UNIT - III
COMBINATIONAL LOGIC CIRCUITS

Overview

- Gates, latches, memories and other logic components are used to design computer systems and their subsystems.
- Good understanding of digital logic is necessary in order to learn the fundamentals of computing systems organization and architecture.

- Two types of digital logic:

  - **Combinatorial logic:** Output is a function of inputs

  - **Sequential logic:** Output is a complex function of inputs, previous inputs and previous outputs

- Neither combinatorial logic nor sequential logic is better than the other. In practice, both are used as appropriate in circuit design.

- **Combinatorial logic** – The output of this type of logic is dependent solely on its current inputs. When certain input values are set, a combinatorial circuit generates output values corresponding to those input values. When the input of the combinatorial logic is changed, the outputs are changing to reflect the changes in the new input values. Previous values of the inputs do not matter; the current outputs depend solely on the current inputs.

- **Sequential logic** – The outputs of a sequential logic circuit depend on both the current inputs and on previous inputs and outputs of the circuit. Sequential elements have storage elements that record the state of the circuit. In other words, the state information combined with the inputs is generating the outputs. The state and inputs also combine to generate a new state of the circuit. The same inputs in a sequential circuit may generate different outputs and different new states, depending on the circuit's current state.

  In practice both types of logic are used. The sequential logic include combinatorial logic but the reverse is not true.

Note:

Combinational circuits:

- The outputs are a function of the current inputs.

Sequential circuits:

- Contain memory elements.
- The outputs are a function of the current inputs and the state of the memory elements.
- The outputs also depend on past inputs.

COMBINATIONAL LOGIC CIRCUITS:

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.

- A combinational circuit has m – input variables, 2^m possible combinations of input values and n-output variables.

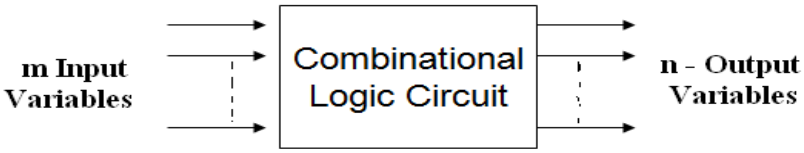


Fig: Block diagram of a combinational circuit.

- 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 [$n < 2^m$].

Specific functions

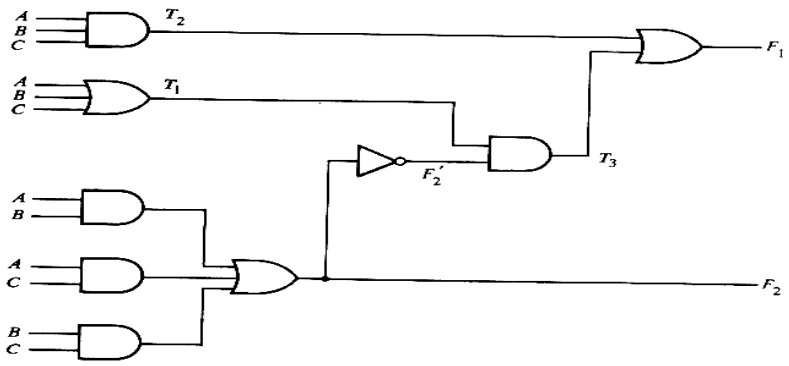
- Adders, subtractors , comparators, decoders, encoders, and multiplexers
- MSI circuits or standard cells.

Analysis Procedure:

The Analysis of a combinational circuit is the Procedure by which we can determine the function that the circuit implements.

- Make sure that it is combinational not sequential with no feedback path or memory elements.
- Label all the gate outputs that are function of input variables with arbitrary symbols, and determine the Boolean function for each gate output.
- Label the gates that are function of input variables and previously labelled gates and determine the Boolean function for them.
- Repeat the above step until the Boolean function for outputs of the circuit are obtained.
- Finally, substituting previously defined Boolean functions, obtain the output Boolean function in terms of input variables.

Example:



A straight-forward procedure

- F_2 = AB+AC+BC
- T_1 = A+B+C
- T_2 = ABC
- T_3 = $F_2'T_1$
- F_1 = T_3+T_2
- F_1 = T_3+T_2 = $F_2'T_1$+ABC
  = (AB+AC+BC)'(A+B+C) +ABC
  = (A'+B') (A'+C') (B'+C') (A+B+C) +ABC
  = (A'+B'C') (AB'+AC'+BC'+B'C) +ABC
  = A'BC'+A'B'C+AB'C'+ABC

Once the Boolean function of outputs of circuit is known we can easily determine the truth table using the following steps:

- Determine the number of input variables in the circuit. For n inputs, list the binary numbers from 0 to 2^{n-1} in a table.
- Determine the output of selected gates for all input combinations.
- Obtain the truth table for the outputs of those gates that are a function of previously defined values.
- Repeat the above step until we get truth table for final outputs.

The truth table:

A	B	C	F_2	F_2'	T_1	T_2	T_3	F_1
0	0	0	0	1	0	0	0	0
0	0	1	0	1	1	0	1	1
0	1	0	0	1	1	0	1	1
0	1	1	1	0	1	0	0	0
1	0	0	0	1	1	0	1	1
1	0	1	1	0	1	0	0	0
1	1	0	1	0	1	0	0	0
1	1	1	1	0	1	1	0	1

Design Procedure:

The design procedure of combinational circuits

- State the problem definition (system specification).
- Determine the inputs and outputs.
- The input and output variables are assigned symbols.
- Derive the truth table indicating the relationship between input and output variables. List all the combination of input variables.
- Derive the simplified Boolean functions for each output in terms of input variables.
- Draw the logic diagram and verify the correctness.

Functional description

- Boolean function.
- HDL (Hardware description language)
  - Verilog HDL
  - VHDL
- Schematic entry.

Logic minimization

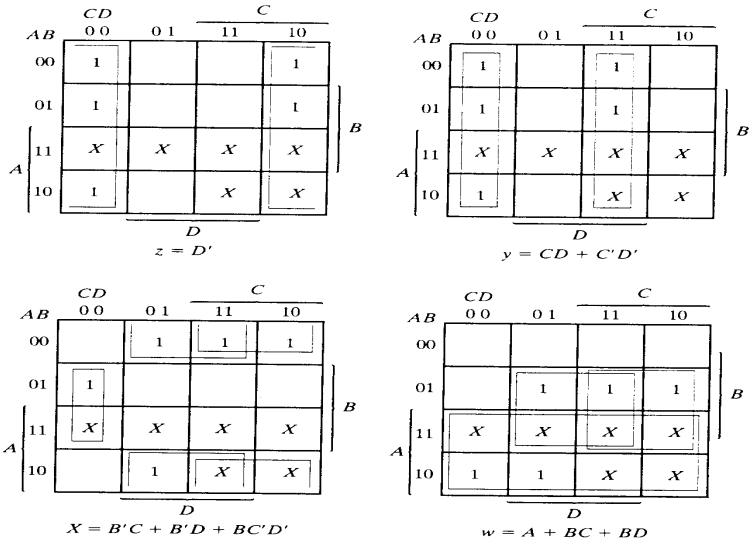
- Number of gates
- Number of inputs to a gate
- Propagation delay
- Number of interconnection &
- Limitations of the driving capabilities.

Example: **Code conversion:** BCD to excess-3 code

The truth table

A	B	C	D	w	x	y	z
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	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

Simplification using K-Map:



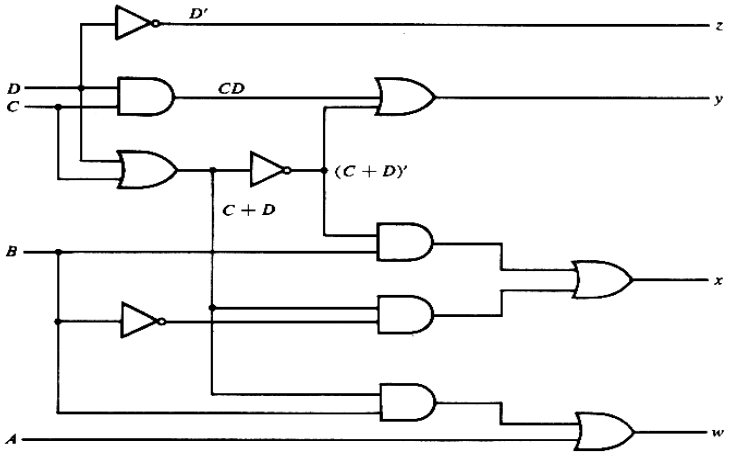
The simplified functions

- z = D'
- y = CD +C'D'
- x = B'C + B'D+BC'D'
- w = A+BC+BD

Another implementation

- z = D'
- y = CD +C'D'
  = CD + (C+D)'
- x = B'C + B'D+BC'D‘
  = B'(C+D) +B(C+D)'
- w = A+BC+BD

The logic diagram:



ARITHMETIC 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.

- Adders are used not only to perform addition but also to perform subtraction, multiplication and division.
- The most basic adder is the half adder.
- Inputs two 1-bit values, A and B, and outputs their 2-bit sum as bits C (Carry) and S (Sum).

Half Adder

1. Fig below shows the block diagram of half adder (HA).

- 0+0=0 ; 0+1=1 ; 1+0=1 ; 1+1=10
- Two input variables: A, B.
- wo output variables: C (carry), S (sum)

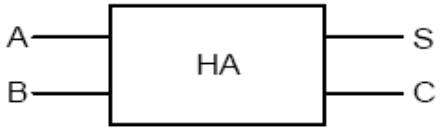


Fig: Half adder

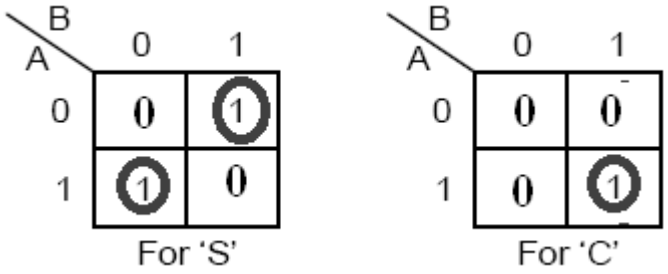
It has two inputs A and B, that are two 1-bit members, and two output sum (S) and carry (C) produced by addition of two bits.

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

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.

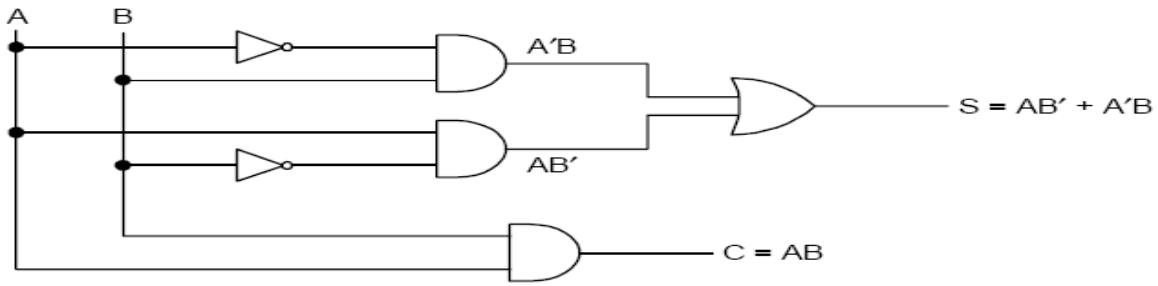


S = AB' + A'B
= A ⊕ B.

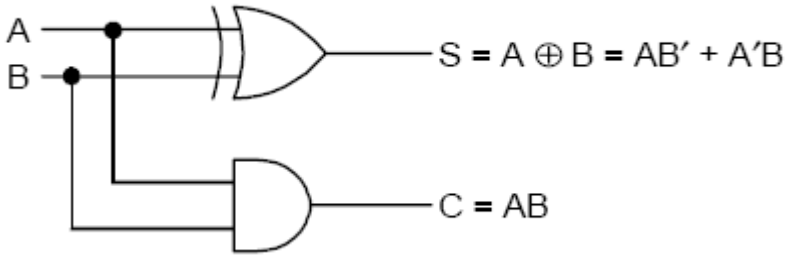
C= AB

4. Logical Implementation.

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



(ii) Using XOR gate as shown in Fig. below.



Full Adder:

Full adder is a combinational circuit that performs the addition of three binary digits.

1. Fig. below shows a full adder (FA). It has three inputs A, B and C and two outputs S and Co produced by addition of three input bits. Carry output is designated Co just to avoid confusion between with I/p variable C.

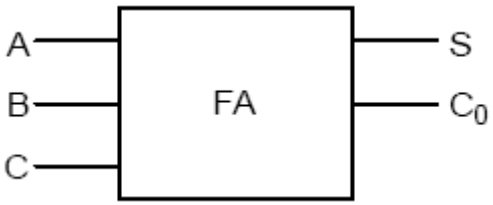
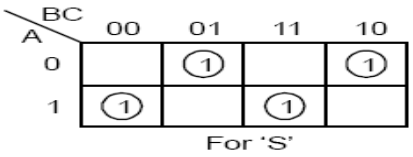


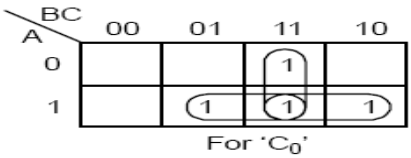
Fig: Full adder

2. **Truth Table:** The eight possible combinations of three input variables with their respective outputs are 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_o
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

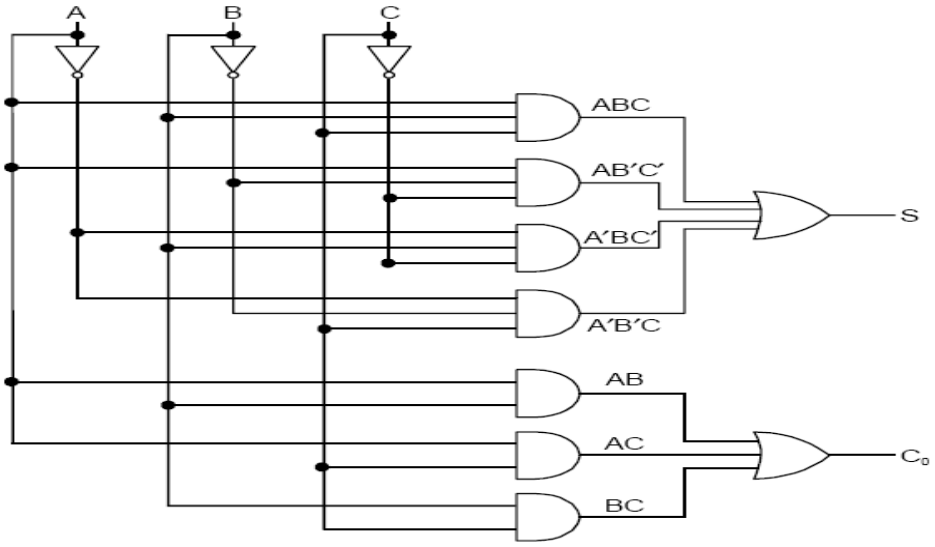
3. Using a three variable map for both outputs.





S = ABC + AB'C' + A'BC' + A'B'C and C_0 = AB + AC + BC.

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



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

The Sum, 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 ⊕ B) ⊕ C

and the carry 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 ⊕ B)
= AB + C (A ⊕ B)

Therefore, S = (A ⊕ B) ⊕ C and C_0 = AB + C (A ⊕ B)

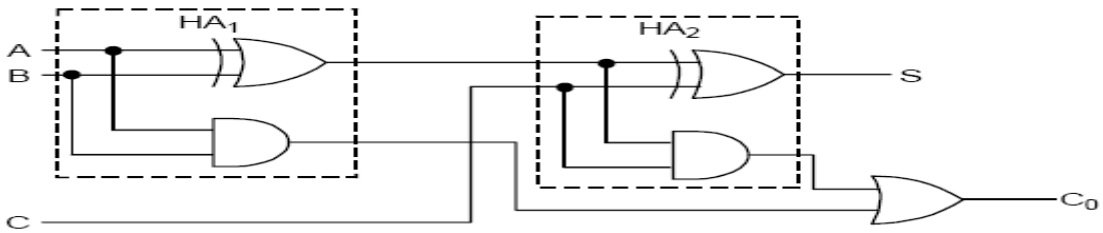
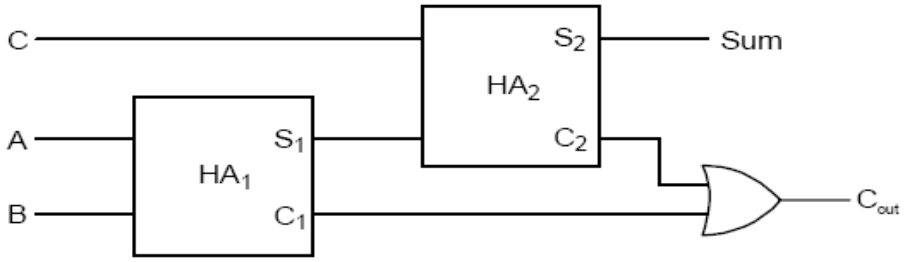


Fig: Implementation of a full- adder.

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



- S1 and C1 are outputs of first half adder (HA1)
- S2 and C2 are outputs of second half adder (HA2)
- A, B and C are inputs of Full adder.
- Sum and Cout 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 bits.

1. Fig. below shows a half subtractor. (HS) It has two inputs, A (minuend) and B (subtrahend) and two outputs D (difference) and B0 (Borrow). [The symbol for borrow (B0) is taken to avoid confusion with input variable B] produced by subtractor of two bits.

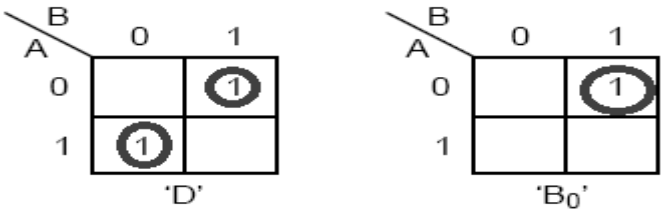
Fig: Half subtractor.

2. **Truth Table**

The difference output is 0 if A = B and 1 if A ≠ 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

3. Using a two variable map, for outputs D and B_0.



D = AB' + A'B
= A ⊕ B

B0 = A'B

4. Logical Implementation shown in Fig.

(a) Using Basic gates (b) using XOR gate

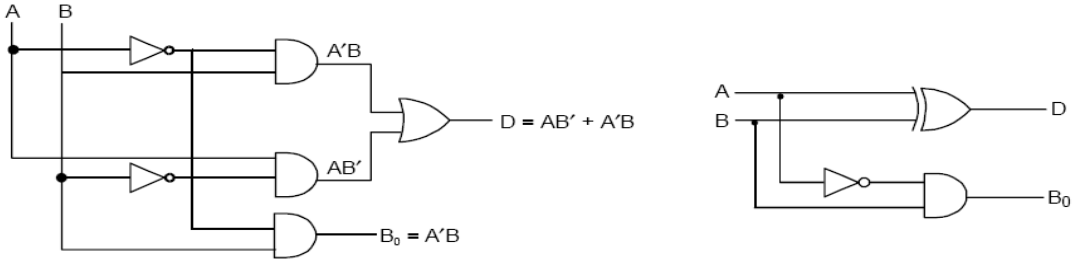


Fig: Basic gate implementation & X-OR gate implementation of half subtractor.

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

1. Fig. 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.

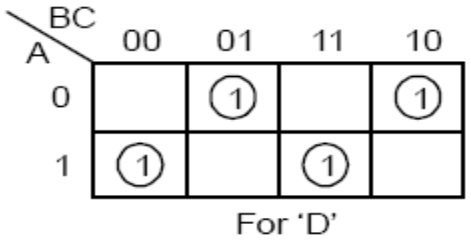
Fig: Full subtractor.

2. **Truth Table:**

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

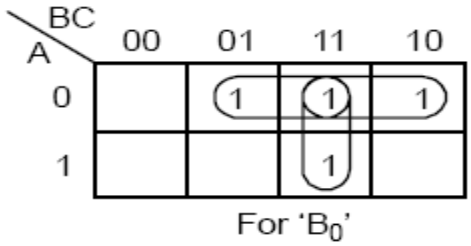
Inputs			*Output*	
A	*B*	*C*	*D*	B_0
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

3. Using a three variable map for both outputs.



For 'D'

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



For 'B_0'

B0 = A'B + A'C + BC

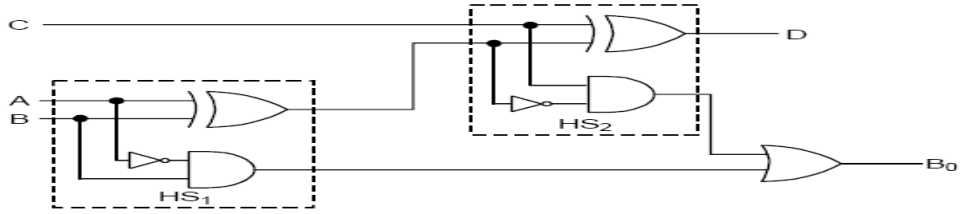
4. Logical implementation:

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

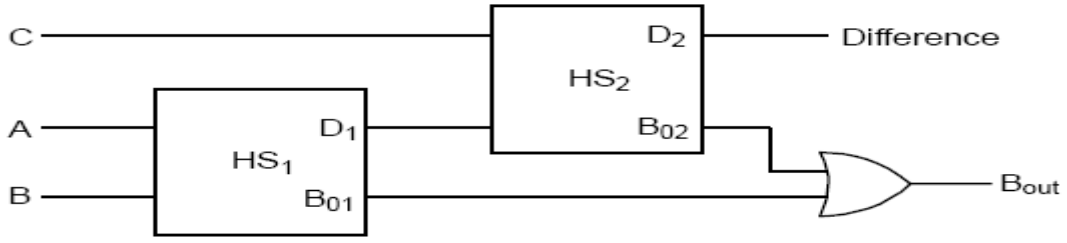
The difference D = ABC + AB'C' + A'BC' + A'B'C
= ABC + A'B'C + AB'C' + A'BC'
= C (AB + A'B') + C' (AB' + A'B)
= C (AB' + A'B)' + C' (AB' +A'B)
= C (A ⊕B)' + C' (A ⊕ B)
= (A ⊕ B) ⊕ C

and the borrow B0 = 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 ⊕ B)'
= A'B + C (A ⊕ B)'

D = (A ⊕ B) ⊕ C and B_0 = A'B + C (A ⊕ B)'



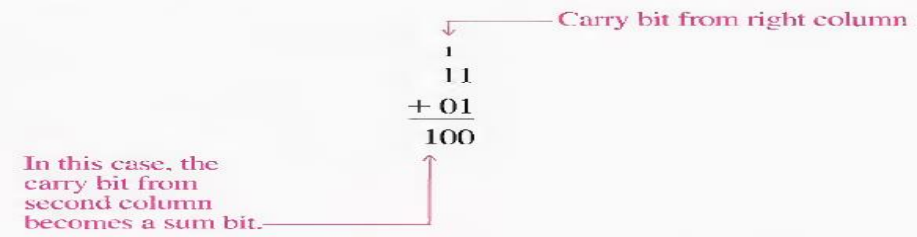
Block Diagram Representation of a full subtractor using two half subtractors :



- 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 Bout are outputs of full subtractor.

PARALLEL BINARY ADDERS

- Two or more full-adders are connected to form parallel binary adders. Here we will learn the basic operation of this type of adder and its associated input and output functions.
- As we know that, a single full-adder is capable of adding two 1-bit numbers and an input carry. To add binary numbers with more than one bit, we must use additional full-adders.
- When one binary number is added to another, each column generates a sum bit and a 1 or 0 carry bit to the next column to the left, as illustrated here with 2-bit numbers.



- To add two binary numbers, a full-adder is required for each bit in the numbers. So for 2-bit numbers, two adders are needed; for 4-bit numbers, four adders are used; and so on.
- In general n-bit numbers, n full adders are used. The carry output of each adder is connected to the carry input of the next higher-order adder, as shown in Figure below for a 2-bit adder.

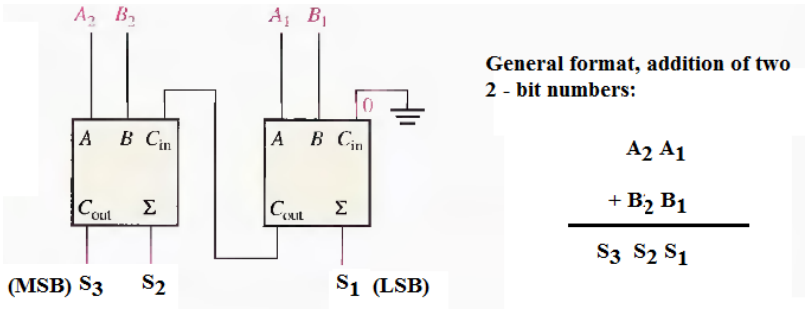


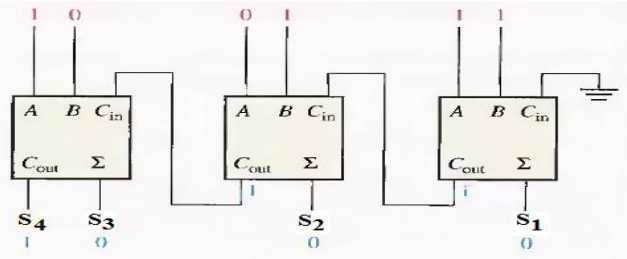
Fig: Block diagram of a basic 2-bit parallel adder using two full-adders.

