MSI and PLD Components

5-1 INTRODUCTION

The purpose of Boolean-algebra simplification is to obtain an algebraic expression that, when implemented, results in a low-cost circuit. However, the criteria that determine a low-cost circuit must be defined if we are to evaluate the success of the achieved simplification. The design procedure for combinational circuits presented in Section 4-2 minimizes the number of gates required to implement a given function. This procedure assumes that given two circuits that perform the same function, the one that requires fewer gates is preferable because it will cost less. This is not necessarily true when integrated circuits are used.

The circuit complexity of integrated circuits (ICs) has been classified in Section 2-8 as having four levels of integration: small- (SSI), medium- (MSI), large- (LSI), and very large- (VLSI) scale integration. A combinational circuit designed with individual gates can be implemented with SSI circuits that contain several independent gates. The number of gates in an SSI circuit is limited by the number of pins in the package, typically 14 or 16. Since several gates are included in a single IC package, it becomes economical to use as many of the gates from an already used package even if, by doing so, we increase the total number of gates. Moreover, some interconnections among the gates in many ICs are internal to the chip and it is more economical to use as many internal interconnections as possible in order to minimize the number of wires between package pins. With integrated circuits, it is not the count of gates that determines the cost, but the number and types of ICs employed and the number of interconnections needed to implement the given digital circuit.

There are several combinational circuits that are employed extensively in the design of digital systems. These circuits are available in integrated circuits and are classified as MSI components. MSI components perform specific digital functions commonly needed in the design of digital systems. In this chapter we introduce the most important combinational circuit-type MSI components that are readily available in IC packages. These are adders, subtractors, comparators, decoders, encoders, and multiplexers. These components are also used as standard modules within more complex LSI and VLSI circuits. The MSI components presented here provide a catalog of elementary digital modules used extensively as basic building blocks in the design of digital computers and systems.

The components of a digital system can be classified as being specific to an application or as being standard circuits. Standard components are taken from a set that has been used in other systems. MSI components are standard circuits and their use results in a significant reduction in the total cost as compared to the cost of using SSI circuits. In contrast, specific components are particular to the system being implemented and are not commonly found among the standard components. The implementation of specific circuits with LSI chips can be done by means of ICs that can be programmed to provide the required logic.

A programmable logic device (PLD) is an integrated circuit with internal logic gates that are connected through electronic fuses. Programming the device involves the blowing of fuses along the paths that must be disconnected so as to obtain a particular configuration. The word "programming" here refers to a hardware procedure that specifies the internal configuration of the device. The gates in a PLD are divided into an AND array and an OR array that are connected together to provide an AND-OR sum of product implementation. The initial state of a PLD has all the fuses intact. Programming the device involves the blowing of internal fuses to achieve a desired logic function.

In this chapter we introduce three programmable logic devices and establish procedures for their use in the design of digital systems. The three types of PLDs differ in the placement of fuses in the AND-OR array. Figure 5-1 shows the fuse locations of the three PLDs. The programmable read-only memory (PROM) has a fixed AND array and programmable fuses for the output OR gates. The PROM implements Boolean functions in sum of minterms, as explained in Section 5-7. The programmable array logic (PAL) has a fused programmable AND array and a fixed OR array. The AND gates are programmed to provide the product terms for the Boolean functions that are logically summed in each OR gate. PALs are presented in Section 5-9. The most flexible PLD is the programmable logic array (PLA), where both the AND and OR arrays can be programmed. The product terms in the AND array may be shared by any OR gate to provide the required sum of products implementation. The operation of the PLA is explained in Section 5-8.

The advantage of using PLDs in the design of digital systems is that they can be programmed to incorporate complex logic functions within one LSI circuit. The use of programmable logic devices is an alternative to another design technology called VLSI design. VLSI design refers to the design of digital systems that contain thousands of

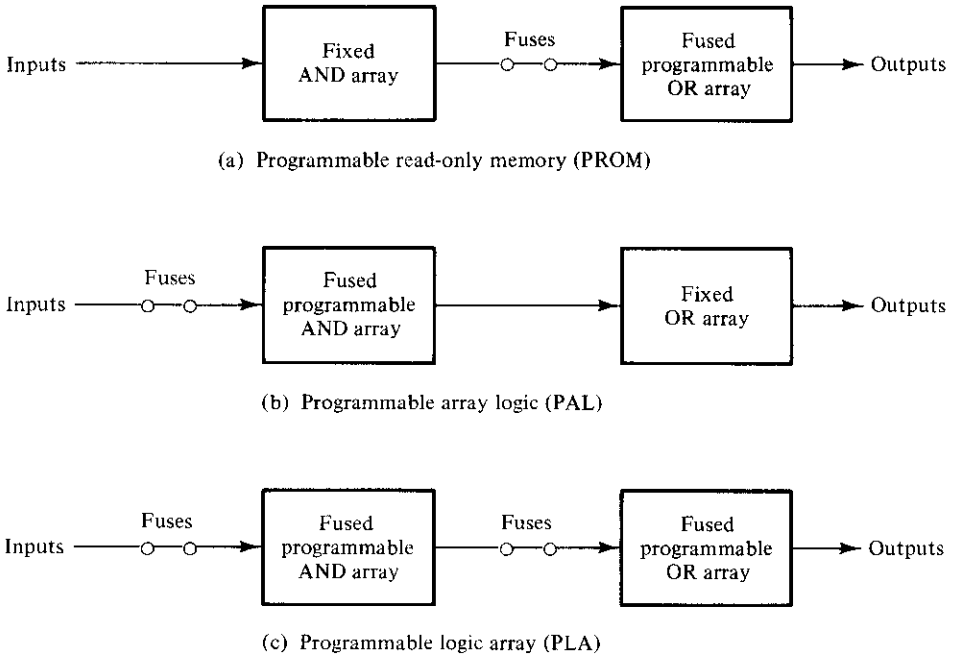


FIGURE 5-1
Basic configuration of three PLDs

gates within a single integrated-circuit chip. The basic component used in VLSI design is the *gate array*. A gate array consists of a pattern of gates fabricated in an area of silicon that is repeated thousands of times until the entire chip is covered with identical gates. Arrays of 1000 to 10,000 gates can be fabricated within a single integrated-circuit chip, depending on the technology used. The design with gate arrays requires that the designer specify the layout of the chip and the way that the gates are routed and connected. The first few levels of the fabrication process are common and independent of the final logic function. Additional fabrication levels are required to interconnect the gates in order to realize the desired function. This is usually done by means of computer-aided design methods. Both the gate array and the programmable logic device require extensive computer software tools to facilitate the design procedure.

5-2 BINARY ADDER AND SUBTRACTOR

The full-adder introduced in Section 4-3 forms the sum of two bits and a previous carry. Two binary numbers of n bits each can be added by means of this circuit. To demonstrate with a specific example, consider two binary numbers, $A = 1011$ and $B = 0011$, whose sum is $S = 1110$. When a pair of bits are added through a full-adder, the circuit produces a carry to be used with the pair of bits one significant position higher. This is shown in the following table:

Subscript i	4	3	2	1		Full-adder of Fig. 4-5
Input carry	0	1	1	0	C_i	z
Augend	1	0	1	1	A_i	x
Addend	0	0	1	1	B_i	y
Sum	1	1	1	0	S_i	S
Output carry	0	0	1	1	C_{i+1}	C

The bits are added with full-adders, starting from the least significant position (subscript 1), to form the sum bit and carry bit. The inputs and outputs of the full-adder circuit of Fig. 4-5 are also indicated. The input carry C_1 in the least significant position must be 0. The value of C_{i+1} in a given significant position is the output carry of the full-adder. This value is transferred into the input carry of the full-adder that adds the bits one higher significant position to the left. The sum bits are thus generated starting from the rightmost position and are available as soon as the corresponding previous carry bit is generated.

