

MODULE-2 - Logic Design with MSI Components and Programmable Logic Devices

Module-2

Logic Design with MSI Components and Programmable Logic Devices: Binary Adders and Subtractors, Comparators, Decoders, Encoders, Multiplexers, Programmable Logic Devices (PLDs) (Section 5.1 to 5.7 of Text 2)

Binary Adders and Subtractors

-
- The Binary Adder
-
- Another common and very useful combinational logic circuit is that of the Binary Adder circuit. The Binary Adder is made up from standard AND and Ex-OR gates and allow us to "add" single bits of data together to produce two outputs, the SUM ("S") of the addition and a CARRY ("C"). One of the main uses for the Binary Adder is in arithmetic and counting circuits.
- Consider the addition of two denary (base 10) numbers below.
-

123	A	(Augend)
+ 789	B	(Addend)
912	SUM	

- Each column is added together starting from the right hand side. As each column is added together a carry is generated if the result is greater or equal to ten, the base number. This carry is then added to the result of the addition of the next column to the left and so on, simple school math's addition. Binary addition is based on similar principals but a carry is only generated when the result in any column is greater or equal to "2", the base number of binary.
-

Binary Addition

-
- Binary Addition follows the same basic rules as for the denary addition above except in binary there are only two digits and the largest digit is "1", so any "SUM" greater than 1 will result in a "CARRY". This carry 1 is passed over to the next column for addition and so on. Consider the single bit addition below.
-

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

+ 0	+ 1	+ 0	+ 1
0	1	1	10

- The single bits are added together and "0 + 0", "0 + 1", or "1 + 0" results in a sum of "0" or "1" until you get to "1 + 1" then the sum is equal to "2". For a simple 1-bit addition problem like this, the resulting carry bit could be ignored which would result in an output truth table resembling that of an Ex-OR Gate as seen in the Logic Gates section and whose result is the sum of the two bits but without the carry. An Ex-OR gate only produces an output "1" when either input is at logic "1", but not both. However, all microprocessors and electronic calculators require the carry bit to correctly calculate the equations so we need to rewrite them to include 2 bits of output data as shown below.

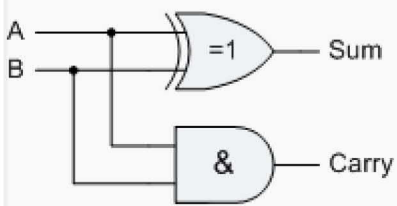
00	00	01	01
+ 00	+ 01	+ 00	+ 01
00	01	01	10

- From the above equations we know that an Ex-OR gate will only produce an output "1" when "EITHER" input is at logic "1", so we need an additional output to produce a carry output, "1" when "BOTH" inputs "A" and "B" are at logic "1" and a standard AND Gate fits the bill nicely. By combining the Ex-OR gate with the AND gate results in a simple digital binary adder circuit known commonly as the "Half-Adder" circuit.

The Half-Adder Circuit

1-bit Adder with Carry-Out

•

Symbol	Truth Table			
	A	B	SUM	CARRY
	0	0	0	0
	0	1	1	0
	1	0	1	0
	1	1	0	1
Boolean Expression: Sum = $A \oplus B$ Carry = $A \cdot B$				

•

- From the truth table we can see that the SUM (S) output is the result of the Ex-OR gate and the Carry-out (CO) is the result of the AND gate. One major disadvantage of the Half-Adder circuit when used as a binary adder, is that there is no provision for a "Carry-in" from the previous circuit when adding together multiple data bits. For example, suppose we want to add together two 8-bit bytes of data, any resulting carry bit would need to be able to "ripple" or move across the bit patterns starting from the least significant bit (LSB). As the Half-Adder

•

•

- has no carry input the resultant added value would be incorrect. One simple way to overcome this problem is to use a "Full-Adder" type binary adder circuit.

•

The Full-Adder Circuit

•

- The main difference between the "Full-Adder" and the previous seen "Half-Adder" is that a Full-Adder has 3-inputs, the two same data inputs "A" and "B" as before plus an additional "Carry-In" (C-in) input as shown below.

•

Full-Adder with Carry-In

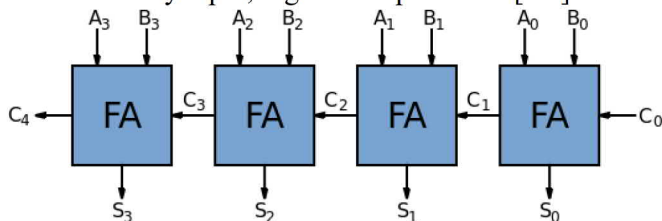
•

Symbol	Truth Table				
	A	B	C-in	Sum	C-out
	0	0	0	0	0
	0	1	0	1	0
	1	0	0	1	0
	1	1	0	0	1
	0	0	1	1	0
	0	1	1	0	1
	1	0	1	0	1
	1	1	1	1	1
	Boolean Expression: $\text{Sum} = A \oplus B \oplus \text{C-in}$				

-
- The Full-Adder circuit above consists of three Ex-OR gates, two AND gates and an OR gate. The truth table for the Full-Adder includes an additional column to take into account the Carry-in input as well as the summed output and Carry-out. 4-bit Full-Adder circuits are available as standard IC packages in the form of the TTL 74LS83 or the 74LS283 which can add together two 4-bit binary numbers and generate a SUM and a CARRY output.

Ripple Carry Adder:

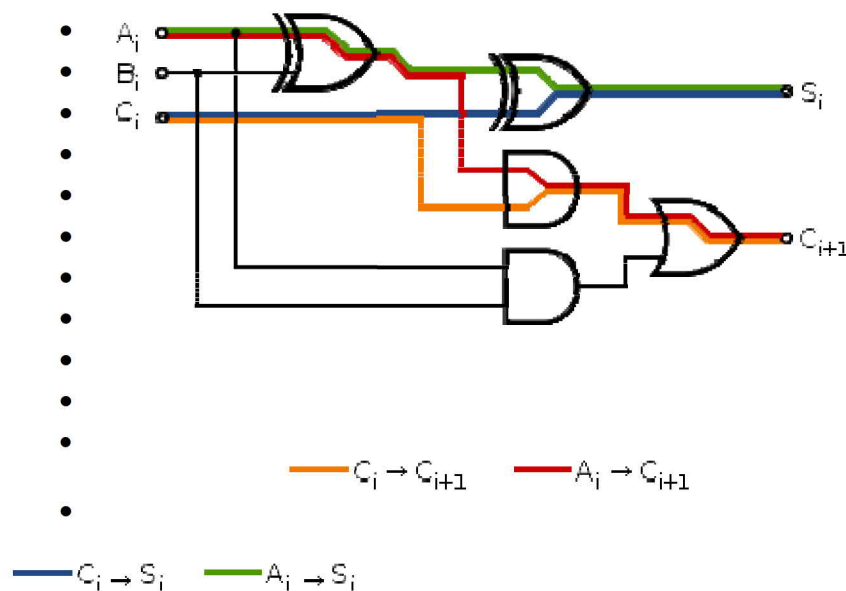
-
- A ripple carry adder is simply several full adders connected in a series so that the carry must propagate through every full adder before the addition is complete. Ripple carry adders require the least amount of hardware of all adders, but they are the slowest.
- The following diagram shows a four-bit adder, which adds the numbers $A[3:0]$ and $B[3:0]$, as well as a carry input, together to produce $S[3:0]$ and the carry output



Propagation Delay in Full Adders

- Real logic gates do not react instantaneously to the inputs, and therefore digital circuits have a maximum speed. Usually, the delay through a digital circuit is measured in gate-delays, as this allows the delay of a design to be calculated for different devices. AND and OR gates have a nominal delay of 1 gate-delay, and XOR gates have a delay of 2, because they are really made up of a combination of ANDs and ORs.
- A full adder block has the following worst case propagation delays:

- From A_i or B_i to C_{i+1} : 4 gate-delays (XOR $\rightarrow$ AND $\rightarrow$ OR)
- From A_i or B_i to S_i : 4 gate-delays (XOR $\rightarrow$ XOR)
- From C_i to C_{i+1} : 2 gate-delays (AND $\rightarrow$ OR)
- From C_i to S_i : 2 gate-delays (XOR)
- Because the carry-out of one stage is the next's input, the worst case propagation delay is then:
- 4 gate-delays from generating the first carry signal ($A_0/B_0 \rightarrow C_1$).
- 2 gate-delays per intermediate stage ($C_i \rightarrow C_{i+1}$).
- 2 gate-delays at the last stage to produce both the sum and
- $C_1 \rightarrow C_n$ and S_{n-1}).
- Carry-out outputs (C_n).
- So for an n -bit adder, we have a total propagation delay, t_p of:



- $$t_p = 4 + 2(n - 2) + 2 = 2n + 2$$
- This is linear in n , and for a 32-bit number, would take 66 cycles to complete the calculation. This is rather slow, and restricts the word length in our device somewhat. We would like to find ways to speed it up.

Carry Look ahead Adder

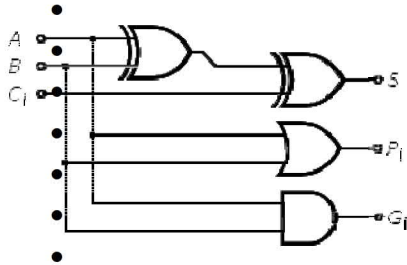
- A fast method of adding numbers is called carry-lookahead. This method doesn't require the carry signal to propagate stage by stage, causing a bottleneck. Instead it uses additional logic to expedite the propagation and generation of carry information, allowing fast addition at the expense of more hardware.
- In a ripple adder, each stage compares the carry-in signal, C_i , with the inputs A_i and B_i and generates a carry-out signal C_{i+1} accordingly. In a carry-lookahead adder, we define two new functions.
- The generate function, G_i , indicates whether that stage causes a carry-out signal C_i to be generated if no carry-in signal exists. This occurs if both the addends contain a 1 in that bit:
- $G_i = A_i \cdot B_i$
-
- The propagate function, P_i , indicates whether a carry-in to the stage is passed to the carry-out for the stage. This occurs if either the addends have a 1 in that bit
- $P_i = A_i + B_i$
-
- Note that both these values can be calculated from the inputs in a constant time of a single gate delay.
- Now, the carry-out from a stage occurs if that stage generates a carry ($G_i = 1$) or there is a carry-in and the stage propagates the carry ($P_i \cdot C_i = 1$)
-
-
- $C_{i+1} = A_i B_i + A_i C_i + B_i C_i$
 $C_{i+1} = A_i B_i + C_i (A_i + B_i)$
 $C_{i+1} = G_i + P_i C_i$

Truth table

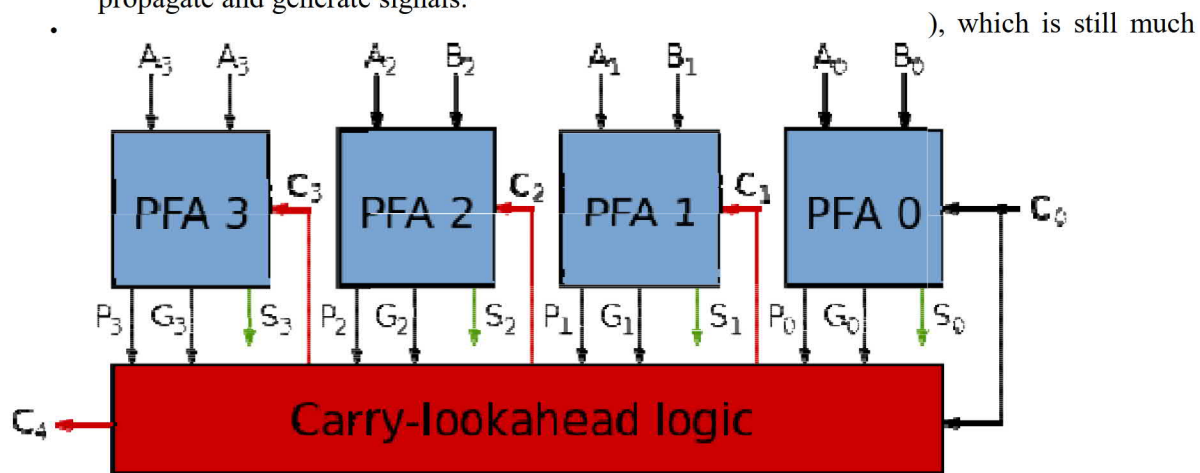
A_i	B_i	C_i	G_i	P_i	C_{i+1}
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	1	0
0	1	1	0	1	1
1	0	0	0	1	0
1	0	1	0	1	1
1	1	0	1	1	1
1	1	1	1	1	1

- $C_{i+1} = G_i + P_i C_i$
 $C_{i+1} = G_i + P_i (G_{i-1} + P_{i-1} C_{i-1})$
 $C_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} (G_{i-2} + P_{i-2} C_{i-2})$
 $C_{i+1} = \vdots$
 $C_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + P_i P_{i-1} P_{i-2} G_{i-3} + \dots + P_i P_{i-1} \dots P_1 P_0 C_0$
-
-

- Note that this does not require the carry-out signals from the previous stages, so we don't have to wait for changes to ripple through the circuit. In fact, a given stage's carry signal can be computed once the propagate and generate signals are ready with only two more gate delays (one AND and one OR). Thus the carry-out for a given stage can be calculated in constant time, and therefore so can the sum.
- The S , P , and G signals are all generated by a circuit called a "partial full adder" (PFA), which is similar to a full adder.



- For a slightly smaller circuit, the propagate signal can be taken as the output of the first XOR gate instead of using a dedicated OR gate, because if both A and B are asserted, the generate signal will force a carry. However, this simplification means that the propagate signal will take two gate delays to produce, rather than just one.
- A carry lookahead adder then contains n PFAs and the logic to produce carries from the stage propagate and generate signals:



- Two numbers can therefore be added in constant time, $O(1)$, of just 6 gate delays, regardless of the length, n of the numbers. However, this requires AND and OR gates with up to n inputs. If logic gates are available with a limited number of inputs, trees will need to be constructed to compute these, and the overall computation time better than the linear time for ripple adders.