- Note that either a half-adder can be used for the least significant position or the carry input of a full-adder can be made 0 (grounded) because there is no carry input to the least significant bit position.

- In Figure above the least significant bits (LSB) of the two numbers are represented by A_1 and B_1. The next higher-order bits are represented by A_2 and B_2. The three sum bits are S_1, S_2 and S_3. Note that the output carry from the left-most full-adder becomes the most significant bit (MSB) in the sum, S_3.

Example:

Determine the sum generated by the 3-bit parallel adder in Figure below and show the intermediate carries when the binary numbers 101 and 011 are being added.



Solution: The LSBs of the two numbers are added in the Right-most full-adder. The Sum bits and the intermediate carries are indicated as shown in Figure above.

Four-Bit Parallel Adders

A group of four bits is called a nibble. A basic 4-bit parallel adder is implemented with four full-adder stages as shown in Figure.

Again, the LSBs (A1 and B1) in each number being added go into the right-most full-adder: the higher-order bits are applied as shown to the successively higher-order adder, with the MSBs (A4 and B4) in each number being applied to the left-most full-adder. The Carry output of each adder is connected to the carry input of the next higher-order adder as indicated. These are called internal carries.

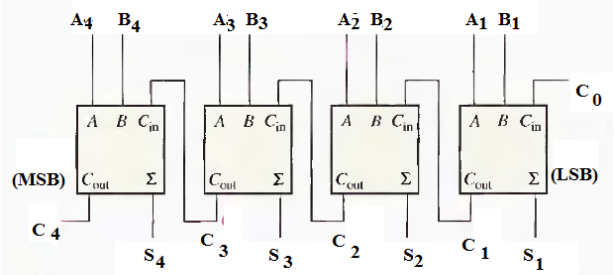


Fig: Block diagram of 4 – Bit Parallel Adder.

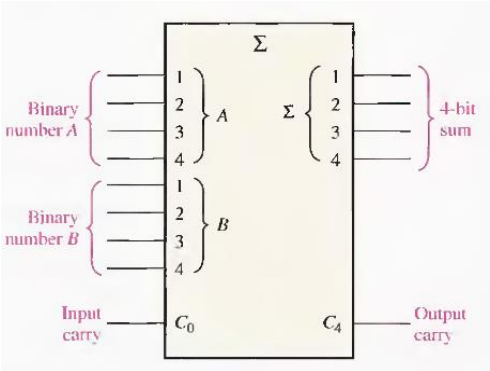


Fig: logic symbol of 4 – Bit Parallel Adder.

4 bit 2's complement Subtractor:

- The subtraction A – B can be done by taking the 2's complement of B and adding it to A because A- B = A + (-B)
- It means if we use the inverters to make 1's complement of B (connecting each Bit to an inverter) and then add 1 to the least significant bit (by setting carry C0 to 1) of binary adder, then we can make a binary subtractor.

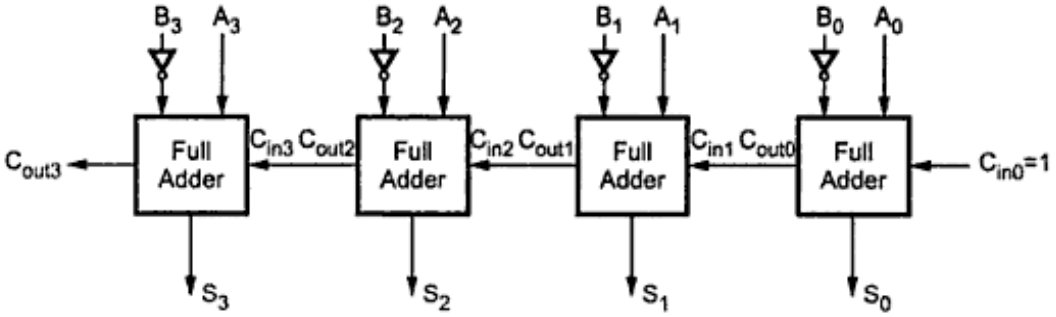


Fig. 4-bit parallel subtractor

4 - Bit Adder / Subtractor:

- The addition and subtraction can be combined into one circuit with one common binary adder.
- The mode M controls the operation. When M=0 the circuit is an adder when M=1 the circuit is subtractor. It can be done by using exclusive-OR for each Bi and M.

Note: $1 \oplus x = x'$ and $0 \oplus x = x$

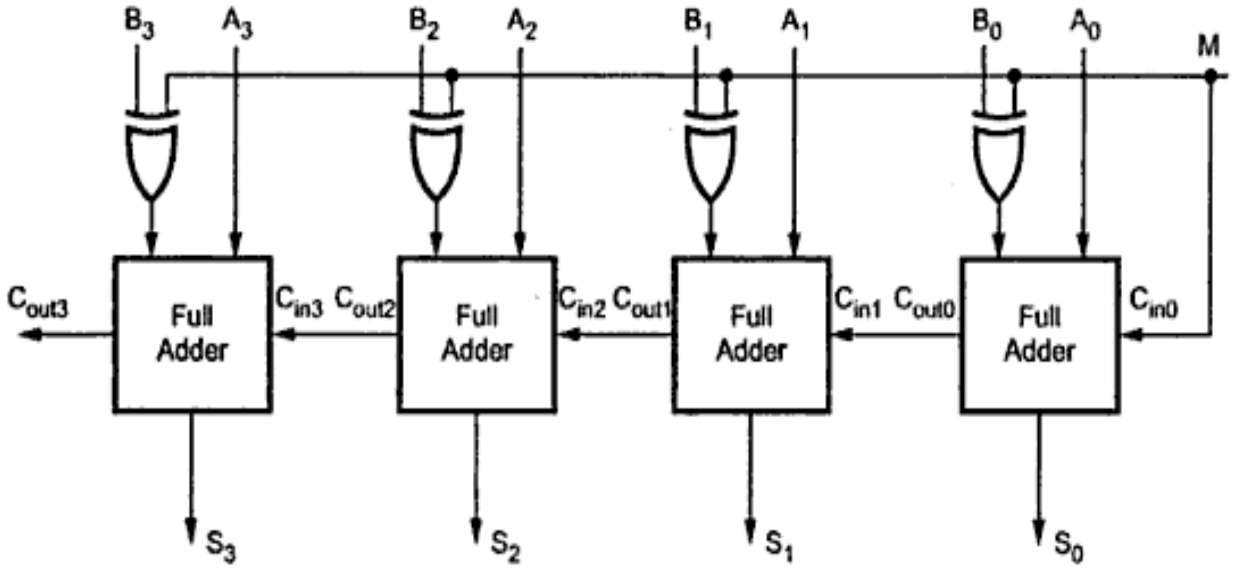


Fig. 4-bit adder-subtractor

RIPPLE CARRY VERSUS LOOK-AHEAD CARRY ADDERS

Parallel adders can be classified into two categories based on the way in which internal carries from stage to stage are handled.

The categories are:

1. Ripple carry Adder and
2. Look-ahead carry.

Externally, both types of adders are the same in terms of inputs and outputs. The difference is the speed at which they can add numbers. The look-ahead carry adder is much faster than the ripple carry adder.

The Ripple Carry Adder:

- A ripple carry adder is one in which the carry output of each full-adder is connected to the carry input of the next higher-order stage (a stage is one full-adder).
- The sum and the output carry of any stage cannot be produced until the input carry occurs; this causes a time delay in the addition process, as illustrated in Figure below.
- The carry propagation delay for each full-adder is the time from the application of the input carry until the output carry occurs, assuming that the A and B inputs are already present.

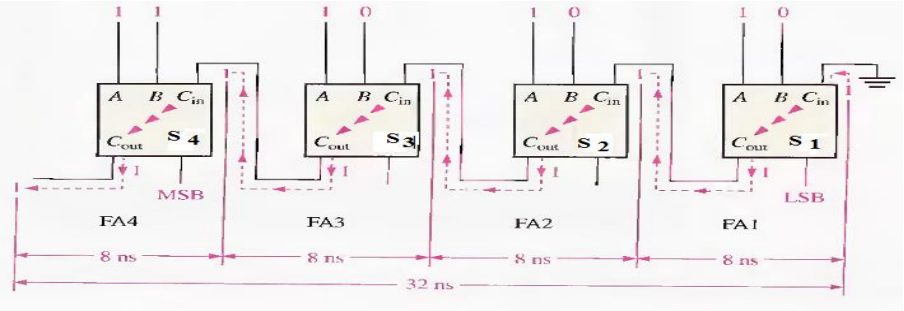


Fig: A 4-bit parallel ripple carry adder showing "worst-case" carry propagation delays.

- Full-adder 1 (FA 1) cannot produce an output carry until an input carry is applied.
- Full-adder 2 (FA2) cannot produce an output carry until full-adder 1 produces an output carry.
- Full-adder 3 (FA3) cannot produce an output carry until an output carry is produced by FA1 followed by an output carry from FA2, and so on.
- As we can see in Figure, the input carry to the least significant stage has to ripple through all the adders before a final sum is produced. The cumulative delay through all the adder stages is a "worst-case" addition time. The total delay can vary, depending on the carry bit produced by each full-adder.
- If two numbers are added such that no carries (0) occur between stages, the addition time is simply the propagation time through a single full-adder from the application of the data bits on the inputs to the occurrence of a sum output.

The look-Ahead Carry (more complex mechanism, yet faster):

- The speed with which an addition can be performed is limited by the time required for the carries to propagate, or ripple, through all the stages of a parallel adder.
- A clear solution for reducing the carry propagation delay time is to employ faster gates with reduced delays. However, physical circuits have a limit to their capability.

- Another solution is to increase the equipment complexity in such a way that the carry delay time is reduced.
- There are several techniques for reducing the carry propagation time in a parallel adder. The most widely used technique of speeding up the addition process is by eliminating this ripple carry delay is called look- ahead carry addition. The look-ahead carry adder anticipates the output carry of each stage, and based on the inputs, produces the output carry by either carry generation or carry propagation.

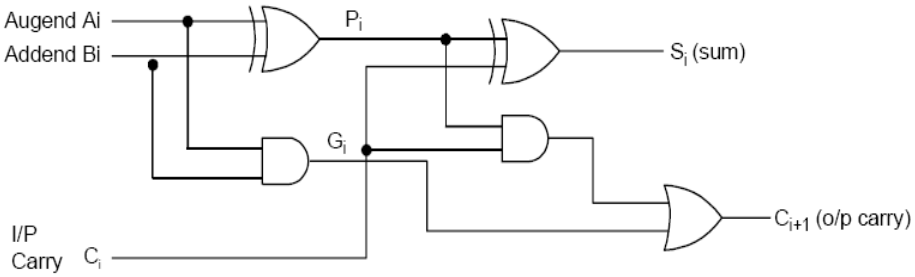


Fig. Full adder

Fig: Full Adder with P_i (Carry propagate) and G_i (Carry Generate).

Consider the full adder shown in figure above. Here we define two new binary variables as

- $P_i = A_i \oplus B_i$
- $G_i = A_i B_i$

The output sum and carry can be expressed in terms of P_i and G_i as

- $S_i = P_i \oplus C_i$
- $C_{i+1} = G_i + P_i C_i$

Where G_i is called carry generate and it produces a carry of 1 when both A_i and B_i are 1, regardless of the input carry C_i .

P_i is called carry propagate because it is the term associated with the propagation of the carry from C_i to C_{i+1}.

The Boolean function for the carry outputs of each stage is as follows and substitute for each C_i its value from the previous equations.

$$C_0 = \text{input carry}$$

$$C_1 = G_0 + P_0 C_0$$

$$C_2 = G_1 + P_1 C_1$$

$$= G_1 + P_1 (G_0 + P_0 C_0)$$

$$= G_1 + P_1 G_0 + P_1 P_0 C_0$$

$$C_3 = G_2 + P_2 C_2$$

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

$$= G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0$$

Since the Boolean function for each output carry is expressed in sum of products, each function can be implemented with one level of AND gates followed by an OR gate (or by two level NAND). The 3 Boolean functions are implemented in the carry look ahead generator as shown in figure below.

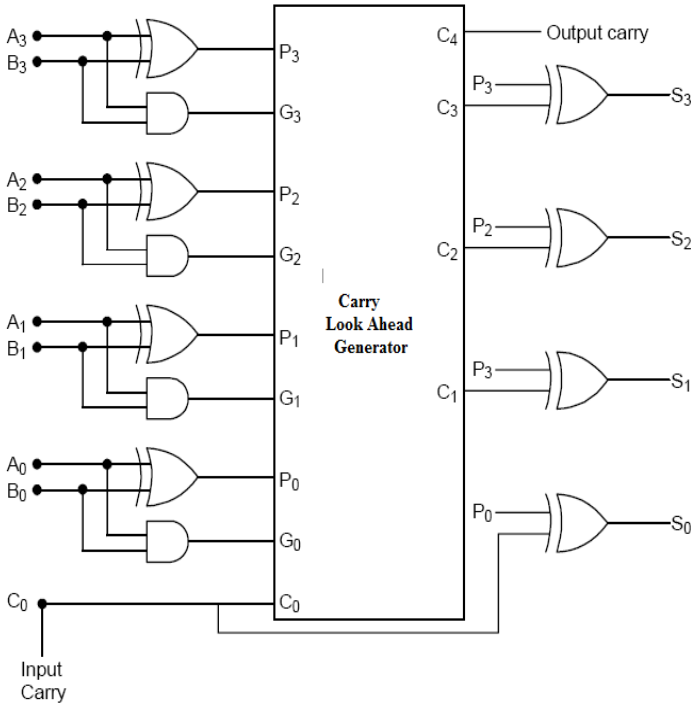
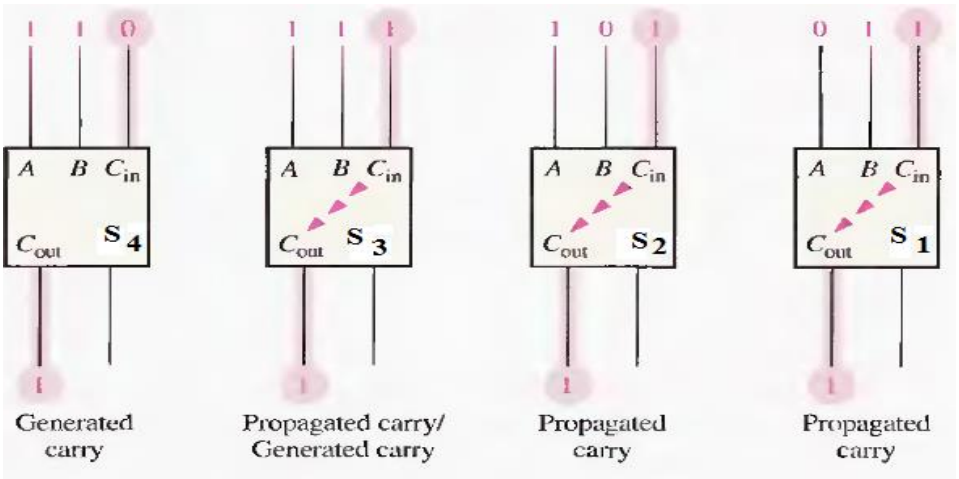


Fig: 4 – Bit Adder with Carry Look Ahead.

- We can observe 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_2 and C_1.
- Each sum output requires two X –OR gates. The output of the first X-OR gate generates Pi Variable, and the AND gate generates G_i variable.
- The carries are propagated through the carry look ahead generator and applied as inputs to the second X-OR gate.
- All output carries are generated after a delay through the two levels of gates. Thus, outputs S_1 through S_3 have equal propagation delay times. The two-level circuits for the output carry C_4 is not shown. This circuit can be easily derived by the equation substitution method.



Serial Adder:

Shift register with added circuitry can be used to design a serial adder.

It consists of :

- Two right shift registers,
- Full adder and
- D- flip – flop.

Serial Addition Process:

- A full adder is used to perform bit by bit addition and D – flip – flop is used to store the carry output generated after addition. This carry is used as carry input for the next addition.
- Initially, the D- flip – flop is cleared and addition starts with LSBs of both register.
- After each clock pulse data within the right shift register are shifted right, 1 – bit and we get the bits from next digit and carry of previous addition as new inputs for the full adder.
- The sum bit of the full adder is connected as serial input o shift register A. Thus the result of the serial addition gets stored in register A.
- The new number can be added to the contents of register A by loading a new number into register B and repeating the process of serial addition.

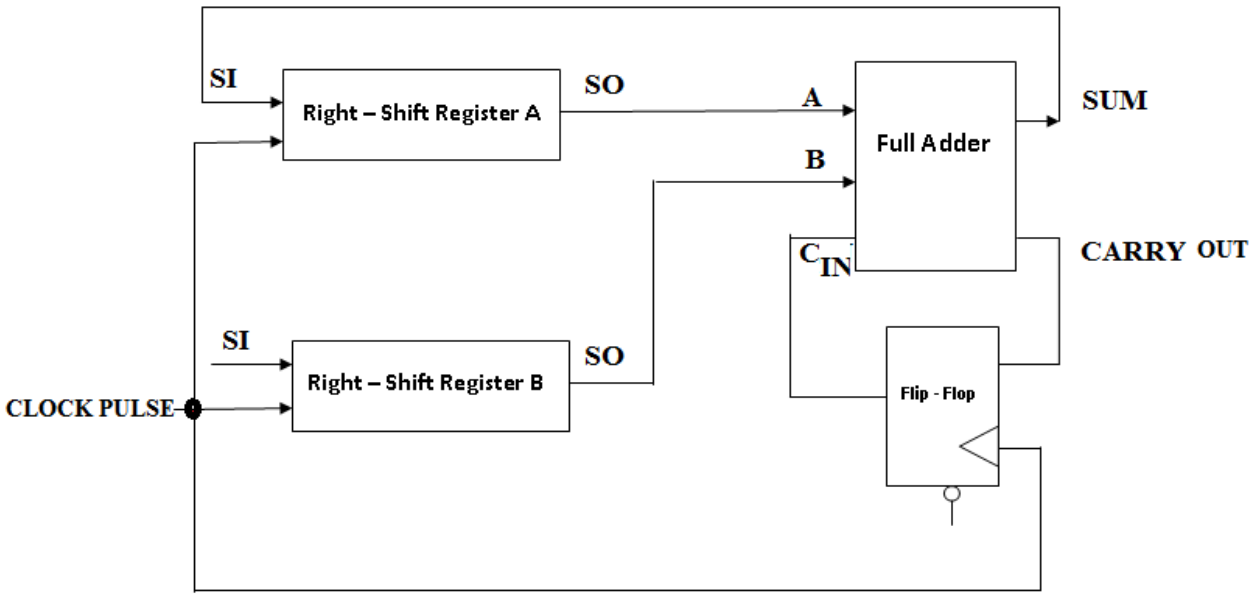


Fig: Serial Adder

Comparison between Serial and Parallel Adder:

S.No:	Serial Adder	Parallel Adder
1	Serial adder uses shift registers	Parallel adder uses registers with parallel loads
2	The serial adder requires only one full – adder circuit.	The number of full – adder circuits in the parallel adder equals to the number of bits in the binary numbers.
3	The serial adder is a sequential circuit.	Excluding the register, the parallel adder is purely combinational circuit.
4	Time required for addition depends on number of bits.	Time required for addition does not depend on number of bits.
5	It is slower.	It is faster.

Overflow:

- When 2 numbers of n digits each are added and the sum occupies n+ 1 digit, we say that an overflow occurred. This is true for binary or decimal numbers whether signed or unsigned.
- When the addition is performed with paper and pencil, an overflow is not a problem, since there is no limit by the width of the page to write down the sum.
- Overflow is a problem in digital computers because the number of bits that hold the number is finite and the result contains n+ 1 bit cannot be accommodated.
- For this reason, many computers detect the occurrence of an overflow, and when it occurs, a corresponding flip-flop is set that can then be checked by the user.
- The detection of an overflow after the addition of 2 binary numbers depends on whether the numbers are considered to be signed or unsigned.
- When 2 unsigned numbers are added, an overflow is detected from the end carry out of the most significant position.
- In the case of signed numbers, the left most bit always represent the sign and negative numbers are in 2's complement form. When 2 signed numbers are added, the sign bit is treated as part of the number and the end carry does not indicate an overflow.
- An overflow cannot occur after an addition if one number is positive and other number is negative, since adding a positive number to a negative number produces a result which is smaller than the larger of the two numbers.
- An overflow may occur if the 2 numbers added are both positive and both negative.

For example,

Carries:	0 1	
+70	0 1000110	
+ 80	0 1010000	
+ 150	1 0010110	
Carries:	1 0	
-70	1 0111010	
- 80	1 0110000	
- 150	0 1101010	

- Note that the 8 – bit result that should have been positive has a negative sign bit and the 8 – bit result that should have been negative has a positive sign bit.
- If however, the carry out of sign bit position is taken as the sign bit of the result, then the 9-bit answer so obtained will be correct. Since the answer cannot be accommodated within 8-bits, we say that an overflow has occurred.
- An overflow condition can be detected by observing the carry into the sign bit and carry out of sign bit position. If these two carries are not equal, an overflow has occurred. This is indicated in the example where the two carries are not equal. If the 2 carries are applied to an X-OR gate an overflow is detected when the output is equal to 1.

The binary adder-subtractor circuit with outputs C and V is shown in figure below.

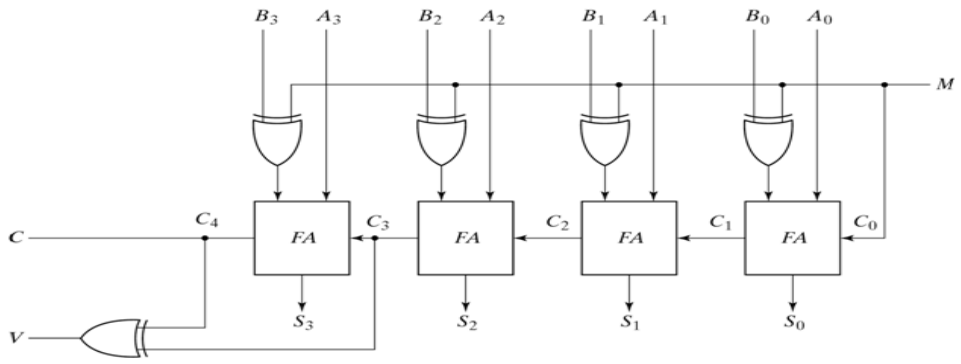


Fig. 4-Bit Adder Subtractor

Fig: Binary adder-subtractor circuit with outputs C and V.

DECIMAL (BCD) ADDER

BCD is a numerical code and can be used in arithmetic operations. Addition is the most important operation because the other three operations (subtraction, multiplication, and division) can be accomplished by the use of addition.

Steps for addition of two BCD numbers:

Add the two BCD numbers, using the rules for binary addition.

- If a 4-bit sum is equal to or less than 9, it is a valid BCD number.
- If a 4-bit sum is greater than 9, or if a carry out of the 4-bit group is generated, it is an invalid result. Add 6 (0110) to the 4-bit sum in order to skip the six invalid states and return the code to 8421.
- If a carry results when 6 is added, simply add the carry to the next 4-bit group.

Logic circuit to detect Sum greater than 9

Inputs				Outputs
S3	S2	S1	S0	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

$S_3 S_2$ \ $S_1 S_0$	00	01	11	10
00	**0**	**0**	**0**	**0**
01	0	0	0	0
11	1	1	1	1
10	0	0	1	1

$$\mathbf{Y = S_3 S_2 + S_3 S_1}$$

Y = 1 indicates sum is greater than 9. We can put one more term, C_{out} in the above expression to check whether carry is 1. If any one condition is satisfied we add 6 (0110) in the sum.

