(AB' + A'B) \\ &= AB(1 + C) + C(A \oplus B) \\ &= AB + C(A \oplus B) \end{aligned}$$

Therefore, $S = (A \oplus B) \oplus C$ and $C_0 = AB + C(A \oplus B)$

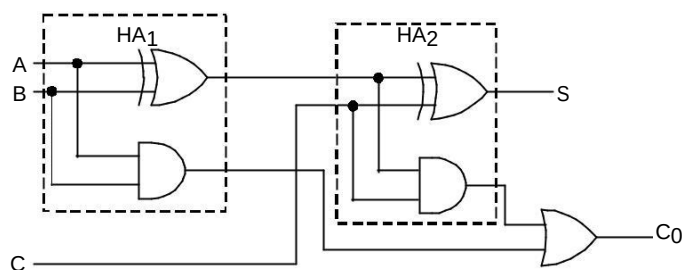
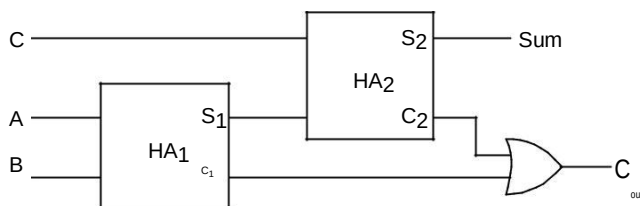


Fig. 4.7 Implementation of full adder

Block Diagram representation of a full adder using two half adders :



S_1 and C_1 are outputs of first half adder (HA_1)

S_2 and C_2 are outputs of second half adder (HA_2)

A, B and C are inputs of Full adder.

Sum and C_{out} are outputs of full adder.

Subtractors

- The logic circuits used for binary subtraction, are known as 'binary subtractors'.

Half Subtractor :

- The half subtractor is a combinational circuit which is used to perform the subtraction of two bit
- Fig. 4.8 shows a half subtractor. (HS)
- It has two inputs, A (minuend) and B (subtrahend) and two outputs D (difference) and B_0 (Borrow). [The symbol for borrow (B_0) is taken to avoid confusion with input variable B] produced by subtractor of two bits.

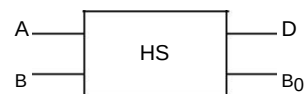


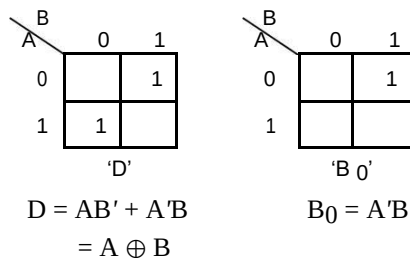
Fig. 4.8 Half subtractor

Truth Table

- The difference output is 0 if $A = B$ and 1 if $A \neq B$; the borrow output is 1 whenever $A < B$. If $A < B$, the subtraction is done by borrowing 1 from the next higher order bit.

Inputs		Outputs	
A	B	D	B_0
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

Using a two variable map, for outputs D and B.



Logical Implementation shown in Fig. 4.9

(a) Using Basic gates

(b) using XOR gate

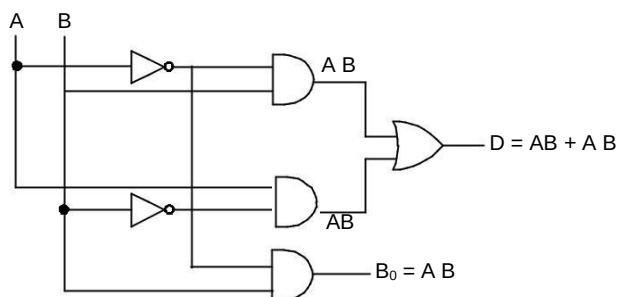


Fig. 4.9 (a) Basic gate implementation half subtractor.

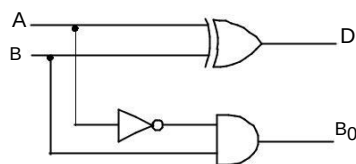


Fig. 4.9 (b) X-OR gate implementation of half subtractor

Full subtractor:

- Full subtractor is a combinational circuit that performs the subtraction of three binary digits.

1. Fig. 4.10 shows a full subtractor (FS).

It has three inputs A, B and C and two outputs D and B₀ produced by subtraction of three input bits.

2. **Truth Table**

The eight possible combinations of three input variables with their respective outputs is shown. We observe that when all the three inputs are 1, the difference and borrow both outputs are 1.

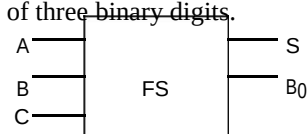
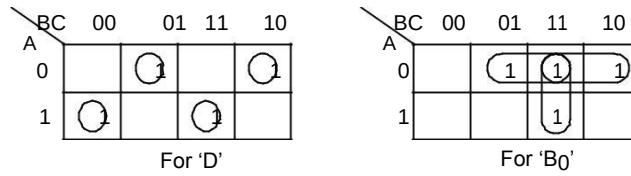


Fig. 4.10 Full subtractor

Inputs			Output	
A	B	C	B ₀	D
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

Using a three variable map for both outputs.



$$D = ABC + AB'C' + A'BC' + A'B'C, B_0 = A'B + A'C + BC$$

Logical implementation—

(i) Using basic gates : Left as an exercise.

(ii) A 'full subtractor' can also be implemented using two 'half subtractors' and an 'OR' gate as shown in Fig. 4.11.

The difference

$$'D' = ABC + AB'C' + A'BC' + A'B'C$$

$$ABC + A'B'C + AB'C' + A'BC'$$

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

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

$$C(A \oplus B)' + C'(A \oplus B)$$

$$(A \oplus B) \oplus C$$

and the borrow

$$B_0 = A'B + A'C + BC$$

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

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

$$A'B + C[A'B + AB + A'B']$$

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

$$A'B(C + 1) + C(A \oplus B)'$$

$$A'B + C(A \oplus B)'$$

$$D = (A \oplus B) \oplus C \text{ and } B_0 = A'B + C(A \oplus B)'$$

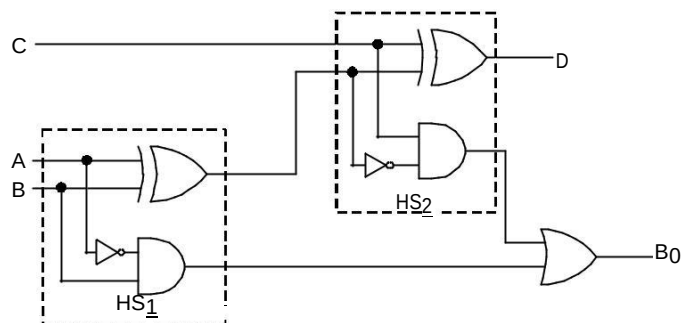


Fig. 4.11 (a)

Block Diagram Representation of a full subtractor using two half subtractors :

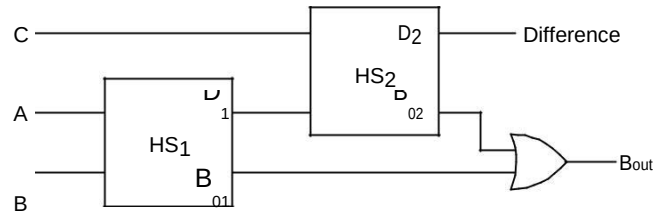


Fig. 4.11 (b)

D_1 and B_{01} are outputs of first half subtractor (HS_1)

D_2 and B_{02} are outputs of second half subtractor (HS_2)

A, B and C are inputs of full subtractor.

Difference and B_{out} are outputs of full subtractor.

Code Converters

- In the previous study of codes, coding was defined as the use of groups of bits to represent items of information that are multivalued.
- Assigning each item of information a unique combination of bits makes a transformation of the original information.
- This we recognize as information being processed into another form. Moreover, we have seen that there are many coding schemes exist.
- Different digital systems may use different coding schemes. It is sometimes necessary to use the output of one system as the input to other.
- Therefore a sort of code conversion is necessary between the two systems to make them compatible for the same information.
- 'A code converter is a combinational logic circuit that changes data presented in one type of binary code to another type of binary code.'
- A general block diagram of a code converter is shown in Fig. 4.12.

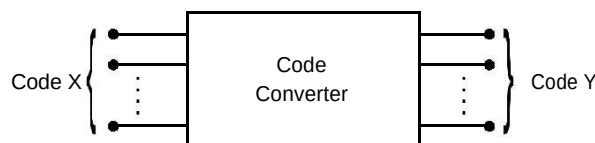


Fig. 4.12 Code converter

- To understand the design procedure; we will take a specific example of 4-bit Binary to Gray code conversion.
- The block diagram of a 4-bit binary to gray code converter is shown in Fig. 4.13.

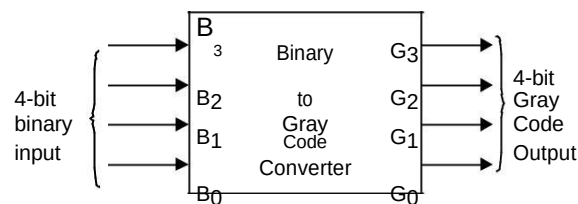


Fig. 4.13

- It has four inputs (B_3 B_2 B_1 B_0) representing 4-bit binary numbers and four outputs (G_3 G_2 G_1 G_0) representing 4-bit gray code.

Truth table for binary to gray code converters.

Binary Inputs				Gray code Outputs			
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	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

Now we solve all the gray outputs distantly with respect to binary inputs From the truth table; the logic expressions for the gray code outputs can be written as

$$G_3 = \Sigma (8, 9, 10, 11, 12, 13, 14, 15)$$

$$G_2 = \Sigma (4, 5, 6, 7, 8, 9, 10, 11)$$

$$G_1 = \Sigma (2, 3, 4, 5, 10, 11, 12, 13)$$

$$G_0 = \Sigma (1, 2, 5, 6, 9, 10, 13, 14).$$

The above expressions can be simplified using K-map

Map for G_3 :

From the octet, we get

$$G_3 = B_3$$

$B_1 B_0$

$B_3 B_2$	00	01	11	10
00				
01				
11	1	1	1	1
10	1	1	1	1

Map for G_2 :

From the two quads, we get

$$\begin{aligned}
 G_2 &= B_3' B_2 + B_3 B_2' \\
 &= B_3 \oplus B_2.
 \end{aligned}$$

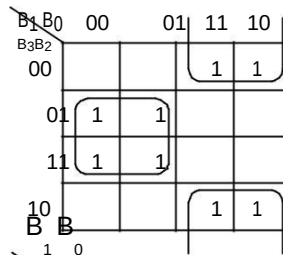
$B_1 B_0$

$B_3 B_2$	00	01	11	10
00				
01	1	1	1	1
11				
10	1	1	1	1

Map for G_1 :

From the two quads, we get

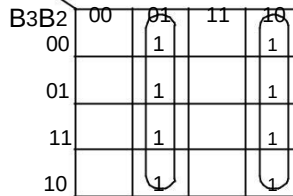
$$G_1 = B_2 B_1' + B_2' B_1 = B_2 \oplus B_1$$



Map for G_0 :

From the two quads, we get

$$G_0 = B_1' B_0 + B_1 B_0' = B_1 \oplus B_0$$



Now the above expressions can be implemented using X-OR gates to yield the desired code converter circuit shown in Fig. 4.14.