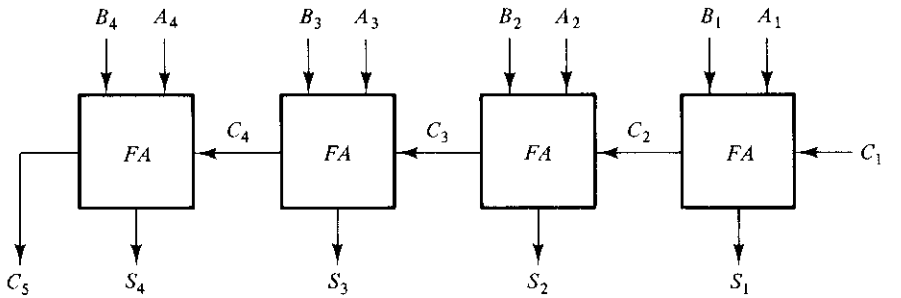
The sum of two n-bit binary numbers, A and B, can be generated in two ways: either in a serial fashion or in parallel. The serial addition method uses only one full-adder circuit and a storage device to hold the generated output carry. The pair of bits in A and B are transferred serially, one at a time, through the single full-adder to produce a string of output bits for the sum. The stored output carry from one pair of bits is used as an input carry for the next pair of bits. The parallel method uses n full-adder circuits, and all bits of A and B are applied simultaneously. The output carry from one full-adder is connected to the input carry of the full-adder one position to its left. As soon as the carries are generated, the correct sum bits emerge from the sum outputs of all full-adders.

Binary Parallel Adder

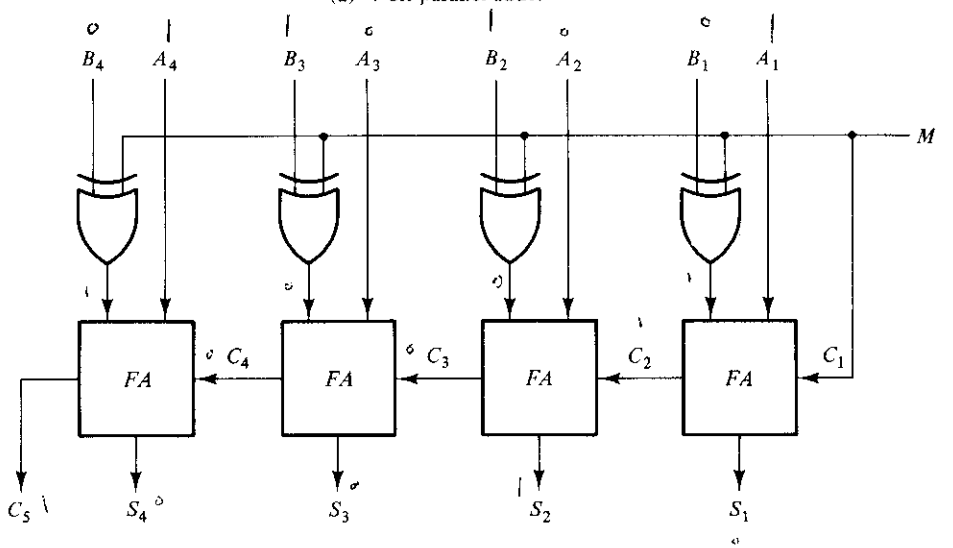
A binary parallel adder is a digital circuit that produces the arithmetic sum of two binary numbers in parallel. It consists of full-adders connected in a chain, with the output carry from each full-adder connected to the input carry of the next full-adder in the chain.

Figure 5-2(a) shows the interconnection of four full-adder (FA) circuits to provide a 4-bit binary parallel adder. The augend bits of A and the addend bits of B are designated by subscript numbers from right to left, with subscript 1 denoting the low-order bit. The carries are connected in a chain through the full-adders. The input carry to the adder is C_1 and the output carry is C_5. The S outputs generate the required sum bits. When the 4-bit full-adder circuit is enclosed within an IC package, it has four terminals for the augend bits, four terminals for the addend bits, four terminals for the sum bits, and two terminals for the input and output carries.

An n-bit parallel adder requires n full-adders. It can be constructed from 4-bit, 2-bit, and 1-bit full-adders ICs by cascading several packages. The output carry from one



(a) 4-bit parallel adder



(b) 4-bit adder-subtractor

FIGURE 5-2
Adder and subtractor circuits

package must be connected to the input carry of the one with the next higher-order bits.

The 4-bit full-adder is a typical example of an MSI function. It can be used in many applications involving arithmetic operations. Observe that the design of this circuit by the classical method would require a truth table with $2^9 = 512$ entries, since there are nine inputs to the circuit. By using an iterative method of cascading an already known function, we were able to obtain a simple and well-organized implementation.

Binary Adder-Subtractor

The subtraction of binary numbers can be done most conveniently by means of complements, as discussed in Section 1-5. Remember that the subtraction $A - B$ can be done by taking the 2's complement of B and adding it to A. The 2's complement can be ob-

tained by taking the 1's complement and adding one to the least significant pair of bits. The 1's complement can be implemented with inverters and a one can be added to the sum through the input carry.

The circuit for subtracting $A - B$ consists of a parallel adder with inverters placed between each data input B and the corresponding input of the full-adder. The input carry C_1 must be equal to 1 when performing subtraction. The operation thus performed becomes A plus the 1's complement of B plus 1. This is equal to A plus the 2's complement of B. For unsigned numbers, this gives $A - B$ if $A \geq B$ or the 2's complement of $B - A$ if $A < B$ (see Section 1-5). For signed numbers, the result is $A - B$ provided there is no overflow. (See Section 1-6.)

The addition and subtraction operations can be combined into one circuit with one common binary adder. This is done by including an exclusive-OR gate with each full-adder. A 4-bit adder-subtractor circuit is shown in Fig. 5-2(b). The mode input M controls the operation. When $M = 0$, the circuit is an adder, and when $M = 1$, the circuit becomes a subtractor. Each exclusive-OR gate receives input M and one of the inputs of B. When $M = 0$, we have $B \oplus 0 = B$. The full-adders receive the value of B, the input carry is 0, and the circuit performs A plus B. When $M = 1$, we have $B \oplus 1 = B'$ and $C_1 = 1$. The B inputs are all complemented and a 1 is added through the input carry. The circuit performs the operation A plus the 2's complement of B.

Carry Propagation

The addition of two binary numbers in parallel implies that all the bits of the augend and the addend are available for computation at the same time. As in any combinational circuit, the signal must propagate through the gates before the correct output sum is available in the output terminals. The total propagation time is equal to the propagation delay of a typical gate times the number of gate levels in the circuit. The longest propagation delay time in a parallel adder is the time it takes the carry to propagate through the full-adders. Since each bit of the sum output depends on the value of the input carry, the value of S_i in any given stage in the adder will be in its steady-state final value only after the input carry to that stage has been propagated. Consider output S_4 in Fig. 5-2(a). Inputs A_4 and B_4 reach a steady value as soon as input signals are applied to the adder. But input carry C_4 does not settle to its final steady-state value until C_3 is available in its steady-state value. Similarly, C_3 has to wait for C_2, and so on down to C_1. Thus, only after the carry propagates through all stages will the last output S_4 and carry C_5 settle to their final steady-state value.