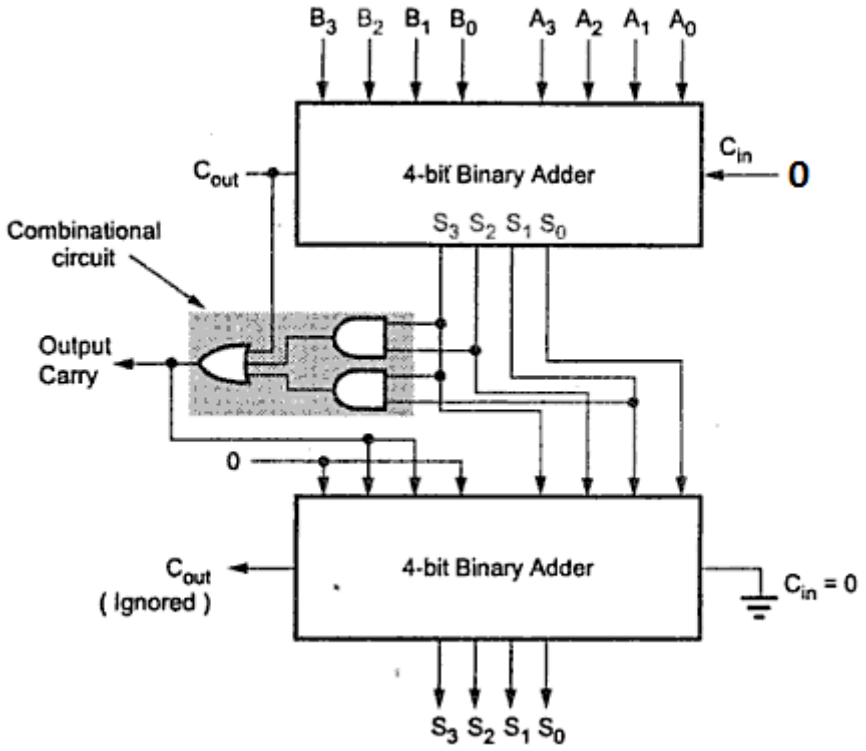


Fig: Block diagram of BCD Adder.

Block diagram of 8 – Bit BCD Adder:

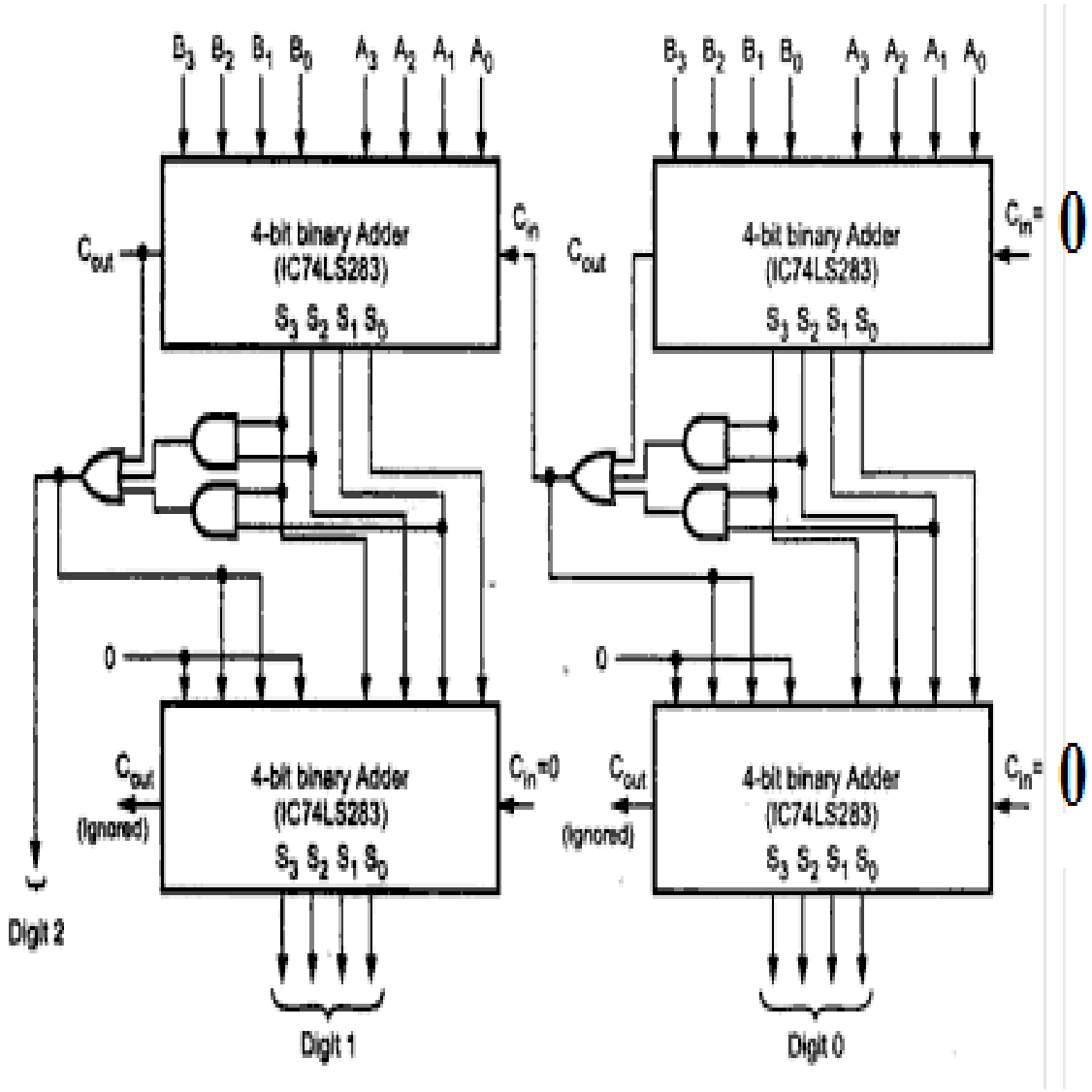
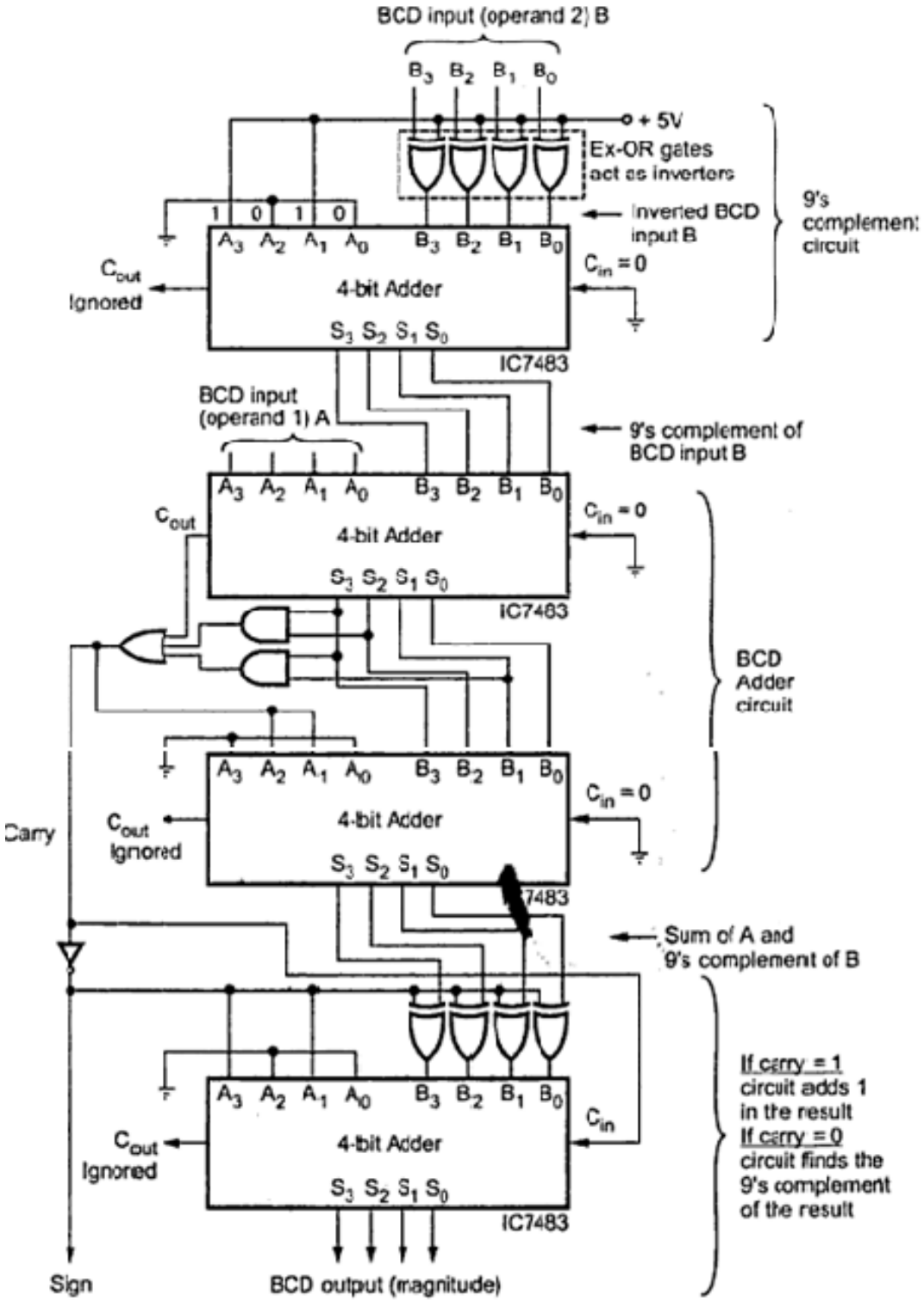


Fig: 8 - Bit BCD Adder.

DECIMAL SUBTRACTOR:

Steps to perform 9's complement BCD subtraction:

- Find the 9's complement of a negative number
- Add two numbers using BCD addition
- If carry is generated add the carry to the result otherwise find the 9's complement of the result.



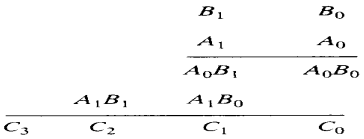
Binary Multiplier: The multiplication process for the binary numbers is similar to the decimal numbers. The binary multiplication is simple than decimal multiplication since it involves only 1's and 0's.

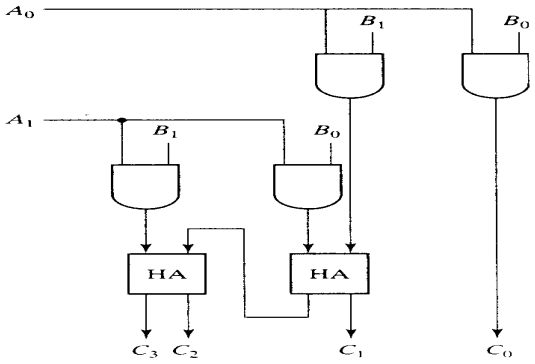
Rules for binary multiplication:

- $0 \times 0 = 0$
- $0 \times 1 = 0$
- $1 \times 0 = 0$
- $1 \times 1 = 1$

Generalized multiplication process for 2 × 2 for two unsigned 2 - bit numbers:

Partial products – AND operations





4-bit by 3-bit binary multiplier:

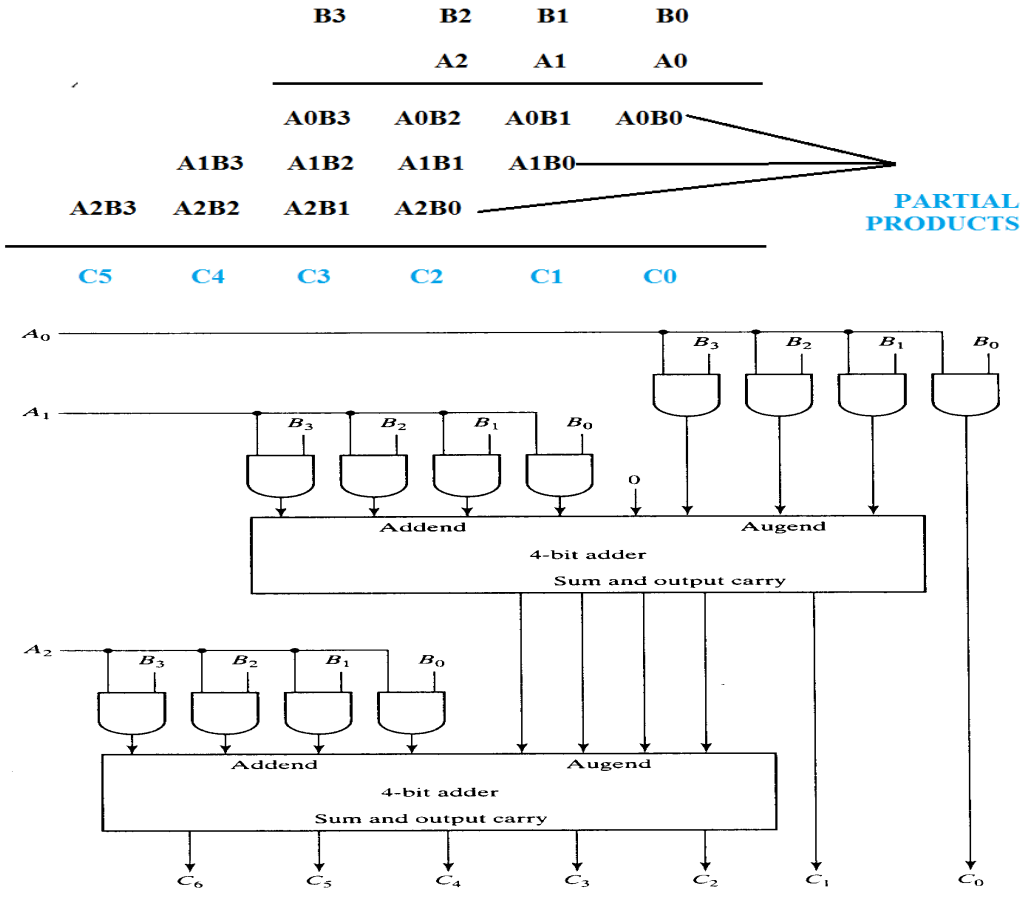


Fig: 4-bit by 3-bit binary multiplier.

MODULAR DESIGN USING IC CHIPS

MAGNITUDE COMPARATOR

- Magnitude comparator is a combinational circuit that compares two numbers, A and B, and determines their relative magnitudes. The outcome of the comparison is specified by three binary variables that indicates whether A>B, A=B and A<B.
- The circuit for comparing two n-bit numbers has 2^{2n} entries in the truth table and becomes too cumbersome even with n= 3.
- The comparison of two numbers
  - outputs: A>B, A=B, A<B
- Design Approaches
  - The truth table
    - 2^{2n} entries - too cumbersome for large n
  - use inherent regularity of the problem
    - reduce design efforts
    - reduce human errors

2 – Bit Comparator

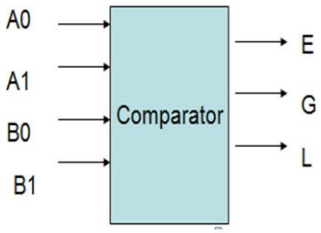


Fig: 2 – Bit Comparator

Let the 2 numbers be

- A = A1 A0
- B = B1 B0

Truth table for 2- Bit Comparator:

INPUTS				OUTPUTS		
A1	A0	B1	B0	A > B	A = B	A < B
0	0	0	0	0	1	0
0	0	0	1	0	0	1
0	0	1	0	0	0	1
0	0	1	1	0	0	1
0	1	0	0	1	0	0
0	1	0	1	0	1	0
0	1	1	0	0	0	1
0	1	1	1	0	0	1
1	0	0	0	1	0	0
1	0	0	1	1	0	0
1	0	1	0	0	1	0
1	0	1	1	0	0	1
1	1	0	0	1	0	0
1	1	0	1	1	0	0
1	1	1	0	1	0	0
1	1	1	1	0	1	0

The circuit for comparing two 2 – bit numbers requires 16 entries in the truth table and we can construct. But, the circuit for comparing two n-bit numbers has 2^{2n} entries in the truth table and becomes too cumbersome even with n= 3.

So we follow finite set of steps to overcome the above problem.

The algorithm is a direct process, which is used to compare the relative magnitude of 2 numbers. Consider 2 numbers A and B, with 4 bits each. Write down the coefficients with descending significance

$A = A_3\ A_2\ A_1\ A_0$

$B = B_3\ B_2\ B_1\ B_0$

- The 2 numbers are equal if all the pairs of significant digits are equal: $A_3 = B_3$ and $A_2 = B_2$ and $A_1 = B_1$ and $A_0 = B_0$.
- When the numbers are binary, the digits are either 1 or 0, and the equality relation of each pair of bits can be expressed logically with an X-NOR gate as

  $X_i = A_i B_i + A_i^1 B_i^1$ for i = 0,1,2,3

  Where X_i =1 only if the pairs of bits in positions I are equal (i.e., if both are 1 or both are 0).
- The equality of 2 numbers, A and B, is displayed in a combinational circuit by an output binary variable that we designate by the symbol (A=B). This binary variable is equal to 1 if the input numbers, A and B are equal, and it is equal to 0 otherwise. For the condition to exist, all X_i variables must be equal to 1.

  $(A=B) = X_3\ X_2\ X_1\ X_0$
- The binary variable (A=B) is equal to 1 only if all pairs of digits of the two numbers are equal.
- The procedure for binary numbers with more than 2 bits can also be found in the similar way. The expressions for the output of 4-bit magnitude comparator is as follows, in which

  $(A= B) = X_3X_2X_1X_0$

  $(A> B) = A_3B_3^1 + X_3\ A_2B_2^1 + X_3X_2\ A_1B_1^1 + X_3X_2X_1\ A_0B_0^1$

$(A< B) = A_3{}^1B_3 + X_3\,A_2{}^1B_2 + X_3X_2\,A_1{}^1B_1 + X_3X_2X_1\,A_0{}^1B_0$

- The symbols (A>B) and (A<B) are binary output variables that are equal to 1 when A>B or A<B, respectively.

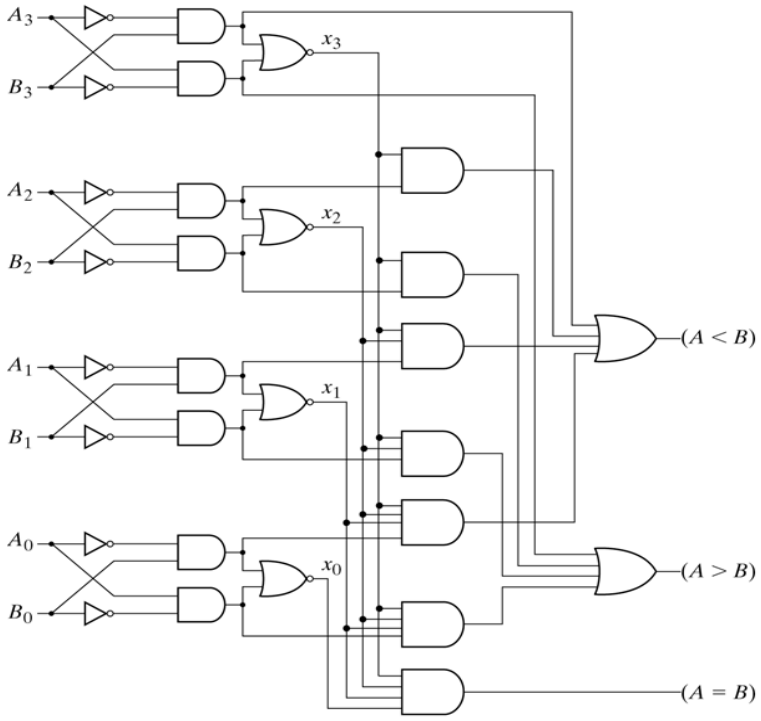


Fig: 4 - bit Magnitute comparator.

Decoders:

A Decoder is a multiple-input, multiple-output logic circuit which converts coded inputs into coded outputs, where the input and output codes are different. The input code generally has fewer bits than the output code. Each input code word produces a different output code word, i.e., there is one-to-one mapping from input code words to output code words.

- A decoder accepts a binary value as input and decodes it.
- It has n inputs and 2^n outputs, numbered from 0 to 2^n -1.
- Each output represents one minterm of the inputs
- The output corresponding to the value of the n inputs is activated

- An n-to-2^n decoder takes an n-bit input and produces 2^n outputs. The n inputs represent a binary number that determines which of the 2^n outputs are uniquely true.

- For n=2, 2-to-2^2 i.e., we get 2-to-4 decoder.

- A 2-to-4 decoder operates according to the following truth table.
  - Let the 2-bit input is called S1S0, and the four outputs are Q0-Q3.
  - If the input is the binary number i, and then output Qi is uniquely true.

- For example, if the input S1 S0 = 10 (decimal 2), then output Q2 is true, and Q0, Q1, Q3 are all false.

- This circuit “decodes” a binary number into a “one-of-four” code.

- By following the design procedures, we have a truth table, so we can write equations for each of the four outputs (Q0-Q3), based on the two inputs (S0-S1).

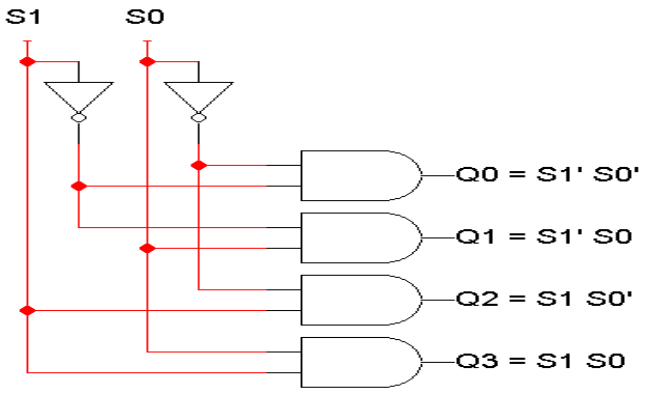
S1	S0	Q0	Q1	Q2	Q3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

- In this case there's not much to be simplified. Here are the equations:

  - Q0 = S1' S0'
  - Q1 = S1' S0
  - Q2 = S1 S0'
  - Q3 = S1 S0

A truth table and logic diagram of a 2-to-4 decoder is as follows:

S1	S0	Q0	Q1	Q2	Q3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1



2-to-4 decoder with Enable inputs:

- Many devices have an additional enable input, which is used to "activate" or "deactivate" the device.
- For a decoder,
  - EN=1 activates the decoder, so it behaves as specified earlier. Exactly one of the outputs will be 1.
  - EN=0 "deactivates" the decoder. By convention, that means all of the decoder's outputs are 0.
- We can include this additional input in the decoder's truth table as shown below:

EN	S1	S0	Q0	Q1	Q2	Q3
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

- In the above table, note that whenever EN=0, the outputs are always 0, regardless of inputs S1 and S0.
- So, we can abbreviate the table by writing x's in the input columns for S1 and S0 as below.

EN	S1	S0	Q0	Q1	Q2	Q3	
0	x	x	0	0	0	0	Q0 = S1' S0'
1	0	0	1	0	0	0	Q1 = S1' S0
1	0	1	0	1	0	0	Q2 = S1 S0'
1	1	0	0	0	1	0	
1	1	1	0	0	0	1	Q3 = S1 S0

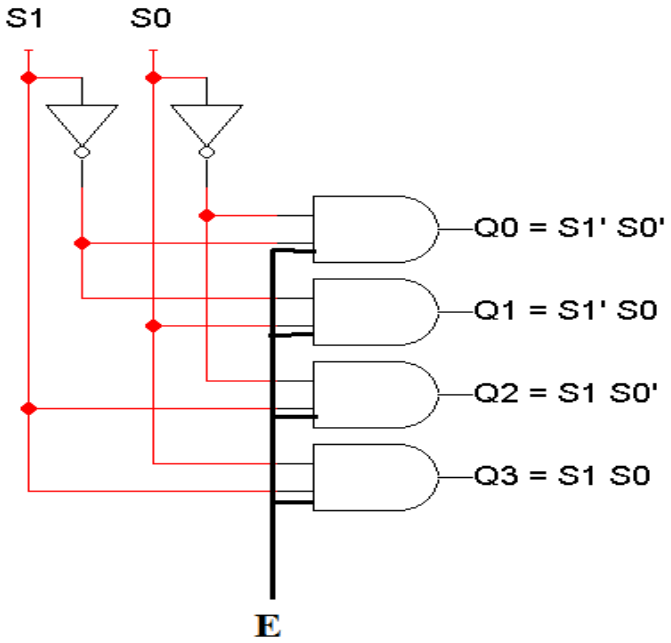
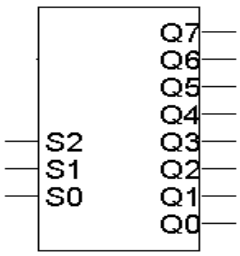


Fig: logic diagram of a 2-to-4 decoder with Enable input.