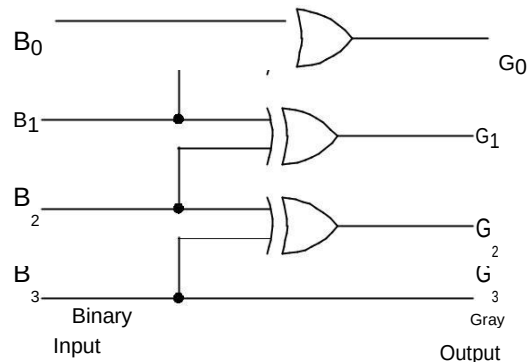


Fig. 4.14

- Let us see one more example of XS-3 to BCD code converter.
- The block diagram of an XS-3 to BCD code converter is shown in Fig. 4.15.
- It has four inputs (E_3, E_2, E_1, E_0) representing 4 bit XS-3 number and four outputs ($B_3 B_2 B_1 B_0$) representing 4-bit BCD code.

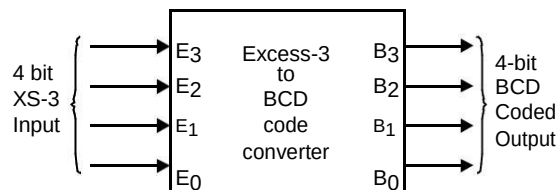


Fig. 4.15

- Truth Table for XS-3 to BCD code converter.
- XS-3 codes are obtained from BCD code by adding 3 to each coded number. Moreover 4 binary variables may have 16 combinations, but only 10 are listed.
- The six not listed are don't care-combinations. Since they will never occur, we are at liberty to assign to the output variable either a 1 or a 0, whichever gives a simpler circuit. In this particular example, the unused i/o combinations are listed below the truth table.

Min Terms	Excess-3 Inputs				BCD Outputs				Decimal Equivalent
	E_3	E_2	E_1	E_0	B_3	B_2	B_1	B_0	
m_3	0	0	1	1	0	0	0	0	0
m_4	0	1	0	0	0	0	0	1	1
m_5	0	1	0	1	0	0	1	0	2
m_6	0	1	1	0	0	0	1	1	3
m_7	0	1	1	1	0	1	0	0	4
m_8	1	0	0	0	0	1	0	1	5
m_9	1	0	0	1	0	1	1	0	6
m_{10}	1	0	1	0	0	1	1	1	7
m_{11}	1	0	1	1	1	0	0	0	8
m_{12}	1	1	0	0	1	0	0	1	9
Unused I/Ps					Outputs				
m_0	0	0	0	0	x	x	x	x	
m_1	0	0	0	1	x	x	x	x	
m_2	0	0	1	0	x	x	x	x	
m_{13}	1	1	0	1	x	x	x	x	
m_{14}	1	1	1	0	x	x	x	x	
m_{15}	1	1	1	1	x	x	x	x	

XS-3 is also a class of BCD codes.

- Now we solve all the BCD outputs. From the truth table, the logic expressions for the BCD coded outputs can be written as :

$$B_3 = \Sigma (m_{11}, m_{12}), d (m_0, m_1, m_2, m_{13}, m_{14}, m_{15})$$

$$B_2 = \Sigma (m_7, m_8, m_9, m_{10}), d (m_0, m_1, m_2, m_{13}, m_{14}, m_{15})$$

$$B_1 = \Sigma (m_5, m_6, m_9, m_{10}), d (m_0, m_1, m_2, m_{13}, m_{14}, m_{15})$$

$$B_0 = \Sigma (m_4, m_6, m_8, m_{10}, m_{12}), d (m_0, m_1, m_2, m_{13}, m_{14}, m_{15}).$$

- These expressions can be simplified using k-map

Map for B_3

$E_3 E_2 \backslash E_1 E_0$	00	01	11	10
00	x	x		x
01				
11	1	x	x	x
10			1	

$$B_3 = E_3 E_2 + E_3 E_1 E_0$$

Map for B_2

$E_3 E_2 \backslash E_1 E_0$	00	01	11	10
00	x	x		x
01			1	
11		x	x	x
10	1	1		1

$$B_2 = E_2' E_0' + E_2 E_1 E_0 + E_2' E_1'$$

Map for B₁

E E	1 0	00	01	11	10
$\begin{smallmatrix} E_3 \\ E_2 \end{smallmatrix}$					
00	X	X			X
01		1			1
11		X	X	X	
10		1			1

$$B_1 = E_1' E_0 + E_1 E_0'$$

$$= E_1 \oplus E_0$$

$$B_3 = E_3 E_2 + E_3 E_1 E_0$$

$$B_2 = E_2' E_0 + E_2 E_1 E_0 + E_2' E_1'$$

$$B_1 = E_1 \oplus E_0$$

$$B_0 = E_0'$$

Map for B₀

E E	1 0	00	01	11	10
$\begin{smallmatrix} E_3 \\ E_2 \end{smallmatrix}$					
00	X	X			X
01	1				1
11	1	X	X	X	
10	1				1

$$B_0 = E_0'$$

- The expressions for BCD outputs (B₃ B₂ B₁ B₀) can be implemented for terms of inputs (E₃ E₂ E₁ E₀) to form a XS-3 to BCD code converter circuit.
- The implementation is left for you as an exercise.

Parity Generators and Checkers

- When digital data is transmitted from one location to another, it is necessary to know at the receiving end, whether the received data is free of error.
- To help make the transmission accurate, special error detection methods are used.
- To detect errors, we must keep a constant check on the data being transmitted.
- To check accuracy we can generate and transmit an extra bit along with the message (data).
- This extra bit is known as the parity bit and it decides whether the data transmitted is error free or not.
- There are two types of parity bits, namely, even parity and odd parity that we have discussed in chapter 1 under error detecting codes.
- Fig. 4.16 shows an error detecting circuit using a parity bit.

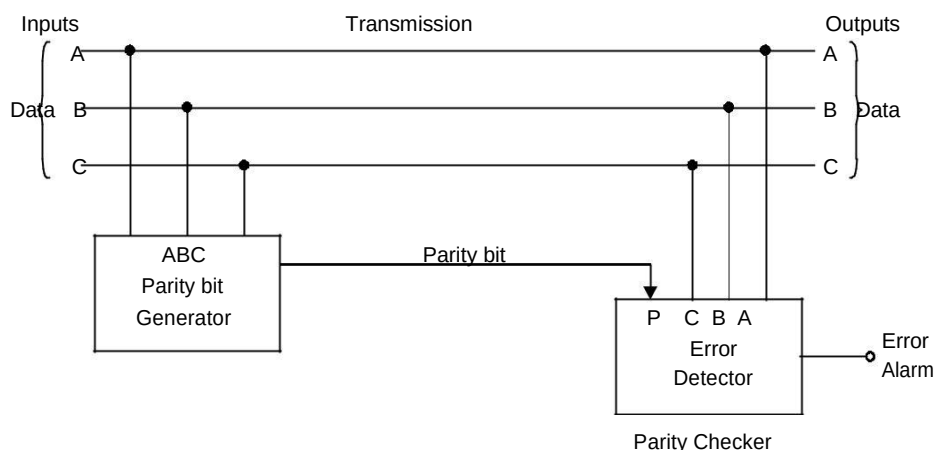


Fig. 4.16

- In these system three parallel bits A, B and C and being transmitted over a long distance. Near the input they are fed into a parity bit generator circuit.
- This circuit generates what is called a parity bit.
- It may be either even or odd. For example, if it is a 3-bit even parity generator, the parity bit generated is such that it makes total member of 1s even.
- We can make a truth table of a 3-bit even parity generator circuit.
- Truth Table for a 3-bit even parity generator.

Inputs Data				Output Even parity bit
A	B		C	P
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	0
1	1		1	1

- Next, the truth table is converted to a logic circuit shown in Fig. 4.17.

$$P = A'B'C + A'BC' + AB'C' + ABC$$

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

$$= A' (B \oplus C) + A (BC' + B'C)$$

$$= A' (B \oplus C) + A (B \oplus C)'$$

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

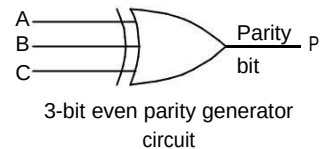


Fig. 4.17

- The generated parity bit is transmitted with the data and near the output it is fed to the error detector (parity checker) circuit.
- The detector circuit checks for the parity of transmitted data. As soon as the total number of 1's in the transmitted data is found 'odd' it sounds an alarm, indicating an error.
- If total member of 1's are even, no alarm sounds, indicating no error.
- In above example we are transmitting 4 bits. (3 bits of message plus 1 even parity bit). So, it is easy to understand that.
- Error detector is nothing but a 4 bit even-parity checker circuit. Fig. 4.18 (a) shows a truth table of a 4 bit even parity checker circuit.

Inputs Transmitted Data with parity bit					Outputs Parity error check
A	B		C	P	E
0	0		0	0	0
0	0		0	1	1
0	0		1	0	1
0	0		1	1	0
0	1		0	0	1
0	1		0	1	0

(Contd.)

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

Fig. 4.18 (a)

- Now, we convert this truth table into logic circuit shown in Fig. 4.18(b).

$$= A'B'C'P + A'B'CP' + A'BC'P' + A'BCP + AB'C'P' + AB'CP + ABC'P' + ABCP$$

- $A'B'(C \oplus P) + A'B(C \oplus P)' + AB'(C \oplus P)' + AB(C \oplus P)$
- $(C \oplus P)(A \oplus B)' + (C \oplus P)'(A \oplus B)$
- $(A \oplus B) \oplus (C \oplus P)$

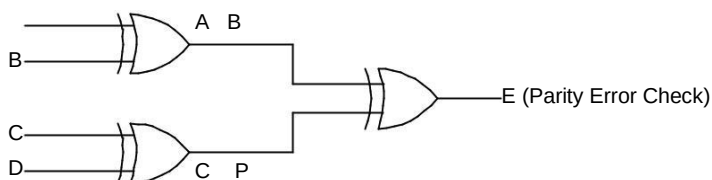


Fig. 4.18 (b) 4-bit even parity checker

If $E = 1$, Alarm sounds means error.

If $E = 0$, No alarm sounds means no error.

- Now, it is possible to implement the parity generator with the circuit of parity checker. If the input P is connected to logic-0, causing the value of C to pass through the gate unchanged. (because $C \oplus 0 = C$).
- The advantage of this is that the same circuit can be used for both parity generation and checking.