The number of gate levels for the carry propagation can be found from the circuit of the full-adder. This circuit was derived in Fig. 4-5 and is redrawn in Fig. 5-3 for convenience. The input and output variables use the subscript i to denote a typical stage in the parallel adder. The signals at P_i and G_i settle to their steady-state values after the propagation through their respective gates. These two signals are common to all full-adders and depend only on the input augend and addend bits. The signal from the input carry, C_i, to the output carry, C_{i+1}, propagates through an AND gate and an OR gate, which constitute two gate levels. If there are four full-adders in the parallel adder, the

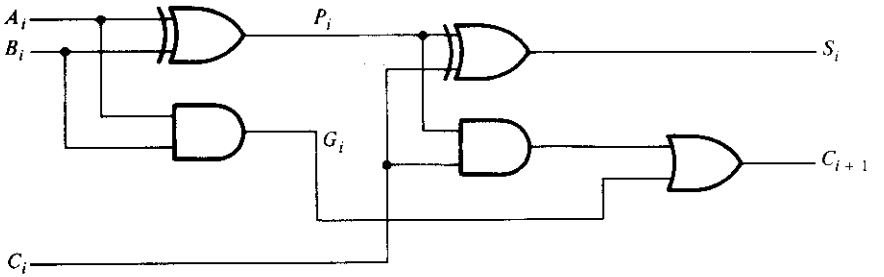


FIGURE 5-3
Full-adder circuit

output carry C_5 would have $2 \times 4 = 8$ gate levels from C_1 to C_5. The total propagation time in the adder would be the propagation time in one half-adder plus eight gate levels. For an n-bit parallel adder, there are $2n$ gate levels for the carry to propagate through.

The carry propagation time is a limiting factor on the speed with which two numbers are added in parallel. Although a parallel adder, or any combinational circuit, will always have some value at its output terminals, the outputs will not be correct unless the signals are given enough time to propagate through the gates connected from the inputs to the outputs. Since all other arithmetic operations are implemented by successive additions, the time consumed during the addition process is very critical. An obvious solution for reducing the carry propagation delay time is to employ faster gates with reduced delays. But 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 employs the principle of *look-ahead* carry and is described below.

Consider the circuit of the full-adder shown in Fig. 5-3. If we define two new binary variables:

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

the output sum and carry can be expressed as

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

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

We now write the Boolean function for the carry output of each stage and substitute for each C_i its value from the previous equations:

$$C_2 = G_1 + P_1C_1$$

$$C_3 = G_2 + P_2C_2 = G_2 + P_2(G_1 + P_1C_1) = G_2 + P_2G_1 + P_2P_1C_1$$

$$C_4 = G_3 + P_3C_3 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1C_1$$

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 a two-level NAND). The three Boolean functions for C_2, C_3, and C_4 are implemented in the look-ahead carry generator shown in Fig. 5-4. Note that C_4 does not have to wait for C_3 and C_2 to propagate; in fact, C_4 is propagated at the same time as C_2 and C_3.

The construction of a 4-bit parallel adder with a look-ahead carry scheme is shown in Fig. 5-5. Each sum output requires two exclusive-OR gates. The output of the first exclusive-OR gate generates the P_i variable, and the AND gate generates the G_i variable. All the P's and G's are generated in two gate levels. The carries are propagated through the look-ahead carry generator (similar to that in Fig. 5-4) and applied as in-