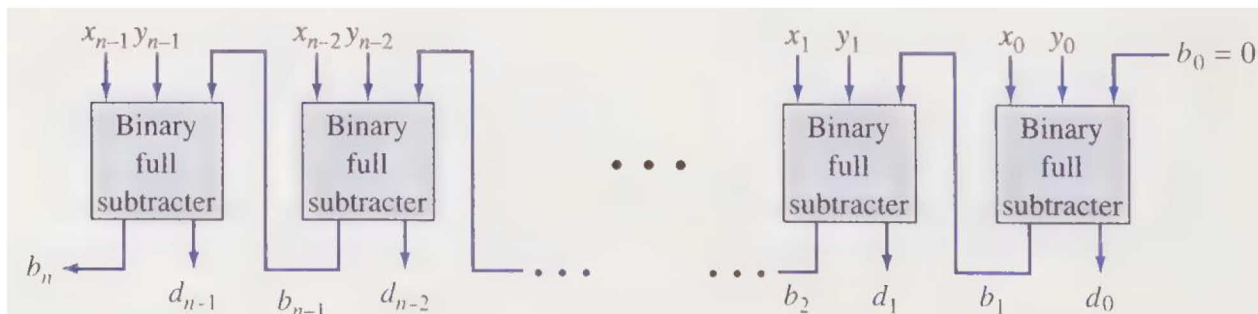
Binary Subtractors

- A binary subtractor can be designed using the same approach as that for a binary adder. The binary subtraction process is summarized in Table.

x_i	y_i	b_i	b_{i+1}	d_i
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

- Again, in general, three bits are involved at each bit order, a minuend bit, x_i a subtrahend bit, y_i and a borrow-in bit from the previous bit-order position, b_i . The result of the subtraction is a difference bit, d_i and a borrow-out bit, b_{i+1} . The difference bit at each order is obtained by subtracting both the subtrahend and borrow-in bits from the minuend bit. To achieve this result, however, a borrow-out from the next higher-order bit position may be necessary.
- From the Truth-table, it may be summarized that:
 - $d_i = x_i \oplus (y_i \oplus b_i)$
 - $b_{i+1} = \bar{x}_i y_i + \bar{x}_i b_i + y_i b_i$

As was done for addition, by cascading n binary full subtractors, a ripple binary subtractor is realized for handling two n -bit operands. The structure of such a realization is shown in Fig, where x_n is the n -bit minuend and y_n is the n -bit subtrahend.



Parallel binary subtractor

Subtraction can be replaced by addition through the use of complements. For example, adding the 2's-complement of the subtrahend to the minuend results in the difference between the two numbers.

The 2's-complement of a binary number is readily obtained by adding one to its 1's-complement. Figure 5.5 shows the design of a subtractor using this approach. The 1's-complement of the subtrahend is formed

by inverting each of its bits, and a carry-in of 1 in the least-significant-bit position provides for the addition of 1 to the 1's-complement.

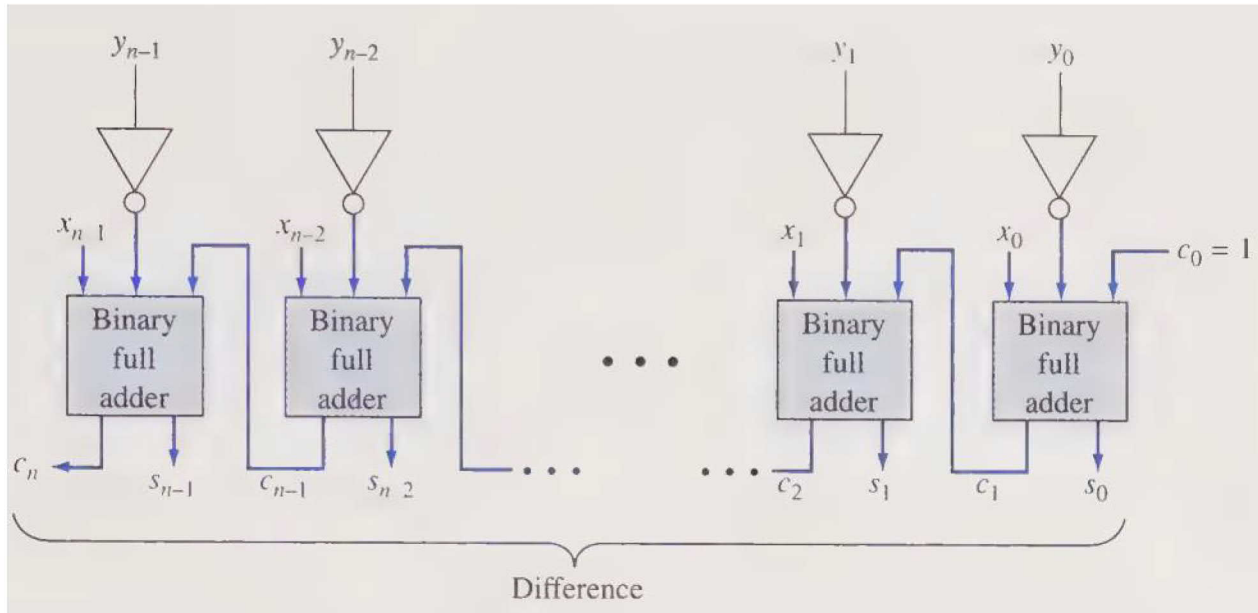
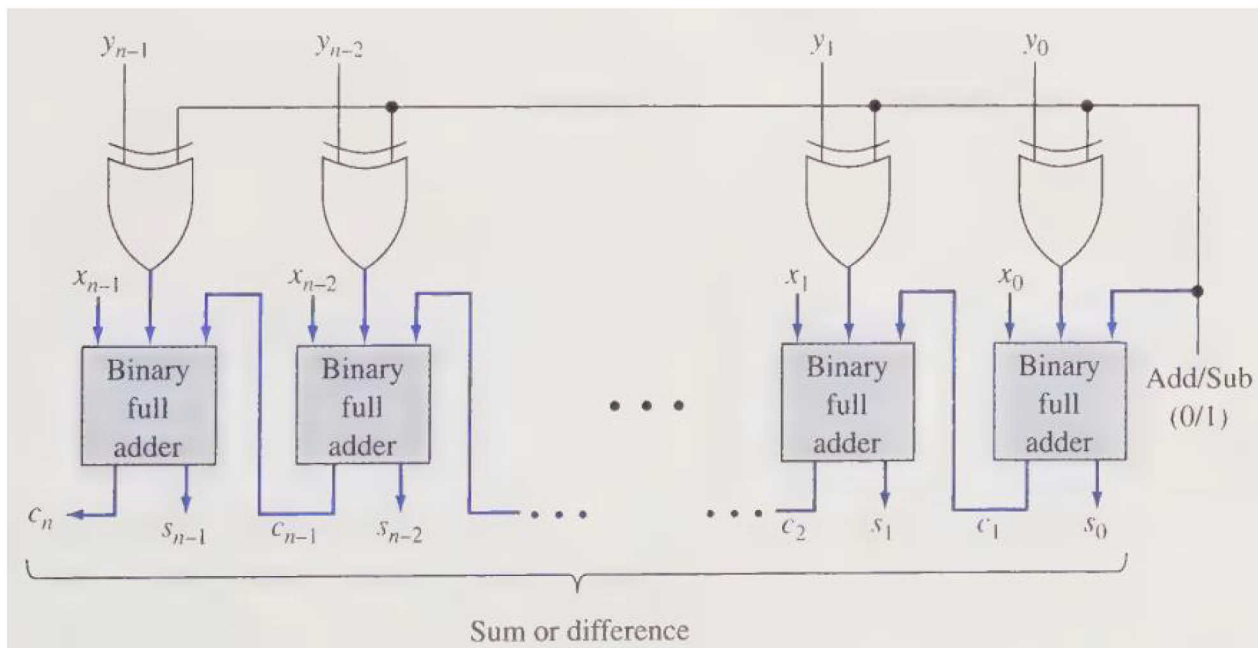


Figure Parallel binary subtracter constructed using a parallel binary adder.

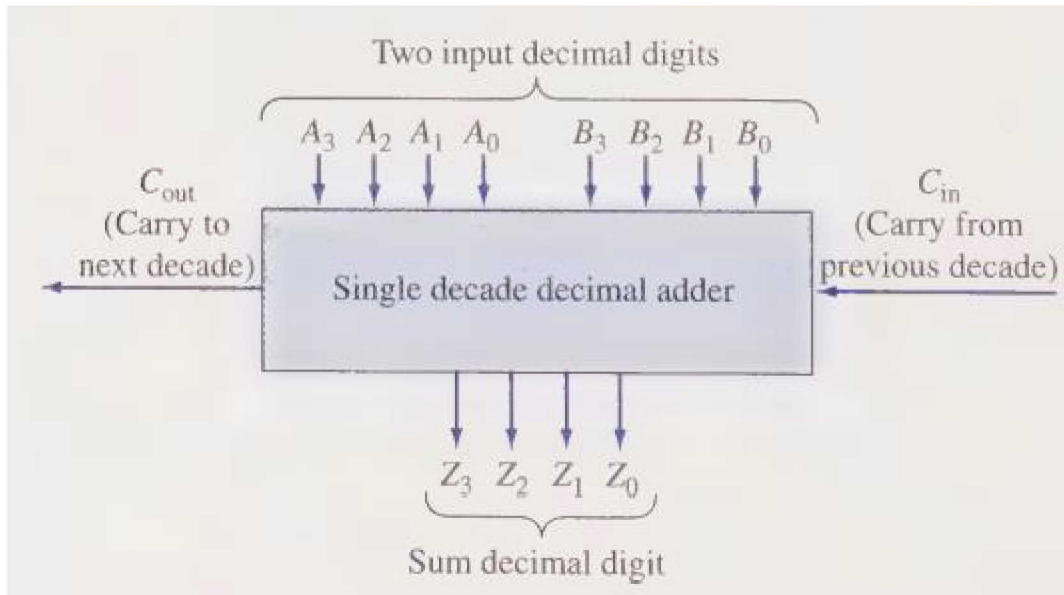
By making use of the fact that $y_i \oplus 1 = \bar{y}_i$, Fig. gives a realization of a parallel adder/subtractor. The behavior of this network is determined by the control signal Add/Sub. The subtraction operation is obtained by letting Add/Sub = 1, in which case 1's are appropriately entered into the exclusive-or-gates to provide the 1's-complement of the subtrahend and the initial carry-in of 1.



The parallel binary adder then produces the difference. On the other hand, when the two operands are to be added, Add/Sub = 0. The bits of the addend are not modified by the exclusive-or-gates prior to entering the parallel binary adder and the necessary initial carry-in of 0 is provided.

Decimal Adders

The general form of a single-decade decimal adder, i.e., an adder corresponding to a single-order digit position, is given in Fig.



In this figure it is assumed that a 4-bit code is used for each decimal digit. Here the two decimal digits serving as operands are denoted by $A_3A_2A_1A_0$ and $B_3B_2B_1B_0$. A carry, denoted by C_{in} , also appears as an input from the addition of the previous decade. The outputs from the adder are a single sum digit, $Z_3Z_2Z_1Z_0$, and a carry, C_{out} . Although the signal values denoting the decimal digits depend upon the code being used, the carries, C_{in} , and C_{out} , are only 0 or 1. Thus, only a single line is shown in the figure for each of them.

BCD Adders

The 8421 weighted coding scheme is the most commonly occurring in digital systems and is frequently referred to as simply BCD for binary-coded decimal. When using BCD, a single-decade decimal adder can be constructed by first performing conventional binary addition on the two binary-coded operands and then applying a corrective procedure. This approach is illustrated in Fig.

The code groups for the two decimal digits are added using a 4-bit binary adder as discussed in the previous section to produce intermediate results $KP_3P_2P_1P_0$. These results are then modified so as to obtain the appropriate output carry and code group for the sum digit, i.e., $C_{out}Z_3Z_2Z_1Z_0$. Since each operand digit has a decimal value from 0 to 9 along with the fact that a carry from a previous digit position is at most 1, the decimal sum at each digit position must be in the range from 0 to 19. Table summarizes the various outputs from the 4-bit binary adder and the required outputs from the single-decade decimal adder.

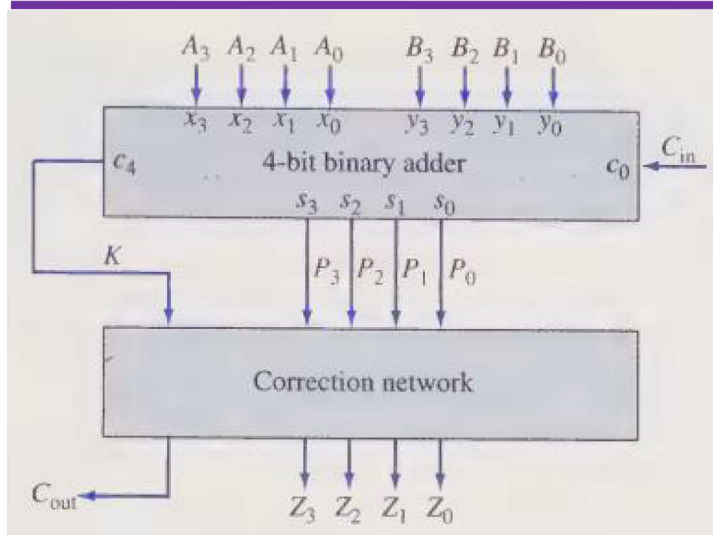


Figure Organization of a single-decade BCD adder.

Table Comparing binary and BCD sums

Decimal sum	Binary sum					Required BCD sum				
	K	P_3	P_2	P_1	P_0	C_{out}	Z_3	Z_2	Z_1	Z_0
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	1	0	0	0	0	1
2	0	0	0	1	0	0	0	0	1	0
3	0	0	0	1	1	0	0	0	1	1
4	0	0	1	0	0	0	0	1	0	0
5	0	0	1	0	1	0	0	1	0	1
6	0	0	1	1	0	0	0	1	1	0
7	0	0	1	1	1	0	0	1	1	1
8	0	1	0	0	0	0	1	0	0	0
9	0	1	0	0	1	0	1	0	0	1
10	0	1	0	1	0	1	0	0	0	0
11	0	1	0	1	1	1	0	0	0	1
12	0	1	1	0	0	1	0	0	1	0
13	0	1	1	0	1	1	0	0	1	1
14	0	1	1	1	0	1	0	1	0	0
15	0	1	1	1	1	1	0	1	0	1
16	1	0	0	0	0	1	0	1	1	0
17	1	0	0	0	1	1	0	1	1	1
18	1	0	0	1	0	1	1	0	0	0
19	1	0	0	1	1	1	1	0	0	1

From the Truthtable it may be noticed that the addition of a decimal 6, i.e., binary 0110, to the output from the 4-bit binary adder is the necessary correction whenever the carry bit K is 1.