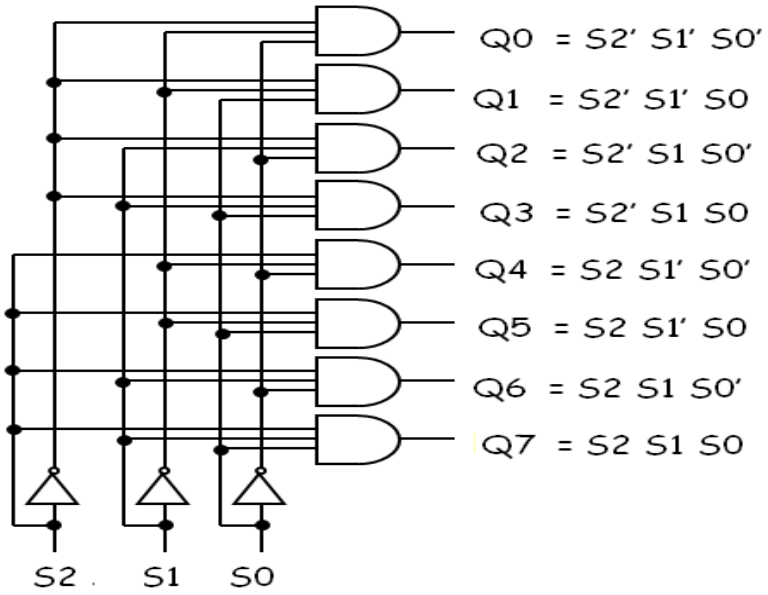
3-to-8 decoder:

- Larger decoders are similar. Here is a 3-to-8 decoder.
  - The block symbol is shown below.



The truth table (without EN) and output equations are as follows.

S2	S1	S0	Q0	Q1	Q2	Q3	Q4	Q5	Q6	Q7	
0	0	0	1	0	0	0	0	0	0	0	Q0 = S2′ S1′ S0′
0	0	1	0	1	0	0	0	0	0	0	Q1 = S2′ S1′ S0
0	1	0	0	0	1	0	0	0	0	0	Q2 = S2′ S1 S0′
0	1	1	0	0	0	1	0	0	0	0	Q3 = S2′ S1 S0
1	0	0	0	0	0	0	1	0	0	0	Q4 = S2 S1′ S0′
1	0	1	0	0	0	0	0	1	0	0	Q5 = S2 S1′ S0
1	1	0	0	0	0	0	0	0	1	0	Q6 = S2 S1 S0′
1	1	1	0	0	0	0	0	0	0	1	Q7 = S2 S1 S0



Note: Only one output is true for any input combination.

Advantages:

- The truth table and equations look familiar and are easy to form the equations.

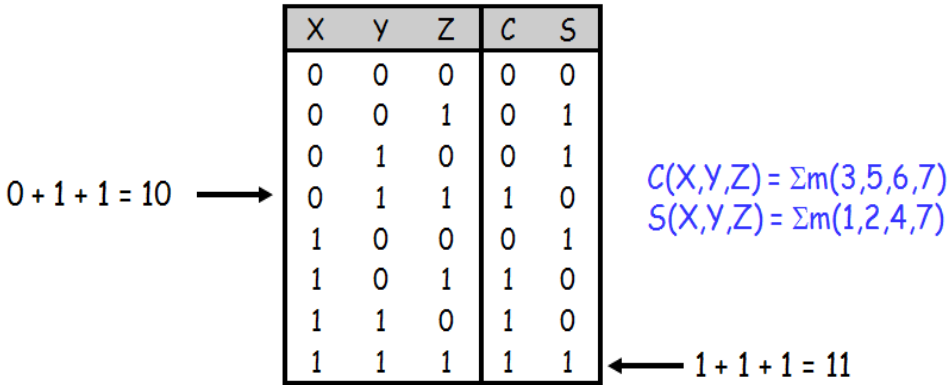
S1	S0	Q0	Q1	Q2	Q3	
0	0	1	0	0	0	Q0 = S1′ S0′
0	1	0	1	0	0	Q1 = S1′ S0
1	0	0	0	1	0	Q2 = S1 S0′
1	1	0	0	0	1	Q3 = S1 S0

- Decoders are sometimes called minterm generators.
  - For each of the input combinations, exactly one output is true.
  - Each output equation contains all of the input variables.
  - These properties hold for all sizes of decoders.
- This means that we can implement any arbitrary functions with decoders. If we have a sum of minterms equation for a function, you can easily use a decoder (a minterm generator) to implement that function.

Example: 1

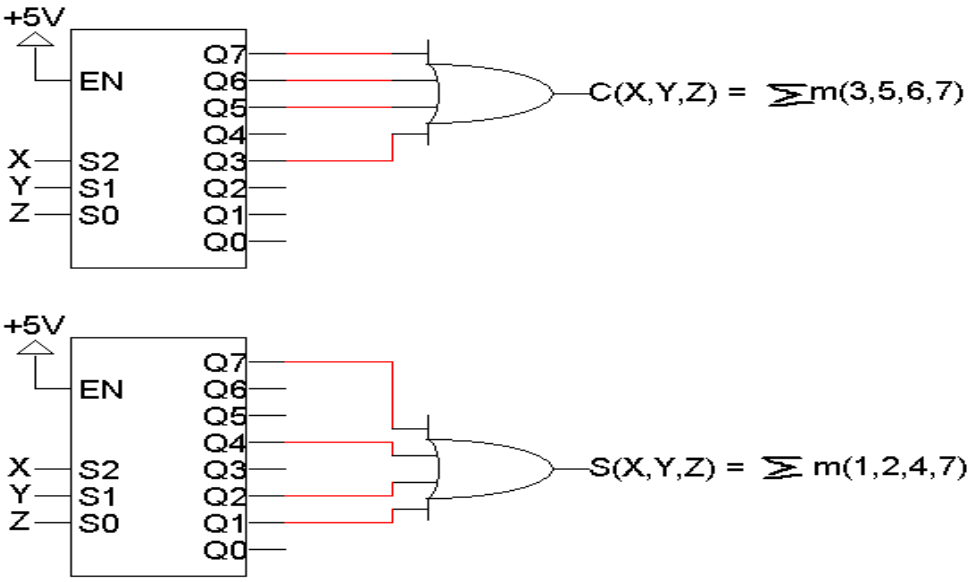
Designing a Full- Adder using Decoder.

- Let's make a circuit that adds three 1-bit inputs X, Y and Z.
- We need two bits to represent the total; let's call them C and S, for "carry" and "sum." Note that C and S are two separate functions of the same inputs X, Y and Z.
- Here are a truth table and sum-of-minterms equations for C and S.



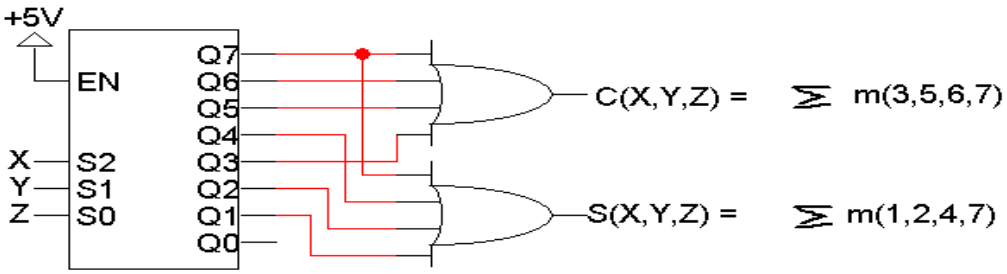
Decoder-based adder:

- Here, two 3-to-8 decoders implement C and S as sums of minterms.
- The "+5V" symbol ("5 volts") is how we represent a constant 1 or true in Logic Works. We use it here so the decoders are always active.



Using just one decoder:

- Since the two functions C and S both have the same inputs, we could use just one decoder instead of two.



$C(X,Y,Z) = \Sigma m(3,5,6,7)$
$S(X,Y,Z) = \Sigma m(1,2,4,7)$

Construction of a 3-to-8 decoder using two 2 – to – 4 decoder:

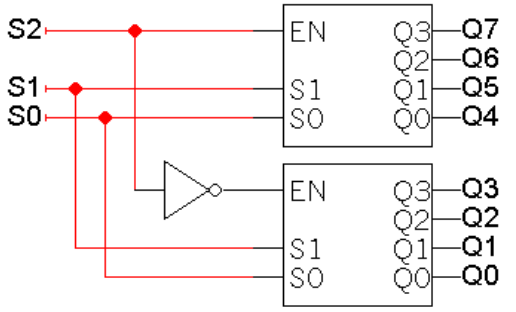
- We can construct a 3-to-8 decoder directly from the truth table and equations below, just like how we built the 2-to-4 decoder.
- Another way to design a decoder is to break it into smaller pieces.
- Notice some patterns in the table below:
  - When S2 = 0, outputs Q0-Q3 are generated as in a 2-to-4 decoder.
  - When S2 = 1, outputs Q4-Q7 are generated as in a 2-to-4 decoder.

S2	S1	S0	Q0	Q1	Q2	Q3	Q4	Q5	Q6	Q7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

$Q0 = S2'\ S1'\ S0' = m_0$
$Q1 = S2'\ S1'\ S0 = m_1$
$Q2 = S2'\ S1\ S0' = m_2$
$Q3 = S2'\ S1\ S0 = m_3$
$Q4 = S2\ S1'\ S0' = m_4$
$Q5 = S2\ S1'\ S0 = m_5$
$Q6 = S2\ S1\ S0' = m_6$
$Q7 = S2\ S1\ S0 = m_7$

Decoder expansion:

- We can use enable inputs to string decoders together. Here's a 3-to-8 decoder constructed from two 2-to-4 decoders:



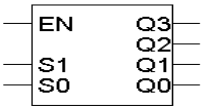
S2	S1	S0	Q0	Q1	Q2	Q3	Q4	Q5	Q6	Q7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Fig : Block diagram and truth table of 3- to – 8 Decoder.

- Be careful not to confuse with the "inner" inputs and outputs of the 2-to-4 decoders with the "outer" inputs and outputs of the 3-to-8 decoder (which are in boldface).
- We can verify that this circuit is a 3-to-8 decoder, by using equations for the 2-to-4 decoders to derive equations for the 3-to-8.

A variation of the standard decoder:

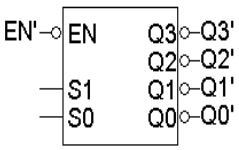
- The decoders we've seen so far are **active-high** decoders.



EN	S1	S0	Q0	Q1	Q2	Q3
0	x	x	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

$Q3 = S1\ S0$
$Q2 = S1\ S0'$
$Q1 = S1'\ S0$
$Q0 = S1'\ S0'$

- **Active-high decoders** generate **minterms** , as we've already seen.
- An **active-low** decoder is similar to active- High decoder, but with an inverted EN input and inverted outputs.



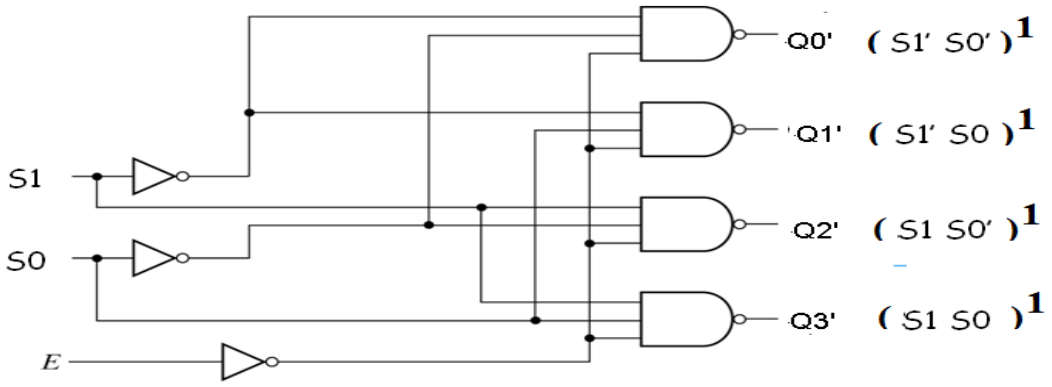
EN	S1	S0	Q0	Q1	Q2	Q3
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0
1	x	x	1	1	1	1

$Q3' = (S1\ S0)' = S1' + S0'$
$Q2' = (S1\ S0')' = S1' + S0$
$Q1' = (S1'\ S0)' = S1 + S0'$
$Q0' = (S1'\ S0')' = S1 + S0$

- The output equations for an active-low decoder are similar, yet somehow different.
- The **active-low** decoders generate **maxterms.**

Building a 2 – Bit Decoder with NAND Gates:

- Add an enable signal (E)



Note: Implementation with NANDs has only one 0 active, for each input combination,

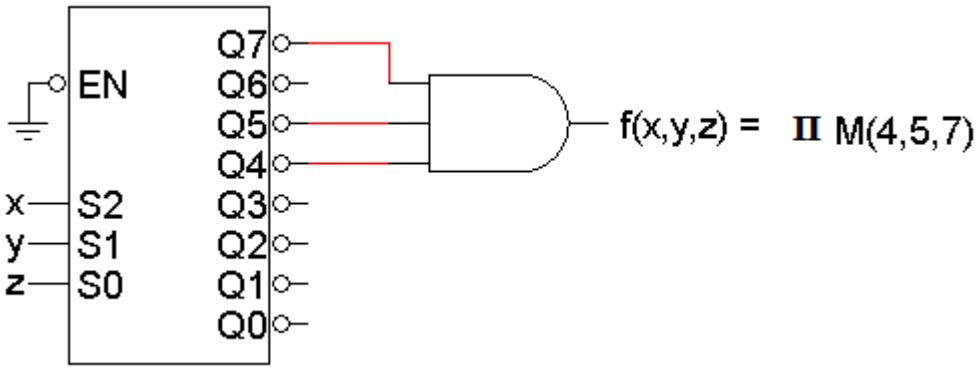
if E = 0.

Active-low decoder example:

- We can use active-low decoders to implement arbitrary functions too, but as a product of maxterms.

Example: 2

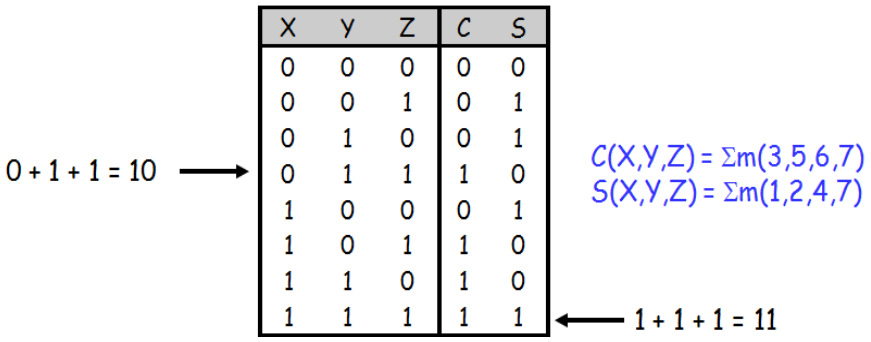
The implementation of the function, f(X, Y, z) = ΠM (4, 5, 7), using an active-low decoder is as follows:



- The "ground" symbol connected to EN represents logical 0, so this decoder is always enabled.

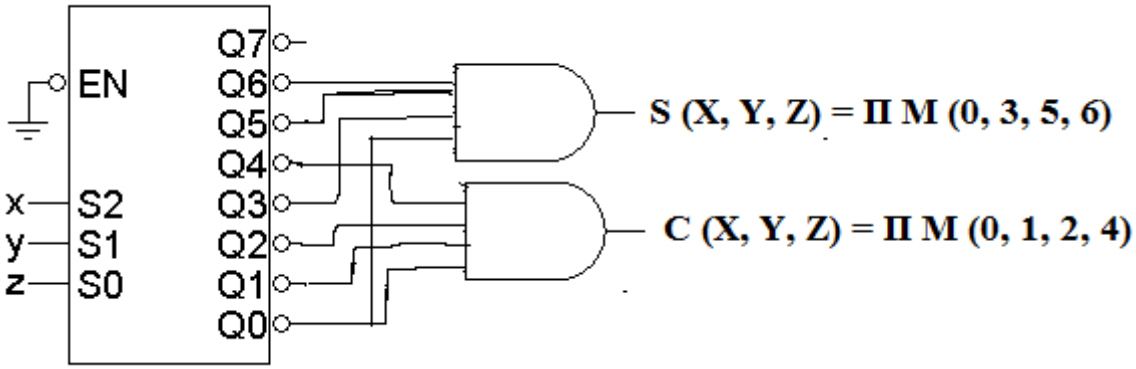
Example: 3

Implement the Full – Adder, using an active-low decoder.



X	Y	Z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

- C (X, Y, Z) = Π M (0, 1, 2, 4)
- S (X, Y, Z) = Π M (0, 3, 5, 6)



Implementation of 4 – to – 16 Decoder using 3 – to – 8 Decoders:

- Enable can also be active high
- In this example, only one decoder can be active at a time.
- x, y, z effectively select output line for w

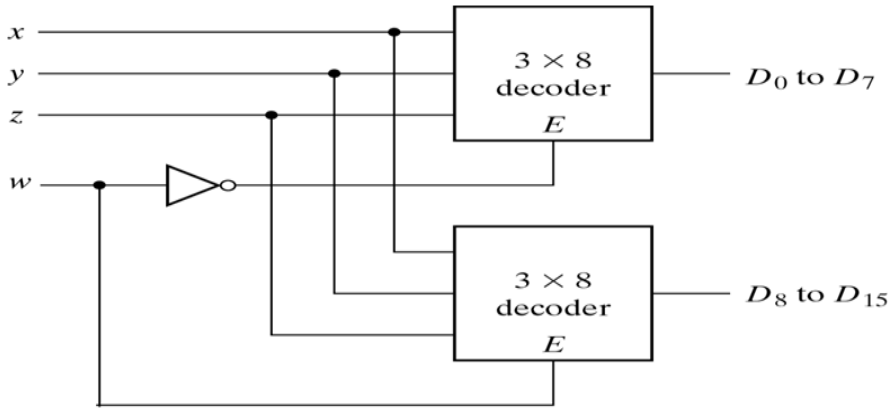


Fig: 4 X 16 Decoder Constructed with Two 3X8 Decoders

ENCODER

An encoder is a digital circuit that performs the inverse operation of a decoder. An encoder has 2^n input lines and n output lines. The output lines generate the binary code corresponding to the input value.

- If the decoder's output code has fewer bits than the input code, the device is usually called an encoder.
  E.g. 2^n-to-n
- One of 2^n inputs = 1
- Output is an n-bit binary number

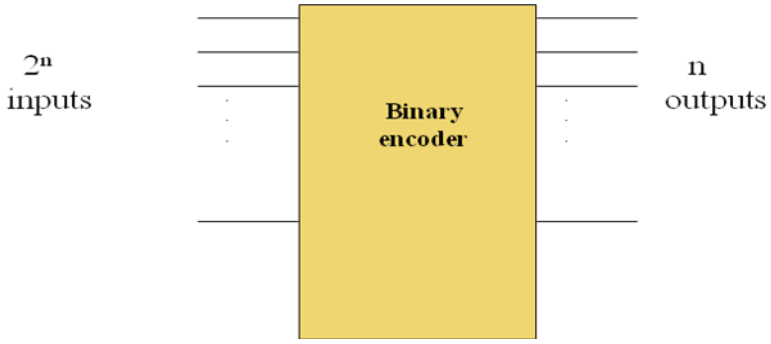


Fig: Block Diagram of Encoder.

8 –to- 3 Binary Encoder

Inputs								Outputs		
I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_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

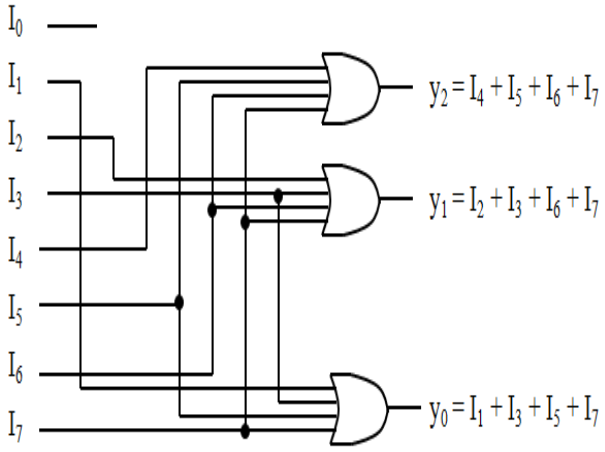


Fig: Truth table & Logic Diagram of 8 – to – 3 Encoder.

Note: At any one time, only one input line has a value of 1.

The encoder can be implemented with OR gates whose inputs are determined directly from the truth table. The output y_2 is equal to 1 for the input octal digits I_4, I_5, I_6, I_7. The output y_1 is equal to 1 for the input octal digits I_2, I_3, I_6, I_7. The output y_0 is equal to 1 for the input octal digits I_1, I_3, I_5, I_7.

Limitation of Encoder:

The Encoder has a limitation that only one input can be active at any given time.

Ambiguities:

1. If two inputs are active simultaneously, the output produces an undefined combination. For example, if I_3 and I_6 are 1 simultaneously, the output of the Encoder will be 111 because all the 3 outputs are equal to 1. This does not represent either binary 3 or binary 6. To resolve this ambiguity, encoder circuits must establish an input priority to ensure that only one input is encoded. If we establish a higher priority for inputs with higher subscripts numbers, and if both I_3 and I_6 are 1 at the same time, the output will be 110 because I_6 has higher priority than I_3.

2. Another ambiguity in the Octal – to – Binary encoder is that an output with all 0's is generated when all the inputs are 0; this output is same as when I_0 is equal to 1. The discrepancy can be resolved by providing one more output to indicate that at least one input is equal to 1.

To overcome the above Ambiguities we use Priority Encoder.

Priority Encoder:

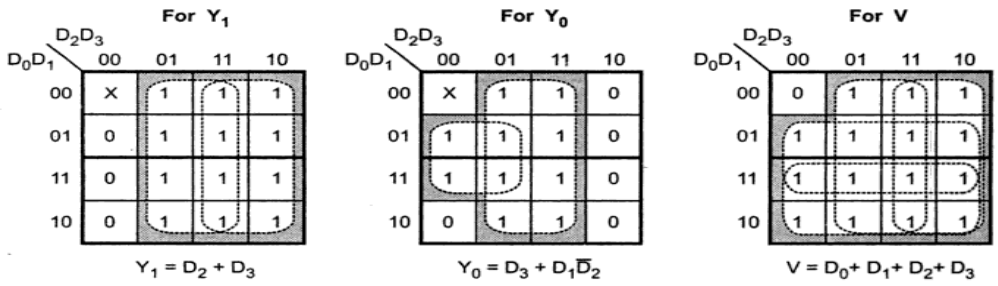
- A priority encoder is an encoder circuit that includes the priority function. The operation of the priority encoder is such that if two or more inputs are equal to 1 at the same time, the input having the highest priority will take precedence.

- In addition to 2 outputs Y_1 and Y_0, the circuit has a 3 rd output designated by V; this is a valid bit indicator that is set to 1 when one or more inputs are equal to 1. If all inputs are 0, there is no valid input and V=0. The other two outputs are not inspected when V=0 and are specified as Don't care condition

Inputs				Outputs		
D_0	D_1	D_2	D_3	Y_1	Y_0	V
0	0	0	0	×	×	0
1	0	0	0	0	0	1
×	1	0	0	0	1	1
×	×	1	0	1	0	1
×	×	×	1	1	1	1

Truth table of 4-Bit Priority Encoder

K-map simplification



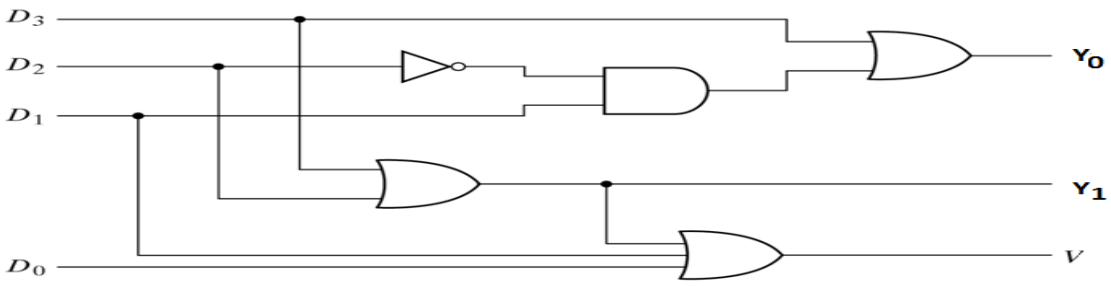


Fig: Input Priority Encoder

Multiplexer (Data Selector)

- It is a combinational circuit that selects binary information from one of the input lines and directs it to a single output line based on the selection input.
- Usually there are 2^n input lines and n selection lines whose bit combinations determine which input line is selected.
- In a 2^n – to – 1 multiplexer, there are 2^n inputs, n selection lines and 1 output line.
- For example, in 2^1 – to - 1 multiplexer if selection S is zero, then I_0 has the path to output and if S is one, I_1 has the path to output .