Unit 5

MSI AND LSI CIRCUITS

- When designing logic circuits, the “discrete logic gates”; *i.e.*, individual AND, OR, NOT *etc.* gates, are often neither the simplest nor the most economical devices we could use.
- There are many standard MSI (medium scale integrated) and LSI (large scale integrated) circuits, or functions available, which can do many of the things commonly required in logic circuits.
- Often these MSI and LSI circuits do not fit our requirements exactly, and it is often necessary to use discrete logic to adapt these circuits for our application.
- However, the number and type of these LSI and VLSI (very large scale integrated) circuits is steadily increasing, and it is difficult to always be aware of the best possible circuits available for a given problem.
- Also, systematic design methods are difficult to devise when the types of logic device available keeps increasing.
- **In general the “best” design procedure is to attempt to find a LSI device which can perform the required function, or which can be modified using other devices to perform the required function.**
- If nothing is available, then the function should be implemented with several MSI devices. Only as a last option should the entire function be implemented with discrete logic gates.
- In fact, with present technology, it is becoming increasingly cost-effective to implement a design as one or more dedicated VLSI devices.
- When designing all but the simplest logic devices, a “top-down” approach should be adopted.
- The device should be specified in block form, and attempt to implement each block with a small number of LSI or MSI functions.
- Each block which cannot be implemented directly can be then broken into smaller blocks, and the process repeated, until each block is fully implemented.
- Of course, **a good knowledge of what LSI and MSI functions are available in the appropriate technology makes this process simpler.**

The Digital Multiplexers

- One MSI function which has been available for a long time is the digital selector, or multiplexer.
- It is the digital equivalent of the rotary switch or selector switch (*e.g.*, the channel selector on a TV set). Its function is to accept a binary number as a “selector input,” and present the logic level connected to that input line as output from the data selector.
- A digital multiplexer (MUX) is a combinational circuits that selects one input out of several inputs and direct it to a single output.
- The particular input selection is controlled by a set of select inputs. Fig. 5.1 shows block diagram of a digital multiplexer with n inputs lines and single output line.
- For selecting one out of n input, a set of m select inputs is required where

$$n = 2^m$$

- On the basis of digital (binary) code applied at the select inputs, one output of n data sources is selected.
- Usually, an enable (or strobe) input (E) is built-in for cascading purpose.
- Enable input is generally active-low, *i.e.*, it performs its in-tended operation when it is low (logic).

Note. 16:1 are the largest available ICs, therefore for larger input requirements there should be provision for expansion. This is achieved through enable/strobe input (multiplexer stacks or trees are designed).

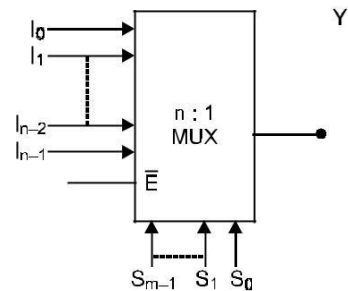


Fig. 5.1 Block diagram of the digital multiplexer

- A circuit diagram for a possible 4-line to 1-line data selector/multiplexer (abbreviated as MUX for multiplexer) is shown in Fig. 4.20.
- Here, the output Y is equal to the input I_0, I_1, I_2, I_3 depending on whether the select lines S_1 and S_0 have values 00, 01, 10, 11 for S_1 and S_0 respectively.
- That is, the output Y is *selected* to be equal to the input of the line given by the binary value of the select lines S_1S_0 .
- The logic equation for the circuit shown in Fig. 5.2 is:

+ + +

- This device can be used simply as a data selector/multiplexer, or it can be used to perform logic functions. Its simplest application is to implement a truth table directly, *e.g.*, with a 4 line to 1 line MUX, it is possible to implement any 2-variable function directly, simply by connecting I_0, I_1, I_2, I_3 to logic 1 in logic 0, as dictated by a truth table.
- In this way, a MUX can be used as a simple look-up table for switching functions.
- This facility makes the MUX a very general purpose logic device.

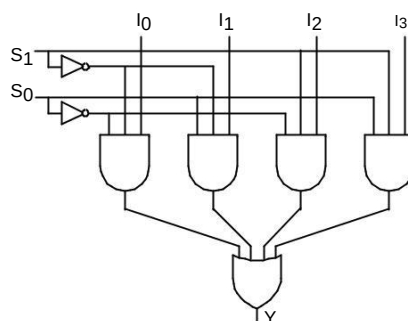


Fig 5.2 A four-line to 1-line multiplexer

Example 1. Use a 4 line to 1 line MUX to implement the function shown in the following truth table .

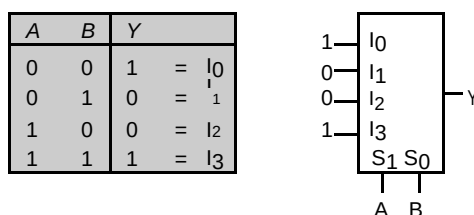


Fig. 5.3 A 4-line to 1-line MUX implementation of a function of 2 variables

Simply connecting $I_0 = 1$, $I_1 = 0$, $I_2 = 0$, $I_3 = 1$, and the inputs A and B to the S_1 and S_0 selector inputs of the 4-line to 1-line MUX implement this truth table, as shown in Fig. 5.3.

- The 4-line to 1-line MUX can also be used to implement any function of three logical variables, as well.
- To see this, we need note only that the only possible functions of one variable C, are C, and the constants 0 or 1. (i.e., C, , and 0).
- We need only connect the appropriate value, C, , 0 or 1, to I_0 , I_1 , I_2 , I_3 to obtain a function of 3 variables.
- The MUX still behaves as a table lookup device; it is now simply looking up values of another variable.

Decoders (Demultiplexers)

- Another commonly used MSI device is the decoder.
- Decoders, in general, transform a set of inputs into a different set of outputs, which are coded in a particular manner; e.g., certain decoders are designed to decode binary or BCD coded numbers and produce the correct output to display a digit on a 7 segment (calculator type) display.
- Decoders are also available to convert numbers from binary to BCD, from binary to hexadecimal, etc.
- Normally, however, the term “decoder” implies a device which performs, in a sense, the inverse operation of a multiplexer.
- A decoder accepts an n digit number as its n “select” inputs and produces an output (usually a logic 0) at one of its 2^n possible outputs.
- Decoders are usually referred to as n line to 2^n line decoders; e.g. a 3 line to 8 line decoder.
- This type of decoder is really a binary to unary number system decoder.
- Most decoders have inverted outputs, so the selected output is set to logic 0, while all the other outputs remain at logic 1.
- As well, most decoders have an “enable” input , which “enables” the operation of the decoder—when the input is set to 0, the device behaves as a decoder and selects the output determined by the select inputs; when the input $\bar{E}$ is set to 1, the outputs of the decoder are *all* set to 1.
- (The bar over the E indicates that it is an “active low” input; that is, a logic 0 enables the function).

- The enable input also allows decoders to be connected together in a treelike fashion, much as we saw for MUX's, so large decoders can be easily constructed from smaller devices.
- The enable input also allows the decoder to perform the inverse operation of a MUX; a MUX selects as output one of 2^n inputs, the decoder can be used to present an input to one of 2^n outputs, simply by connecting the input signal to the input; the signal at the selected output will then be the same as the input at — this application is called “demultiplexing.”
- The demulti-plexer (DEMUX) performs the reverse operation of a multiplexer.
- A demultiplexer is a circuit that accepts single input and transmit it over several (one of 2^n possible) outputs.
- In the block diagram (Fig. 5.4) a demultiplexer, the number of output lines is n and the number of select lines is m , where $n = 2^m$.
- One the basis of select input code, to which output the data will be transmitted is determined. There is an active-low (low-logic) enable/data input. The output for these devices are also active-low.

Note. 4-line to 16-line decoders are the largest available circuits in ICs.

- A typical 3 line to 8 line decoder with an enable input behaves according to the following truth table, and has a circuit symbol as shown in Fig.

E	S_2	S_1	S_0	O_0	O_1	O_2	O_3	O_4	O_5	O_6	O_7
1	x	x	x	1	1	1	1	1	1	1	1
0	0	0	0	0	1	1	1	1	1	1	1
0	0	0	1	1	0	1	1	1	1	1	1
0	0	1	0	1	1	0	1	1	1	1	1
0	0	1	1	1	1	1	0	1	1	1	1
0	1	0	0	1	1	1	1	0	1	1	1
0	1	0	1	1	1	1	1	1	0	1	1
0	1	1	0	1	1	1	1	1	1	0	1
0	1	1	1	1	1	1	1	1	1	1	0

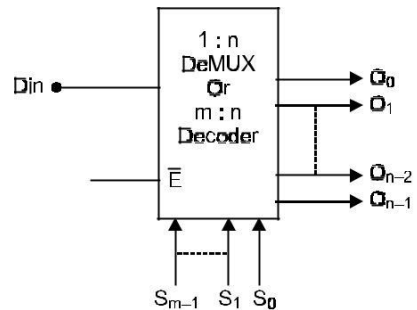
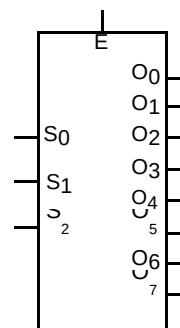


Fig. 45.4 Block diagram of the demultiplexer/deco



- Note that, when the input is enabled, an output of 0 is produced corresponding to each minterm of S_2, S_1, S_0 . —
- These minterm can be combined together using other logic gates to form any required logic function of the input variables.
- In fact, several functions can be produced at the same time.
- If the selected output was a logic 1, then the required minterms could simply be ORed together to implement a switching function directly from its minterm form.
- Using de Morgans theorem, we can see that when the outputs are inverted, as is normally the case, then the minterm form of the function can be obtained by NANDing the required terms together.

Example 1. An implementation the functions defined by the following truth table using a decoder and NAND gates is shown in Fig. 5.5.

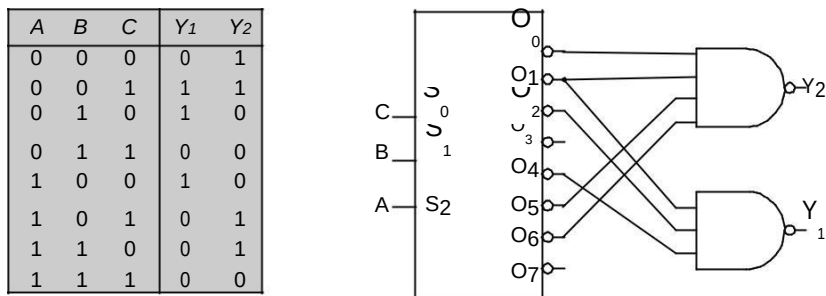


Fig. 5.5

IMPLEMENTATION EXAMPLES OF COMBINATIONAL LOGIC DESIGN USING MUX/DEMUX

We have already seen how to implement combinational circuits using MUX/DEMUX. The standard ICs available for multiplexers are 2:1, 4:1, 8:1 and 16:1. The different digital ICs are given in appendix B, but for sake of convenience some of the MUX/DEMUX ICs are given here in Tables A and B.