The K-map to implement this using Pi bits is as shown:

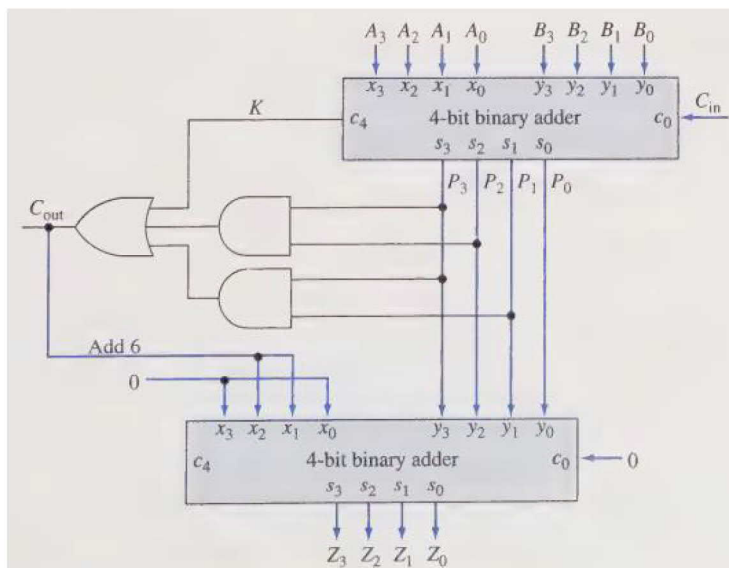
	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	1	1	1	1
10	0	0	1	1

The resulting expression for the circuit is as follows:

$$Add\ 6 = K + P_3P_2 + P_3P_1$$

The first term corresponds to the situation when $10000 \leq KP_3P_2P_1P_0 \leq 10011$ i.e., whenever the carry bit K is 1, and the remaining two terms correspond to the situation when whenever the code group for the sum digit is invalid.

Figure shows the logic diagram of a single-decade BCD adder.



In this diagram whenever $C_{out} = 0$, the outputs from the upper 4-bit binary adder are sent to the lower 4-bit binary adder and the decimal quantity of zero is added to it, which results in no corrective action.

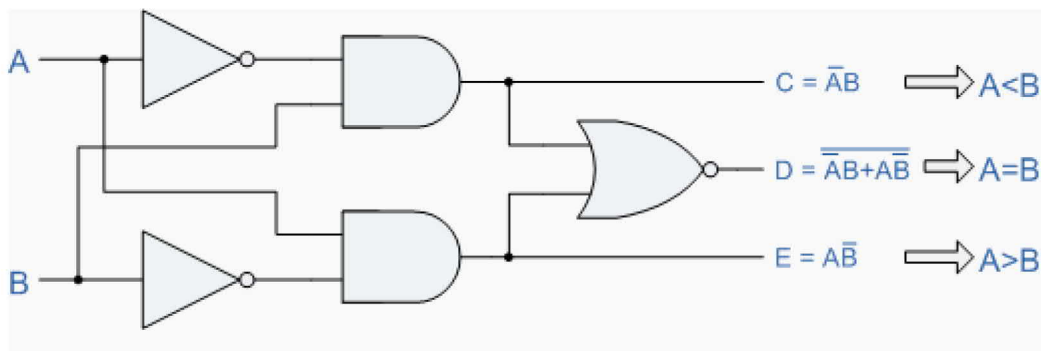
Comparators

- Another common and very useful combinational logic circuit is that of the Digital Comparator circuit. Digital or Binary Comparators are made up from standard AND, NOR and NOT gates that compare the digital signals at their input terminals and produces an output depending upon the condition of the inputs. For example, whether input A is greater than, smaller than or equal to input B etc.
- Digital Comparators can compare a variable or unknown number for example A (A1, A2, A3, ... An, etc) against that of a constant or known value such as B (B1, B2, B3, ... Bn, etc) and produce an output depending upon the result. For example, a comparator of 1-bit, (A and B) would produce the following three output conditions.
- This is useful if we want to compare two values and produce an output when the condition is

$$A > B, A = B, A < B$$

achieved. For example, produce an output from a counter when a certain count number is reached. Consider the simple 1-bit comparator below.

1-bit Comparator



- Then the operation of a 1-bit digital comparator is given in the following Truth Table.

Truth Table

Inputs		Outputs		
B	A	A > B	A = B	A < B
0	0	0	1	0
0	1	1	0	0
1	0	0	0	1
1	1	0	1	0

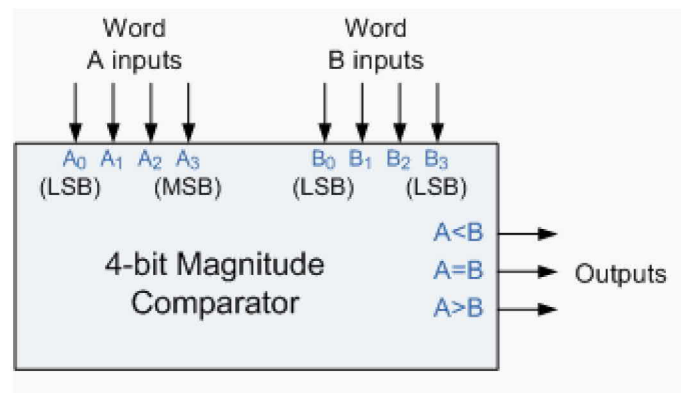
- You may notice two distinct features about the comparator from the above truth table. Firstly, the circuit does not distinguish between either two "0" or two "1"s as an output A = B is produced when they are both equal, either A = B = "0" or A = B = "1". Secondly, the output condition for A = B resembles that of a commonly available logic gate, the Exclusive-NOR or Ex-NOR gate giving $Q = A \oplus B$

-
- Digital comparators actually use Exclusive-NOR gates within their design for comparing the respective pairs of bits in each of the two words with single bit comparators cascaded together to produce Multi-bit comparators so that larger words can be compared.
-

Magnitude Comparators

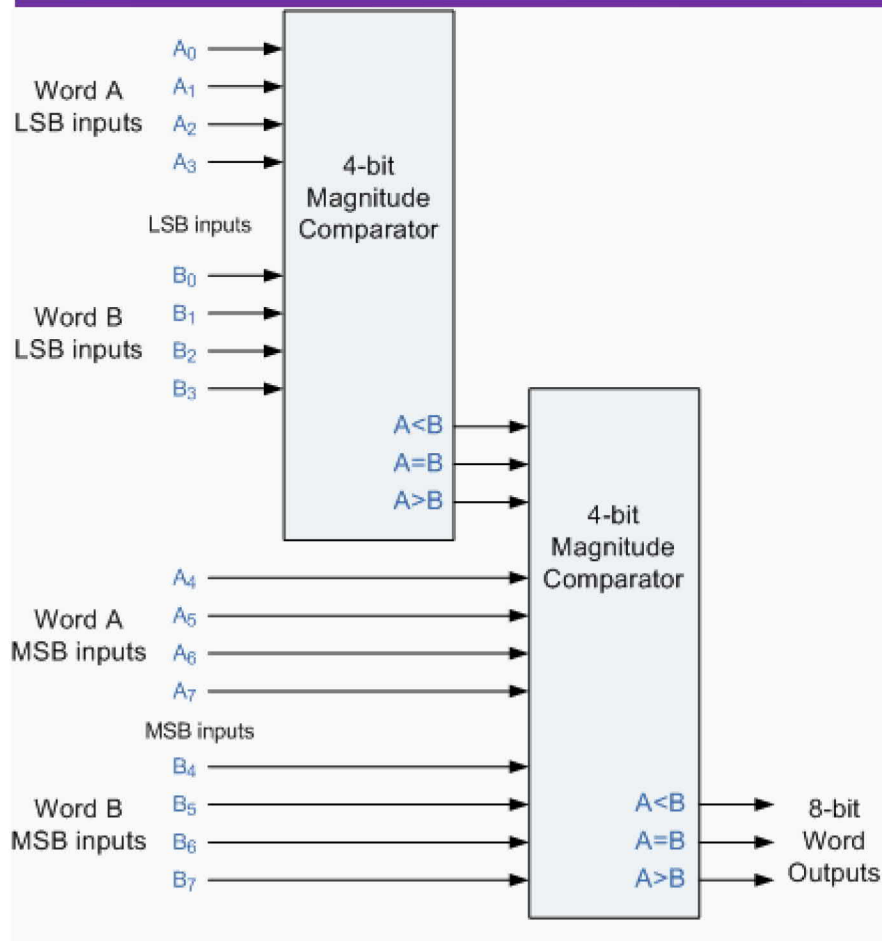
- As well as comparing individual bits, multi-bit comparators can be constructed to compare whole binary or BCD words to produce an output if one word is larger, equal to or less than the other. A very good example of this is the 4-bit Magnitude Comparator. Here, two 4-bit words ("nibbles") are compared to produce the relevant output with one word connected to inputs A and the other to be compared against connected to input B as shown below.

4-bit Magnitude Comparator



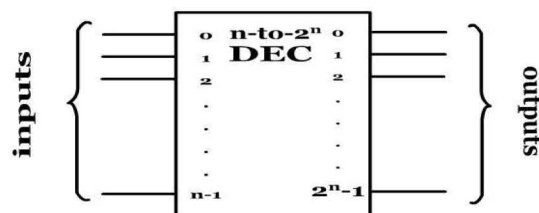
- Some commercially available Magnitude Comparators such as the 7485 have additional input terminals that allow more individual comparators to be "cascaded" together to compare words larger than 4-bits with magnitude comparators of "n"-bits being produced. These cascading inputs are connected directly to the corresponding outputs of the previous comparator as shown to compare 8, 16 or even 32-bit words.
-
-
-
-
-
-

8-bit Word Comparator



Decoders

- A Decoder is a multiple input, multiple output logic circuit. The block diagram of a decoder is shown below.
-



An n -to- 2^n line decoder symbol.

-

The most commonly used decoder is a n –to 2^n decoder which has n inputs and 2^n Output lines .

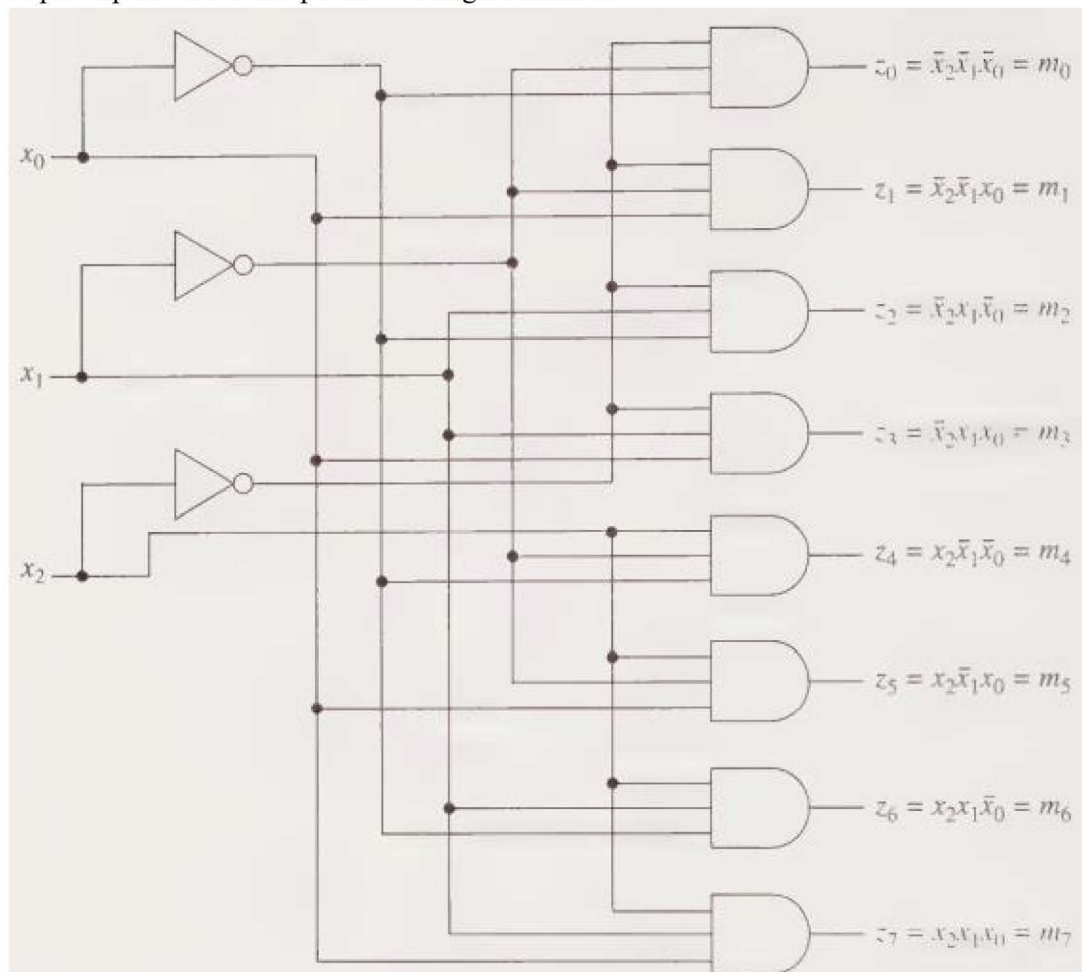
The n -to- 2^n -line decoder is only one of several types of decoders. Functionspecific decoders exist having fewer than 2^n outputs. For example, a decoder having 4 inputs and 10 outputs in which a single

responding output line corresponds to a combination of the 8421 code is referred to as a BCD-to-decimal decoder.

There are also function-specific decoders in which more than one output line responds to a given input combination. For example, there is a four-input-line, seven-output-line decoder that accepts the 4 bits of the 8421 code and is used to drive a seven segment display. However, the n-to-2ⁿ-line decoders are more flexible than the function-specific decoders.