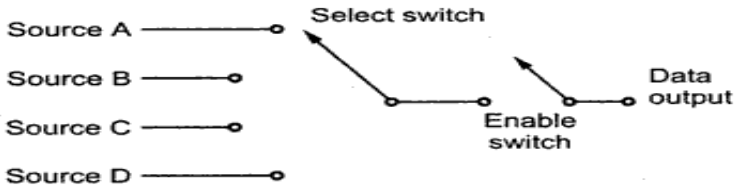
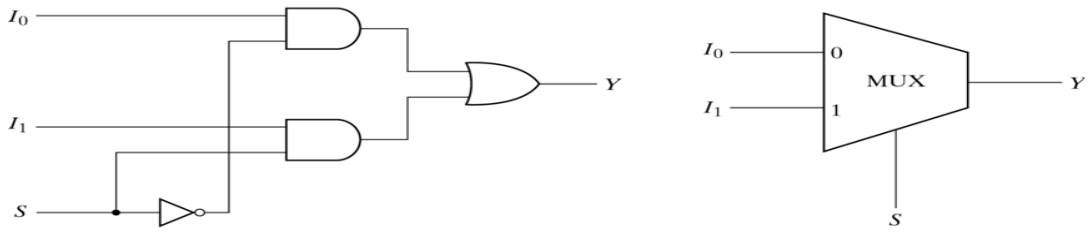


Fig: Analog selector switch

Note: Multiplexer acts as a switch.

2^1 – to - 1 multiplexer

Function Table	
S	Output, Y
0	I_0
1	I_1



(a) Logic diagram (b) Block diagram

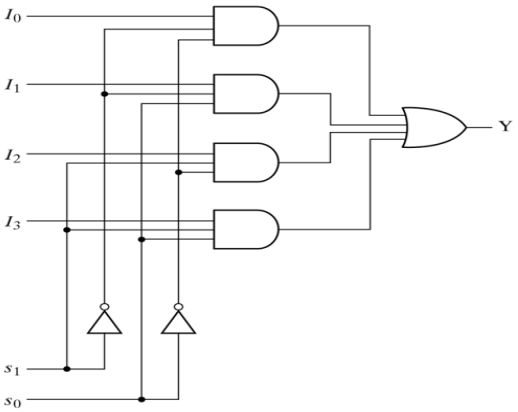
Fig : 2 - to - 1 Line Multiplexer

Note:

In general, a 2^n – to -1 line Mux is constructed from an n – to – 2^n Decoder by adding to it 2^n input lines, one to each AND gate. The outputs of the AND gates are applied to a single OR gate.

MUX: A decoder + an OR gate.

2^2 – to - 1 Multiplexer:



(a) Logic diagram

s_1	s_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

(b) Function table

Fig: 4 - to 1 Line Multiplexer

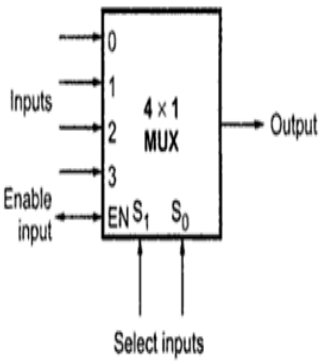
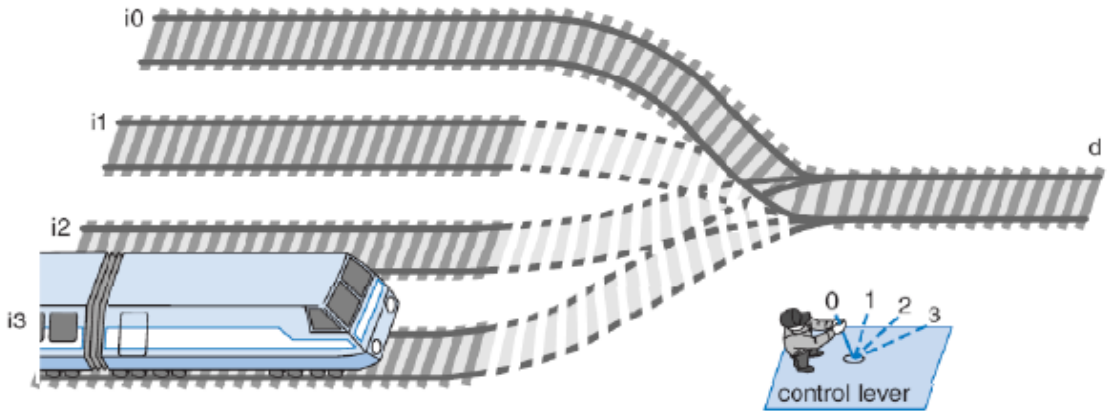


Fig: Logic symbol of 4 - to - 1 Mux.

Example: Like a railyard switch



2^3 – to – 1 Multiplexer:

Selection Lines			**Output**
S_2	**S_1**	**S_0**	**Y**
0	0	0	D_0
0	0	1	D_1
0	1	0	D_2
0	1	1	D_3
1	0	0	D_4
1	0	1	D_5
1	1	0	D_6
1	1	1	D_7

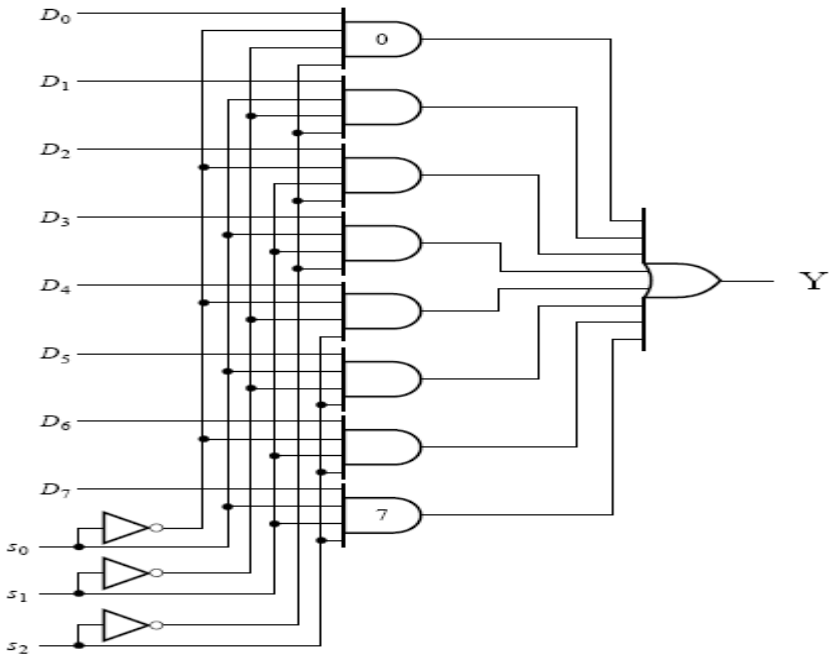
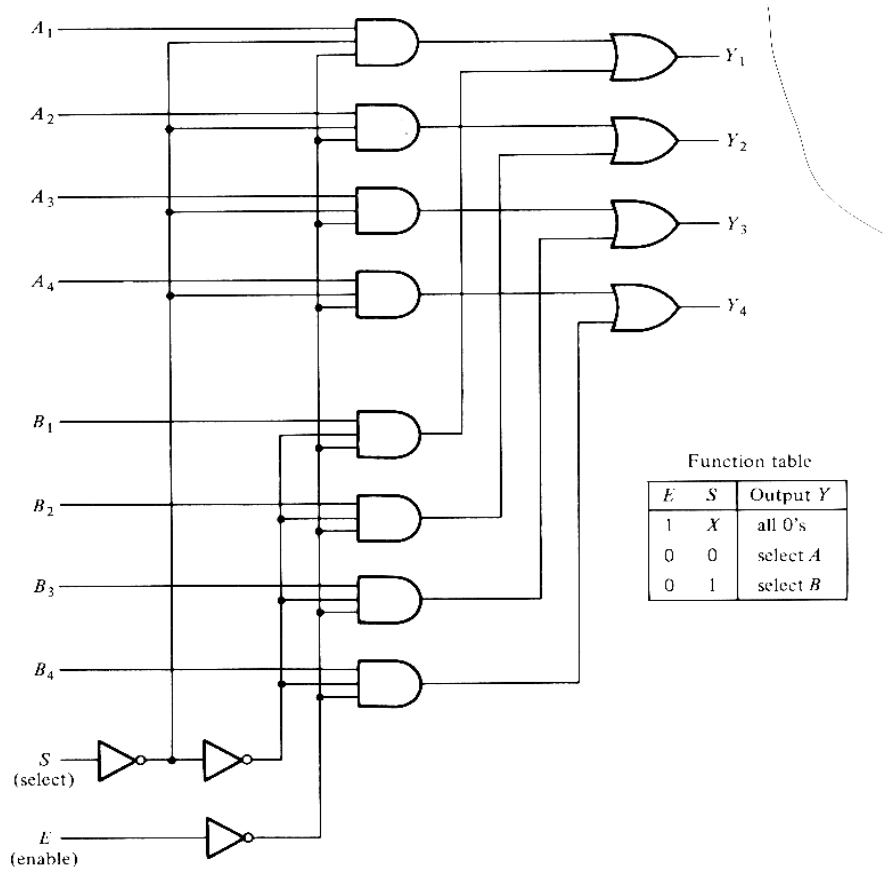


Fig: Multiplexer with 8 Data inputs.

Quadruple 2 –to – 1 Line Mux :



Function table

E	S	Output Y
1	X	all 0's
0	0	select A
0	1	select B

Boolean function Implementation

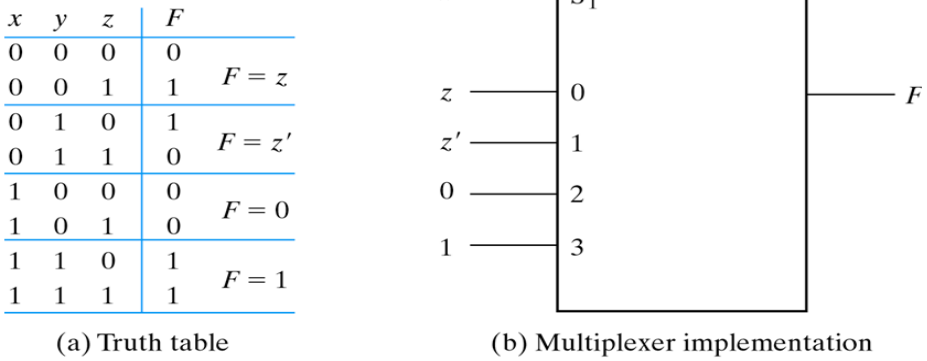
The method for implementing Boolean function using multiplexer is as follows:

For doing that assume Boolean function has n variables. We have to use multiplexer with n-1 selection lines and

- The first n-1 variables of function are used for input: Selection lines.
- The remaining single variable (named z) is used for data input. Each data input can be z, z', 1 or 0. From truth table we have to find the relation of F and z to be able to design input lines.

Example: 1

F (X, y, z) = ∑ (1, 2, 6, 7)



x	y	z	F	
0	0	0	0	$F = z$
0	0	1	1	
0	1	0	1	$F = z'$
0	1	1	0	
1	0	0	0	$F = 0$
1	0	1	0	
1	1	0	1	$F = 1$
1	1	1	1	

(a) Truth table

(b) Multiplexer implementation

Fig: Implementing the Boolean function with a Multiplexer

Example: 2 F (A,B,C,D) = ∑(1,3,4,11,12,13,14,15)

A	B	C	D	F	
0	0	0	0	0	$F = D$
0	0	0	1	1	
0	0	1	0	0	$F = D$
0	0	1	1	1	
0	1	0	0	1	$F = D'$
0	1	0	1	0	
0	1	1	0	0	$F = 0$
0	1	1	1	0	
1	0	0	0	0	$F = 0$
1	0	0	1	0	
1	0	1	0	0	$F = D$
1	0	1	1	1	
1	1	0	0	1	$F = 1$
1	1	0	1	1	
1	1	1	0	1	$F = 1$
1	1	1	1	1	

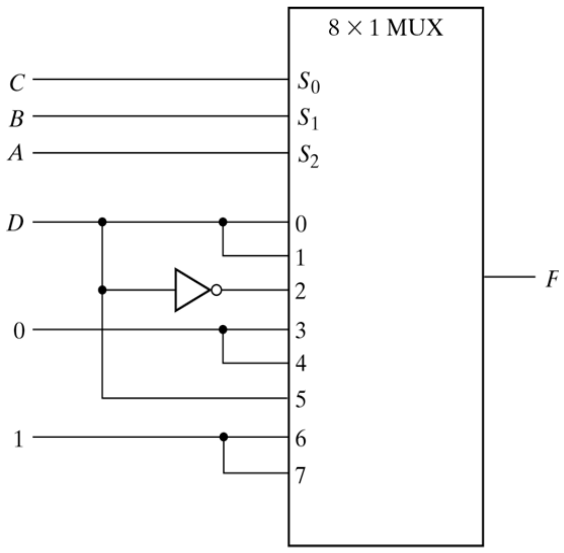
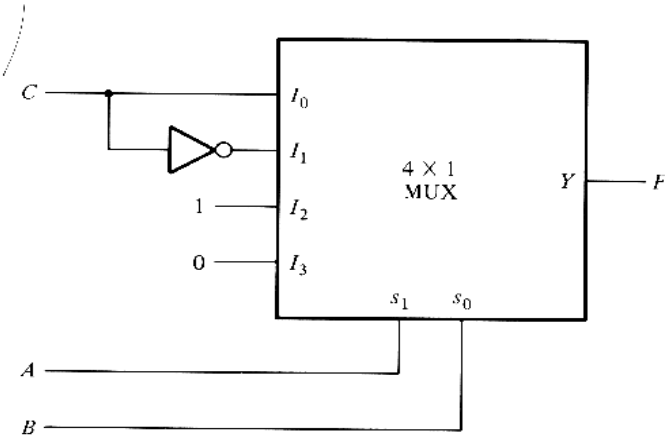


Fig: implementing a 4 Input Function with Multiplexer

Note: It is possible to use any other variable of the function F (A, B, C) = ∑ **m (1, 2, 4, 5)** for the MUX data inputs as follows:

A	B	C	F	
0	0	0	0	$F = C$
0	0	1	1	
0	1	0	1	$F = C'$
0	1	1	0	
1	0	0	1	$F = 1$
1	0	1	1	
1	1	0	0	$F = 0$
1	1	1	0	

(a) Truth table



(b) Multiplexer implementation

	I_0	I_1	I_2	I_3
C'	0	(2)	(4)	6
C	(1)	3	(5)	7
	C	C'	1	0

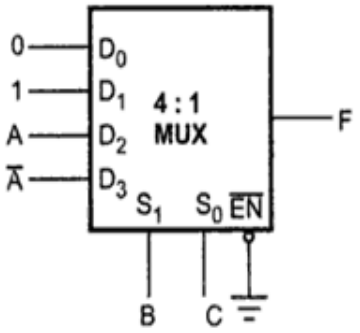
(c) Implementation table

Example: 3

Implement the Boolean function using 4: 1 Mux. F (A, B, C) = ∑ m (1, 3, 5, 6)

Minterm	A	B	C	F
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

(a) Truth table



(b) Multiplexer implementation

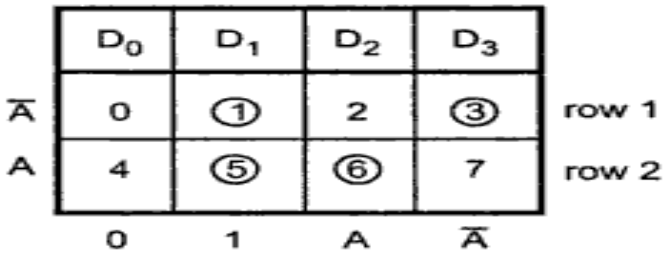
Steps to be followed to obtain the implementation Table

The implementation table is the List of the inputs of the Mux and under them the list of all the minterms in two rows. The first row lists all those minterms where 'A' is complemented, and the second row lists all the minterms with 'A' uncomplemented. The minterms given in the function are circled and then each column is inspected separately as follows:

- If the two minterms in a column are not circled, 0 is applied to the corresponding mux Input (see column 0).
- If the two minterms in a column are circled, 1 is applied to the corresponding mux Input (see column 1).
- If the minterms in the second row is circled, and minterm in the first row is not circled, A is applied to the corresponding mux Input (see column 2).
- If the minterms in the first row is circled, and minterm in the second row is not circled, A^1 is applied to the corresponding mux Input (see column 3).

Procedure:

- assign an ordering sequence of the input variable
- The leftmost variable (A) will be used for the input lines
- Assign the remaining n-1 variables to the selection lines w.r.t. their corresponding sequence
- List all the minterms in two rows (A' and A)
- Circle all the minterms of the function
- Determine the input lines



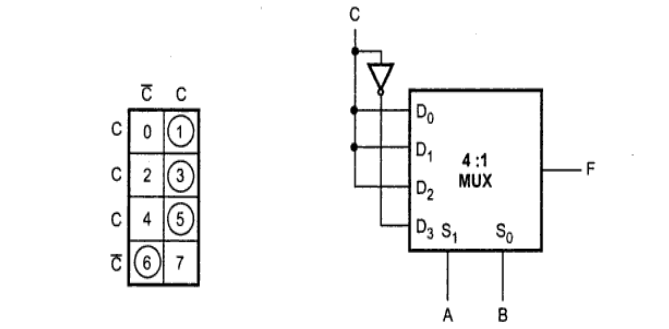
(c) Implementation table

The Mux implementation is also possible by connecting the most significant variables i.e., A and B to the select lines of the Mux. The procedure is same but with Minor changes:

Alternate method:

Implementation table consists of two columns. The first column lists all those minterms where least significant variable C is complemented (C^1), and the second column lists all the minterms with C un-complemented. The minterms given in the function are circled and then each row is inspected separately as follows:

- If the two minterms in a row are not circled, 0 is applied to the corresponding mux Input (see row 0).
- If the two minterms in a row are circled, 1 is applied to the corresponding mux Input (see row 1).
- If the minterms in the second column is circled, and minterm in the first column is not circled, C is applied to the corresponding mux Input (see row 2).
- If the minterms in the first column is circled, and minterm in the second column is not circled, C^1 is applied to the corresponding mux Input (see row 3).



(a) Implementation table Fig. 6.52 (b) Multiplexer Implementation

Example: 4

Implement the following Boolean function using 8:1 Mux. F(A,B,C,D) =∑ m (0,1,3,4,8,9,15)

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

Fig: Implementation table

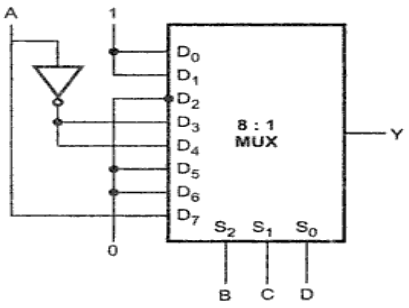
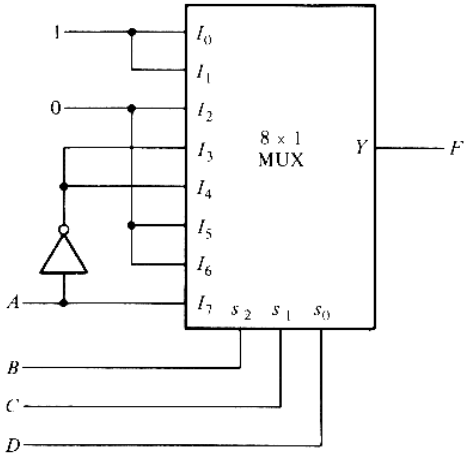


Fig: Multiplexer Implementation

Example: 5

Example: F(A,B,C,D)= ∑ m (0,1,3,4,8,9,15)

	I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7
A'	⓪	①	2	③	④	5	6	7
A	⑧	⑨	10	11	12	13	14	⑮
	1	1	0	A'	A'	0	0	A



Example: 6

Implement the following Boolean function using 4:1 Mux. F(A,B,C,D) =∑ m (0,1,2,4,6,9,12,14)

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

Fig: Implementation Table

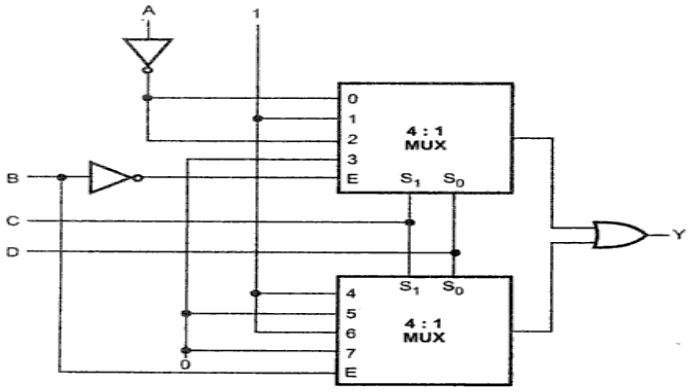


Fig: Implementation using two 4:1 Mux

Example: 7

Implement the following Boolean function using 8:1 Mux.

F (A, B, C, D) = A^1 B D^1 + ACD + B^1 C D + A^1 C^1D.

The given function is not in SOP form. First convert it into Standard SOP form.

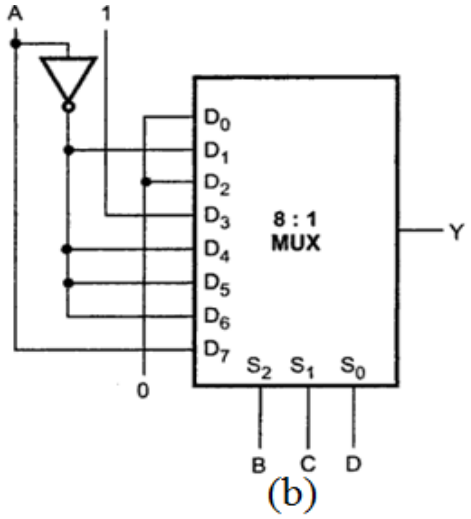
F(A,B,C,D) = A^1 B D^1 ($C+C^1$) + ACD ($B+B^1$) + B^1CD ($A+A^1$) +A^1C^1D ($B+B^1$)
= A^1BCD^1 + $A^1BC^1D^1$ + ABCD + AB^1C D + AB^1CD + A^1B^1CD + A^1BC^1D + $A^1B^1C^1D$.
= A^1BCD^1 + $A^1BC^1D^1$ + ABCD + A B^1CD + A^1B^1CD + A^1BC^1D + $A^1B^1C^1D$

F (A, B, C, D) = $A^1 B^1 C^1 D + A^1 B^1 C D + A^1 B C^1 D^1 + A^1 B C^1 D + A^1 B C D^1 + A B^1 C D + ABCD$

Number	Minterm	A	B	C	D	Y
0		0	0	0	0	0
1	$A^1B^1C^1D$	0	0	0	1	1
2		0	0	1	0	0
3	A^1B^1CD	0	0	1	1	1
4	$A^1 B C^1 D^1$	0	1	0	0	1
5	A^1BC^1D	0	1	0	1	1
6	A^1BCD^1	0	1	1	0	1
7		0	1	1	1	0
8		1	0	0	0	0
9		1	0	0	1	0
10		1	0	1	0	0
11	AB^1CD	1	0	1	1	1
12		1	1	0	0	0
13		1	1	0	1	0
14		1	1	1	0	0
15	ABCD	1	1	1	1	1

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

(a)



(b)

Fig: (a) Implementation Table and (b)Mux Implementation

Example: 8

Implement the following Boolean function using 8:1 Mux.

F(A,B,C,D) = Π M (0,3,5,6,8,9,10,12,14)

Instead of minterms, maxterms are specified. So, we have to circle maxterms which are not included in the function.

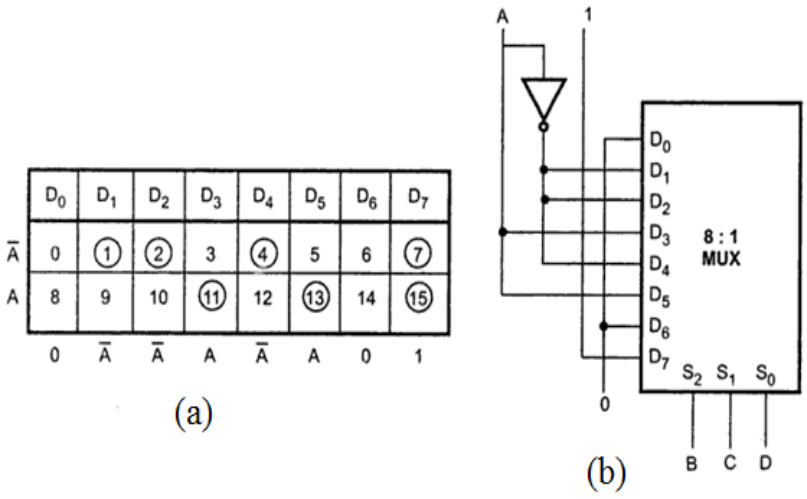
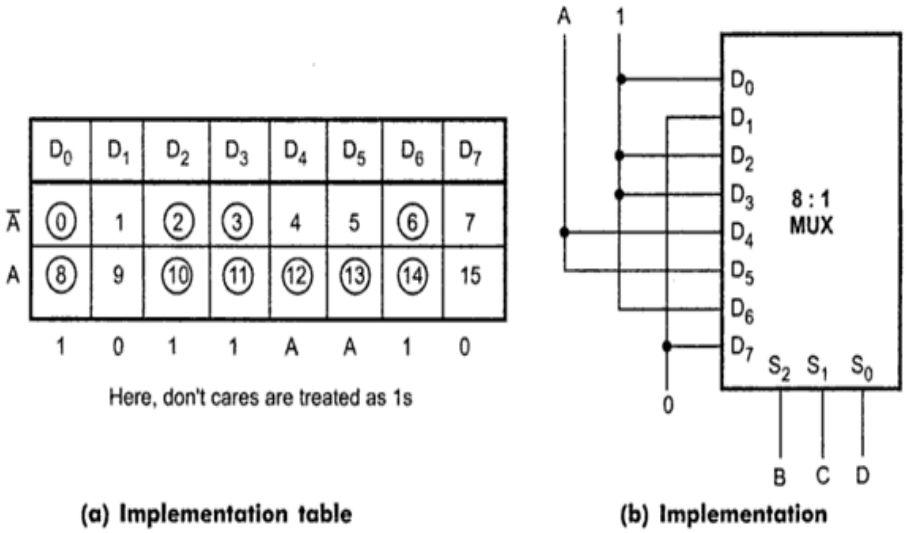


Fig: (a) Implementation table and (b) Mux Implementation

Example: 9

Implement the following Boolean function using 8:1 Mux.