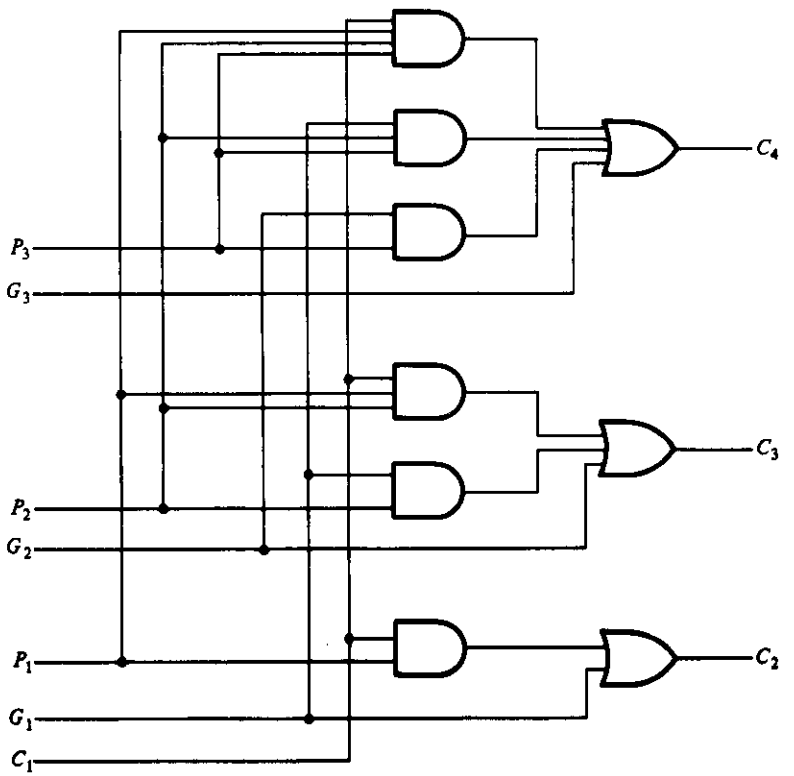


FIGURE 5-4
Logic diagram of a look-ahead carry generator

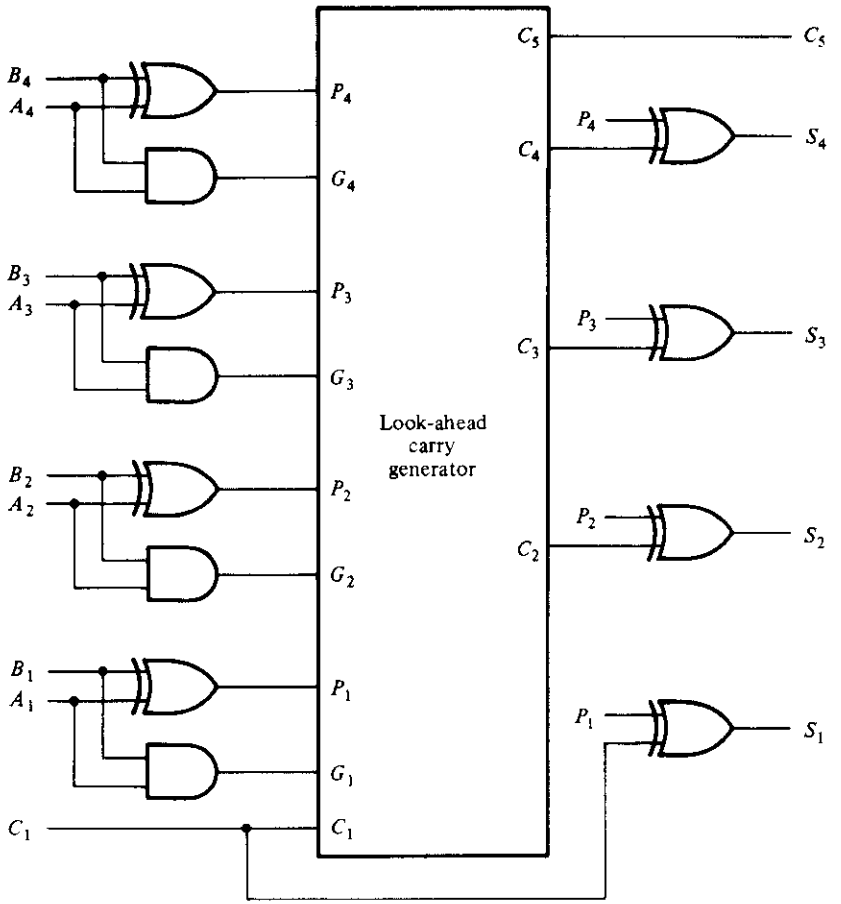


FIGURE 5-5
4-bit full-adders with look-ahead carry

puts to the second exclusive-OR gate. After the P and G signals settle into their steady-state values, all output carries are generated after a delay of two levels of gates. Thus, outputs S_2 through S_4 have equal propagation delay times. The two-level circuit for the output carry C_5 is not shown in Fig. 5-4. This circuit can be easily derived by the equation-substitution method, as done above (see Problem 5-8).

5-3 DECIMAL ADDER

Computers or calculators that perform arithmetic operations directly in the decimal number system represent decimal numbers in binary-coded form. An adder for such a computer must employ arithmetic circuits that accept coded decimal numbers and

present results in the accepted code. For binary addition, it was sufficient to consider a pair of significant bits at a time, together with a previous carry. A decimal adder requires a minimum of nine inputs and five outputs, since four bits are required to code each decimal digit and the circuit must have an input carry and output carry. Of course, there is a wide variety of possible decimal adder circuits, dependent upon the code used to represent the decimal digits.

The design of a nine-input, five-output combinational circuit by the classical method requires a truth table with $2^9 = 512$ entries. Many of the input combinations are don't-care conditions, since each binary code input has six combinations that are invalid. The simplified Boolean functions for the circuit may be obtained by a computer-generated tabular method, and the result would probably be a connection of gates forming an irregular pattern. An alternate procedure is to add the numbers with full-adder circuits, taking into consideration the fact that six combinations in each 4-bit input are not used. The output must be modified so that only those binary combinations that are valid combinations of the decimal code are generated.

BCD Adder

Consider the arithmetic addition of two decimal digits in BCD, together with a possible carry from a previous stage. Since each input digit does not exceed 9, the output sum cannot be greater than $9 + 9 + 1 = 19$, the 1 in the sum being an input carry. Suppose we apply two BCD digits to a 4-bit binary adder. The adder will form the sum in *binary* and produce a result that may range from 0 to 19. These binary numbers are listed in Table 5-1 and are labeled by symbols K, Z_8, Z_4, Z_2, and Z_1. K is the carry, and the subscripts under the letter Z represent the weights 8, 4, 2, and 1 that can be assigned to the four bits in the BCD code. The first column in the table lists the binary sums as they appear in the outputs of a 4-bit *binary* adder. The output sum of two *decimal digits* must be represented in BCD and should appear in the form listed in the second column of the table. The problem is to find a simple rule by which the binary number in the first column can be converted to the correct BCD-digit representation of the number in the second column.

In examining the contents of the table, it is apparent that when the binary sum is equal to or less than 1001, the corresponding BCD number is identical, and therefore no conversion is needed. When the binary sum is greater than 1001, we obtain a nonvalid BCD representation. The addition of binary 6 (0110) to the binary sum converts it to the correct BCD representation and also produces an output carry as required.

The logic circuit that detects the necessary correction can be derived from the table entries. It is obvious that a correction is needed when the binary sum has an output carry $K = 1$. The other six combinations from 1010 to 1111 that need a correction have a 1 in position Z_8. To distinguish them from binary 1000 and 1001, which also have a 1 in position Z_8, we specify further that either Z_4 or Z_2 must have a 1. The condition for a correction and an output carry can be expressed by the Boolean function

$$C = K + Z_8Z_4 + Z_8Z_2$$

TABLE 5-1
Derivation of a BCD Adder

Binary Sum					BCD Sum					Decimal
K	Z_8	Z_4	Z_2	Z_1	C	S_8	S_4	S_2	S_1	
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9
0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19

When $C = 1$, it is necessary to add 0110 to the binary sum and provide an output carry for the next stage.

A *BCD adder* is a circuit that adds two BCD digits in parallel and produces a sum digit also in BCD. A BCD adder must include the correction logic in its internal construction. To add 0110 to the binary sum, we use a second 4-bit binary adder, as shown in Fig. 5-6. The two decimal digits, together with the input carry, are first added in the top 4-bit binary adder to produce the binary sum. When the output carry is equal to zero, nothing is added to the binary sum. When it is equal to one, binary 0110 is added to the binary sum through the bottom 4-bit binary adder. The output carry generated from the bottom binary adder can be ignored, since it supplies information already available at the output-carry terminal.

The BCD adder can be constructed with three IC packages. Each of the 4-bit adders is an MSI function and the three gates for the correction logic need one SSI package. However, the BCD adder is available in one MSI circuit. To achieve shorter propagation delays, an MSI BCD adder includes the necessary circuits for look-ahead carries. The adder circuit for the correction does not need all four full-adders, and this circuit can be optimized within the IC package.

A decimal parallel adder that adds n decimal digits needs n BCD adder stages. The

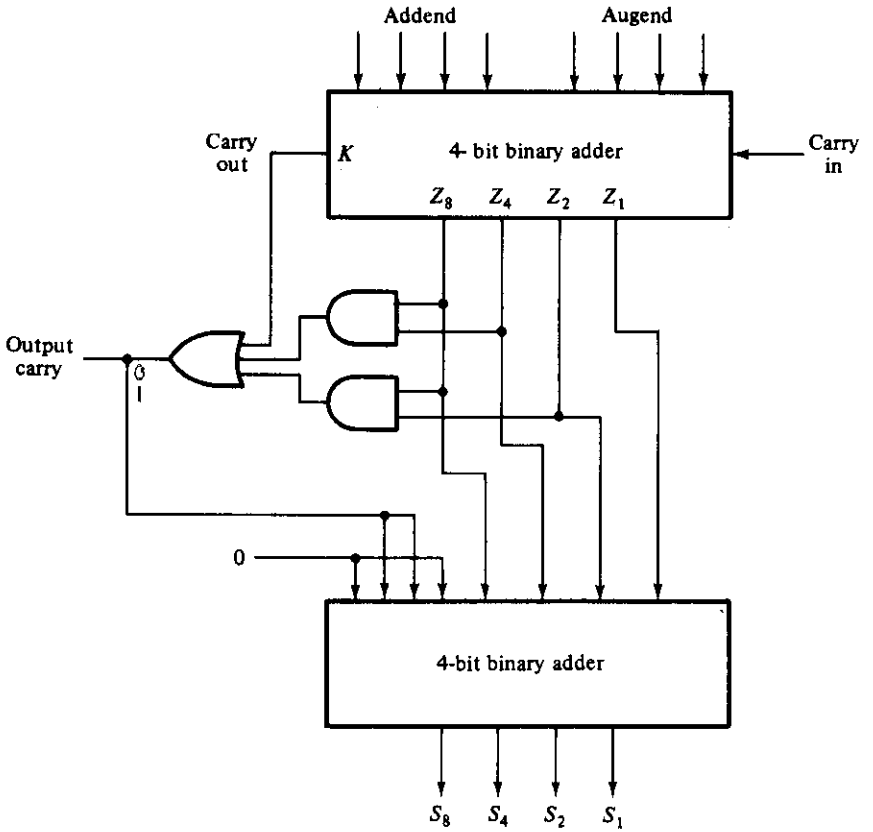


FIGURE 5-6
Block diagram of a BCD adder

output carry from one stage must be connected to the input carry of the next higher-order stage.