Table A: Standard multiplexer ICs

IC No.	Description	Output
74157	Quad. 2:1 Multiplexer	Same as input
74158	Quad 2:1 MUX	Inverted input
74153	Dual 4:1 MUX	Same as input
74352	Dual 4:1 MUX	Inverted input
74151A	8:1 MUX	Complementary outputs
74152	8:1 MUX	Inverted input
74150	16:1 MUX	Inverted input

Table B: Standard Demultiplexer/Decoder ICs

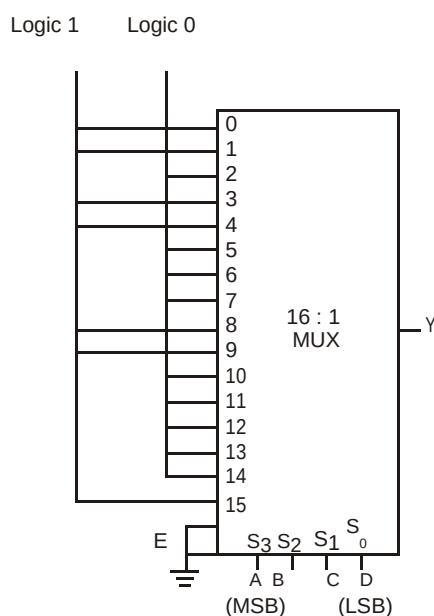
IC No.	Description	Output
74139	Dual 1:4 Demultiplexer (2-line-to-4-line decoder)	Inverted input
74155	Dual 1:4 Demultiplexer (2-line-to-4-line decoder)	1Y-Inverted I/P 2Y-Same as I/P
74138	1:8 Demultiplexer (3-line-to-8-line decoder)	Inverted I/P
74154	1:16 Demultiplexer (4-line-to-16-line decoder)	Same as input

- When using the multiplexer as a logic element either the truth table or one of the standard forms of logic expression must be available.
- The design procedure for combinational circuits using MUX are as follows:

STEP 1: Identify the decimal number correspond-ing to each minterm in the expression. The input lines corresponding to these numbers are to be connected to logic 1 (high).

STEP 2 : All other input lines except that used in step 1 are to be connected to logic 0 (low).

STEP 3 : The control inputs are to be applied to select inputs.



Example 2. Implement the following function with multiplexer.

$$Y = F(A, B, C, D) = \sum m(0, 1, 3, 4, 8, 9, 15)$$

Solution. STEP 1 : The input lines correspond-ing to each minterms (decimal number) are to be connected to logic 1.

Therefore input lines 0, 1, 3, 4, 8, 9, 15 have to be connected to logic 1.

STEP 2 : All other input lines except 0, 1, 3, 4, 8, 9, 15 are to be connected to logic 0.

Qs: Design a full adder using 8:1 multiplexer.

Solution. The truth table of a full adder is given as

A	B	C	S	C _F
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

$$(A, B, C) = \Sigma \quad \text{---} \quad \text{---} \quad \text{---}$$

$$C_F(A, B, C) = \quad \text{---} \quad \text{---} \quad \text{---} \quad \Sigma$$

The implementation for summation expression is

Step 1: The input lines corresponding to 1, 2, 4, 7 are to be connected to logic 1.

Step 2: Other input lines are to be connected to logic 0.

Step 3: Control inputs A, B, C are to be applied to select inputs. Fig. 5.6 A.

Similarly for carry expression.

Step 1: The input lines corresponding to 3, 5, 6, 7 are to be connected to logic 1.

Step 2: Other input lines are to be connected to logic 0.

Step 3: Control inputs A, B, C are to be applied to select inputs. Fig. 5.6 B.

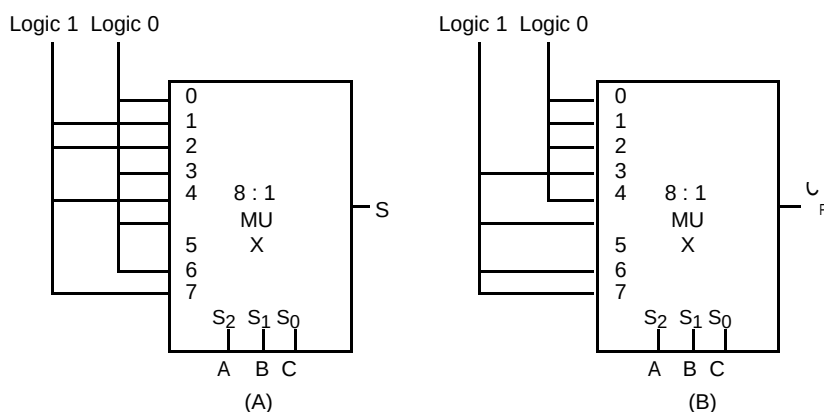


Fig. 5.6 Full adder implementation using 8:1 multiplexer

Encoders

- The encoder is another example of combinational circuit that performs the inverse operation of a decoder.
- It is designed to generate a different output code for each input which becomes active.
- In general, the encoder is a circuit with m input lines $\leq (* m < 2^n \rightarrow$ If unused input combinations occur.) and n output lines that converts an active input signal into a coded output signal.
- In an encoder, the number of outputs is less than the number of inputs. The block diagram of an encoder is shown in Fig..
- An example of an encoder is an octal to binary encoder.
- An octal to binary encoder accept eight inputs and produces a 3-bit output code corresponding to the activated input.
- The truth table for the octal to binary encoder is shown below in table.



Inputs								Outputs		
O_0	O_1	O_2	O_3	O_4	O_5	O_6	O_7	Y_2	Y_1	Y_0
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

The truth table shows that Y_0 must be 1 whenever the input O_1 or O_3 or O_5 or O_7 is high. Thus,

$$Y_0 = O_1 + O_3 + O_5 + O_7$$

Similarly

$$Y_1 = O_2 + O_3 + O_6 + O_7$$

$$Y_2 = O_4 + O_5 + O_6 + O_7$$

Using these three expressions, the circuit can be implemented using three 4-input OR gates as shown in Fig.

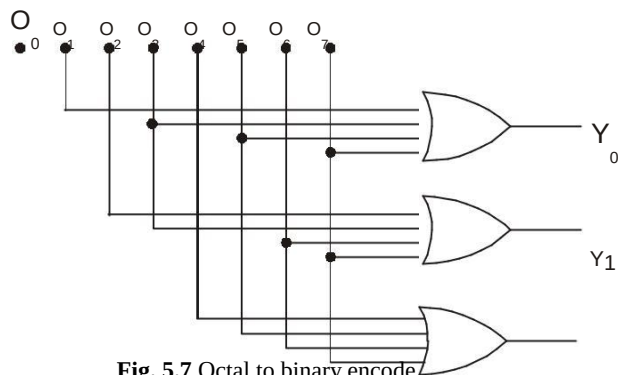


Fig. 5.7 Octal to binary encode

The encoder has two limitations:

- Only one input can be active at any given time. If two or more inputs are equal to 1 at the same time, the O/P is undefined. For example if O_2 and O_5 are active simultaneously, the o/p of encoder will be 111 that is equal to binary 7. This does not represent binary 2 or 5.
- The output with all 0's is generated when all inputs are '0', and is also true when $O_0 = '1'$.
- The first problem is taken care by a circuit, called as 'priority encoder'. It establishes a priority to ensure that only one input is active (High) at a given time.
- The second problem is taken care by an extra line in the encoder output, called 'valid output indicator' that specifies the condition that none of the inputs are active.

Priority Encoder

- A priority encoder is an encoder that includes priority function.
- If two or more inputs are equal to 1 at the same time, the input having the highest priority will take precedence.
- To understand priority encoder, consider a 4 to 2 line encoder which gives priority to higher subscript number input than lower subscript number.
- The truth table is given below.

Truth Table of 4 to 2 line priority encoder:

Inputs				Outputs		
D_0	D_1	D_2	D_3	Y_1	Y_2	V
0	0	0	0	x	x	0
1	0	0	0	0	0	1
x	1	0	0	0	1	1
x	x	1	0	1	0	1
x	x	x	1	1	1	1

- The X's are don't care conditions. Input D_3 has the highest priority, so regardless of values of other inputs, when this input is 1, the output $Y_1 Y_2 = 11$. D_2 has next priority level.
- The o/p is 10 if D_2 is 1, provided $D_3 = 0$, irrespective of the values of the other two lower-priority inputs.
- The o/p is 01 if D_1 is 1, provided both D_2 and D_3 are 0, irrespective of the value of lower-priority input D_0 . The o/p is 00 if $D_0 = 1$, provided all other inputs are 0.
- A valid output indicator, V is set to 1, only when one or more of the inputs are equal to 1. If all the inputs are 0, V is equal to 0 and the other two outputs of the circuit are not used.
- Now, simplifying using k-map the outputs can be written as :

$$Y_1 = D_2 + D_3$$

$$Y_2 = D_3 + D_1 D_2'$$

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

- The logic diagram for a 4 to 2 line priority encoder with 'valid output indicator' is shown below in Fig. 5.8.

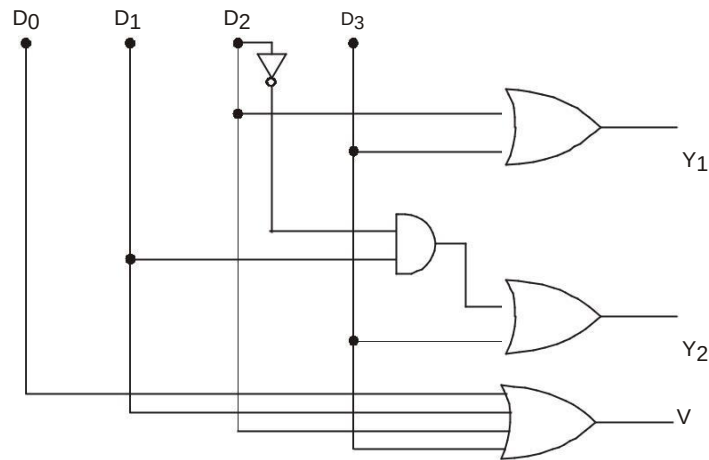


Fig. 5.8

Serial and Parallel Adders

- Full adder is a combinational circuit that adds three binary digits.
- When we add two numbers of any length, the terms we have to deal with are :
- Input carry, Augend, Addend, sum and output carry.
- We simply start adding two binary digits from LSB (rightmost positioned bits).
- At this position, the input carry is always equal to zero.
- After addition, we get sum and output carry.
- This output carry works as the input carry to the next higher positioned augend and addend bits.
- Next we add augend and addend bits alongwith the input carry that again produces sum and output carry.
- The process repeats upto MSB position (leftmost positioned bits).
- We observe that in the process of addition we are actually adding three digits – the input carry, the augend bit and the addend bit.
- And, we are getting two outputs the sum and the output carry.
- This can be illustrated by the following example. Let the 4-bits words to be added be represented by:

$A_3 A_2 A_1 A_0 = 1 1 0 1$ and $B_3 B_2 B_1 B_0 = 0 0 1 1$.

Significant Place	3	2	1	0
Input Carry	1	1	1	0 ← Carry-In
Augend Word	1	1	0	1
Addend Word	0	0	1	1
Sum	0	0	0	0
Output carry	1	1	1	1
Carry-out				

- Now, if we compare this with the full adder circuit, we can easily observe that the two inputs (A and B) are augend and addend bits with the third input (c) as the input carry. Similarly two outputs are sum (s) and output carry (C_0).

- In general, the sum of two n -bit numbers can be generated by using either of the two methods : the serial addition and the parallel addition.