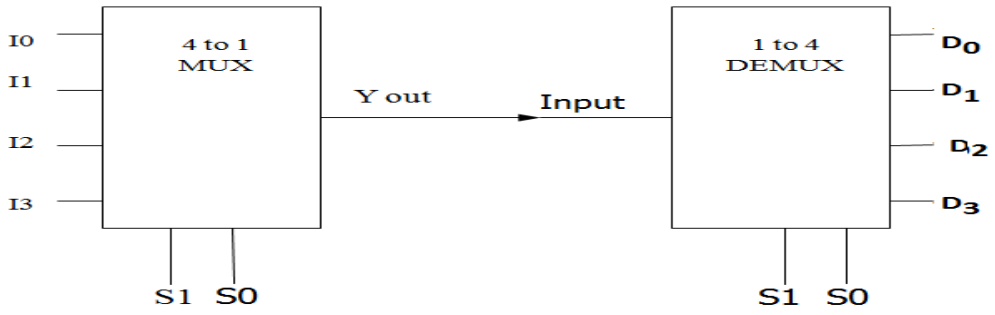
F(A,B,C,D) =∑ m (0,2,6,10,11,12,13) + d (3,8,14)



De-multiplexers

- A **DEMUX** basically reverses the MUX function.
- It takes digital information from one line and distributes it to a given number of output lines based on the selection lines.
- It also known as data distributor.

The diagram below shows the relation between a multiplexer and a De-multiplexer.



Selection Lines		Data Input	Output Lines			
S_1	S_0	D_{in}	D_0	D_1	D_2	D_3
0	0	0	0	0	0	0
0	0	1.	1	0	0	0
0	1	0	0	0	0	0
0	1	1	0	1	0	0
1	0	0	0	0	0	0
1	0	1	0	0	1	0
1	1	0	0	0	0	0
1	1	1	0	0	0	1

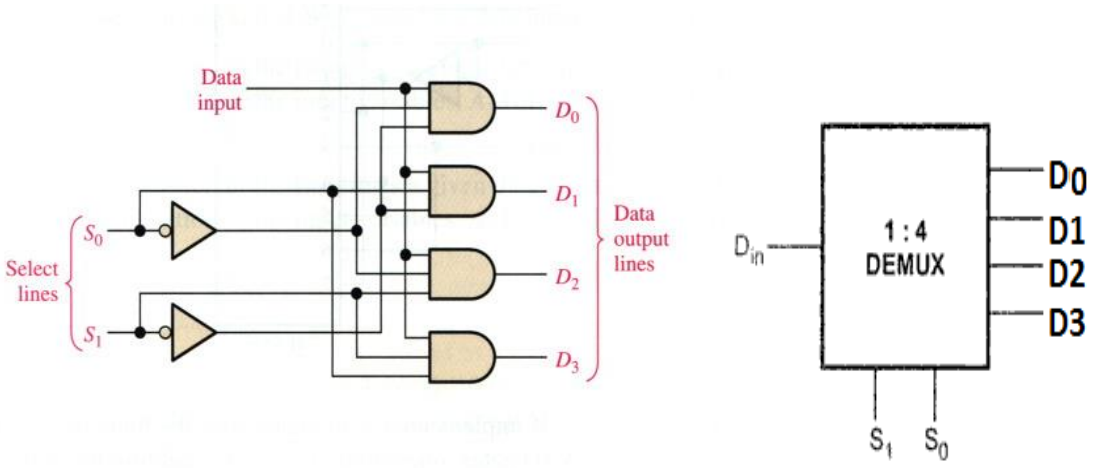


Fig: Logic Diagram and Logic Symbol of 1: 4 De-Mux.

Enable	S1	S0	Din	Y0	Y1	Y2	Y3
0	X	X	X	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	1	1	0	0	0
1	0	1	0	0	0	0	0
1	0	1	1	0	1	0	0
1	1	0	0	0	0	0	0
1	1	0	1	0	0	1	0
1	1	1	0	0	0	0	0
1	1	1	1	0	0	0	1

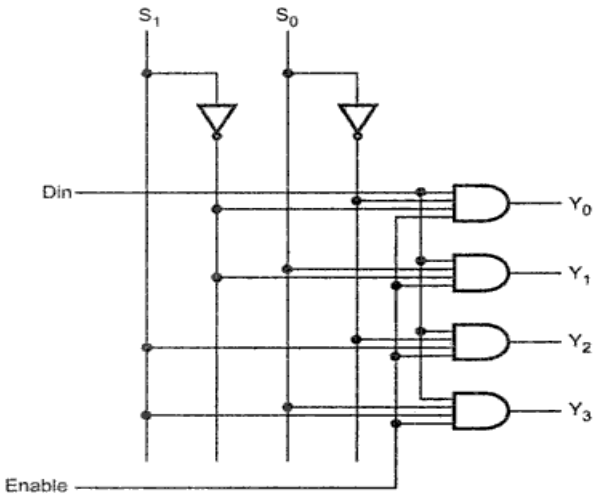


Fig : Truth table and Logic Diagram of 1: 4 De-Mux with Enable Input.

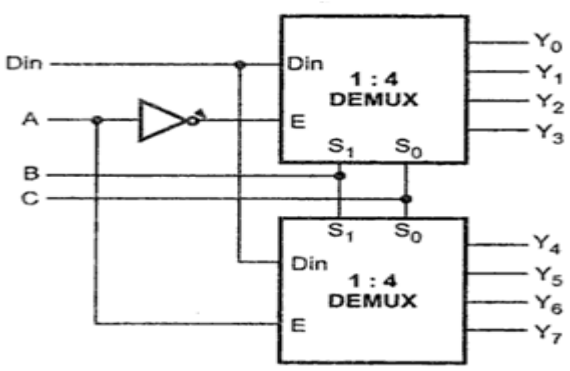


Fig: Implementation of a 1: 8 De-Mux using two 1: 4 De- Mux.

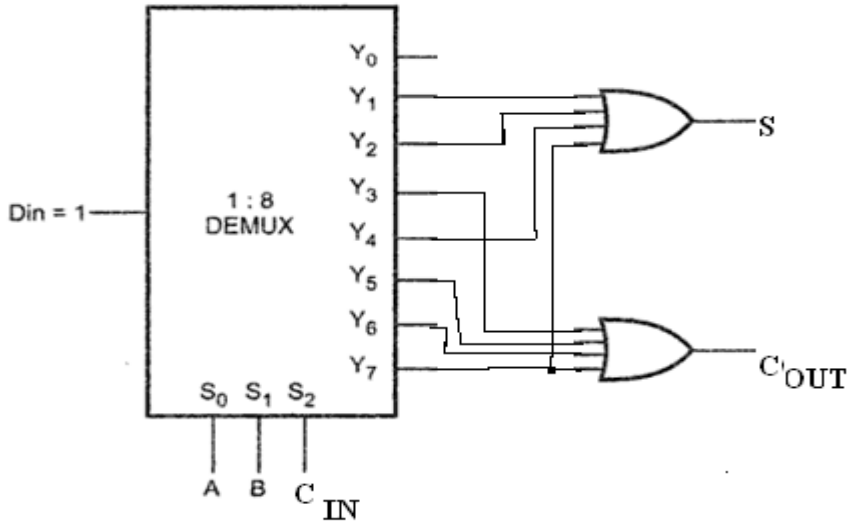
Examples: 1

Implement Full-Adder using De- Multiplexer:

INPUTS			OUTPUT	
A	**B**	**C_{IN}**	**C_{OUT}**	**S**
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$S\ (A, B, C_{IN}) = \sum m\ (1, 2, 4, 7)$

$C_{OUT}\ (A, B, C_{IN}) = \sum m\ (3, 5, 6, 7)$



Example: 2

Implement full-subtractor using De- Multiplexer:

D = F (A, B, Bin) = ∑ m (1, 2, 4, 7)

Bout = F (A, B, Bin) = ∑ m (1, 2, 3, 7)

A	B	Bin	D	Bout
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

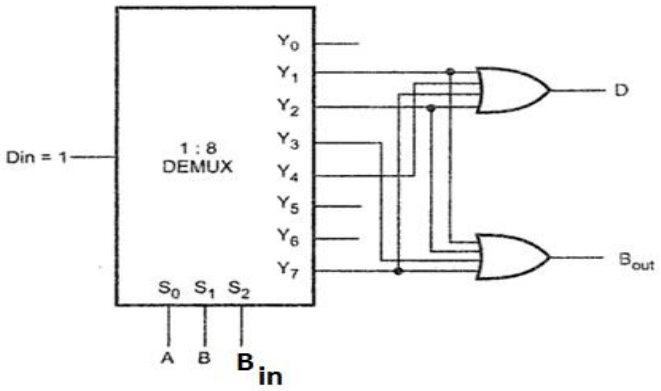
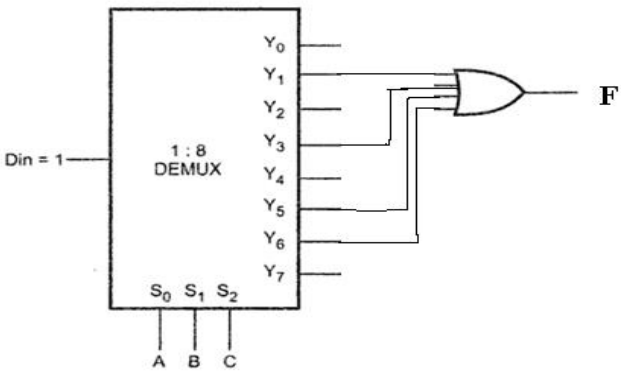


Fig : Implementation of a Full – Subtractor Using 1: 8 De-Mux.

Example: 3

Implement the Boolean function F (A, B, C) = ∑ m (1, 3, 5, 6) using De-Multiplexer.

Minterm	A	B	C	F
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0



CODE CONVERTERS

BINARY TO BCD CONVERTER:

BINARY CODE				BCD CODE				
D	C	B	A	B4	B3	B2	B1	B0
0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	1
0	0	1	0	0	0	0	1	0
0	0	1	1	0	0	0	1	1
0	1	0	0	0	0	1	0	0
0	1	0	1	0	0	1	0	1
0	1	1	0	0	0	1	1	0
0	1	1	1	0	0	1	1	1
1	0	0	0	0	1	0	0	0
1	0	0	1	0	1	0	0	1
1	0	1	0	1	0	0	0	0
1	0	1	1	1	0	0	0	1
1	1	0	0	1	0	0	1	0
1	1	0	1	1	0	0	1	1
1	1	1	0	1	0	1	0	0
1	1	1	1	1	0	1	0	1

For B_0

DC \ BA	00	01	11	10
00	0	1	1	0
01	0	1	1	0
11	0	1	1	0
10	0	1	1	0

$B_0 = A$

For B_1

DC \ BA	00	01	11	10
00	0	0	1	1
01	0	0	1	1
11	1	1	0	0
10	0	0	0	0

$B_1 = DC\overline{B} + \overline{D}B$

For B_2

DC \ BA	00	01	11	10
00	0	0	0	0
01	1	1	1	1
11	0	0	1	1
10	0	0	0	0

$B_2 = \overline{D}C + CB$

For B_3

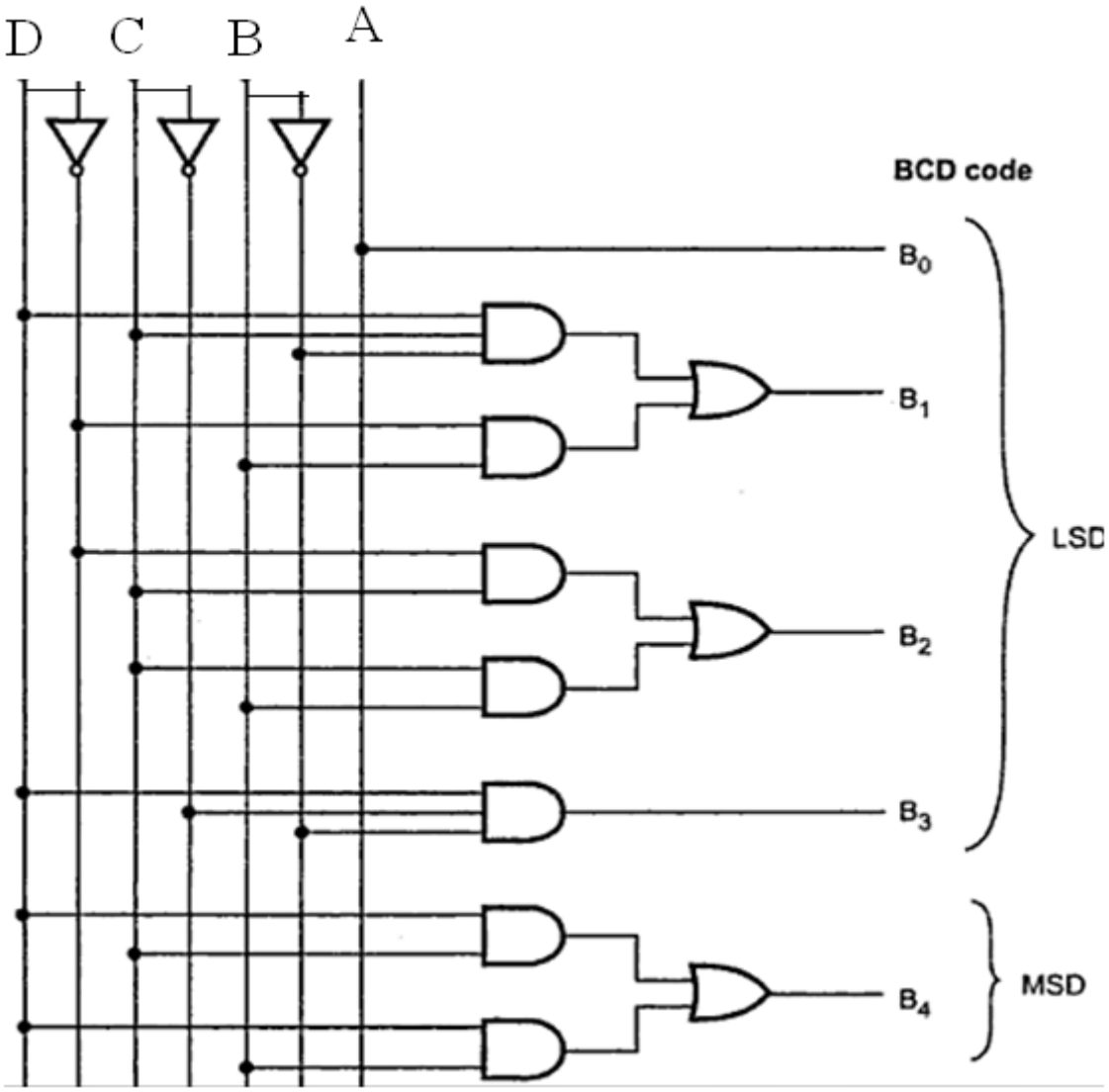
DC \ BA	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	0	0	0	0
10	1	1	0	0

$B_3 = D\overline{C}\,\overline{B}$

For B_4

DC \ BA	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	1	1	1	1
10	0	0	1	1

$B_4 = DC + DB$



BINARY TO GRAY CODE CONVERTER:

DECIMAL	BINARY CODE				GRAY CODE			
	D	C	B	A	G3	G2	G1	G0
0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1
2	0	0	1	0	0	0	1	1
3	0	0	1	1	0	0	1	0
4	0	1	0	0	0	1	1	0
5	0	1	0	1	0	1	1	1
6	0	1	1	0	0	1	0	1
7	0	1	1	1	0	1	0	0
8	1	0	0	0	1	1	0	0
9	1	0	0	1	1	1	0	1
10	1	0	1	0	1	1	1	1
11	1	0	1	1	1	1	1	0
12	1	1	0	0	1	0	1	0
13	1	1	0	1	1	0	1	1
14	1	1	1	0	1	0	0	1
15	1	1	1	1	1	0	0	0

K- Map :

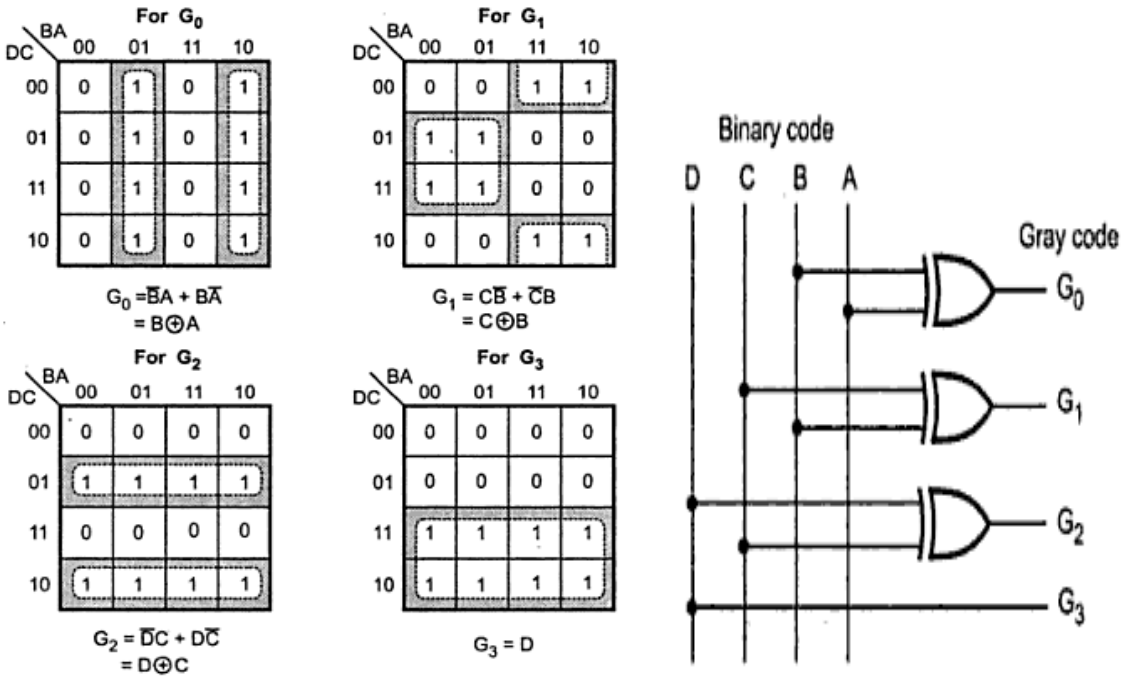


Fig: K- Map and Logic diagram of Binary Code to Gray Code Converter

GRAY TO BINARY CONVERTER:

GRAY CODE				BINARY CODE			
G3	G2	G1	G0	D	C	B	A
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	0	1	0	0	0	1	1
0	1	1	0	0	1	0	0
0	1	1	1	0	1	0	1
0	1	0	1	0	1	1	0
0	1	0	0	0	1	1	1
1	1	0	0	1	0	0	0
1	1	0	1	1	0	0	1
1	1	1	1	1	0	1	0
1	1	1	0	1	0	1	1
1	0	1	0	1	1	0	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

For A

G_3G_2 \ G_1G_0	00	01	11	10
00	0	(1)	0	(1)
01	(1)	0	(1)	0
11	0	(1)	0	(1)
10	(1)	0	(1)	0

$$
\begin{aligned}
A &= (\overline{G}_3G_2 + G_3\overline{G}_2)\,\overline{G}_1\overline{G}_0 + (\overline{G}_3\,\overline{G}_2 + G_3G_2)\,\overline{G}_1G_0 \\
&\quad + (\overline{G}_3G_2 + G_3\,\overline{G}_2)\,G_1G_0 + (\overline{G}_3\overline{G}_2 + G_3G_2)\,G_1\,\overline{G}_0 \\
&= (G_3 \oplus G_2)\,\overline{G}_1\overline{G}_0 + (G_3 \odot G_2)\overline{G}_1G_0 \\
&\quad + (G_3 \oplus G_2)\,G_1G_0 + (G_3 \odot G_2)\,G_1\overline{G}_0 \\
&= (G_3 \oplus G_2)\,(\overline{G}_1\overline{G}_0 + G_1G_0) + (G_3 \odot G_2)\,(\overline{G}_1G_0 + G_1\overline{G}_0) \\
&= (G_3 \oplus G_2)\,(G_1 \odot G_0) + (G_3 \odot G_2)\,(G_1 \oplus G_0) \\
&= (G_3 \oplus G_2)\,\overline{(G_1 \oplus G_0)} + \overline{(G_3 \oplus G_2)}\,(G_1 \oplus G_0) \\
&= (G_3 \oplus G_2) \oplus (G_1 \oplus G_0)
\end{aligned}
$$

For B

G_3G_2 \ G_1G_0	00	01	11	10
00	0	0	1	1
01	1	1	0	0
11	0	0	1	1
10	1	1	0	0

$$B = (\overline{G}_3\overline{G}_2 + G_3G_2)G_1 + (\overline{G}_3G_2 + G_3\overline{G}_2)\overline{G}_1$$
$$= (G_3 \odot G_2)G_1 + (G_3 \oplus G_2)\overline{G}_1$$
$$= (\overline{G_3 \oplus G_2})G_1 + (G_3 \oplus G_2)\overline{G}_1$$
$$= G_3 \oplus G_2 \oplus G_1$$

For C

G_3G_2 \ G_1G_0	00	01	11	10
00	0	0	0	0
01	1	1	1	1
11	0	0	0	0
10	1	1	1	1

For D

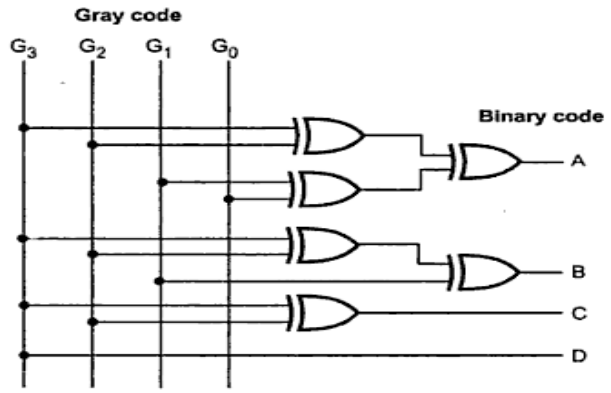
G_3G_2 \ G_1G_0	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	1	1	1	1
10	1	1	1	1

Fig. 6.100

$C = \overline{G}_3G_2 + G_3\overline{G}_2$ $\quad D = G_3$

$= G_3 \oplus G_2$

Logic diagram



Gray code to binary code converter

Fig: Gray Code to Binary Code

BCD TO GRAY CODE CONVERTER:

BCD CODE				GRAY CODE			
B3	B2	B1	B0	G3	G2	G1	G0
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