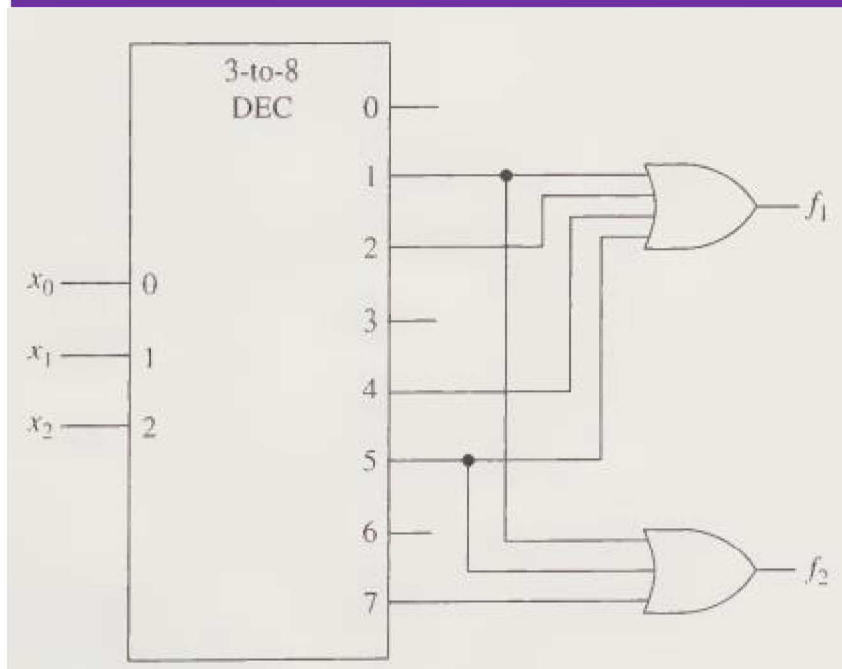
Logic Design using Decoders

In Fig. the Boolean expressions describing the outputs of the decoder are also written. Each of these output expressions corresponds to a single minterm.



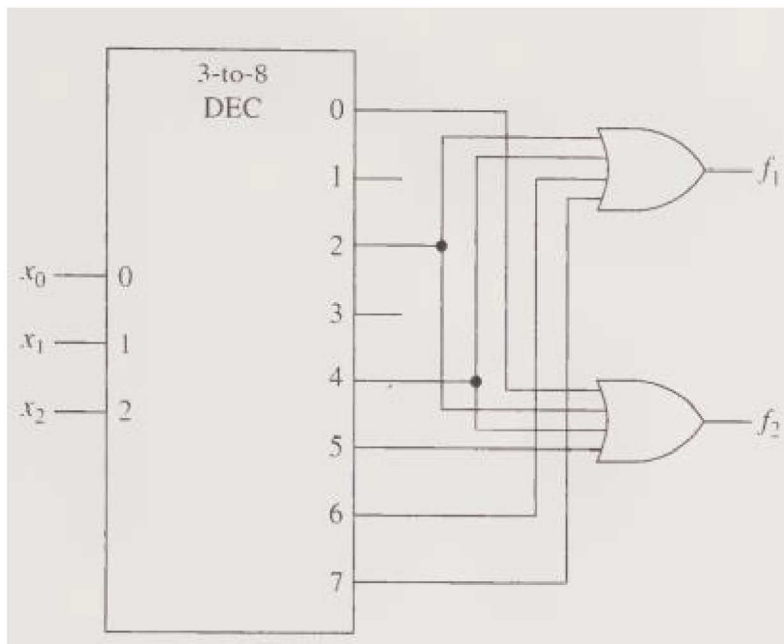
Hence, an n-to-2ⁿ-line decoder is a minterm generator. Recall that any Boolean function is describable by a sum of minterms. Thus, by using or-gates in conjunction with an n-to-2ⁿ-line decoder, realizations of Boolean functions are possible.

Example: Realization of the Boolean expressions $f_1(x_2, x_1, x_0) = \sum m(1,2,4,5)$ and $f_2(x_2, x_1, x_0) = \sum m(1,5,7)$ with a 3-to-8-line decoder and two or-gates.

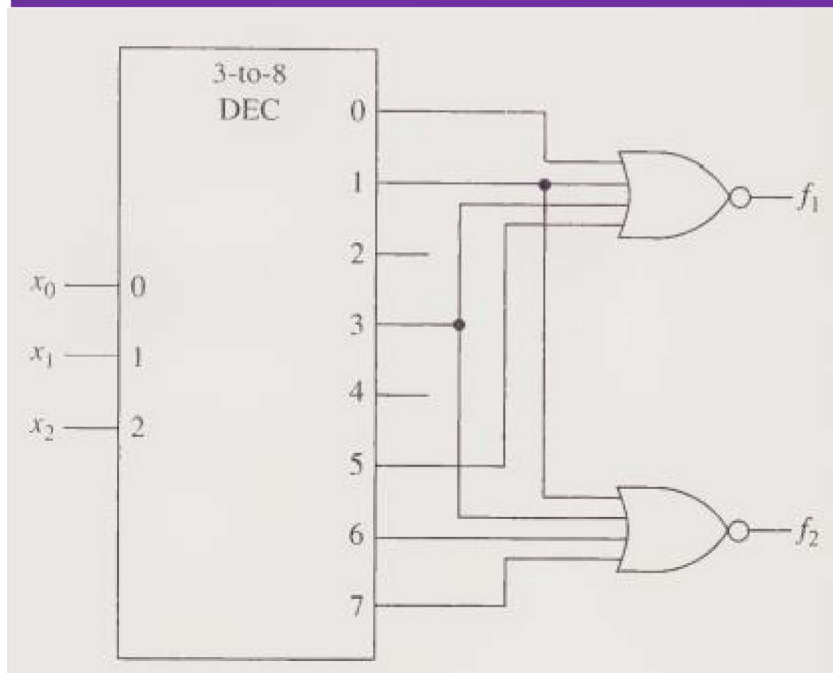


Example: A decoder realization of $f_1(x_2, x_1, x_0) = \prod M(0,1,3,5)$ and $f_2(x_2, x_1, x_0) = \prod M(1,3,6,5)$

(a) Using output or-gates.

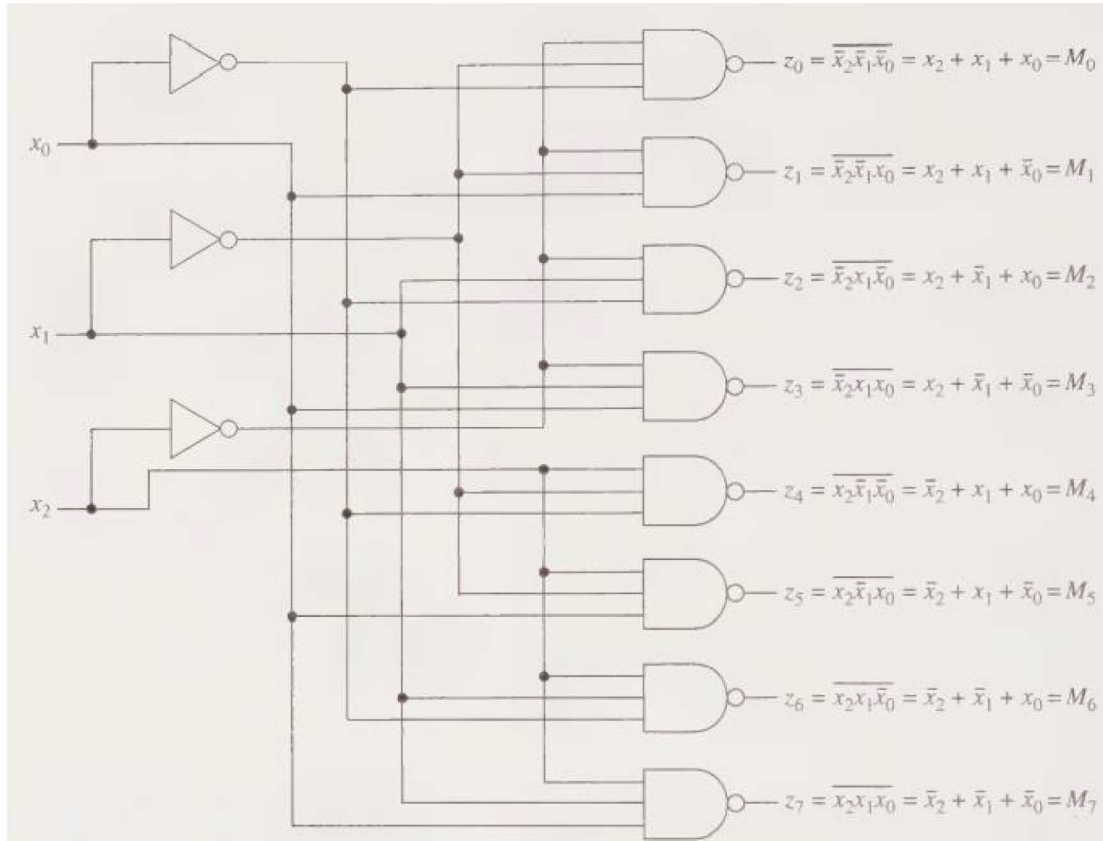


(b) Using output nor-gates.

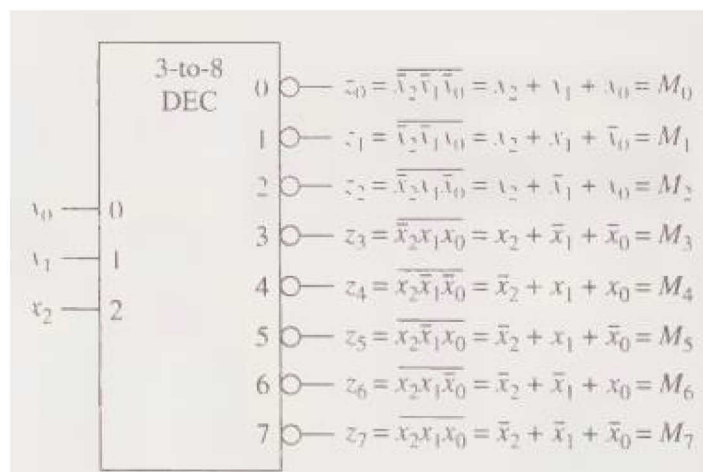


Decoders with Complemented Outputs

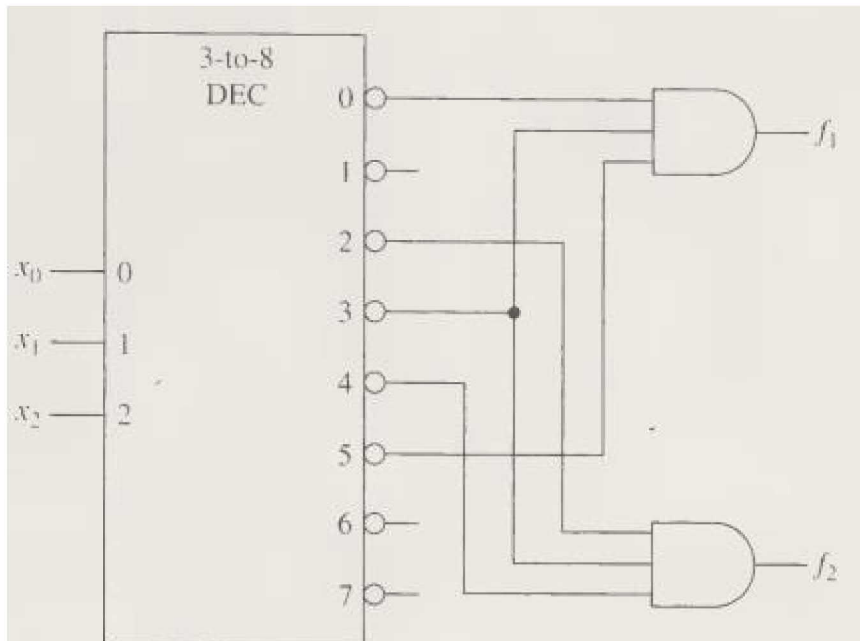
Example: A 3-to-8-line decoder using nand-gates.



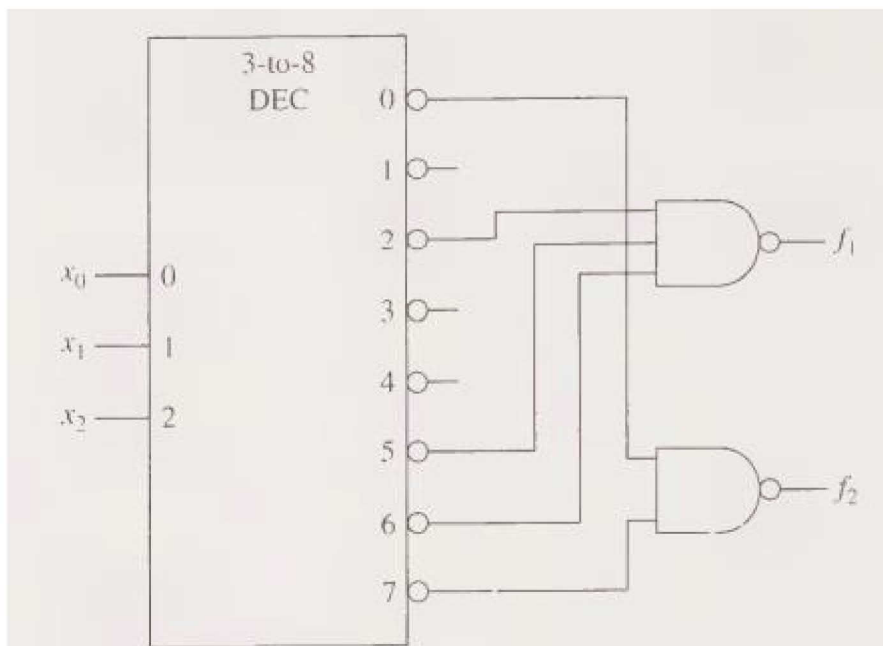
Inputs			Outputs							
x_2	x_1	x_0	z_0	z_1	z_2	z_3	z_4	z_5	z_6	z_7
0	0	0	0	1	1	1	1	1	1	1
0	0	1	1	0	1	1	1	1	1	1
0	1	0	1	1	0	1	1	1	1	1
0	1	1	1	1	1	0	1	1	1	1
1	0	0	1	1	1	1	0	1	1	1
1	0	1	1	1	1	1	1	0	1	1
1	1	0	1	1	1	1	1	1	0	1
1	1	1	1	1	1	1	1	1	1	0



Example: Realization of the pair of maxterm canonical expressions $f_1(x_2, x_1, x_0) = \prod M(0,3,5)$ and $f_1(x_2, x_1, x_0) = \prod M(2,3,4)$ with a 3-to-8-line decoder and two and-gates.