5-4 MAGNITUDE COMPARATOR

The comparison of two numbers is an operation that determines if one number is greater than, less than, or equal to the other number. A *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 indicate whether $A > B$, $A = B$, or $A < B$.

The circuit for comparing two n-bit numbers has 2^{2n} entries in the truth table and becomes too cumbersome even with $n = 3$. On the other hand, as one may suspect, a comparator circuit possesses a certain amount of regularity. Digital functions that possess an inherent well-defined regularity can usually be designed by means of an al-

gorithmic procedure if one is found to exist. An *algorithm* is a procedure that specifies a finite set of steps that, if followed, give the solution to a problem. We illustrate this method here by deriving an algorithm for the design of a 4-bit magnitude comparator.

The algorithm is a direct application of the procedure a person uses to compare the relative magnitudes of two numbers. Consider two numbers, A and B, with four digits each. Write the coefficients of the numbers with descending significance as follows:

$$A = A_3A_2A_1A_0$$

$$B = B_3B_2B_1B_0$$

where each subscripted letter represents one of the digits in the number. The two numbers are equal if all pairs of significant digits are equal, i.e., if $A_3 = B_3$ 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 equivalence function:

$$x_i = A_iB_i + A_i'B_i' \qquad i = 0, 1, 2, 3$$

where $x_i = 1$ only if the pair of bits in position i are equal, i.e., if both are 1's or both are 0's.

The equality of the two 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 equality condition to exist, all x_i variables must be equal to 1. This dictates an AND operation of all variables:

$$(A = B) = x_3x_2x_1x_0$$

The *binary* variable $(A = B)$ is equal to 1 only if all pairs of digits of the two numbers are equal.

To determine if A is greater than or less than B, we inspect the relative magnitudes of pairs of significant digits starting from the most significant position. If the two digits are equal, we compare the next lower significant pair of digits. This comparison continues until a pair of unequal digits is reached. If the corresponding digit of A is 1 and that of B is 0, we conclude that $A > B$. If the corresponding digit of A is 0 and that of B is 1, we have that $A < B$. The sequential comparison can be expressed logically by the following two Boolean functions:

$$(A > B) = A_3B_3' + x_3A_2B_2' + x_3x_2A_1B_1' + x_3x_2x_1A_0B_0'$$

$$(A < B) = A_3'B_3 + x_3A_2'B_2 + x_3x_2A_1'B_1 + x_3x_2x_1A_0'B_0$$

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

The gate implementation of the three output variables just derived is simpler than it seems because it involves a certain amount of repetition. The "unequal" outputs can use the same gates that are needed to generate the "equal" output. The logic diagram of the 4-bit magnitude comparator is shown in Fig. 5-7. The four x outputs are generated with

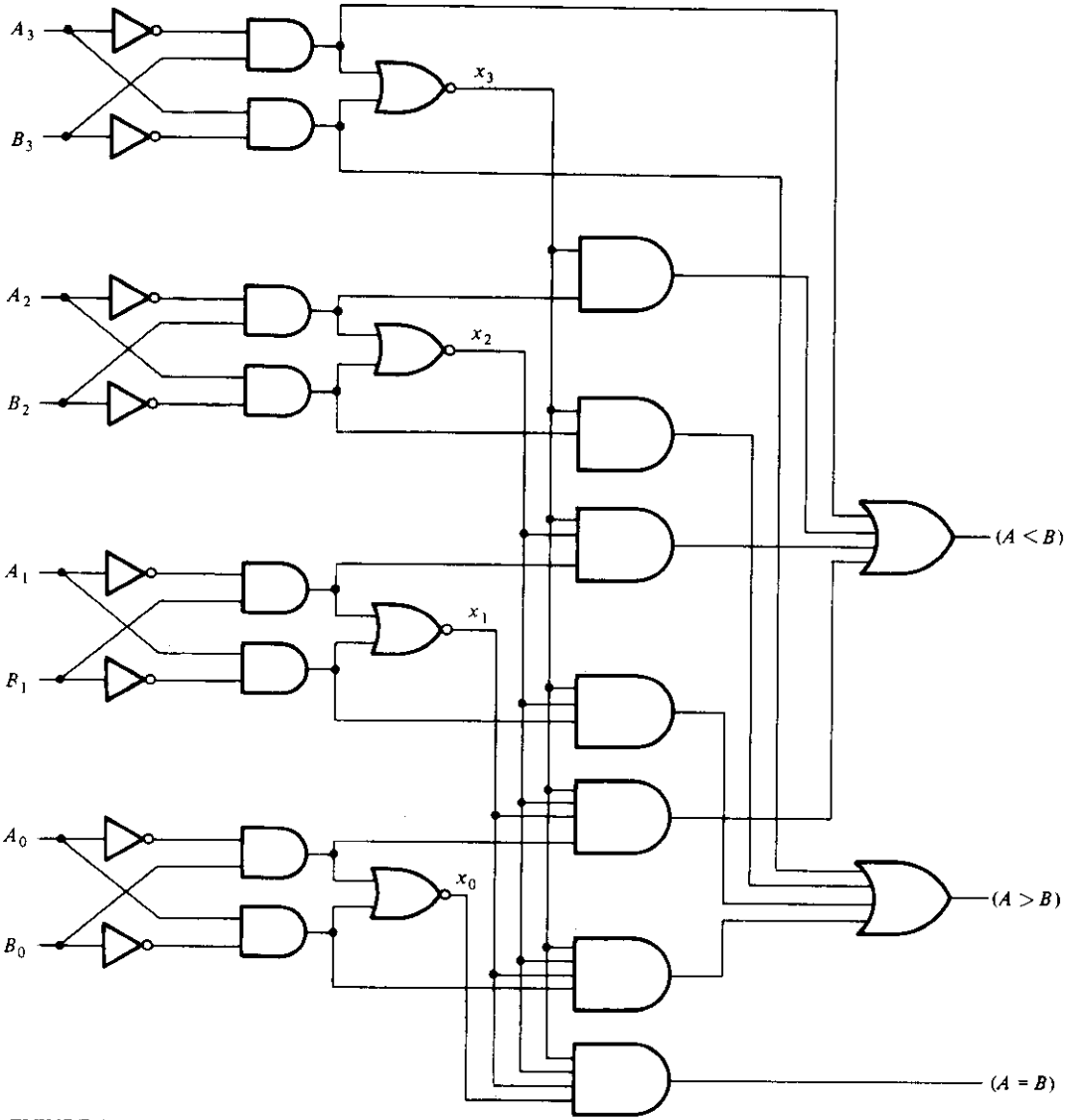


FIGURE 5-7
4-bit magnitude comparator

equivalence (exclusive-NOR) circuits and applied to an AND gate to give the output binary variable $(A = B)$. The other two outputs use the x variables to generate the Boolean functions listed before. This is a multilevel implementation and, as clearly seen, it has a regular pattern. The procedure for obtaining magnitude-comparator circuits for binary numbers with more than four bits should be obvious from this example. The same circuit can be used to compare the relative magnitudes of two BCD digits.