Serial Adder

- In serial addition, the addition operation is carried out bit by bit.
- The serial adder uses one full adder circuit and some storage device (memory element) to hold generated output carry.
- The diagram of a 4 bit serial adder is shown in Fig. 5.9.
- The two bits at the same positions in augend and addend word are applied serially to A and B inputs of the full adder respectively.
- The single full adder is used to add one pair of bits at a time along with the carry C_{in} .
- The memory element is used to store the carry output of the full adder circuit so that it can be added to the next significant position of the numbers in the augend and addend word.
- This produces a string of output bits for sum as S_0, S_1, S_2 and S_3 respectively.

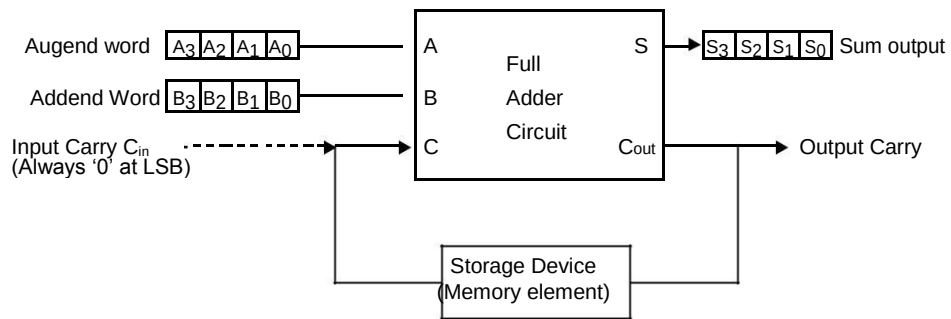


Fig. 5.9 4-bit serial adder

Parallel Adder

- To add two n -bit numbers, the parallel method uses n full adder circuits and all bits of addend and augend bits are applied simultaneously.
- The output carry from one full adder is connected to the input carry of the full adder one position to its left.
- The 4-bit adder using full adder circuit is capable of adding two 4-bit numbers resulting in a 4-bit sum and a carry output as shown in Fig. 4.39.

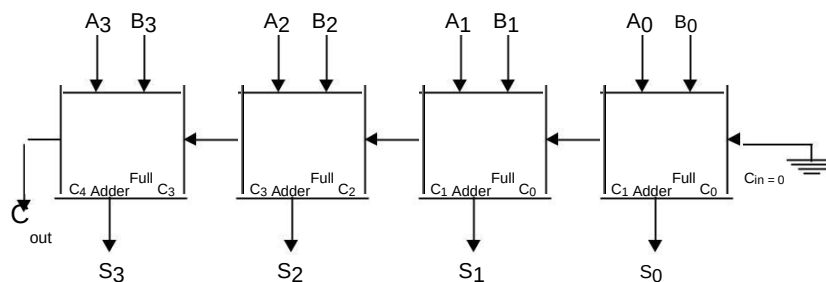


Fig. 4.39 4-bit binary parallel adder

- The addition operation is illustrated in the following example. Let the 4-bit words to be added be represented by $A_3 A_2 A_1 A_0 = 1 0 1 0$ and $B_3 B_2 B_1 B_0 = 0 0 1 1$.

Subscript i	3	2	1	0
Input carry C_i	0	1	0	0
Augend A_i	1	0	1	0
Addend B_i	0	0	1	1
Sum S_i	1	1	0	1
Output carry C_{i+1}	0	0	1	0

← Significant place.

- In a 4-bit parallel adder, the input to each full adder will be A_i , B_i and C_i , and the outputs will be S_i and C_{i+1} , where i varies from 0 to 3.
- In the least significant stage, A_0 , B_0 and C_0 (which is 0) are added resulting in sum S_0 and carry C_1 .
- This carry C_1 becomes the carry input to the second stage. Similarly, in the second stage, A_1 , B_1 and C_1 are added resulting in S_1 and C_2 ; in the third stage, A_2 , B_2 and C_2 are added resulting in S_2 and C_3 ; in the fourth stage A_3 , B_3 and C_3 are added resulting in S_3 and C_4 which is the output carry.
- Thus the circuit results in a sum (S_3, S_2, S_1, S_0) and a carry output (C_{out}).
- An alternative block diagram representation of a 4 bit binary parallel adder is shown in Fig. 5.10.

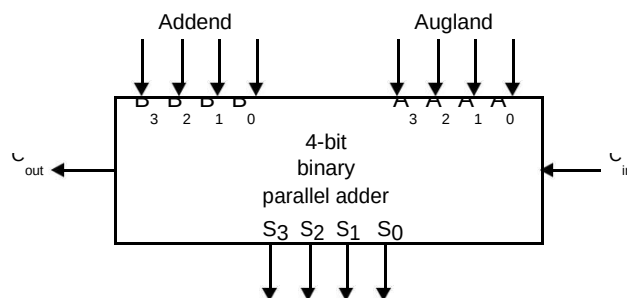


Fig. 5.10 4-bit binary parallel adder

Propagation Delay:

- Though the parallel binary adder is said to generate its output immediately after the inputs are applied, its speed of operation is limited by the carry propagation delay through all stages.
- In the parallel binary adder, the carry generated by the adder is fed as carry input to $(i + 1)$ th adder.
- This is also called as 'ripple carry adder'.
- In such adders, the output ($C_{out}, S_3, S_2, S_1, S_0$) is available only after the carry is propagated through each of the adders, i.e., from LSB to MSB adder through intermediate adders.
- Hence, the addition process can be considered to be complete only after the carry propagation delay through adders, which is proportional to number of stages in it; one of the methods of speeding up this process is look-ahead carry addition, which eliminates the ripple carry delay.
- This method is based on the carry generating and the carry propagating functions of the full adder.

4-bit Look-ahead Carry Generator

- The look-ahead carry generator is based on the principle of looking at the lower order bits of addend and augend if a higher order carry is generated.
- This reduces the carry delay by reducing the number of gates through which a carry signal must propagate.
- To explain its operation consider the logic diagram of a full adder circuit Fig. 5.11.

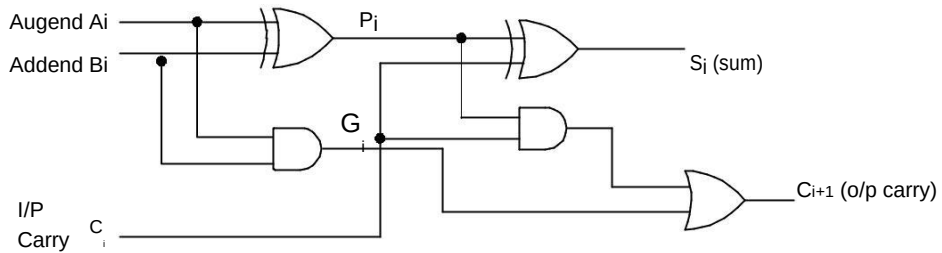


Fig. 5.11 Full adder

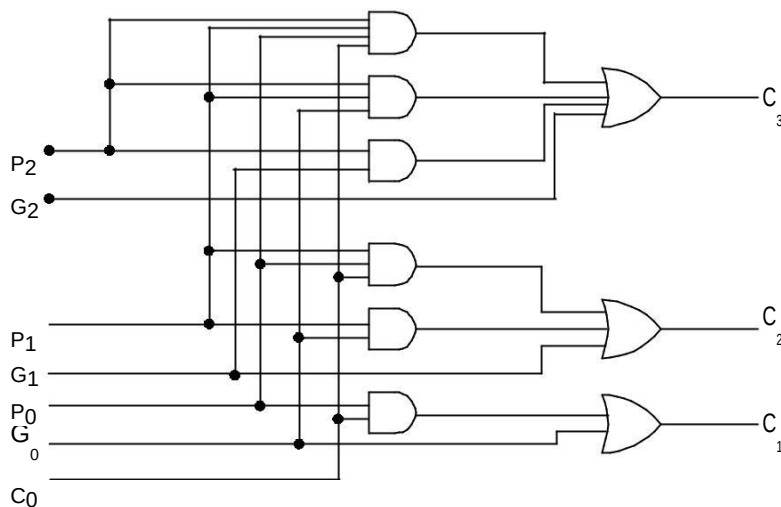
- We define two new intermediate output variable P_i and G_i .
 $P_i = A_i \oplus B_i$; called carry propagate, and
 $G_i = A_i \cdot B_i$; called carry generate.
- We now write the Boolean function for the carry output of each stage and substitute for each C_i its value from the previous equations :

$$C_1 = G_0 + P_0 C_0$$

$$C_2 = G_1 + P_1 C_1 = G_1 + P_1 (G_0 + P_0 C_0) = G_1 + P_1 G_0 + P_1 P_0 C_0.$$

$$C_3 = G_2 + P_2 C_2 = G_2 + P_2 (G_1 + P_1 G_0 + P_1 P_0 C_0)$$

$$= G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0.$$
- Note that C_3 does not have to wait for C_2 and C_1 to propagate; in fact C_3 is propagated at the same time as C_1 and C_2 .
- Next we draw the logic diagram of this 4 bit look-ahead carry generator as shown in Fig. 5.12



4-bit binary parallel adder with a look-ahead carry generator (FAST ADDER)

- In the 4-bit look ahead carry generator.
- We have seen that all the carry outputs are generated simultaneously with the application of Augend word, addend word and the input carry.
- What is remaining are the sum outputs. From the newly defined full adder circuit of Fig. 4.41, we notice that the sum output $S_i = P_i \oplus C_i$.

$$S_0 = P_0 \oplus C_0 = A_0 \oplus B_0 \oplus C_0$$

$$S_1 = P_1 \oplus C_1 = A_1 \oplus B_1 \oplus C_1$$

$$S_2 = P_2 \oplus C_2 = A_2 \oplus B_2 \oplus C_2$$

$$S_3 = P_3 \oplus C_3 = A_3 \oplus B_3 \oplus C_3$$

Similarly carry output

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

Therefore, Final o/p carry

$$C_4 = G_3 + P_3 C_3$$

- Using the above equations, the 4-bit binary parallel adder with a look ahead carry generator can be realized as shown in Fig. 5.13.