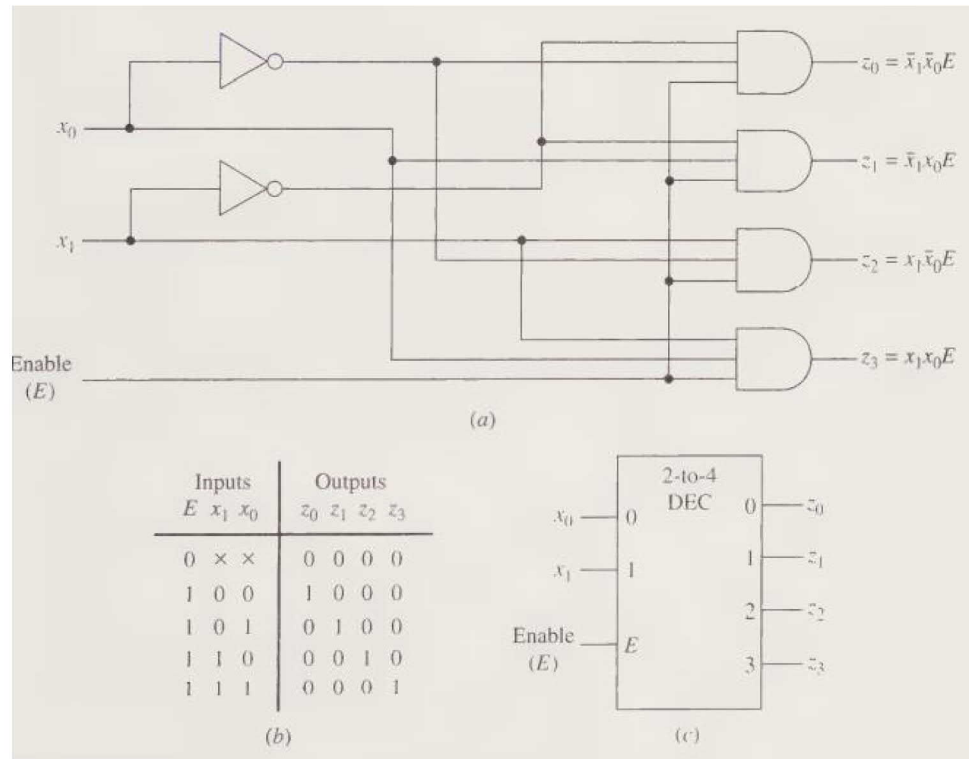


Example: Realization of the pair of maxterm canonical expressions $f_1(x_2, x_1, x_0) = \prod M(0,1,3,4,7) = \overline{\prod M(2,5,6)}$ and $f_2(x_2, x_1, x_0) = \prod M(1,2,3,4,5,6) = \overline{\prod M(0,7)}$ with a 3-to-8-line decoder and two and-gates.

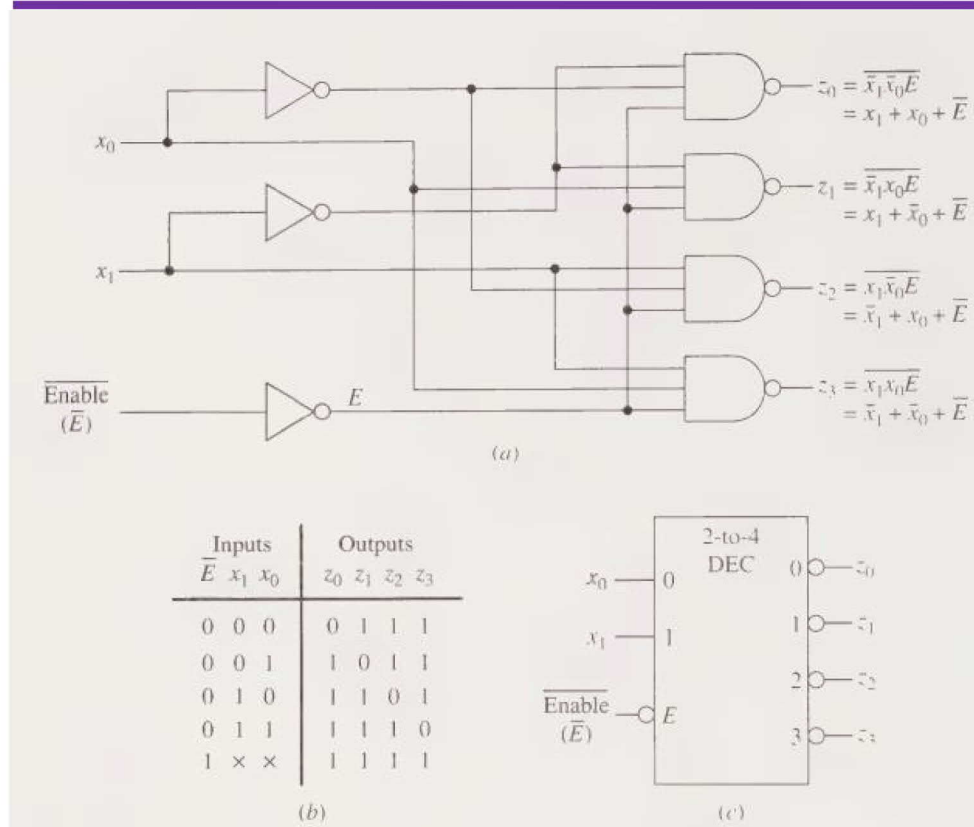


Decoders with enable input

Example of decoder with enable

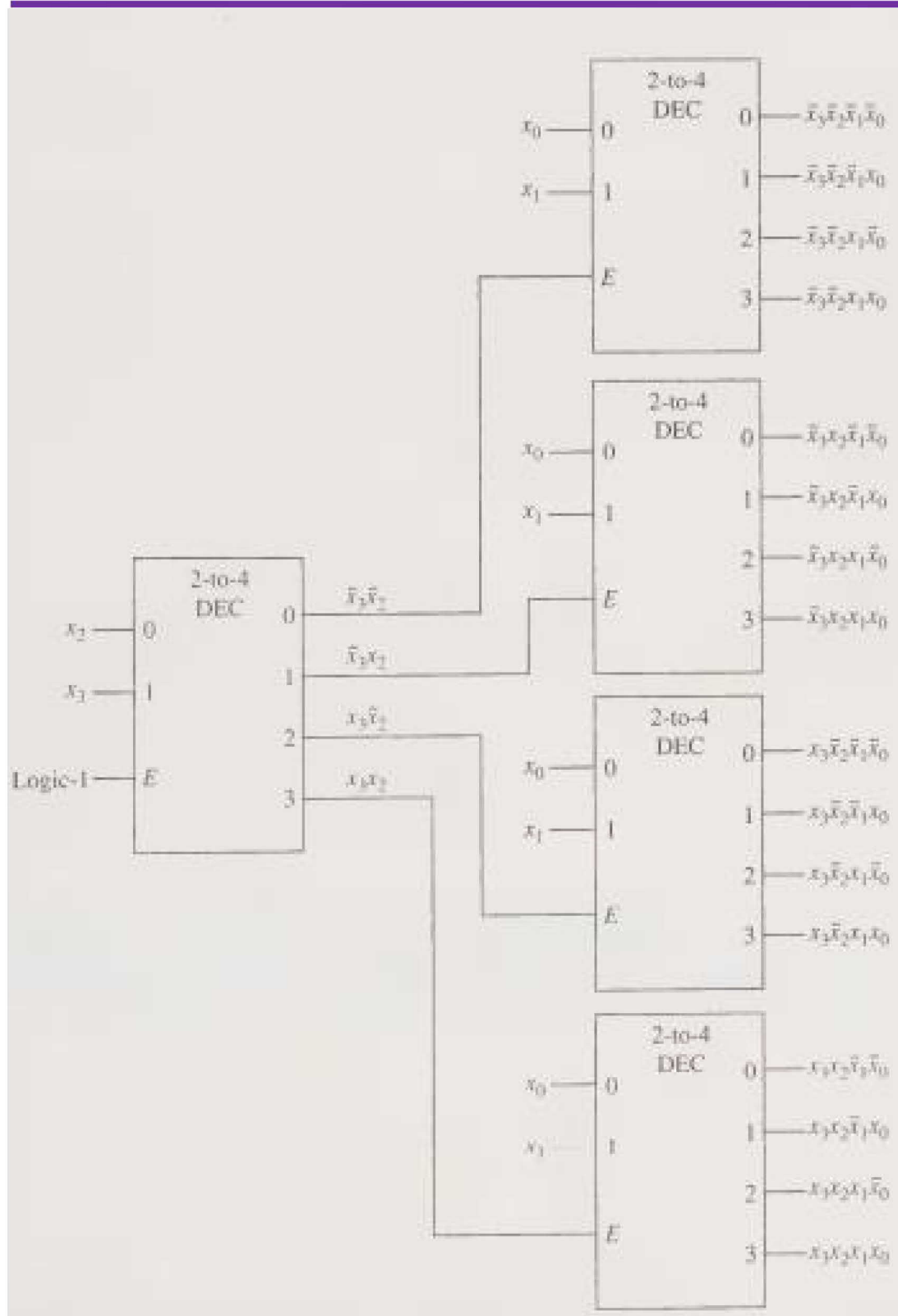


Example of decoder with active low enable



Larger Decoders with Smaller Decoders

Example: A 4-to-16-line decoder constructed from 2-to-4-line decoders.



Encoders

Encoders provide for the conversion of binary information from one form to another. Encoders are essentially the inverse of decoders. Decoders that have more input lines than output lines are usually called encoders.

The simplest encoder is the 2^n -to- n -line encoder in which an assertive logic value, say, logic-1, on one of its 2^n input lines causes the corresponding binary code to appear at the output lines. If it is assumed that at most one input line is asserted at any time, then the 2^n -to- n -line encoder is simply a collection of or-gates.

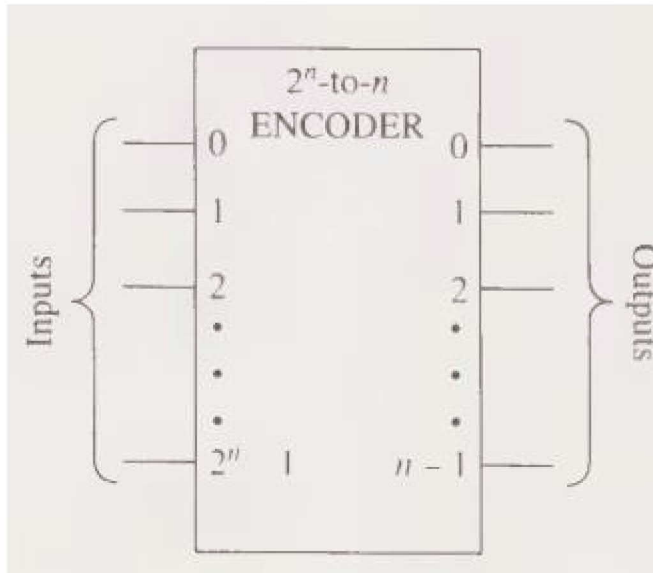
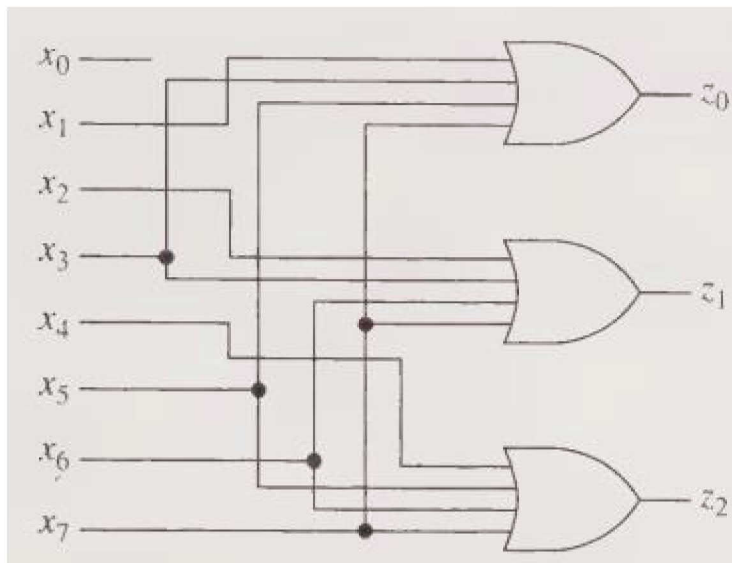


Figure shows a 2^n -to- n -line encoder symbol

Fig. 5.31 shows the logic diagram for an 8-to-3-line encoder.



The equations for the three outputs of Multiplexers

$$z_0 = x_1 + x_3 + x_5 + x_7$$

$$z_1 = x_2 + x_3 + x_6 + x_7$$

$$z_2 = x_4 + x_5 + x_6 + x_7$$

Priority Encoders

In a priority encoder, a priority scheme is assigned to the input lines so that whenever more than one input line is asserted at any time, the output is determined by the input line having the highest priority.

For example, Table 5.5 is a condensed truth table specifying the behavior of a priority encoder where the output is determined by the asserted input having the highest index

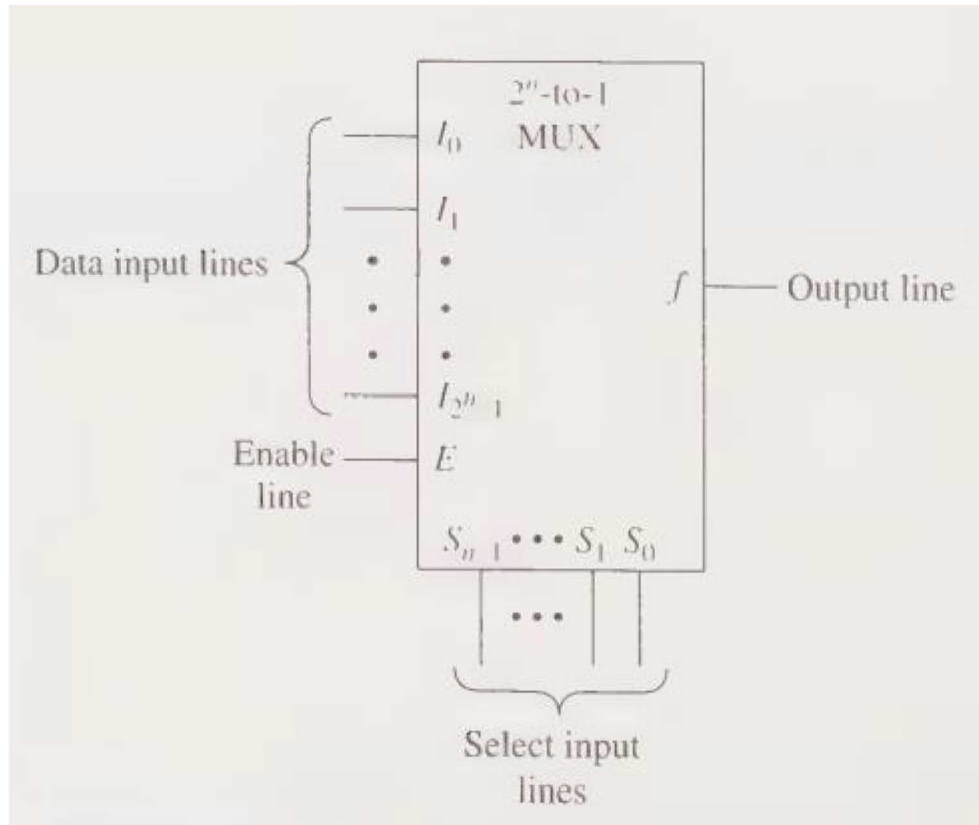
Inputs								Outputs			
x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	z_2	z_1	z_0	Valid
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1
×	1	0	0	0	0	0	0	0	0	1	1
×	×	1	0	0	0	0	0	0	1	0	1
×	×	×	1	0	0	0	0	0	1	1	1
×	×	×	×	1	0	0	0	1	0	0	1
×	×	×	×	×	1	0	0	1	0	1	1
×	×	×	×	×	×	1	0	1	1	0	1
×	×	×	×	×	×	×	1	1	1	1	1

if $x_4 = x_5 = x_6 = x_7 = 0$ and $x_3 = 1$, then $z_2z_1z_0 = 011$ regardless of the values of the x_0 , x_1 , and x_2 , inputs.