5-5 DECODERS AND ENCODERS

Discrete quantities of information are represented in digital systems with binary codes. A binary code of n bits is capable of representing up to 2^n distinct elements of the coded information. A *decoder* is a combinational circuit that converts binary information from n input lines to a maximum of 2^n unique output lines. If the n-bit decoded information has unused or don't-care combinations, the decoder output will have fewer than 2^n outputs.

The decoders presented here are called n-to-m-line decoders, where $m \leq 2^n$. Their purpose is to generate the 2^n (or fewer) minterms of n input variables. The name *decoder* is also used in conjunction with some code converters such as a BCD-to-seven-segment decoder.

As an example, consider the 3-to-8-line decoder circuit of Fig. 5-8. The three inputs are decoded into eight outputs, each output representing one of the minterms of the 3-input variables. The three inverters provide the complement of the inputs, and each one of the eight AND gates generates one of the minterms. A particular application of this decoder would be a binary-to-octal conversion. The input variables may represent a binary number, and the outputs will then represent the eight digits in the octal number

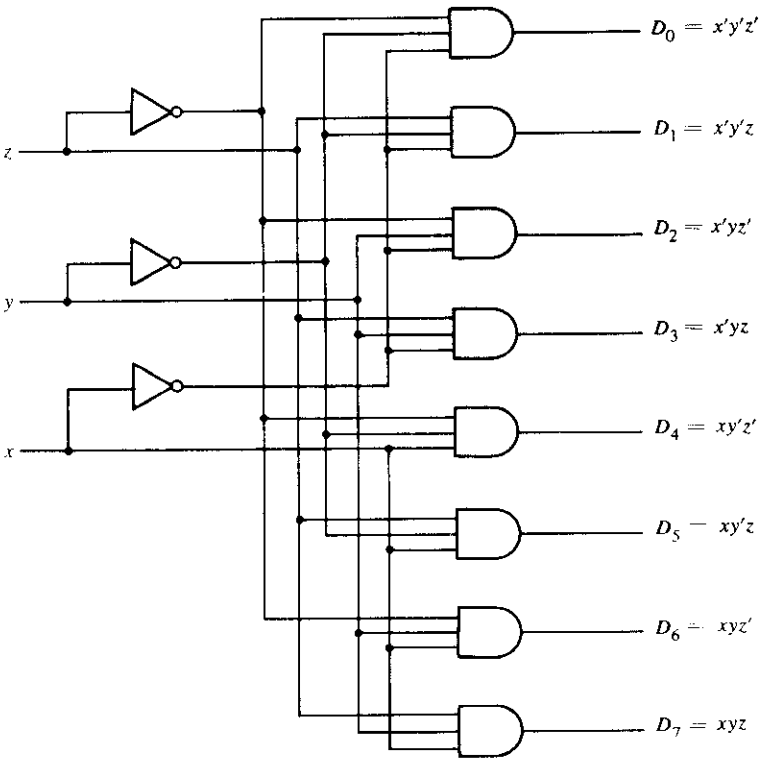


FIGURE 5-8
A 3-to-8 line decoder

TABLE 5-2
Truth Table of a 3-to-8-Line Decoder

Inputs			Outputs							
x	y	z	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
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

system. However, a 3-to-8-line decoder can be used for decoding any 3-bit code to provide eight outputs, one for each element of the code.

The operation of the decoder may be further clarified from its input–output relationship, listed in Table 5-2. Observe that the output variables are mutually exclusive because only one output can be equal to 1 at any one time. The output line whose value is equal to 1 represents the minterm equivalent of the binary number presently available in the input lines.

Combinational Logic Implementation

A decoder provides the 2^n minterm of n input variables. Since any Boolean function can be expressed in sum of minterms canonical form, one can use a decoder to generate the minterms and an external OR gate to form the sum. In this way, any combinational circuit with n inputs and m outputs can be implemented with an n-to-2^n-line decoder and m OR gates.

The procedure for implementing a combinational circuit by means of a decoder and OR gates requires that the Boolean functions for the circuit be expressed in sum of minterms. This form can be easily obtained from the truth table or by expanding the functions to their sum of minterms (see Section 2-5). A decoder is then chosen that generates all the minterms of the n input variables. The inputs to each OR gate are selected from the decoder outputs according to the minterm list in each function.

Example 5-1

Implement a full-adder circuit with a decoder and two OR gates.

From the truth table of the full-adder (Section 4-3), we obtain the functions for this combinational circuit in sum of minterms:

$$S(x, y, z) = \Sigma(1, 2, 4, 7)$$
$$C(x, y, z) = \Sigma(3, 5, 6, 7)$$

Since there are three inputs and a total of eight minterms, we need a 3-to-8-line decoder. The implementation is shown in Fig. 5-9. The decoder generates the eight

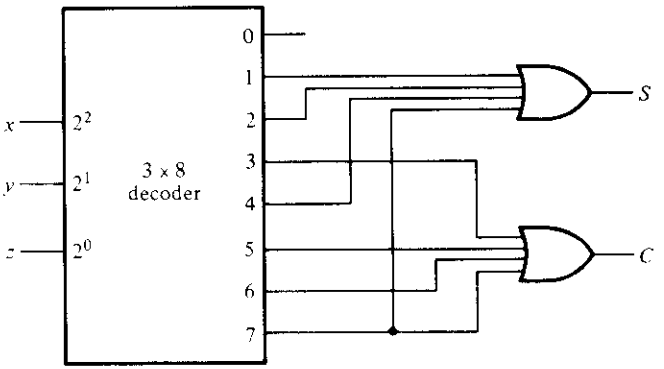


FIGURE 5-9
Implementation of a full-adder with a decoder

minterms for x, y, z. The OR gate for output S forms the sum of minterms 1, 2, 4, and 7. The OR gate for output C forms the sum of minterms 3, 5, 6, and 7. ■

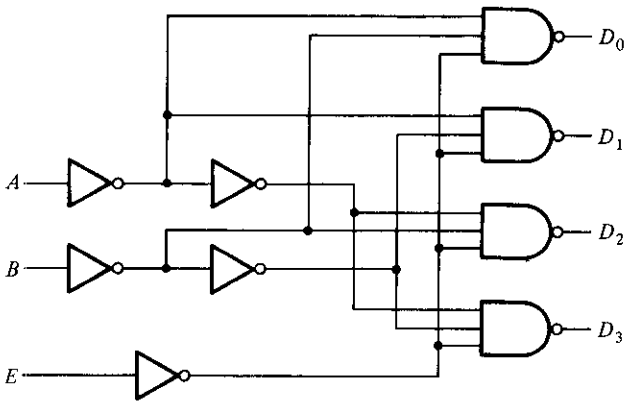
A function with a long list of minterms requires an OR gate with a large number of inputs. A function F having a list of k minterms can be expressed in its complemented form F' with $2^n - k$ minterms. If the number of minterms in a function is greater than $2^n/2$, then F' can be expressed with fewer minterms than required for F. In such a case, it is advantageous to use a NOR gate to sum the minterms of F'. The output of the NOR gate will generate the normal output F.