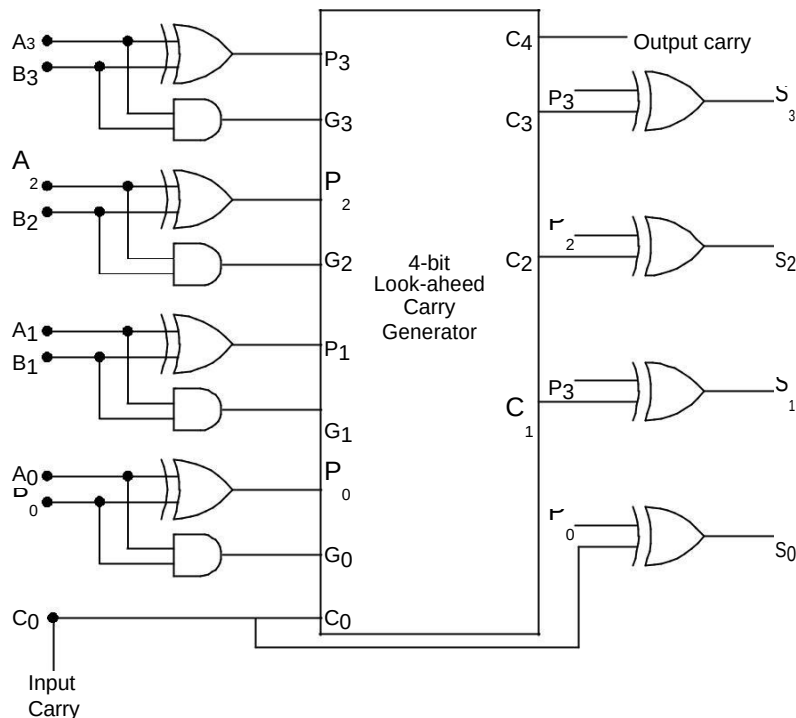


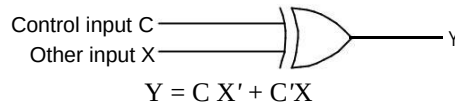
Fig. 5.13 Four bit binary parallel adder with look-ahead carry generator

From the diagram, note that the addition of two 4 bit numbers can be done by a look ahead carry generator in a 4 gate propagation time (4 stage implementation). Also, it is important to realize that the addition of n-bit binary numbers takes the same 4-stage propa-gation delay.

4-bit Parallel Adder/Subtractor

- The 4-bit binary parallel adder/subtractor can be realized with the same circuit taking into consideration the 2's complement method of subtraction and the controlled inversion property of the Exclusive OR gate.
- The subtraction of two binary number by taking 2's complement of the subtrahend and adding to the minuend.

- The 2's complement of the subtrahend can be obtained by adding 1 to the 1's complement of the subtrahend.
- From the Exclusive OR operation, we know when one of the input is low the output is same as the other input and when one of the input is high the output is the complement of the other input.



Naturally, if

$$C = 0, Y = X$$

$$C = 1, Y = X'$$

- The 4-bit binary parallel adder/subtractor circuit is shown in Fig. 5.14. that perform the operation of both addition and subtraction.
- It has two four bit inputs $A_3 A_2 A_1 A_0$ and $B_3 B_2 B_1 B_0$.
- The control input line C , connected with the input carry of the LSB of the full adder, is used to perform both operations.
- To perform subtraction, the C (control input) is kept high.
- The controlled inverter produces the 1's complement of the addend ($B_3' B_2' B_1' B_0'$). Since 1 is given to input carry of the LSB of the adder, it is added to the complemented addend producing 2's complement of the addend before addition.

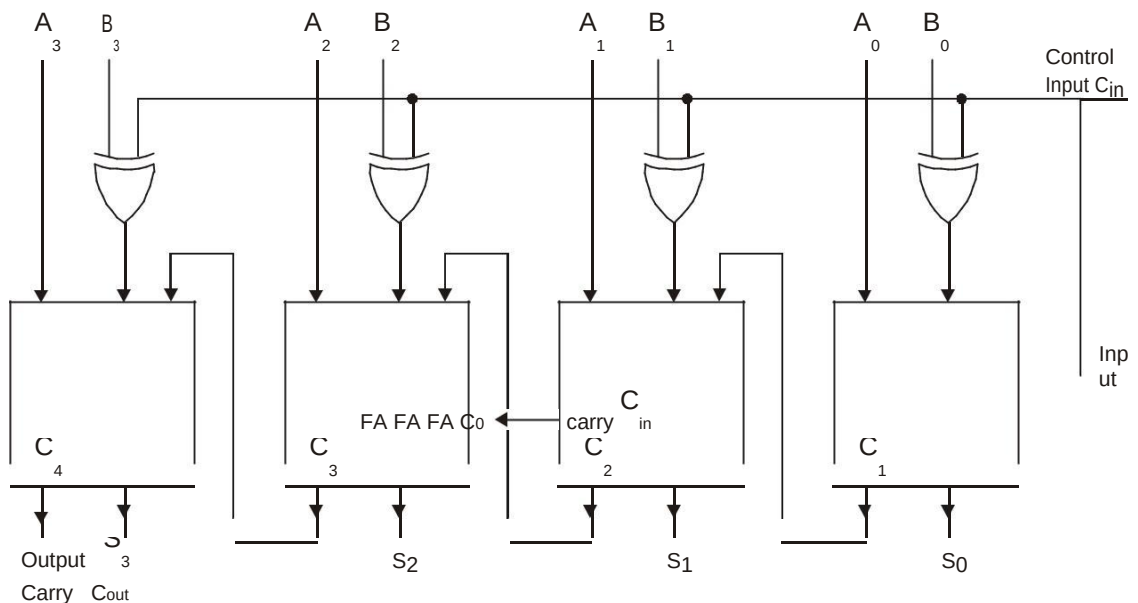


Fig. 5.14 Bit binary parallel adder/subtractor

- Now the augend ($A_3 A_2 A_1 A_0$) will be added to the 2's complement of addend ($B_3 B_2 B_1 B_0$) to produce the sum, i.e., the difference between the addend and augend, and C_{out} (output carry), i.e. the borrow output of the 4-bit subtractor.
- When the control input ' C ' is kept low, the controlled inverter allows the addend ($B_3 B_2 B_1 B_0$) without any change to the input of full adder, and the input carry C_{in} of LSB of full adder, becomes zero
- Now the augend ($A_3 A_2 A_1 A_0$) and addend ($B_3 B_2 B_1 B_0$) are added with $C_{in} = 0$.
- Hence, the circuit functions as 4-bit adder resulting in sum $S_3 S_2 S_1 S_0$ and carry output C_{out} .

Decimal Adder

- A BCD adder is a combinational circuit that adds two BCD digits in parallel and produces a sum digit which is also in BCD.
- The block diagram for the BCD adder is shown in Fig. 5.15. This adder has two 4-bit BCD inputs $A_8 A_4 A_2 A_1$ and $B_8 B_4 B_2 B_1$ and a carry input C_{in} .
- It also has a 4-bit sum output $S_8 S_4 S_2 S_1$ and a carry output C_{out} . Obviously the sum output is in BCD form. (This is why subscripts 8, 4, 2, 1 are used).
- If we consider the arithmetic addition of two decimal digits in BCD, the sum output can not be greater than $9 + 9 + 1 = 19$. (Since each input digit does not exceed 9 and 1 being the possible carry from previous stage).
- Suppose, we apply two BCD digits to a 4-bit binary parallel adder.
- The adder will form the sum in binary and produce a sum that will range from 0 to 19. But if we wish to design a BCD adder, it must be able to do the following.
- Add two 4-bit BCD numbers using straight binary addition.
- If the four bit sum is equal to or less than 9, the sum is in proper BCD form.
- If the four bit sum is greater than 9 or if a carry is generated from the sum, the sum is not in BCD form.
- In this case a correction is required that is obtained by adding the digit 6 (0110) to the sum produced by binary adder.

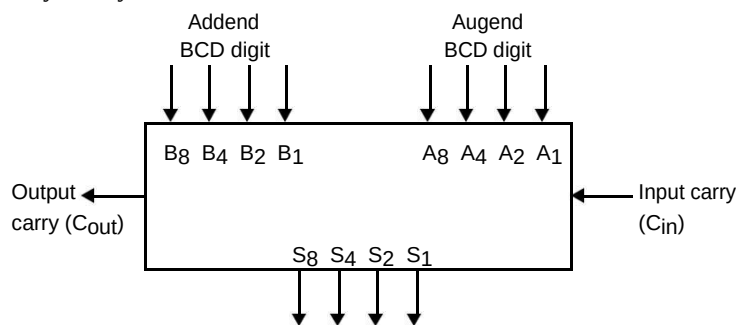


Fig. 5.15 Block diagram of a BCD adder

The BCD adder can be implemented using two 4-bit binary parallel adders as shown in Fig. 5.16.

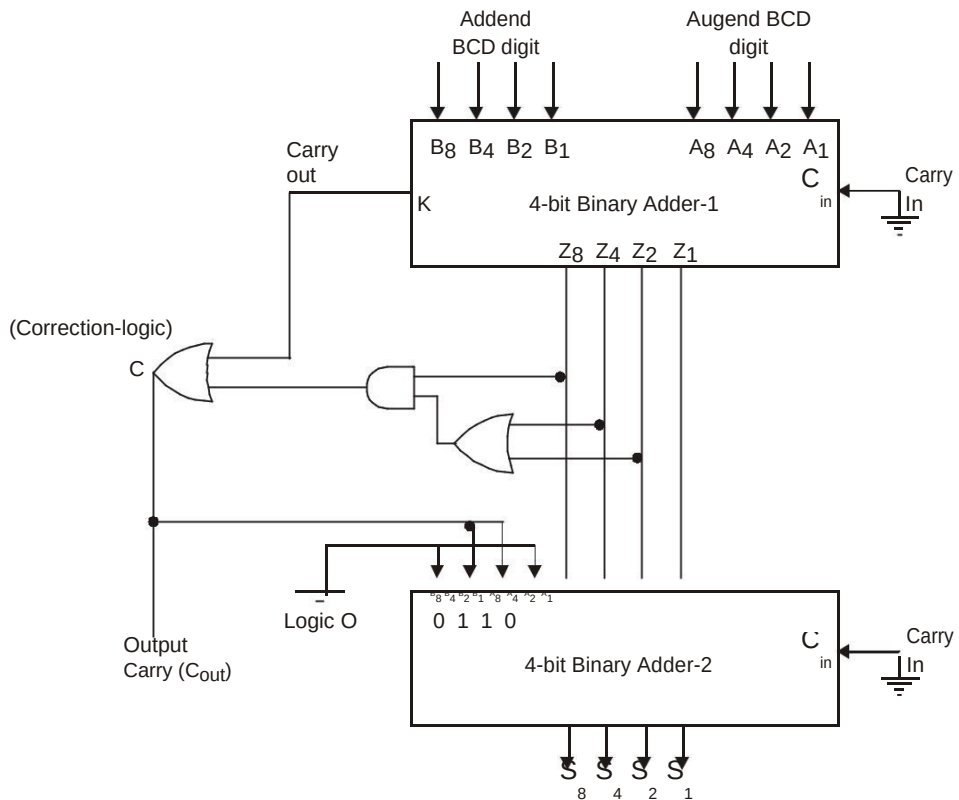


Fig. 5.16 BCD adder using two 4-bit binary adders along with the correction logic C

- Here A₈ A₄ A₂ A₁ and B₈ B₄ B₂ B₁ are the BCD inputs.
- The two BCD inputs with input carry C_{in} are first added in the 4-bit binary adder-1 to produce the binary sum Z₈, Z₄, Z₂, Z₁ and output carry K.
- The outputs of adder-1 are checked to ascertain wheather the output is greater than 9 by AND-OR logic circuitry.
- If correction is required, then a 0110 is added with the output of adder-1.
- Now the 4-bit binary adder-2 forms the BCD result (S₈ S₄ S₂ S₁) with carry out C.
- The output carry generated from binary adder-2 can be ignored, since it supplies information already available at output carry terminal C.