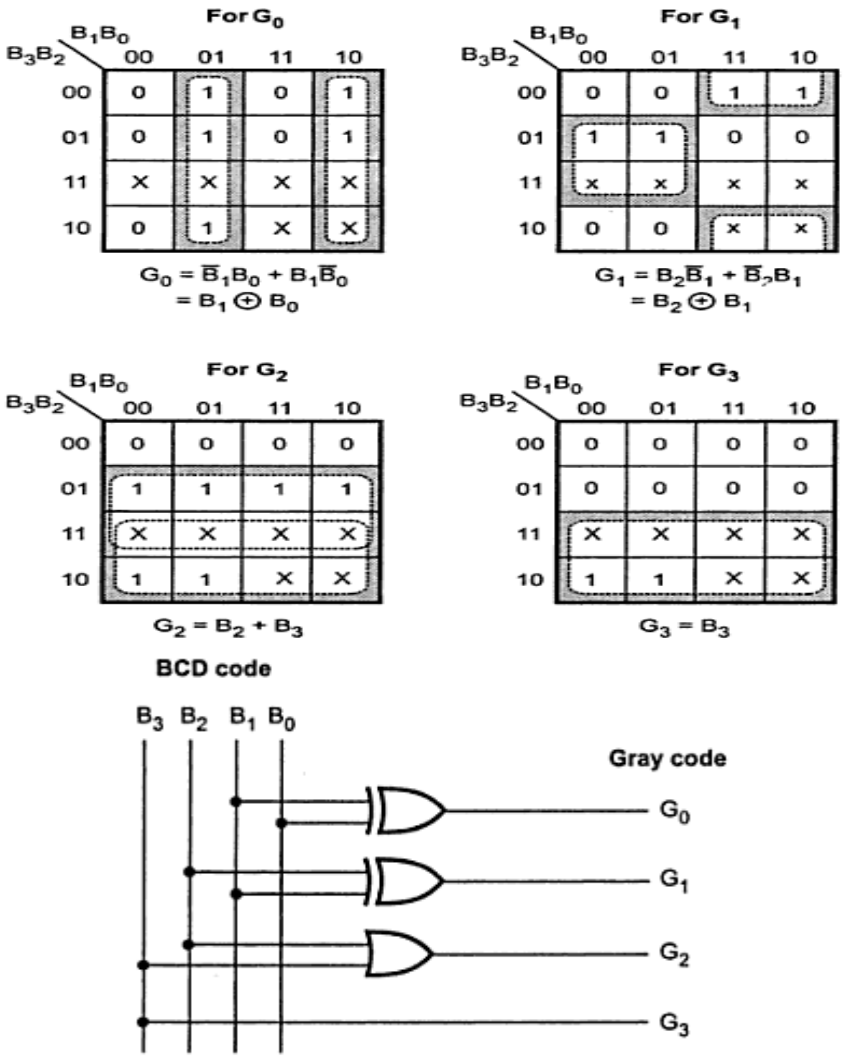
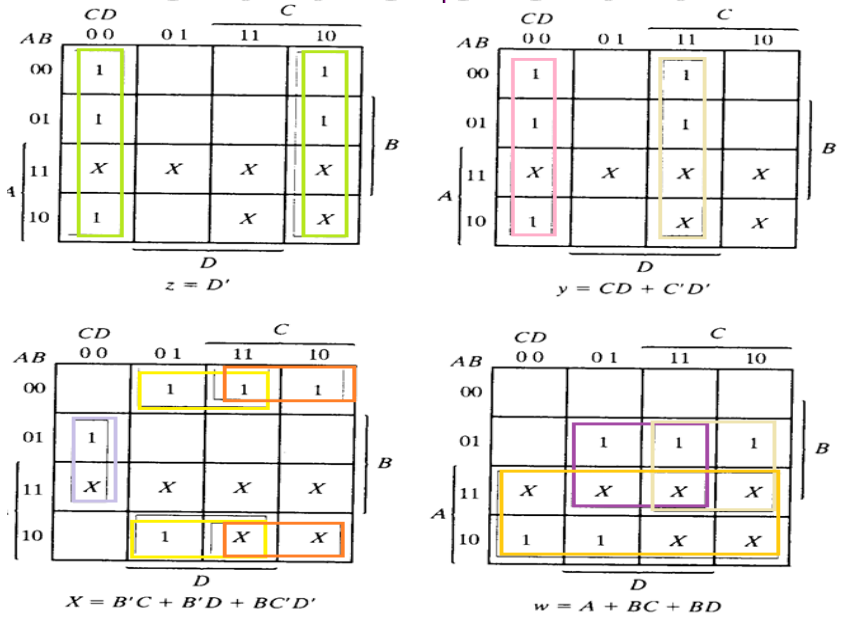


Fig : Logic diagram of BCD to Gray code.

BCD TO EXCESS – 3 CODE CONVERTERS:

Truth table

A	B	C	D	w	x	y	z
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	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



The simplified functions

- z = D'
- y = CD +C'D'
- x = B'C + B'D+BC'D'
- w = A+BC+BD

Another implementation

- z = D'
- y = CD +C'D' = CD + (C+D)'
- x = B'C + B'D+BC'D‘ = B'(C+D) +B(C+D)'
- w = A+BC+BD

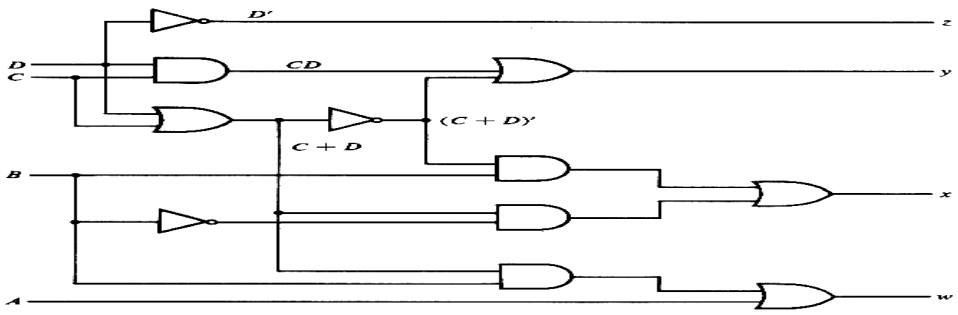


Fig: LOGIC DIAGRAM OF BCD TO EXCESS – 3 CODE CONVERTER.

PROGRAMMABLE LOGIC DEVICES & THRESHOLD LOGIC

There are 3 approaches for the design a digital circuit:

- Use of Fixed function ICs,
- Use of Application Specific Integrated Circuit &
- Use of Programmable Logic Devices.

So for we have seen various digital IC approach for performing basic operations (i.e., NOT,AND and OR) required to implement arbitrary functions and other functions, such as Adders, Comparators, Code converters Multiplexers, De- Mux, Decoders & Encoders etc.

These ICs due to their fix functions are known as fixed function IC's. These IC's are designed by manufacturers and produced in large quantities required for a wide variety of applications. The design of digital circuit using fixed function IC's are seen i.e., these IC's are used to perform only specific function, so they are called fixed function IC's.

In fixed function IC approach, we have to use various fixed function IC's to implement different functional blocks in the digital circuit.

In ASIC approach, a single IC is designed and manufactured to implement the entire circuit.

In PLDs approach, the PLDs are used to implement the Logic functions.

The main advantage of PLD approach is that the PLDs can be easily programmed by individual users for specific application.

S.No.	**Parameter**	**Fixed function IC approach**	**ASIC approach**	**PLD approach**
1	Development Cost	Low	High	Low
2	Space required	Large	Minimum	Less
3	Power required	More	Less	Less
4	Design time	Less	More	Less
5	Reliability	Less Compared to other two approaches	Highest	High
6	Circuit Testing	Easy	Specialized Testing methods are required , which may increase the cost & effort	Easy
7	Design flexibility	Less	Not possible	More
8	Modification in the design	Possible with change in circuit AND/OR with change in components.	Not possible	Possible without any circuit or component changes. But, only by reconfigurable

- PLDs can be reprogrammed in few sec and hence gives more flexibility to experiment with designs.
- Reprogramming feature of PLDs allows changes/modifications in the previously designed circuits. The above advantages and some parameters like cost, design flexibility and design time makes the PLDs more popular in digital design.

According to the architecture, complexity & flexibility in programming, PLDs are classified as

- PROMs,
- PLAs,
- PAL,
- FPGAs and
- CPLDs etc.

ROM (Read Only Memory)

A ROM is memory device in which permanent binary information is stored. The binary information must be specified by the designer and is then embedded in the Unit to form the required interconnection pattern. Once the pattern is established, it stays within the Unit even when power is turned off and on again.

A block diagram of ROM is shown in Figure below.

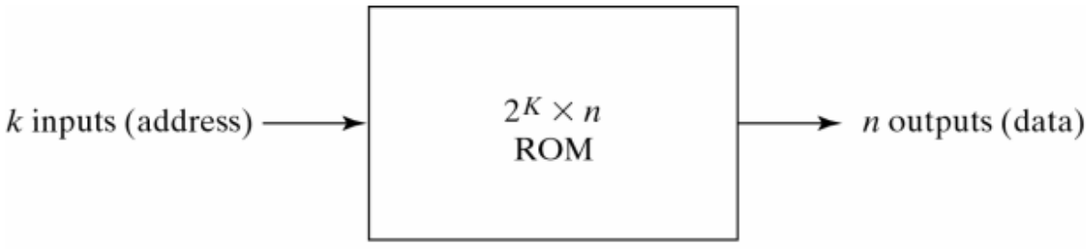


Fig: ROM Block Diagram.

- It consists of K inputs and n output lines.
- Each bit combination of input variable is called an address. Each bit combination that comes out of the output lines is called a word.
- The number of bits / word is equal to the number of output lines 'n'.
- The address specified in binary number denotes one of the minterms of K variable. The number of distinct address possible with K – variable is 2^K.
- An output word can be selected by a unique address and since there are 2^n distinct address in a PROM , there are 2^n distinct words in the PROM.

The word available on the output lines at any given time depends on the address value applied to input lines.

Example:

A 32 x 8 ROM consists of 32 words of 8 bits each. There are five input lines that form the binary numbers from 0 through31 for the address. Figure below shows the internal logic construction of ROM.

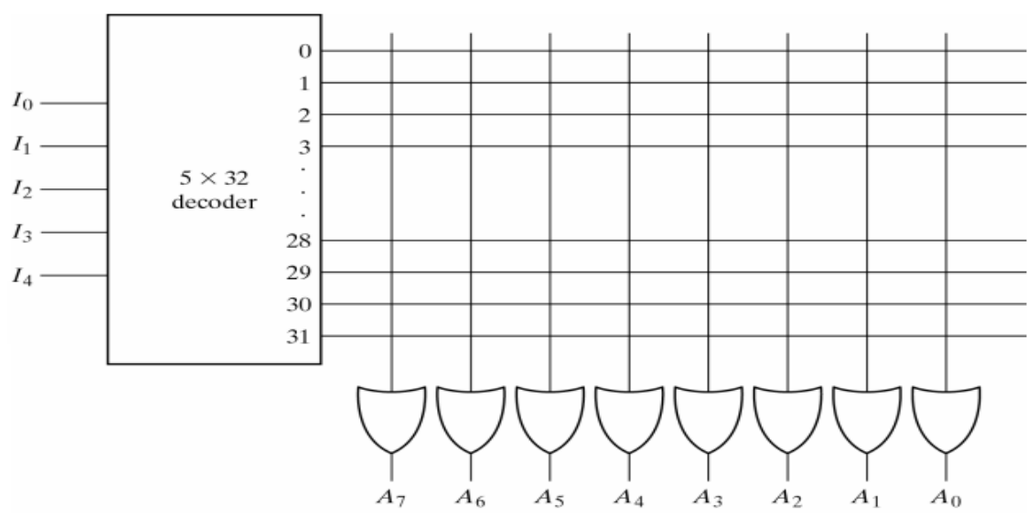


Fig: Internal Logic diagram of 32 × 8 ROM.

- The five input lines are decoded by into 32 distinct outputs (memory addresses) using a 5 x 32 decoder. Each output of decoder represents a memory address or minterm. The 32 outputs of the decoder are connected to each of the 8 OR gates. Each OR gate has 32 input connections (i.e. each output of the decoder is connected to one of the inputs of each OR gate). Since each OR gate has 32 input connections and there are 8 OR gates, the ROM contains 32 x 8 = 256 internal connections 32 x 8.

- In general, a 2^K x n ROM will have an internal k x 2^K decoder and n OR gates with 2^K x n internal connections.

- A programmable connection (a cross point) between two lines is logically equivalent to a switch that can be closed (two lines are connected) or open (two lines are disconnected). The programmable interconnection between two lines is called cross point. A switch can be a fuse that normally connects the two points, but can be opened by blowing the fuse using a high voltage pulse.

Programming by blowing fuses

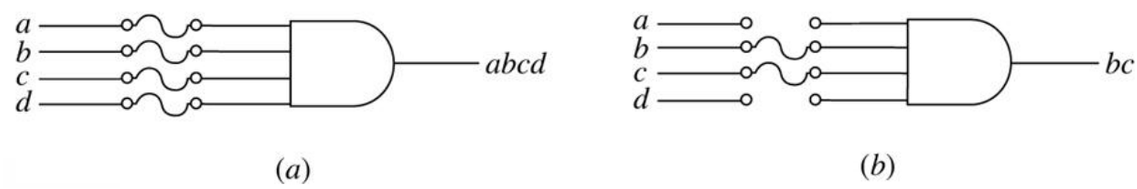


Fig: (a) Before Programming **(b) After Programming**

The internal binary storage of a ROM is specified by a truth table that shows the word content in each address.

For example, the content of a 32 × 8 ROM may be specified with a truth table as follows:

Decimal	**Inputs**					**Outputs**							
No.	**I4**	**I3**	**I2**	**I1**	**I0**	**A7**	**A6**	**A5**	**A4**	**A3**	**A2**	**A1**	**A0**
0	0	0	0	0	0	1	0	1	1	0	1	1	0
1	0	0	0	0	1	0	0	0	1	1	1	0	1
2	0	0	0	1	0	1	1	0	0	0	1	0	1
3	0	0	0	1	1	1	0	1	1	0	0	1	0
30	1	1	1	1	0	0	1	0	0	1	0	1	0
31	1	1	1	1	1	0	0	1	1	0	0	1	1

The truth table shows that the 5 inputs under which all the 32 addresses are listed. At each address, a word of 8 – Bit is stored, which is listed under the output column.

The hardware procedure that programs the ROM results in blowing the fuse links according to the given truth table.
Every 0 listed in the truth table specifies no connection and every 1 listed specifies connection.

Example:

The 8 – Bit word (10110010) in the table is stored permanently at the address 3 (00011). The 4 0's in the word are programmed by blowing the fuse links, between output 3 of the decoder and inputs of the OR gate associated with the outputs A_6 , A_3 , A_2 and A_0. The 4 I's in the word are marked in the diagram with an x to denote a connection in place of a dot used for permanent connection in the logic diagram.

When the inputs of ROM are 00011, all the outputs of the decoder are 0 except for output 3, which is at logic 1. The signal equivalent to logic 1 at the decoder output 3 propagates through the connections to the OR gate output of the A7 , A5 , A4 AND A1. The other 4 outputs remain at 0. The result is that the stored word 101100010 is applied to the eight data outputs as shown below:

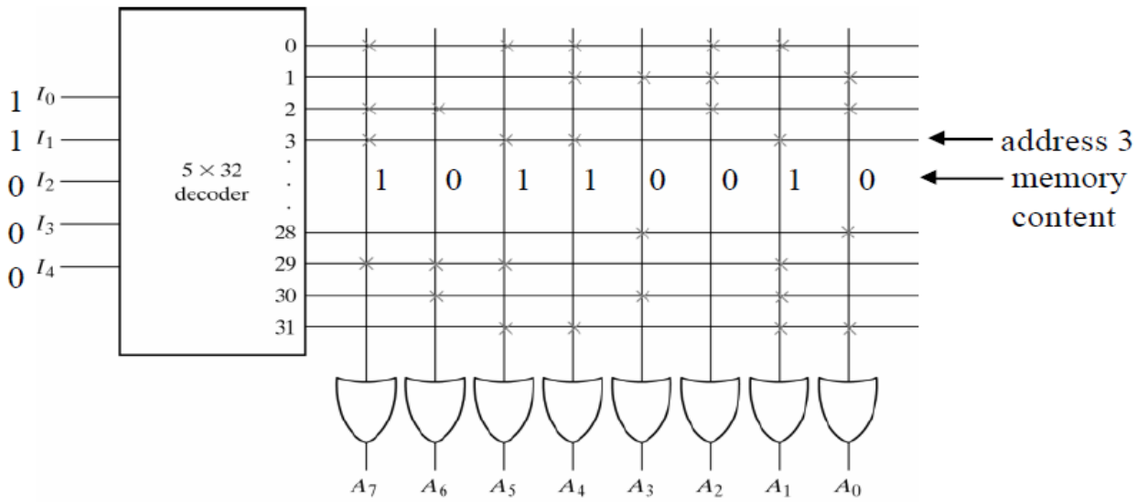


Fig: Programming the ROM according to truth table.

Output A_7 and A_6 can be expressed in sum of minterms as: $A_7\ (I_4,I_3,I_2,I_1,I_0) = \sum m\ (0,2,3...,29)$
$A_6\ (I_4,I_3,I_2,I_1,I_0) = \sum m\ (2,...,29,30)$

In practice, when a combinational circuit is designed by means of a ROM, it is not necessary to design the logic or to show the internal gate connections inside the Unit. The designer work is to specify the particular ROM by its IC number and provide the ROM truth table. The truth table gives all the information needed for programming the ROM. No internal logic diagram is needed.

Example:

Design a combinational circuit using a ROM. The circuit accepts a 3-bit number and generates an output binary number equal to square of the input number.

Step 1: Derive the truth table for combinational circuit.

Inputs			**Outputs**						**Decimal**
A2	**A1**	**A0**	**B5**	**B4**	**B3**	**B2**	**B1**	**B0**	
0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	1	1
0	1	0	0	0	0	1	0	0	4
0	1	1	0	0	1	0	0	1	9
1	0	0	0	1	0	0	0	0	16
1	0	1	0	1	1	0	0	1	25
1	1	0	1	0	0	1	0	0	36
1	1	1	1	1	0	0	0	1	49

Step 2: we can use the partial truth table for the Rom by utilizing certain properties in the output if any like:

- The output B0 is always equal to A0; so there is no need to generate B0 with a ROM. Since it is equal to input variable.
- Output B1 is always equal to 0, so this output is a known constant.

So we need to generate only four outputs with ROM; the other two are readily obtained.

The minimum size ROM needed must have 3 – inputs and 4 – outputs. Three inputs specify 8 words of 4 – bits each.

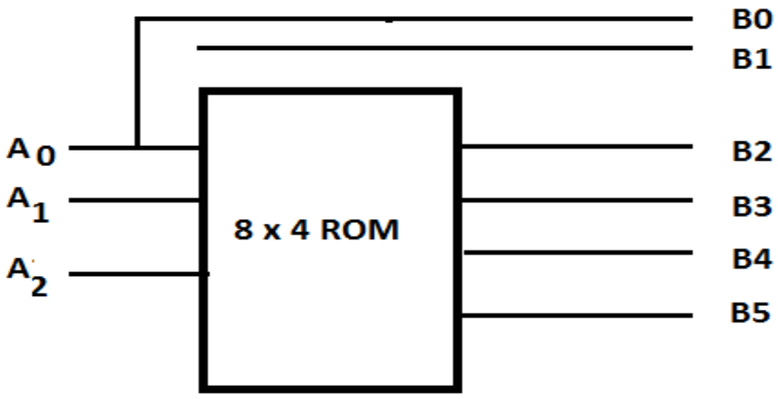
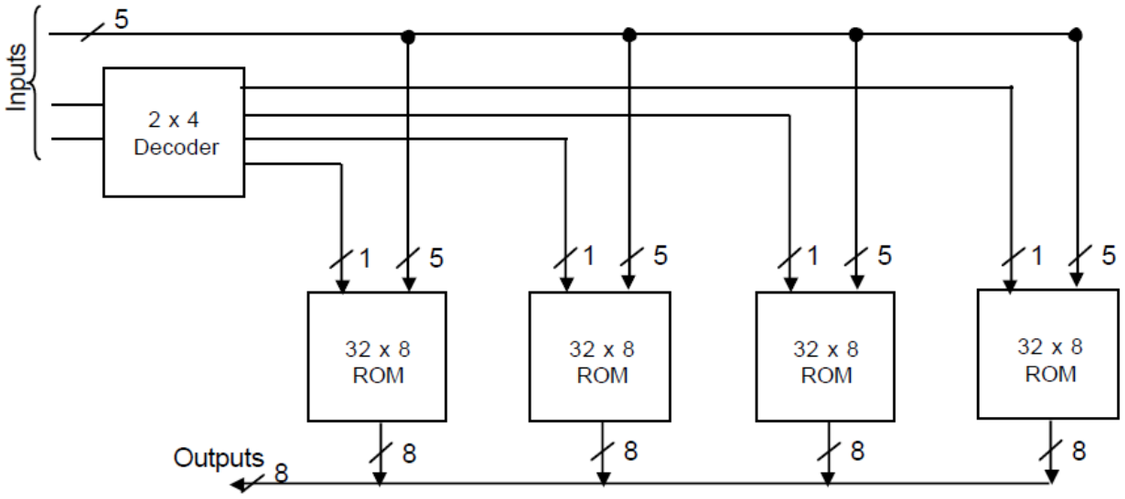


Fig: ROM Implementation.

Implement 128 X 8 ROM chips using 32 x 8 ROM chips:



Combinational PLD's

The PROM is a combinational Programmable Logic Device (PLD).
A combinational PLD is an IC with programmable gates divided into an AND array and OR array to provide an AND – OR sum of product implementation.

Programmable logic devices (PLD) are designed with configurable logic and flip-flops linked together with programmable interconnect.

PLDs provide specific functions, including

- Device-to-device interfacing
- Data communication
- Signal processing
- Data display

Types of combinational PLDs:

There are 3 types of PLDs & they differ in the placement of programmable connections in the AND – OR array.

1. PROM,
2. PLA &
3. PAL.

The basic configuration of three PLDs is shown in figure below:

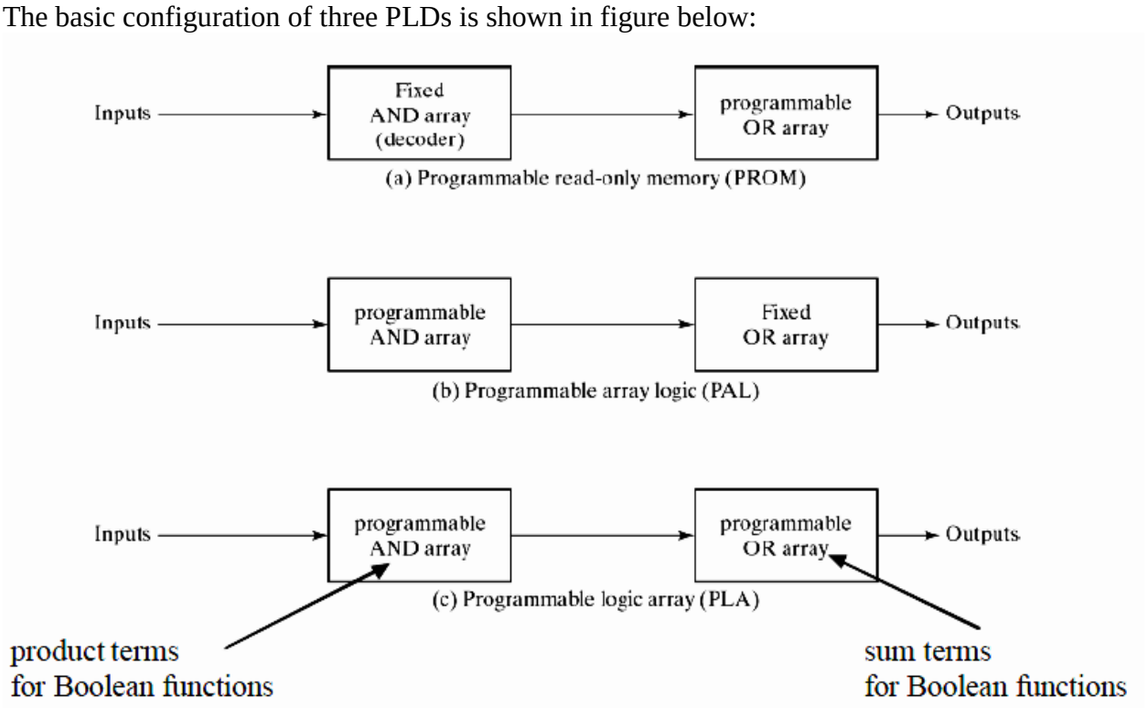


Fig: Basic configuration of three PLDs.

PLA:

The PLA is similar to PROM in concept, except that the PLA does not provide full decoding of the variables and does not generate all min-terms.
In PROM, the decoder is replaced by an array of AND gates that can be programmed to generate any product term of input variables.

The product terms are then connected to OR gates to provide the required Boolean function.

Example : Realize the Boolean function $F_1 = AB^1 + AC + A^1BC^1$ & $F_2 = (AC + BC)^1$.