The decoder method can be used to implement any combinational circuit. However, its implementation must be compared with all other possible implementations to determine the best solution. In some cases, this method may provide the best implementation, especially if the combinational circuit has many outputs and if each output function (or its complement) is expressed with a small number of minterms.

Demultiplexers

Some IC decoders are constructed with NAND gates. Since a NAND gate produces the AND operation with an inverted output, it becomes more economical to generate the decoder minterms in their complemented form. Most, if not all, IC decoders include one or more *enable* inputs to control the circuit operation. A 2-to-4-line decoder with an enable input constructed with NAND gates is shown in Fig. 5-10. All outputs are equal to 1 if enable input E is 1, regardless of the values of inputs A and B. When the enable input is 0, the circuit operates as a decoder with complemented outputs. The truth table lists these conditions. The X's under A and B are don't-care conditions. Normal decoder operation occurs only with $E = 0$, and the outputs are selected when they are in the 0 state.

The block diagram of the decoder is shown in Fig. 5-11(a). The small circle at input E indicates that the decoder is enabled when $E = 0$. The small circles at the outputs indicate that all outputs are complemented.



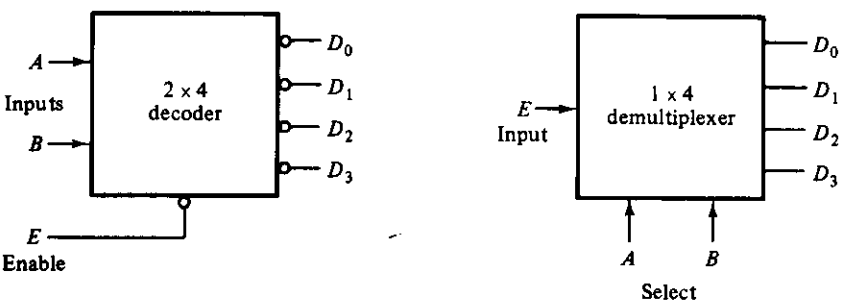
(a) Logic diagram

E	A	B	D_0	D_1	D_2	D_3
1	X	X	1	1	1	1
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

(b) Truth table

FIGURE 5-10
A 2-to-4-line decoder with enable (E) input

A decoder with an enable input can function as a demultiplexer. A *demultiplexer* is a circuit that receives information on a single line and transmits this information on one of 2^n possible output lines. The selection of a specific output line is controlled by the bit values of n selection lines. The decoder of Fig. 5-10 can function as a demultiplexer if the E line is taken as a data input line and lines A and B are taken as the selection lines. This is shown in Fig. 5-11(b). The single input variable E has a path to all four outputs, but the input information is directed to only one of the output lines, as specified by the binary value of the two selection lines, A and B. This can be verified from the truth table of this circuit, shown in Fig. 5-10(b). For example, if the selection lines $AB = 10$, output D_2 will be the same as the input value E, while all other outputs are maintained at 1. Because decoder and demultiplexer operations are obtained from the same circuit, a decoder with an enable input is referred to as a *decoder/demultiplexer*. It is the enable input that makes the circuit a demultiplexer; the decoder itself can use AND, NAND, or NOR gates.



(a) Decoder with enable

(b) Demultiplexer

FIGURE 5-11
Block diagrams for the circuit of Fig. 5-10

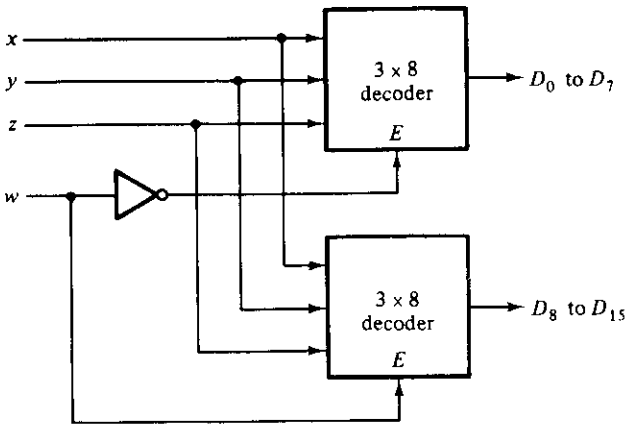


FIGURE 5-12
A 4 × 16 decoder constructed with two 3 × 8 decoders

Decoder/demultiplexer circuits can be connected together to form a larger decoder circuit. Figure 5-12 shows two 3 × 8 decoders with enable inputs connected to form a 4 × 16 decoder. When $w = 0$, the top decoder is enabled and the other is disabled. The bottom decoder outputs are all 0's, and the top eight outputs generate minterms 0000 to 0111. When $w = 1$, the enable conditions are reversed; the bottom decoder outputs generate minterms 1000 to 1111, while the outputs of the top decoder are all 0's. This example demonstrates the usefulness of enable inputs in ICs. In general, enable lines are a convenient feature for connecting two or more IC packages for the purpose of expanding the digital function into a similar function with more inputs and outputs.

Encoders

An encoder is a digital circuit that performs the inverse operation of a decoder. An encoder has 2^n (or fewer) input lines and n output lines. The output lines generate the binary code corresponding to the input value. An example of an encoder is the octal-to-binary encoder whose truth table is given in Table 5-3. It has eight inputs, one for each of the octal digits, and three outputs that generate the corresponding binary number. It is assumed that only one input has a value of 1 at any given time; otherwise the circuit has no meaning.

The encoder can be implemented with OR gates whose inputs are determined directly from the truth table. Output z is equal to 1 when the input octal digit is 1 or 3 or 5 or 7. Output y is 1 for octal digits 2, 3, 6, or 7, and output x is 1 for digits 4, 5, 6, or 7. These conditions can be expressed by the following output Boolean functions:

$$z = D_1 + D_3 + D_5 + D_7$$

$$y = D_2 + D_3 + D_6 + D_7$$

$$x = D_4 + D_5 + D_6 + D_7$$

TABLE 5-3
Truth Table of Octal-to-Binary Encoder

Inputs								Outputs		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	x	y	z
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

The encoder is implemented with three OR gates, as shown in Fig. 5-13.

The encoder defined in Table 5-3 has the limitation that only one input can be active at any given time. If two inputs are active simultaneously, the output produces an undefined combination. For example, if D_3 and D_6 are 1 simultaneously, the output of the encoder will be 111 because all three outputs are equal to 1. This does not represent binary 3 nor binary 6. To resolve this ambiguity, encoder circuits must establish a priority to ensure that only one input is encoded. If we establish a higher priority for inputs with higher subscript numbers, and if both D_3 and D_6 are 1 at the same time, the output will be 110 because D_6 has higher priority than D_3.

Another ambiguity in the octal-to-binary encoder is that an output with all 0's is generated when all the inputs are 0. The problem is that an output with all 0's is also generated when D_0 is equal to 1. This ambiguity can be resolved by providing an additional output that specifies the condition that none of the inputs are active.