An output is also included in Table 5.5, labeled valid, to indicate that at least one input line is asserted. This is done so as to distinguish the situation that no input line is asserted from when the x_i input line is asserted, since in both cases $z_2z_1z_0 = 000$.

Multiplexers

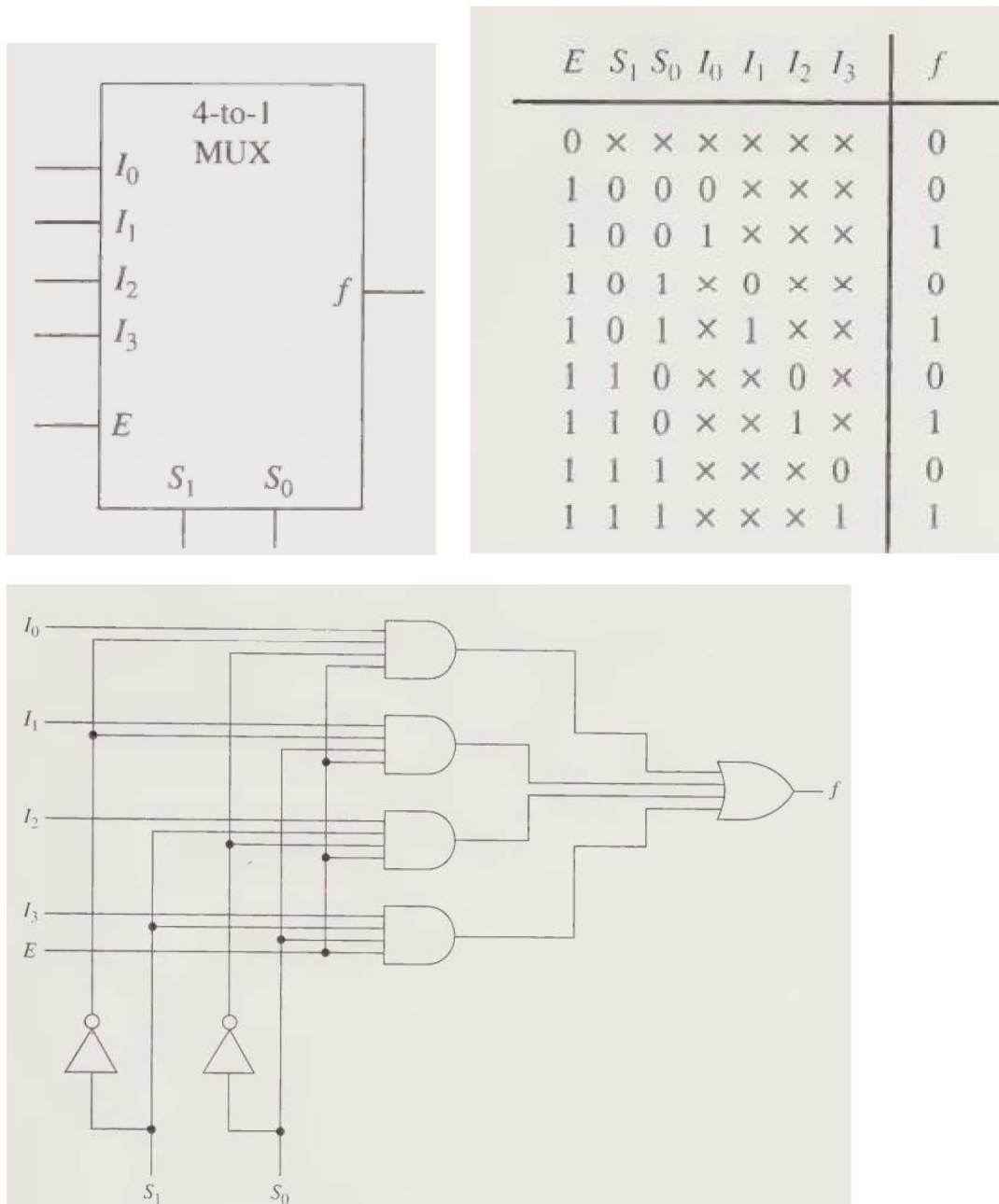
Multiplexers are also called data selectors. The basic function of this device is to select one of its 2^n data input lines and place the corresponding information appearing on this line onto a single output line. Since there are 2^n data input lines, n bits are needed to specify which input line is to be selected. This is achieved by placing the binary code for a desired data input line onto its n select input lines. A symbol for a 2^n -to-1-line multiplexer is shown in Fig



Typically, an enable, or strobe, line is also included to provide greater flexibility as in the case of decoders. The multiplexer shown in Fig. is enabled by applying a logic-1 to the E input terminal.

4-to-1 Mux

A realization of a 4-to-1-line multiplexer is given in Figure along with its compressed truth table and symbol. The X's in the compressed truth table denote irrelevant, i.e., don't-care, conditions.



As shown in the figure, each data input line I_i goes to its own and-gate. The select lines are used to uniquely select one of the and-gates. Thus, if the multiplexer is enabled, then the output corresponds to the value on the data input line of the selected and-gate. As in the case of decoders, the 0-1 combinations on the select lines are regarded as binary numbers. The decimal equivalents of these numbers determine which data input lines are selected and serve to identify the corresponding input terminals in the symbol. The functional table may be written as

E	S_1	S_0	f
0	×	×	0
1	0	0	I_0
1	0	1	I_1
1	1	0	I_2
1	1	1	I_3

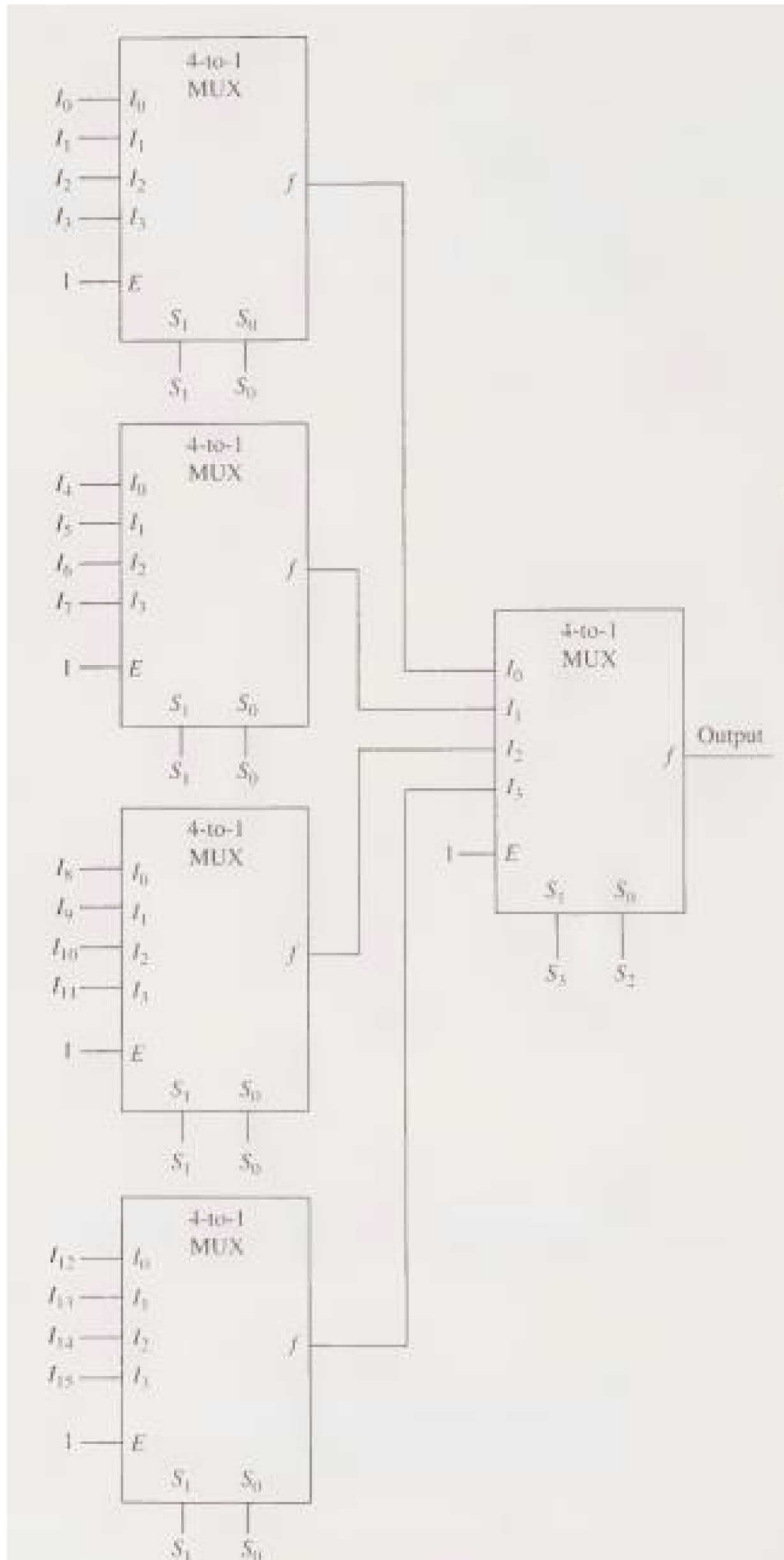
The logic equation of the Mux is

$$f = (I_0 \bar{S}_1 \bar{S}_0 + I_1 \bar{S}_1 S_0 + I_2 S_1 \bar{S}_0 + I_3 S_1 S_0)E$$

Mux Tree using Standard Mux configurations

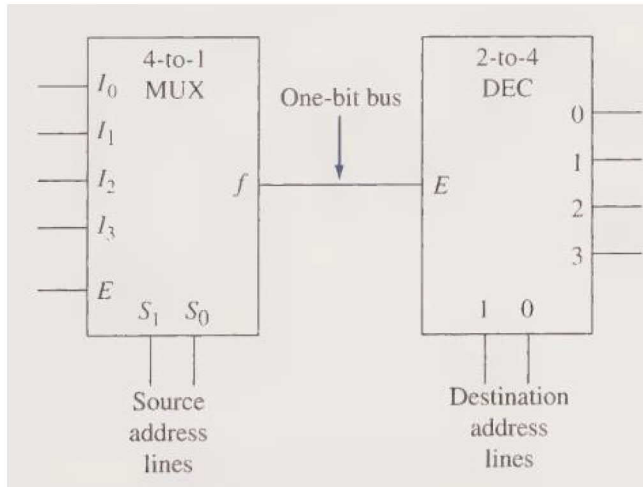
In addition to the 4-to-1-line multiplexer, 2-to-1-line, 8-to-1-line, and 16-to-1-line multiplexers are also commercially available. By interconnecting several multiplexers in a treelike structure, it is possible to produce a larger multiplexer.

For example, Fig. 5.34 illustrates how five 4-to-1-line multiplexers are used to construct a 16-to-1-line multiplexer.



Information Transmission using Mux

One of the primary applications of multiplexers is to provide for the transmission of information from several sources over a single path. This process is known as multiplexing. When a multiplexer is used in conjunction with a demultiplexer, i.e., a decoder with an enable input, an effective means is provided for connecting information from several source locations to several destination locations. This basic application of multiplexers and demultiplexers is illustrated in Fig.

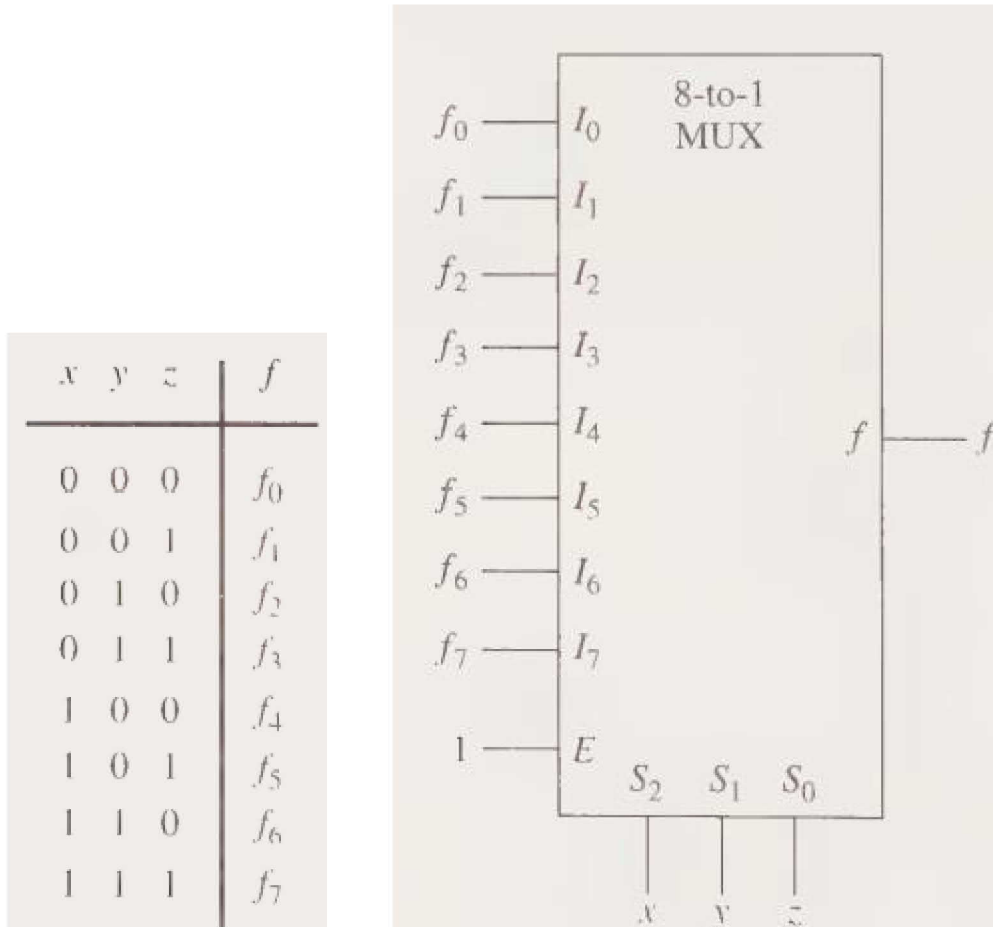


In this figure, one bit of information from any of four sources is selected according to the source address lines. This information is then placed on a wire, known as a bus, that connects to a demultiplexer. The bit combination on the destination address lines then determines on which of the four output lines of the demultiplexer the data information is placed. By using n of the structures shown in Fig in parallel, an n -bit word from any of four source locations is transferred to any of four destination locations.