Magnitude Comparator

- A magnitude comparator is a combinational circuit designed primarily to compare the relative magnitude of the two binary numbers A and B.
- Naturally, the result of this compari-son is specified by three binary variables that indicate, wheather A > B, A = B or A < B.
- The block diagram of a single bit magnitude comparator is shown in Fig. 5.17.

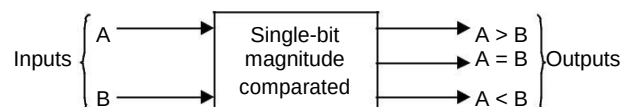


Fig. 5.17 Block diagram of single bit magnitude comparator

- To implement the magnitude comparator the properties of Ex-NOR gate and AND gate is used.

Fig. 5.18(a) shows an EX-NOR gate with two inputs A and B. If $A = B$ then the output of Ex-NOR gate is equal to 1 otherwise 0.

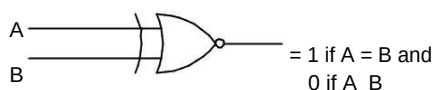


Fig. 5.18 (a)

Fig. 5.18 (b) and (c) shows AND gates, one with A and B' as inputs and another with A' and B as their inputs.

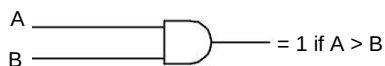


Fig. 5.18 (b)

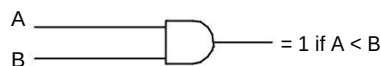


Fig. 5.18(c)

The AND gate output of 4.48(b) is 1 if $A > B$ (i.e. $A = 1$ and $B = 0$) and 0 if $A < B$ (i.e. $A = 0$ and $B = 1$). Similarly the AND gate output of 4.48(c) is 1 if $A < B$ (i.e. $A = 0$ and $B = 1$) and 0 if $A > B$ (i.e. $A = 1$ and $B = 0$).

If the EX-NOR gate and two AND gates are combined as shown in Fig. 5.19(a), the circuit with function as single bit magnitude comparator. For EX-NOR implementation.

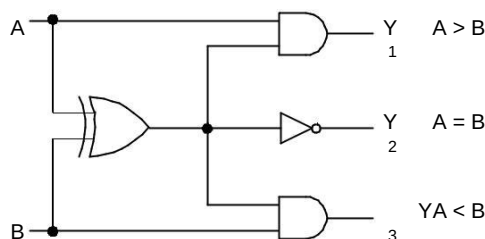


Fig. 5.19 (a) Single bit magnitude comparator

We have used EX-OR followed by an inverter.

Truth table of a single bit magnitude comparator.

Inputs		Output		
A	B	Y_1	Y_2	Y_3
0	0	0	1	0
0	1	0	0	1
1	0	1	0	0
1	1	0	1	0

It clearly shows Y_1 is high when $A > B$.

Y_2 is high when $A = B$

Y_3 is high when $A < B$.

The same principle can be extended to an n-bit magnitude comparator.

4-bit Magnitude Comparator

Consider two numbers A and B, with four digits each.

$$= A_3 A_2 A_1 A_0$$

$$= B_3 B_2 B_1 B_0.$$

- (a) The two numbers are equal if all pairs of significant digits are equal i.e. if $A_3 = B_3$, $A_2 = B_2$, $A_1 = B_1$ and $A_0 = B_0$. We have seen that equality relation is generated by EX-NOR gate. Thus

$x_i = A_i \cdot B_i = A_i B_i + A_i' B_i'$, $i = 0, 1, 2, 3$. Where x_i represents the equality of two numbers $x_i = 1$, if $A = B$.

$x_i = 0$, otherwise.

It follows an AND operation of all variables.

Therefore, $(A = B) = x_3 x_2 x_1 x_0 = 1$ only if all pairs are equal.

- (b) To determine if $A > B$ or $A < B$, we check the relative magnitude of pairs of significant digits starting from MSB. If the two digits are equal, we compare the next lower significant pair of digits. The comparison follows until a pair of unequal digits are reached. If the corresponding digit of A is 1 and that of B is 0, we say that $A > B$. If the corresponding digit A is 0 and that of B is 1 $\Rightarrow A < B$.

This discussion can be expressed logically as :

$$(A > B) = A_3 B_3' + x_3 A_2 B_2' + x_3 x_2 A_1 B_1' + x_3 x_2 x_1 A_0 B_0' \quad (A < B) = A_3' B_3 + x_3 A_2' B_2 + x_3 x_2 A_1' B_1 + x_3 x_2 x_1 A_0' B_0.$$

The logical implementation is shown in Fig. 5.19(b)

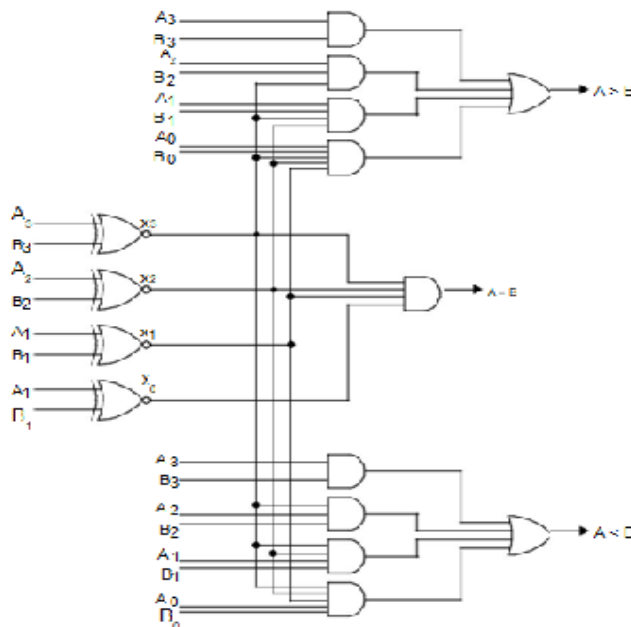


Fig. 4.49(b) Logical implementation of 4-bit magnitude comparator

READ ONLY MEMORY (ROM)

- A Read Only Memory (ROM) as shown in Fig. 5.1 is a matrix of data that is accessed one row at a time. It consists of k input lines and n output lines.
- Each bit combination of output lines is called a word while each bit combination of input variables is called an address.
- The number of bits per word is equal to the number of output lines n . A ROM is describe by the number of words 2^k , the number of bits per word n .
- The A inputs are address lines used to select one row (called a word) of the matrix for access. If $A = i$, then row i is selected and its data appears on the output terminals D .
- In this case we say that the contents of row i are read from the memory.

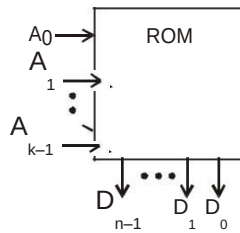


Fig. 5.1

- If there are k address lines, then there are 2^k words in the ROM .
- The number of bits per word is called the word size.
- The data values of the words are called the contents of the memory and are said to be stored in the memory.
- The term read only refers to the property that once data is stored in a ROM, either it cannot be changed, or it is not changed very often.
- ROM can be viewed as a combinational circuit with AND gates connected as a decoder and number of OR gates equal to the number of outputs.
- Internally a ROM contains a decoder and a storage array as shown in the Fig. 5.2.

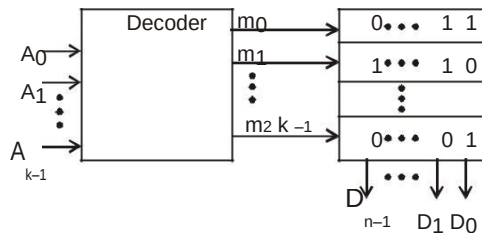


Fig. 5.2

- When the address is i , the i th output of the decoder m_i is activated selecting row i of the data array.
- Functionally the data array can be viewed as a programmable OR array.
- Each column acts as a stack of OR gates as shown in the Fig. 5.3.

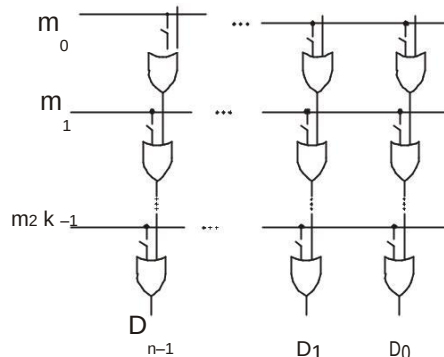


Fig. 5.3

- Depending on the stored value (0/1) switch is open or closed. If a 0 is stored in a row, the switch is open and if a 1 is stored, the switch is closed.
- The type of ROM is determined by the way the switches are set or reset (i.e., programmed).

Mask programmed ROMs:

In the mask programmed ROMs, switch is realized at the time the ROM is manufactured. Either a connection is made by putting in a wire, or the connection is left open by not putting in a wire.

Field programmable ROMs (PROMs):

In the field programmable ROMs, switch is realized by a fuse. When the ROM is manufactured all switches are closed since all the fuses are intact. To open a switch the fuse is blown by sending a larger than usual current through it. Once a fuse is blown, it cannot be reinstalled.

Erasable ROMs (EPROMs):

In the erasable ROMs switch is realized by a special kind of fuse that can be restored to its usual closed state, usually by the insertion of extra energy (e.g., shining ultraviolet light on the fuse). All fuses are reset when this is done.

Electrically Programmable ROMs (EEPROMs):

In the electrically programmable ROMs fuses are reset by the application of larger than usual currents. Sometimes subsections of the ROM can be reset without resetting all fuses.

- Consider a 32×8 ROM as shown in Fig. 5.4.
- The ROM consists of 32 words of bit size 8.
- That is, there are 32 distinct words stored which may be available through eight output lines.
- The five inputs are decoded into 32 lines and each output of the decoder represents one of the minterms of a function of five variables.
- Each one of the 32 addresses selects only one output from the decoder.
- The 32 outputs of the decoder are connected through links to each OR gate.

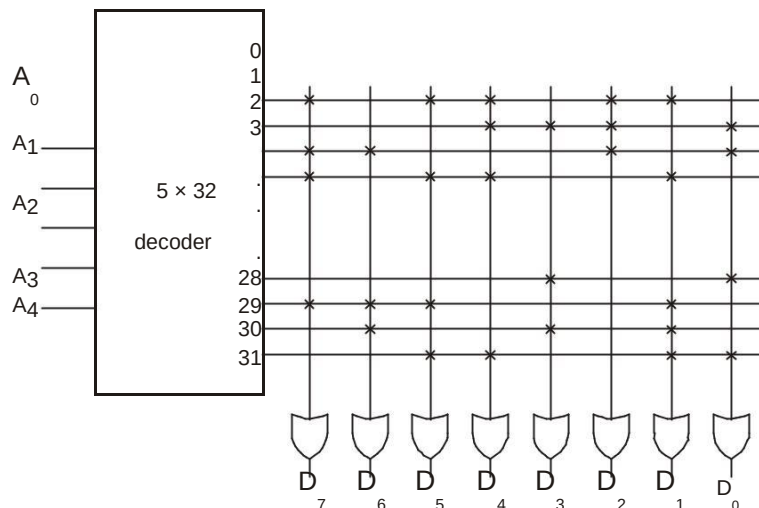


Fig. 5.4

Realizing Logical Functions with ROM

- The ROM is a two-level implementation in sum of minterms form.
- ROMs with k address lines and n data terminals can be used to realize any n logical functions of k variables.
- For this one have to simply store the truth table for each function in a column of the ROM data array.
- The advantage of implementing logical functions by means of some form of program-mable ROM, we have the possibility of reconfigurable logic.