PLA Programmable table:

	Product term	**Inputs**			**Outputs**	
					T	**C**
		A	**B**	**C**	**F1**	**F2**
AB^1	1	1	0	-	1	-
AC	2	1	-	1	1	1
A^1BC^1	3	0	1	0	1	-
BC	4	-	1	1	-	1

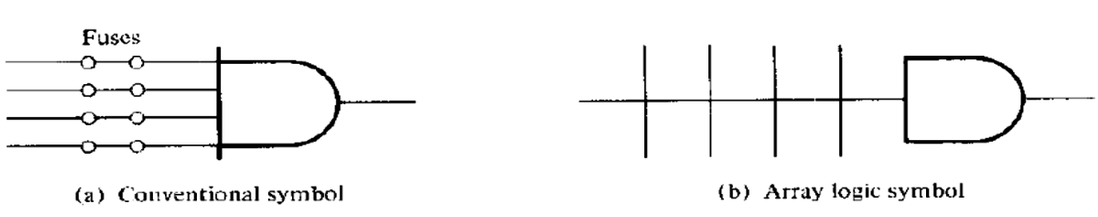


Fig: Two graphic symbols for AND gate.

Implementation:

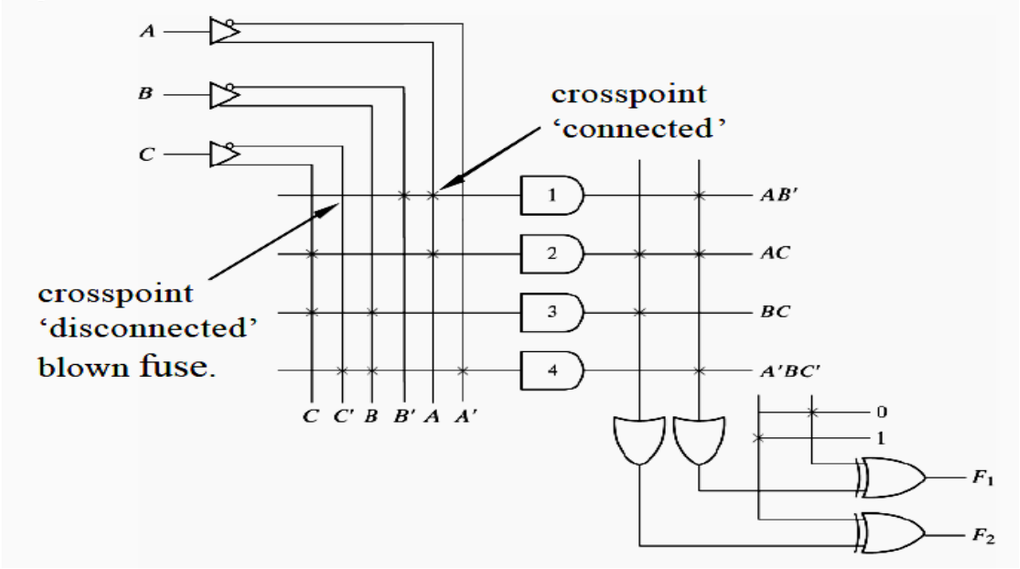


Fig: PLA with 3 Inputs, 4 product terms and 2 Outputs.

Each input goes through a buffer and an inverter as shown in figure with a composite graphic symbol, which has both true and complement outputs. Each input and its complement are connected to the inputs of each AND gate as shown indicated by the intersection between horizontal and vertical lines.

The output of the AND gates are connected to the inputs of each OR gate.

The output of OR gate goes to an X-OR gate where the other input can be programmed to receive a signal equal to either a Logic 1 or Logic 0.

The output is inverted when the XOR input is connected to 1 (since $X \oplus 1 = X^{1}$) & the output does not change when the XOR input is connected to 0 (since $X \oplus 0 = X$).

Example:

Implement the following function with a PLA:

i. $F_1 = \sum m\,(0, 1, 2, 4)$
ii. $F_2 = \sum m\,(0, 5, 6, 7)$

A \ BC	00	01	11	10
0	1	1	0	1
1	1	0	0	0

$F_1 = A'B' + A'C' + B'C'$
$F_1 = (AB + AC + BC)'$

A \ BC	00	01	11	10
0	1	0	0	0
1	0	1	1	1

$F_2 = AB + AC + A'B'C'$
$F_2 = (A'C + A'B + AB'C')'$

PLA programming table

	Product term	Inputs A	B	C	Outputs (C) F_1	(T) F_2
AB	1	1	1	–	1	1
AC	2	1	–	1	1	1
BC	3	–	1	1	1	–
A'B'C'	4	0	0	0	–	1

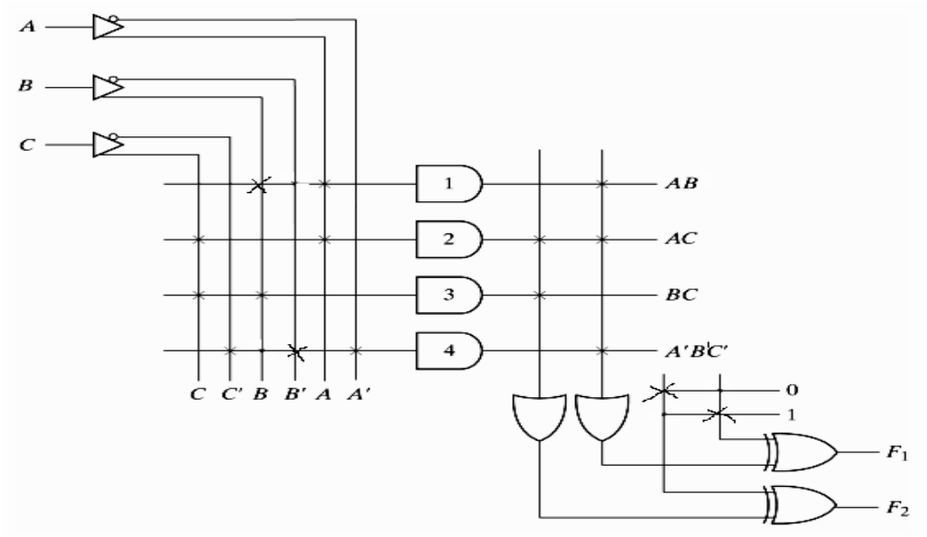


Fig: PLA With 3 inputs, 4 Product terms and 2 outputs.

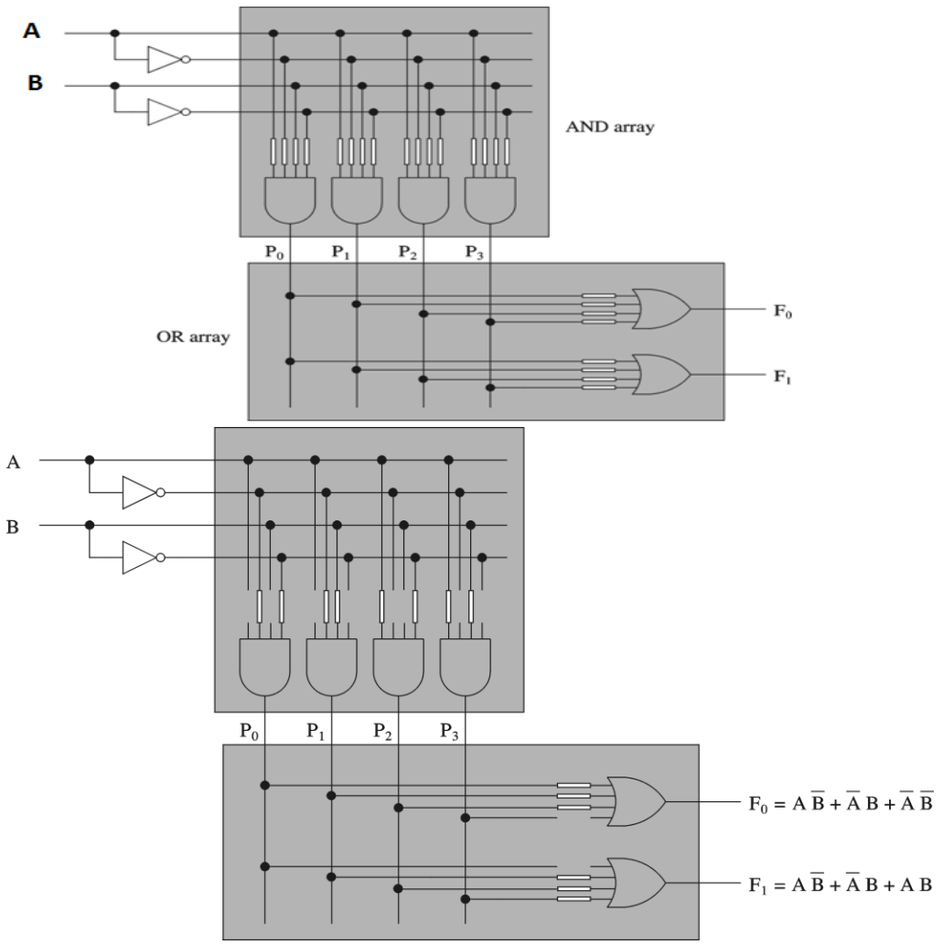
Example:

Implement the following functions using PLA.
$F_0 = AB^1 + A^1 B + A^1 B^1$ &
$F_1 = AB^1 + A^1 B + A B$.

PLA Programming table:

	Product term	Inputs		Outputs	
				T	T
		A	B	F1	F2
AB^1	1	1	0	1	1
$A^1 B$	2	0	1	1	1
$A^1 B^1$	3	0	0	1	-
A B	4	1	1	-	1

A blank PLA with 2 inputs and 2 outputs



Programmable Array Logic (PAL)

The programmable array logic (PAL) is a logic device with fixed OR array and a programmable AND array. It is easier to program but not as flexible as PLA.

Note:

Boolean functions must be simplified to fit into each section. The product term cannot be shared among two or more gates.

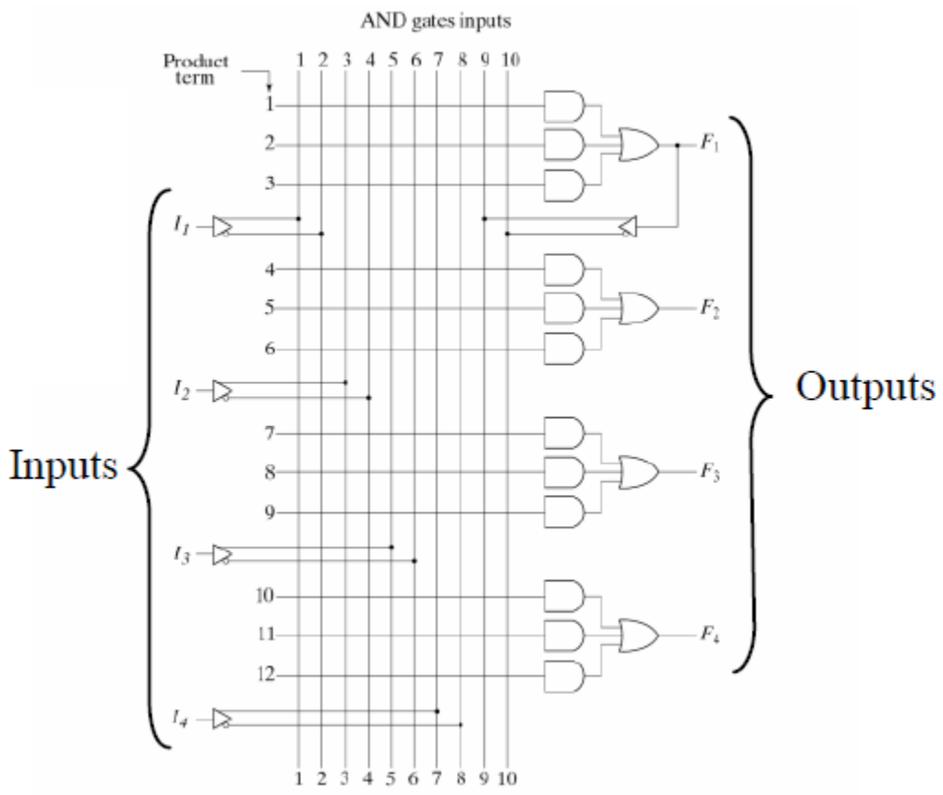


Fig: PAL with 4 inputs, 4 Outputs and 3 wide AND – OR structure.

Example: Implement the following Boolean functions using PAL.

$$w(A, B, C, D) = \Sigma(2, 12, 13)$$
$$x(A, B, C, D) = \Sigma(7, 8, 9, 10, 11, 12, 13, 14, 15)$$
$$y(A, B, C, D) = \Sigma(0, 2, 3, 4, 5, 6, 7, 8, 10, 11, 15)$$
$$z(A, B, C, D) = \Sigma(1, 2, 8, 12, 13)$$

Simplify the above functions using K-Map results as below:

$$w = ABC' + A'B'CD'$$
$$x = A + BCD$$
$$y = A'B + CD + B'D'$$
$$z = ABC' + A'B'CD' + AC'D' + A'B'C'D$$
$$= w + AC'D' + A'B'C'D$$

Product Term	AND Inputs A	B	C	D	W	Outputs
1	1	1	0	–	–	$w = ABC'$
2	0	0	1	0	–	$+ A'B'CD'$
3	–	–	–	–	–	
4	1	–	–	–	–	$x = A$
5	–	1	1	1	–	$+ BCD$
6	–	–	–	–	–	
7	0	1	–	–	–	$y = A'B$
8	–	–	1	1	–	$+ CD$
9	–	0	–	0	–	$+ B'D'$
10	–	–	–	–	1	$z = w$
11	1	–	0	0	–	$+ AC'D'$
12	0	0	0	1	–	$+ A'B'C'D$

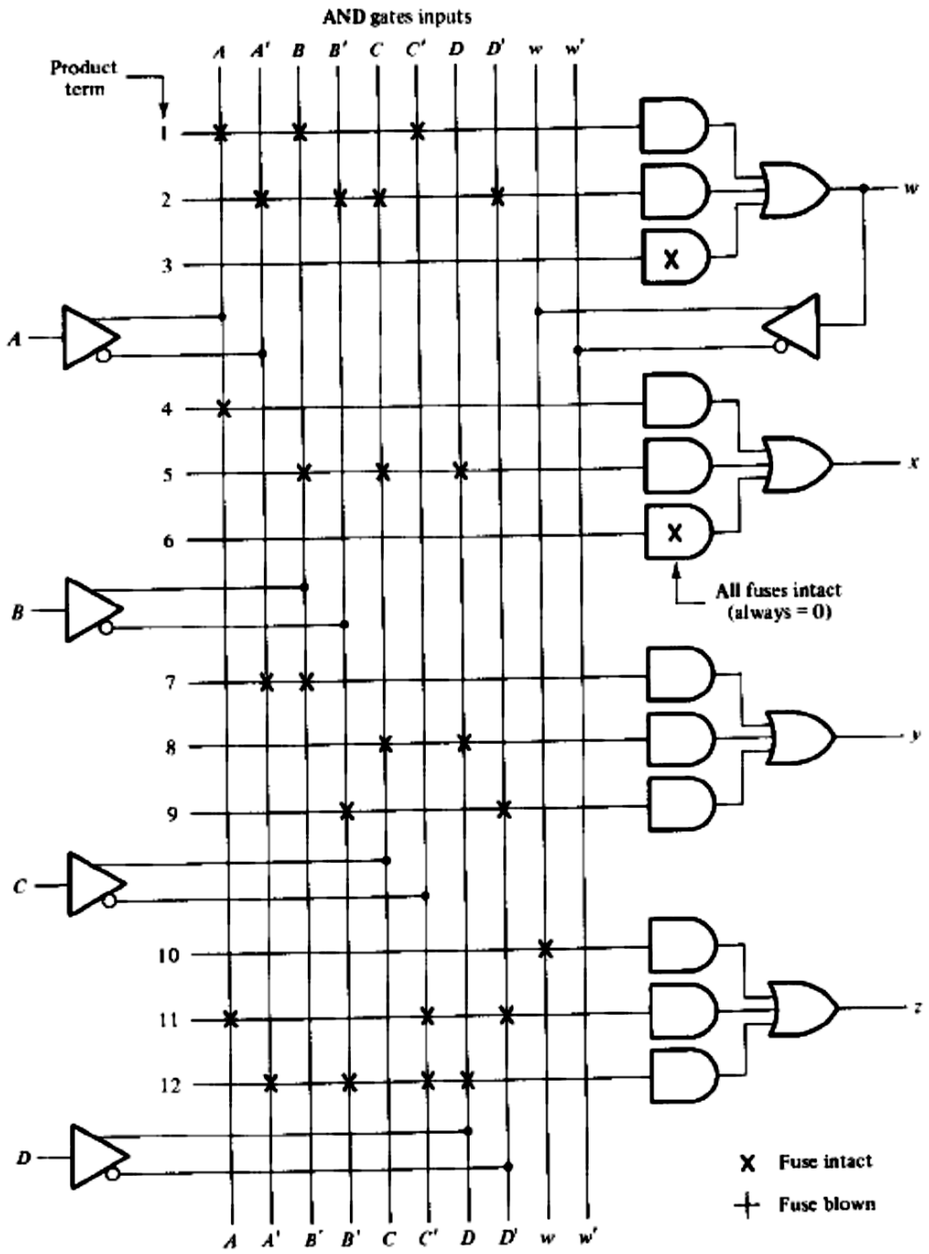


Fig: Fuse map for PAL as shown in programming table.

1. For the 3-input, 4-output truth table of a combinational circuit, tabulate the PAL programming table for the circuit.

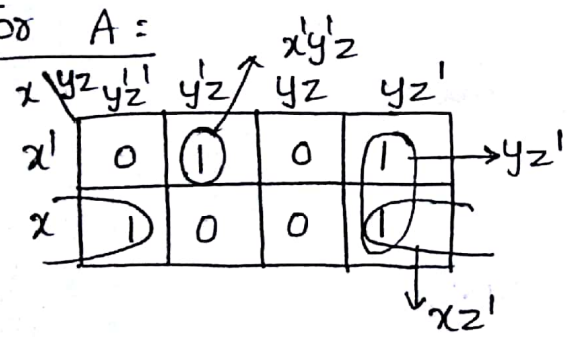
Inputs			outputs			
x	y	z	A	B	C	D
0	0	0	0	1	0	0
0	0	1	1	1	1	1
0	1	0	1	0	1	1
0	1	1	0	1	0	1
1	0	0	1	0	1	0
1	0	1	0	0	0	1
1	1	0	1	1	1	0
1	1	1	0	1	1	1

Sol:

Here, $A = \Sigma m(1,2,4,6)$ $\quad C = \Sigma m(1,2,4,6,7)$

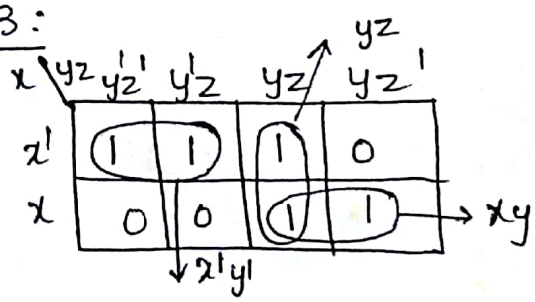
$B = \Sigma m(0,1,3,6,7)$ $\quad D = \Sigma m(1,2,3,5,7)$

For A:



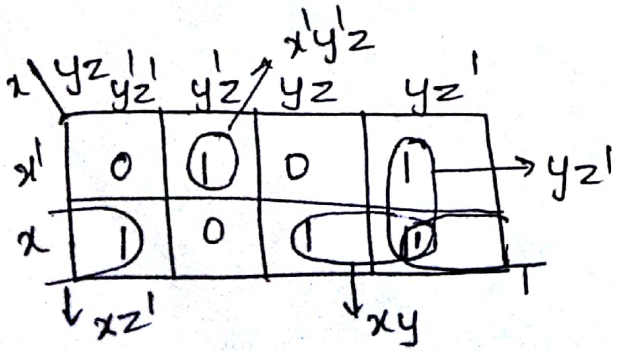
$A = x'y'z + xz' + yz'$

For B:



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

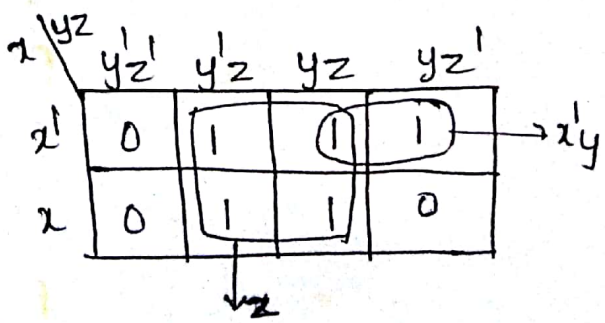
For C:



$C = xz' + x'y'z + xy + yz'$

$= A + xy$

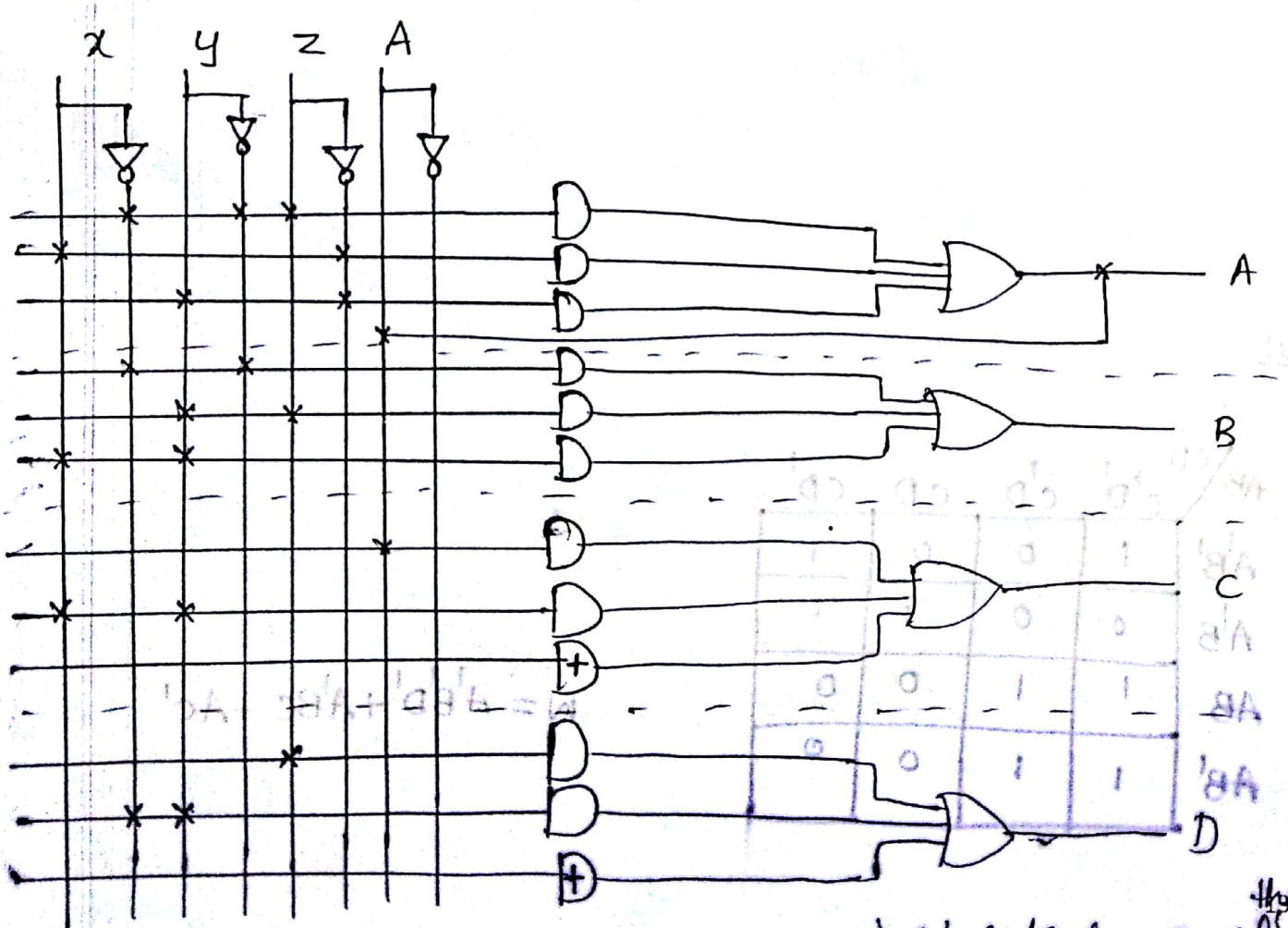
For D:



$D = x'y + z$

Programming Table for PAL :

	Product term	Inputs				Outputs
		x	y	z	A	
$x'y'z$	1	0	0	1	–	$A = x'y'z + xz' + yz'$
xz'	2	1	–	0	–	
yz'	3	–	1	0	–	
$x'y'$	4	0	0	–	–	$B = x'y' + yz + xy$
yz	5	–	1	1	–	
xy	6	1	1	–	–	
A	7	–	–	–	1	$C = A + xy$
xy	8	1	1	–	–	
z	9	–	–	1	–	$D = z + x'y$
$x'y$	10	0	1	–	–	



'+' indicates that 'AND' gate is not used.