Logic Design using Mux

Multiplexers are also used as general logic-design devices for realizing Boolean functions.

Example: Realization of a three-variable function using an 8-to-1-line multiplexer.

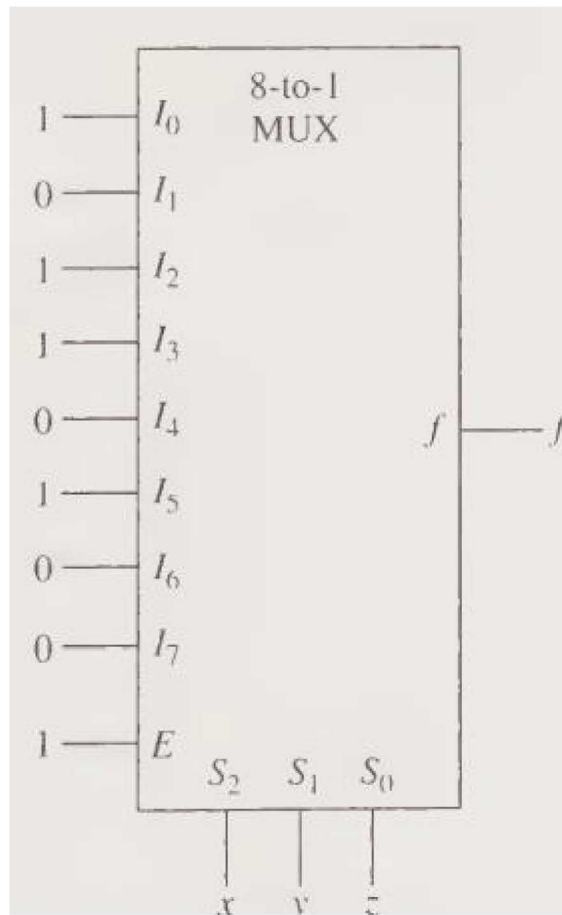


The Boolean expression corresponding to this truth table can be written as

$$f = (I_0 S_2 \bar{S}_1 S_0 + I_1 \bar{S}_2 \bar{S}_1 S_0 + I_2 \bar{S}_2 S_1 \bar{S}_0 + I_3 \bar{S}_2 S_1 S_0 + I_4 S_2 \bar{S}_1 \bar{S}_0 + I_5 S_2 \bar{S}_1 S_0 + I_6 S_2 S_1 \bar{S}_0 + I_7 S_2 S_1 S_0) E$$

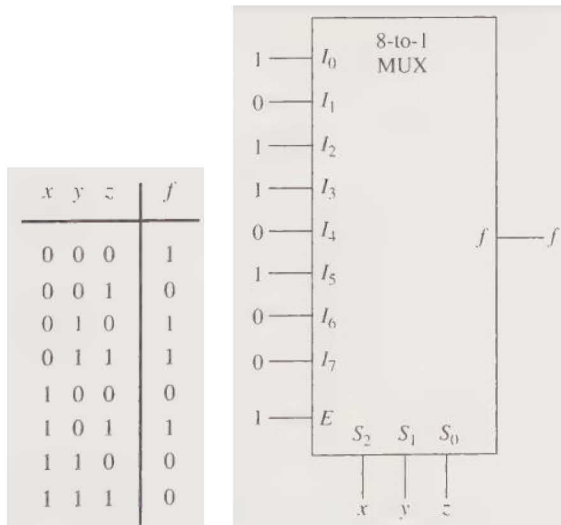
Example: Realization of $f(x, y, z) = \sum m(0, 2, 3, 5)$.

x	y	z	f
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0



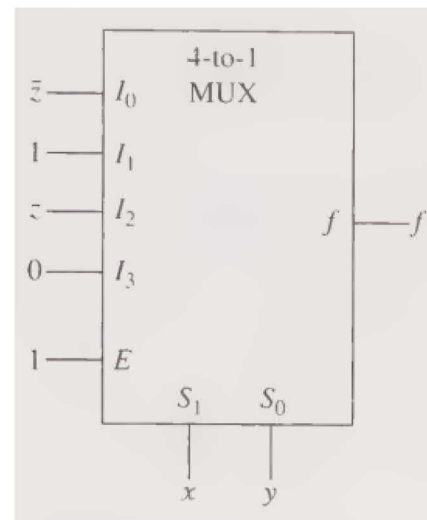
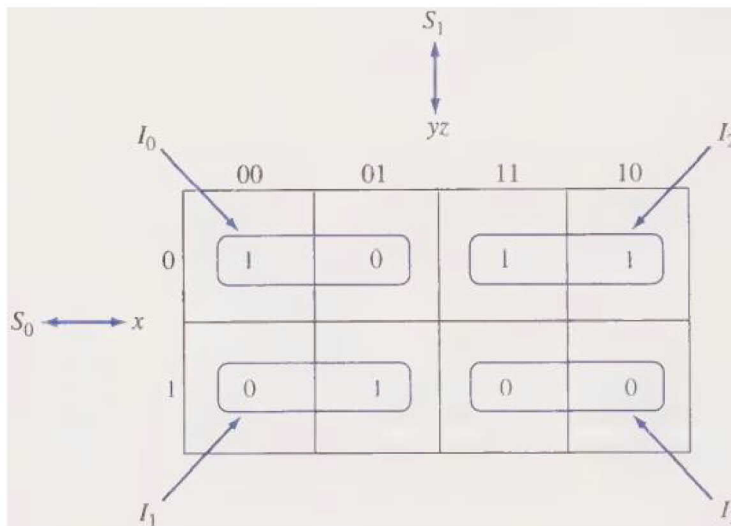
Example: Draw the circuit to realize $f(x, y, z) = \sum m(0,2,3,5)$ using 8-to-1 Mux. Also show at-least 2 realizations of 4-to-1 Mux.

8-to-1 Mux realization:

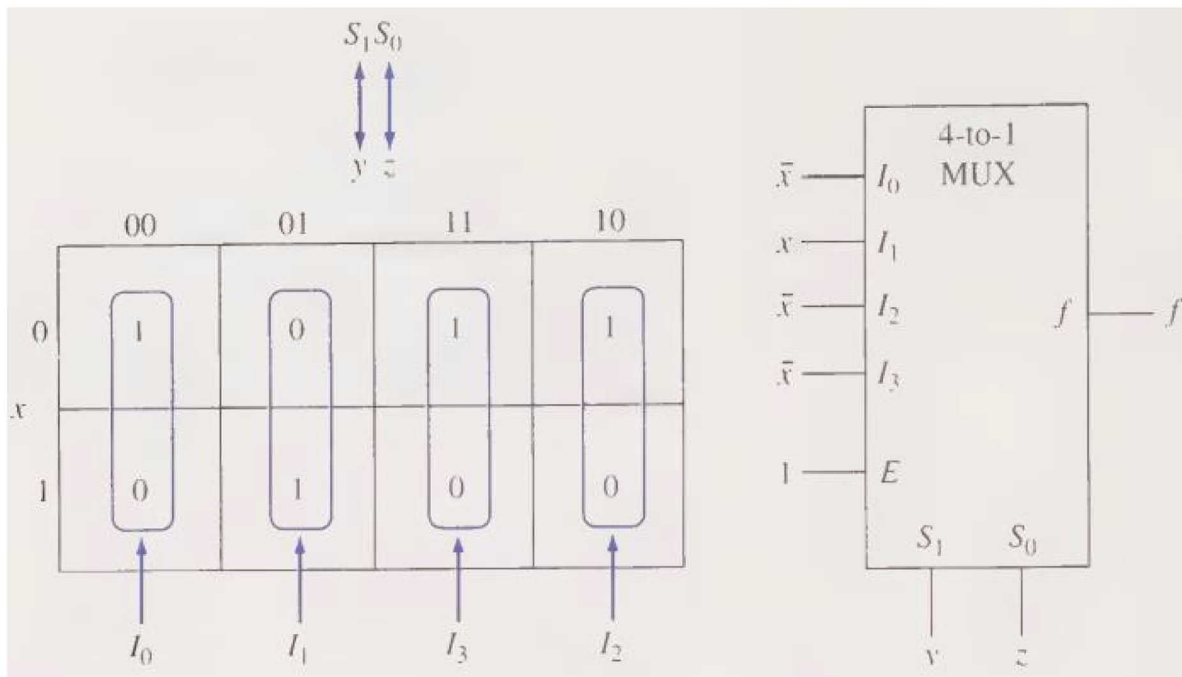


4-to-1 Mux realization:

$$f(x, y, z) = \sum m(0,2,3,5) = \bar{x}\bar{y}\bar{z} + \bar{x}\bar{y}z + \bar{x}yz + x\bar{y}z$$



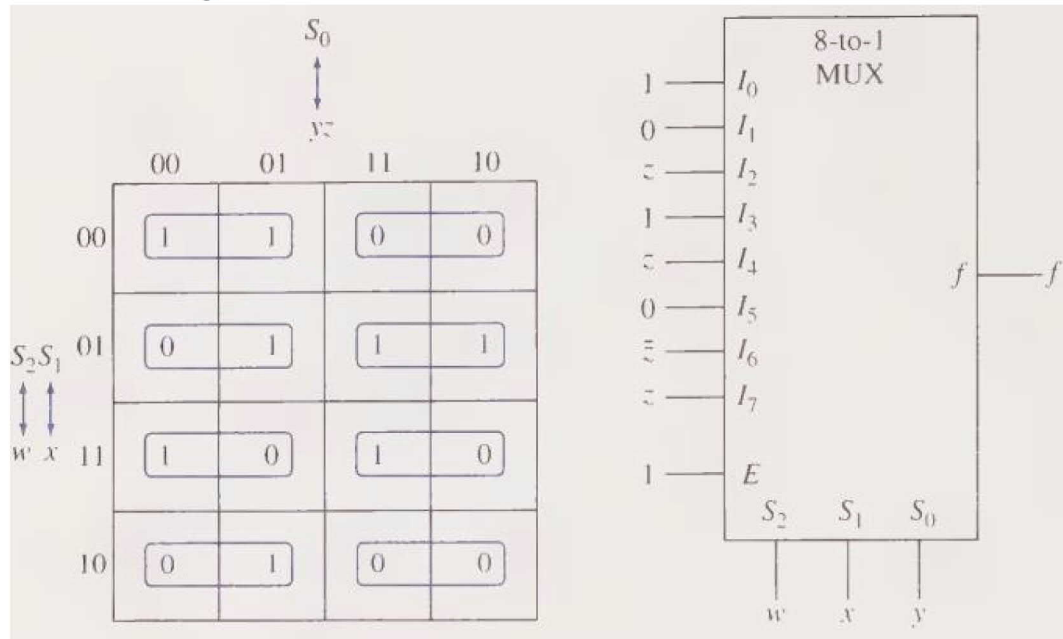
Alternatively, the same may also be done as follows:



Example: Realize the function $f(w, x, y, z) = \sum m(0,1,5,6,7,9,13,14)$ using

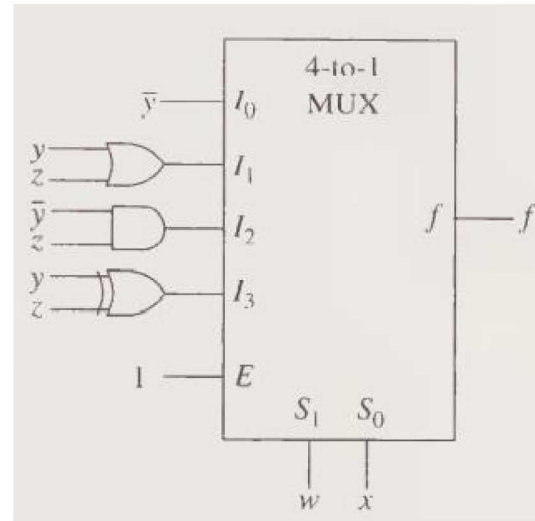
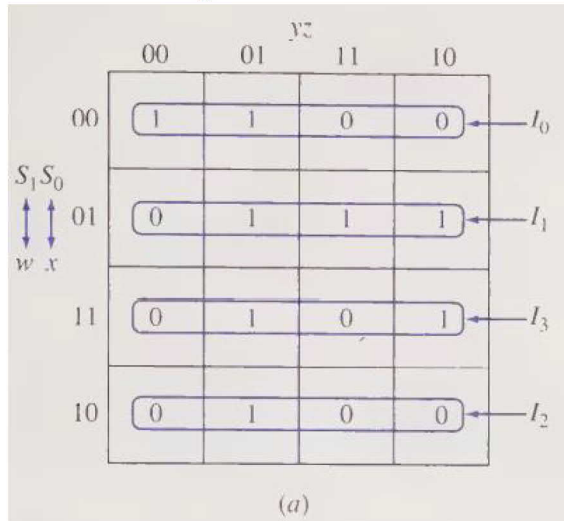
(a) 8-to-1 Mux (b) 4-to-1 Mux (c) 2-to-1 Mux