- That is, the same hardware being reprogrammed to realize different logic at different times.
- But the disadvantage of using large ROMs to realize logical functions is that, the ROMs are much slower than gate realizations of the functions.
- The following example explains the procedure for realizing logical functions.

Example 1. Design a combinational circuit using a ROM that accepts a 2-bit number and generates an output binary number equal to the square of the input number.

Step 1: Derive the truth table for the combinational circuit. For the given example the truth table is

Table 5.1

Inputs		Outputs				Equivalent decimal
A_1	A_0	B_3	B_2	B_1	B_0	
0	0	0	0	0	0	0
0	1	0	0	0	1	1
1	0	0	1	0	0	4
1	1	1	0	0	1	9

Step 2: If possible, reduce the truth table for the ROM by using certain properties in the truth table of the combinational circuit. For the given example, two inputs and four outputs are needed to accommodate all possible numbers.

Since output B_0 is always equal to input A_0 , therefore there is no need to generate B_0 with a ROM. Also B_1 is known as it is always 0. Therefore we need to generate only two outputs with the ROM; the other two are easily obtained.

Step 3: Find the minimum size of ROM from step 2.

The minimum size ROM needed must have two inputs and two outputs. Two inputs specify four words, so the ROM size must be 4×2 . The two inputs specify four words of two bits each. The other two outputs of the combinational circuit are equal to 0 (B_1) and $A_0(B_0)$.

Table 5.2 ROM truth table

A_1	A_0	B_3	B_2
0	0	0	0
0	1	0	0
1	0	0	1
1	1	1	0

The ROM truth table (Table 5.2) specifies complete information for ROM programming and the required connections are shown in Fig. 5.5.

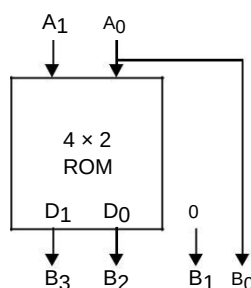


Fig. 5.5 Block diagram

Example. For the function

$$F_1 = \Sigma(0, 2, 5, 6) \quad F_2 = \Sigma(0, 2, 4, 6, 7) \quad F_3 = \Sigma(0, 2, 4, 7) \quad F_0 = \Sigma(1, 2, 3, 5, 7)$$

Design a three bit addressable ROM.

Solution. From truth table, the minimum size of ROM needed having three inputs and four outputs.

Three inputs specify $2^3 = 8$ word so the ROM size must be 8×4 .

Table 5.4

Minterms	Inputs			Outputs			
	A_2	A_1	A_0	F_3	F_2	F_1	F_0
0	0	0	0	1	1	1	0
1	0	0	1	0	0	0	1
2	0	1	0	1	1	1	1
3	0	1	1	0	0	0	1
4	1	0	0	1	1	0	0
5	1	0	1	0	0	1	1
6	1	1	0	0	1	1	0
7	1	1	1	1	1	0	1

The implementation of functions are given in Fig. 5.7.

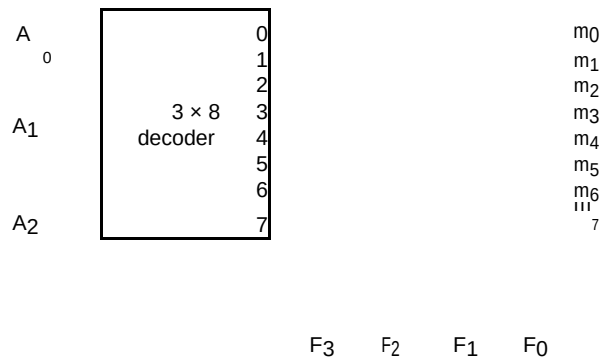


Fig. 5.7

PROGRAMMABLE LOGIC ARRAYS

- The first type of user-programmable chip that could implement logic circuits was the Programmable Read-Only Memory (PROM), in which address lines can be used as logic circuit inputs and data lines as outputs. Fig. 5.8 shows the basic configuration of PROM.

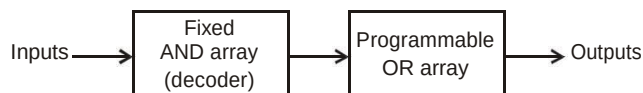


Fig. 5.8 Basic configuration of programmable read-only memory (PROM)

- Logic functions, however, rarely require more than a few product terms, and a PROM contains a full decoder for its address inputs.

- PROMs are thus an inefficient architecture for realizing logic circuits, and so are rarely used in practice for that purpose.
- The first device developed later specifically for implementing logic circuits was the Field-Programmable Logic Array (FPLA), or simply PLA for short.
- A PLA consists of two levels of logic gates: a program-mable “wired” AND-plane followed by a programmable “wired” OR-plane.

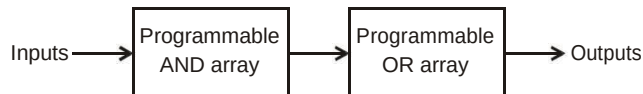
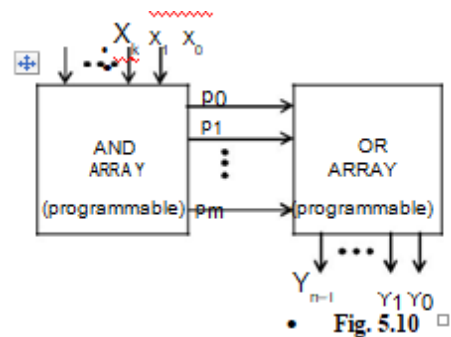


Fig. 5.9 Basic configuration of programmable logic array (PLA)

- A PLA is structured so that any of its inputs (or their complements) can be AND’ed together in the AND-plane; each AND-plane output can thus correspond to any product term of the inputs.
- Similarly, each OR plane output can be configured to produce the logical sum of any of the AND-plane outputs.
- With this structure, PLAs are well-suited for implementing logic functions in sum-of-products form.
- They are also quite versatile, since both the AND terms and OR terms can have many inputs (this feature is often referred to as *wide* AND and OR gates).
- Programmable Logic Arrays (PLAs) have the same programmable OR array as a ROM, but also have a programmable AND array instead of the de-coder as shown in Fig. 5.10.
- The programmable AND array can be used to produce arbitrary product terms, not just minterms.
- While the decoder produces all minterms of k variables.
- Since, the number of possible product terms m in PLA is much less than the number of possible minterms 2^k , so some functions may not be realizable in PLA.



The structure of a row of the programmable AND array is shown in Fig. 5.11 (a).

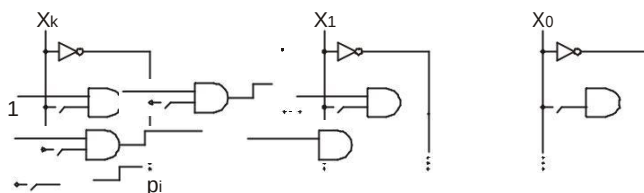


Fig. 5.11 (a)

and of a column of the programmable OR array is shown in Fig. 5.11 (b).

While the notation of a PLA is shown in Fig. 5.12.

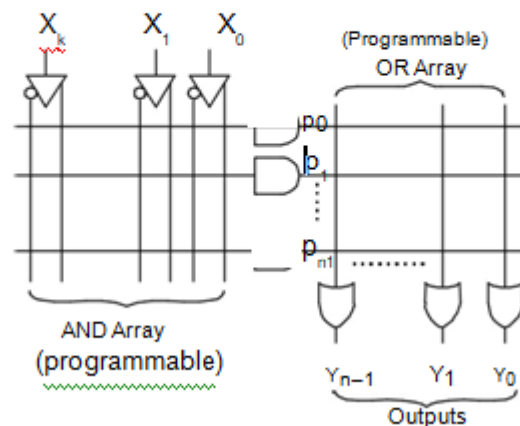


Fig. 5.12

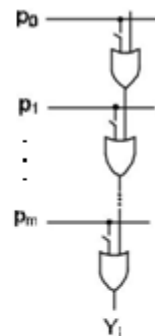
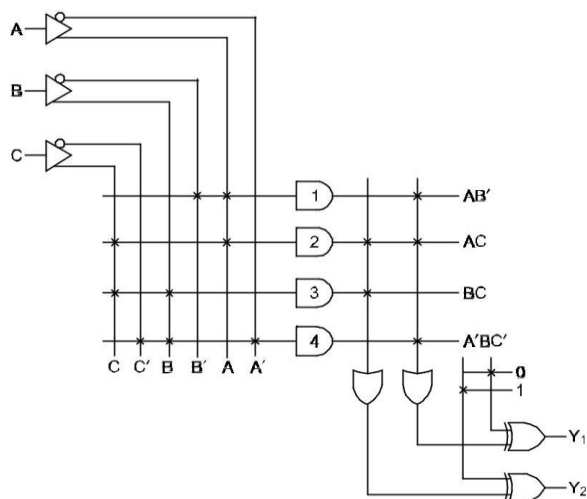


Fig. 5.11 (b)

Realizing Logical Functions with PLAs

- During implementation (or programming) a dot (or cross) is placed at an intersection if the variable is to be included in the product term or to sum term.
- For example a PLA with 3 inputs, 4 product terms and 2 outputs is shown in Fig. 5.13.



From the exclusive OR operation, we know when one of the input is low the output is same as the other input and when one of the input is high the output is the complement of the other input. (Fig. 5.14).

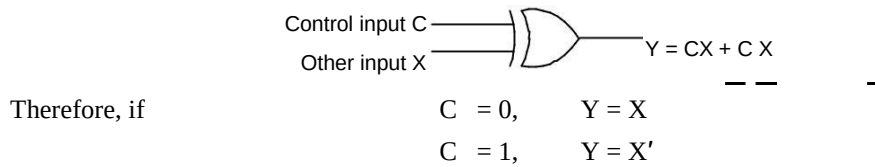


Fig. 5.14

In PLA, the output is present in true form as well as in complement form. In Fig. 5.13, Y_1 is in true form while output Y_2 is complemented.

When realizing (designing) a digital system with a PLA, there is no need to show the internal connections as in Fig. 5.13. The PLA can be programmed with PLA program table which provides appropriate paths.

PLA Program Table

- A typical program table consists of three columns.
- The first column lists the product terms numerically.
- The second column specifies the required paths between inputs and AND gates.
- The third column specifies the paths between the AND gates and the OR gates. Under each output variable we write a T (for true) if the output is in true form and C (for complement) if the output is to be complement.
- For each product term, the inputs are marked as 1, 0 or – (dash).
- A variable in product term is marked as 1 if it appears in normal form, 0 if it is in complemented form and ‘–’ shows that a variable is not present.
- In PLA implementation of combinational circuits the total number of distinct product terms must be reduced by simplifying each function to a minimum number of terms.