(a) Realization using 8-to-1 Mux



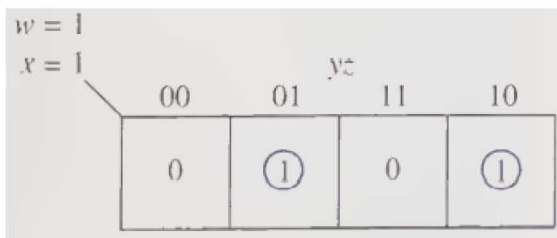
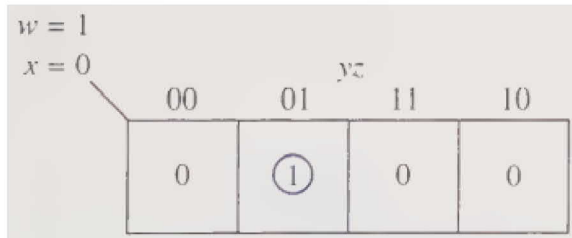
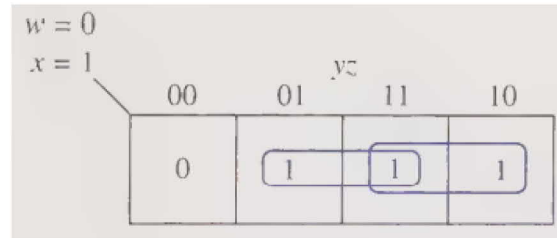
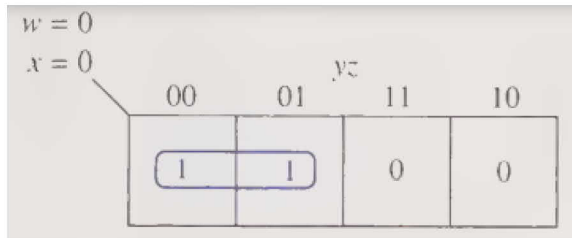
•
•
•

(b) Realization using 4-to-1 Mux



-
-

(c) Realization using 2-to-1 Mux

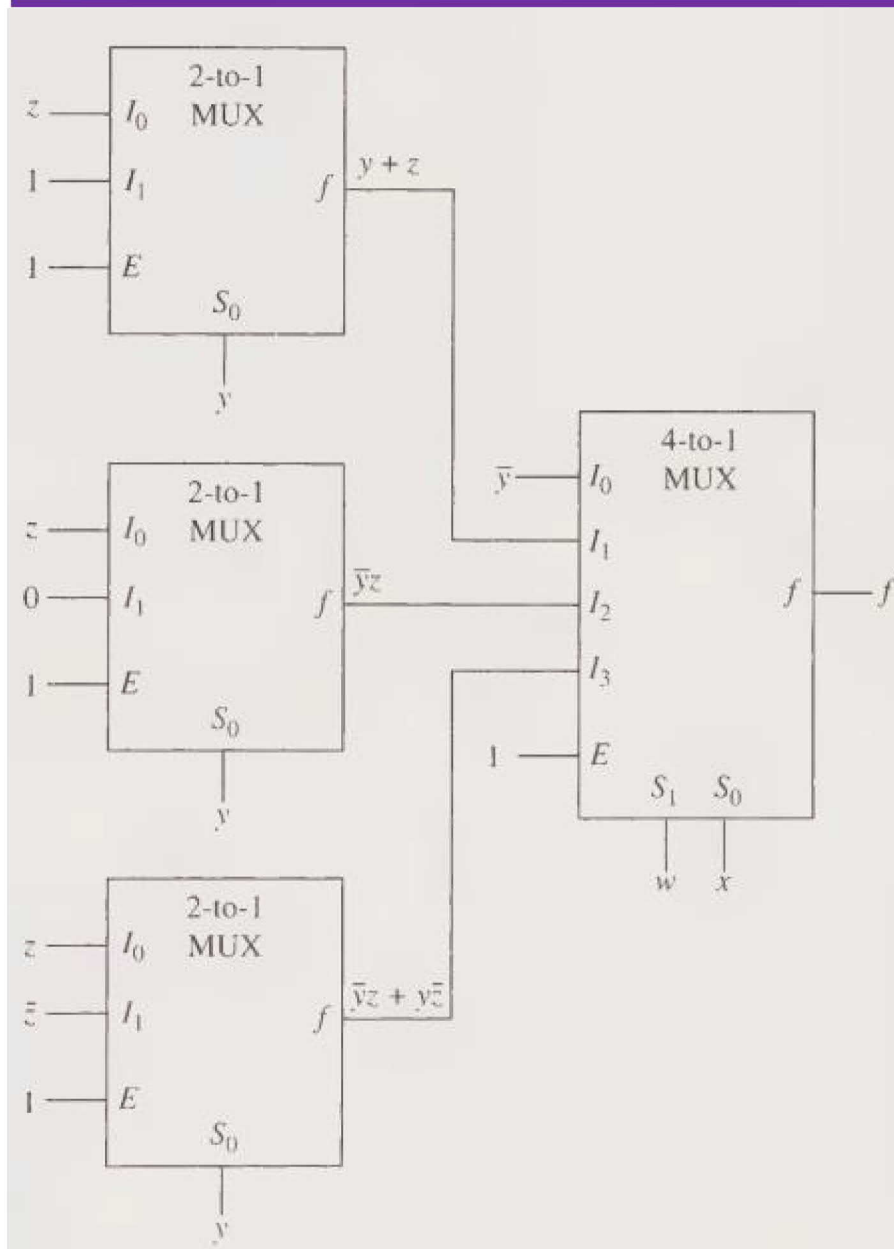


$$I_0 = \bar{y}$$

$$I_1 = y + z$$

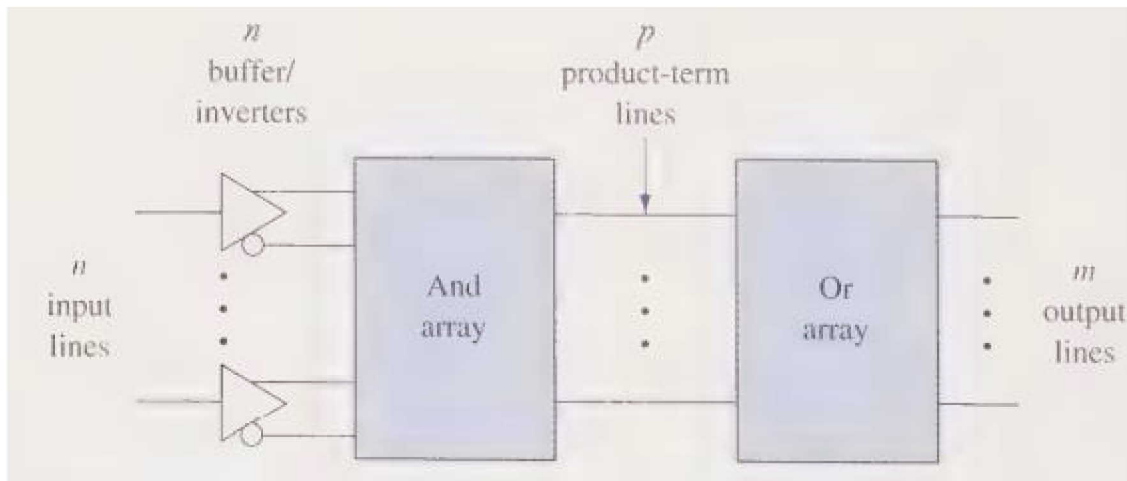
$$I_2 = \bar{y}z$$

$$I_3 = \bar{y}z + y\bar{z} = y \oplus z$$

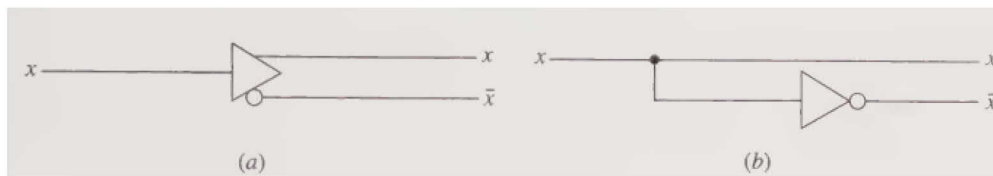


Programmable Logic Devices (PLDs)

The general structure of programmable logic devices is illustrated in Fig below.



The inputs to the PLD are applied to a set of buffer/inverters. The logic equivalent of the buffer/inverter is shown in Fig below.



These devices have both the true value of the input as well as the complemented value of the input as its outputs. In addition, these devices produce the necessary drive for the and-array which follows since, in general, the outputs from these devices serve as inputs to a very large number of gates. The array of and-gates accepts the n input variables and their complements and is used to generate a set of p product terms. These product terms, in turn, serve as inputs to an array of or-gates to realize a set of m sum-of-product expressions.

Types of PLDs

Device	And-array	Or-array
PROM	Fixed	Programmable
PLA	Programmable	Programmable
PAL	Programmable	Fixed

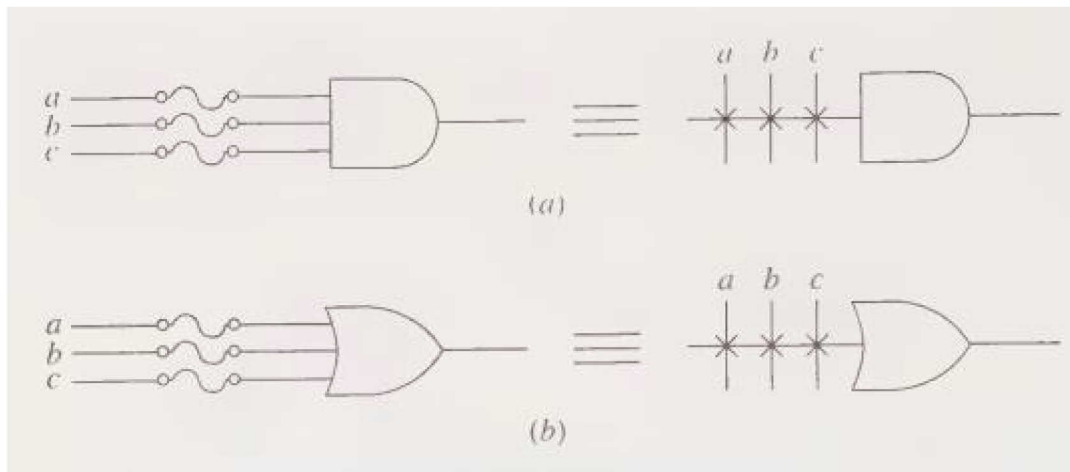
PLD Notation

Rather than drawing all the inputs to the and-gates and or-gates, the gates are drawn with a single input line.

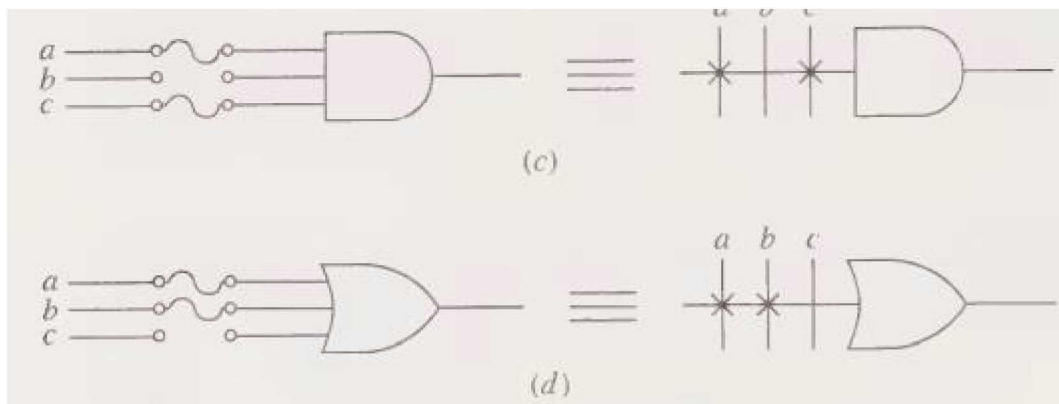
The inputs themselves are indicated by lines at right angles to the single gate line. The intersections between the input lines and the single gate line correspond to the types of connections.

A cross at the intersection denotes a fusible link that is intact; while the lack of a cross indicates the fuse is blown or no connection exists.

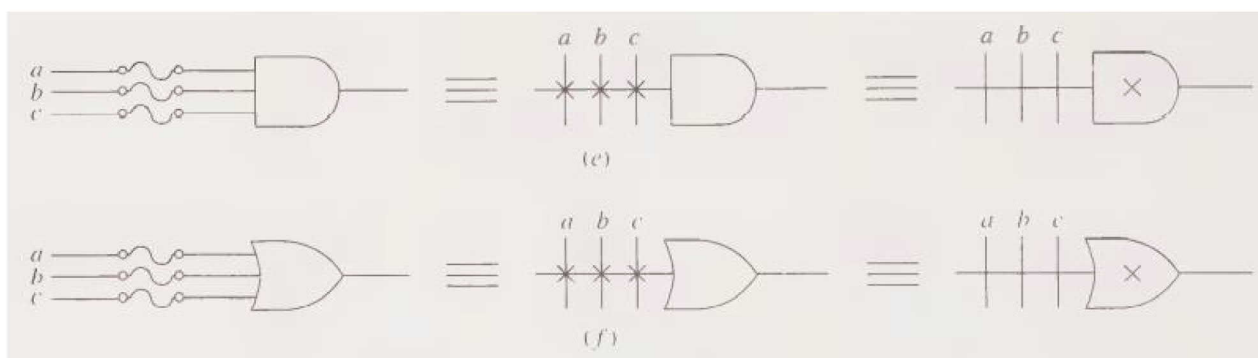
The occurrence of a hard-wired connection, i.e., one that is not fusible, is indicated by a junction dot.



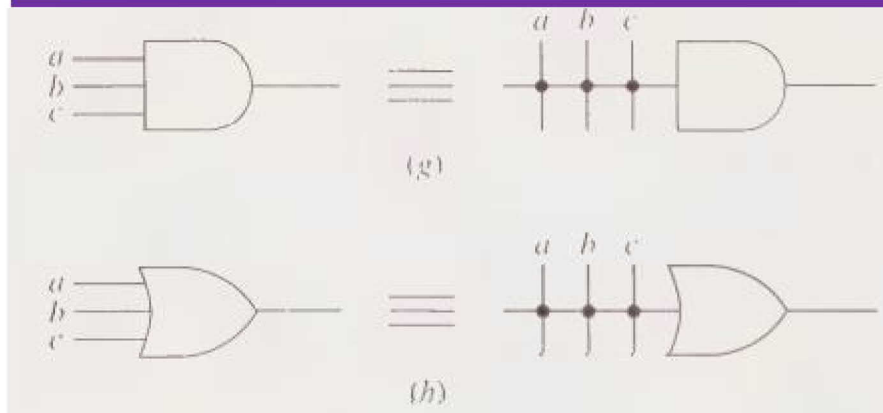
Unprogrammed AND & OR Arrays



Programmed AND & OR Arrays



Special notation for AND & OR arrays having all its input fuses intact



AND & OR arrays with non-fusible inputs