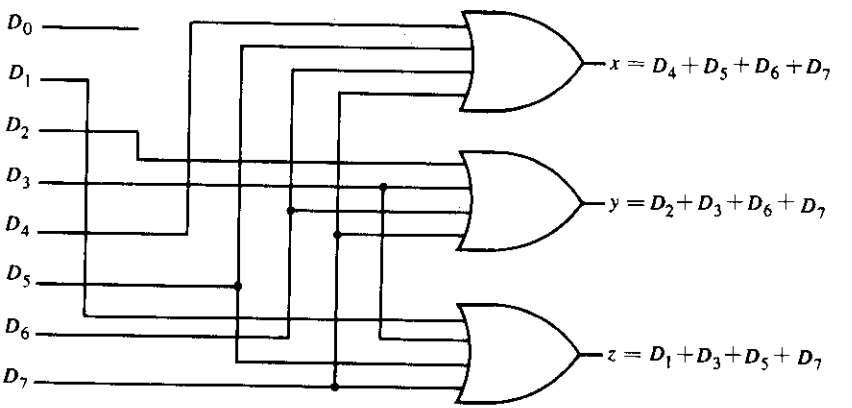


FIGURE 5-13
Octal-to-binary encoder

TABLE 5-4
Truth Table of a Priority Encoder

Inputs				Outputs		
D_0	D_1	D_2	D_3	x	y	V
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1

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. The truth table of a four-input priority encoder is given in Table 5-4. The X's are don't-care conditions that designate the fact that the binary value may be equal either to 0 or 1. Input D_3 has the highest priority; so regardless of the values of the other inputs, when this input is 1, the output for xy is 11 (binary 3). D_2 has the next priority level. The output is 10 if $D_2 = 1$ provided that $D_3 = 0$, regardless of the values of the other two lower-priority inputs. The output for D_1 is generated only if higher-priority inputs are 0, and so on down the priority level. A *valid*-output indicator, designated by V, is set to 1 only when one or more of the inputs are equal to 1. If all inputs are 0, V is equal to 0, and the other two outputs of the circuit are not used.

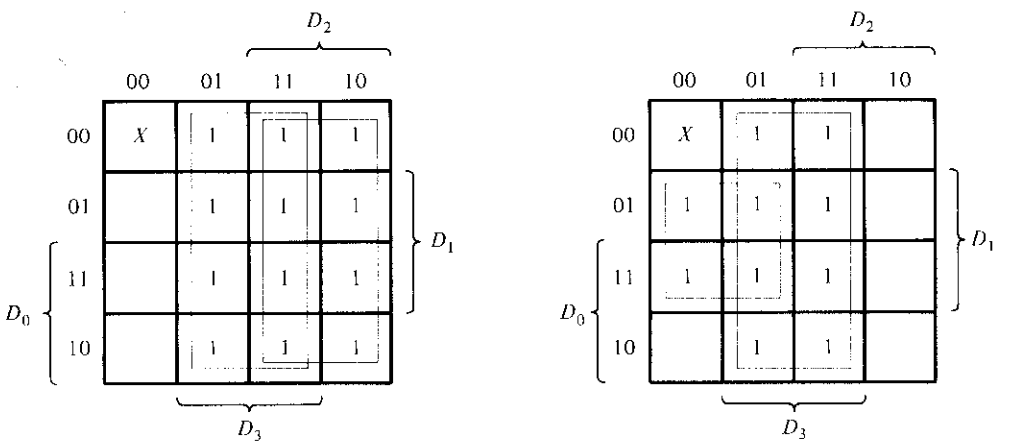


FIGURE 5-14
Maps for a priority encoder

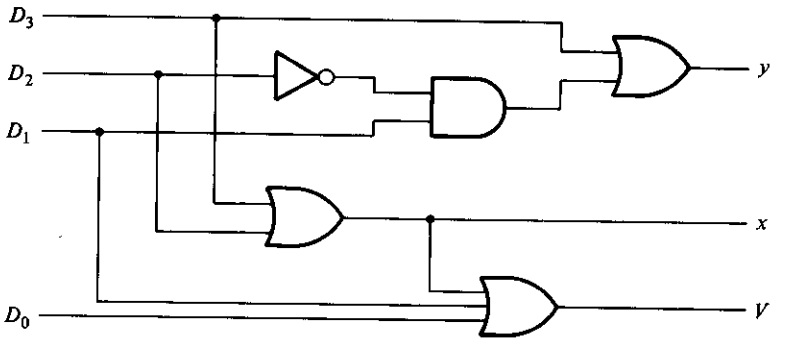


FIGURE 5-15
4-input priority encoder

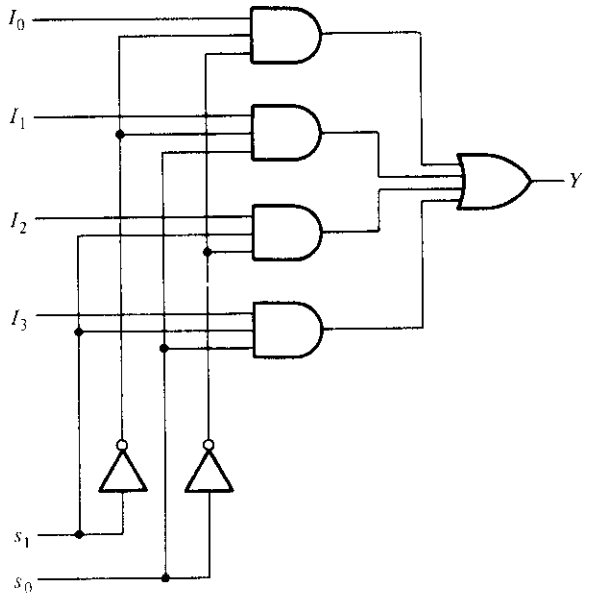
The maps for simplifying outputs x and y are shown in Fig. 5-14. The minterms for the two functions are derived from Table 5-4. Although the table has only five rows, when each don't-care condition is replaced first by 0 and then by 1, we obtain all 16 possible input combinations. For example, the third row in the table with $X100$ represents minterms 0100 and 1100 since X can be assigned either 0 or 1. The simplified Boolean expressions for the priority encoder are obtained from the maps. The condition for output V is an OR function of all the input variables. The priority encoder is implemented in Fig. 5-15 according to the following Boolean functions:

$$x = D_2 + D_3$$
$$y = D_3 + D_1 D_2'$$
$$V = D_0 + D_1 + D_2 + D_3$$

5-6 MULTIPLEXERS

Multiplexing means transmitting a large number of information units over a smaller number of channels or lines. A *digital multiplexer* is a combinational circuit that selects binary information from one of many input lines and directs it to a single output line. The selection of a particular input line is controlled by a set of selection lines. Normally, there are 2^n input lines and n selection lines whose bit combinations determine which input is selected.

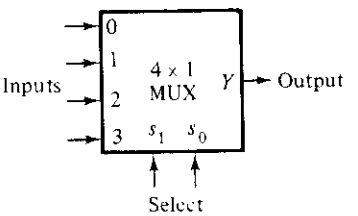
A 4-to-1-line multiplexer is shown in Fig. 5-16. Each of the four input lines, I_0 to I_3, is applied to one input of an AND gate. Selection lines s_1 and s_0 are decoded to select a particular AND gate. The function table, Fig. 5-16(b), lists the input-to-output path for each possible bit combination of the selection lines. When this MSI function is used in the design of a digital system, it is represented in block diagram form, as shown in Fig. 5-16(c). To demonstrate the circuit operation, consider the case when $s_1 s_0 = 10$. The AND gate associated with input I_2 has two of its inputs equal to 1 and the third input connected to I_2. The other three AND gates have at least one input equal to 0, which



(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



(c) Block diagram

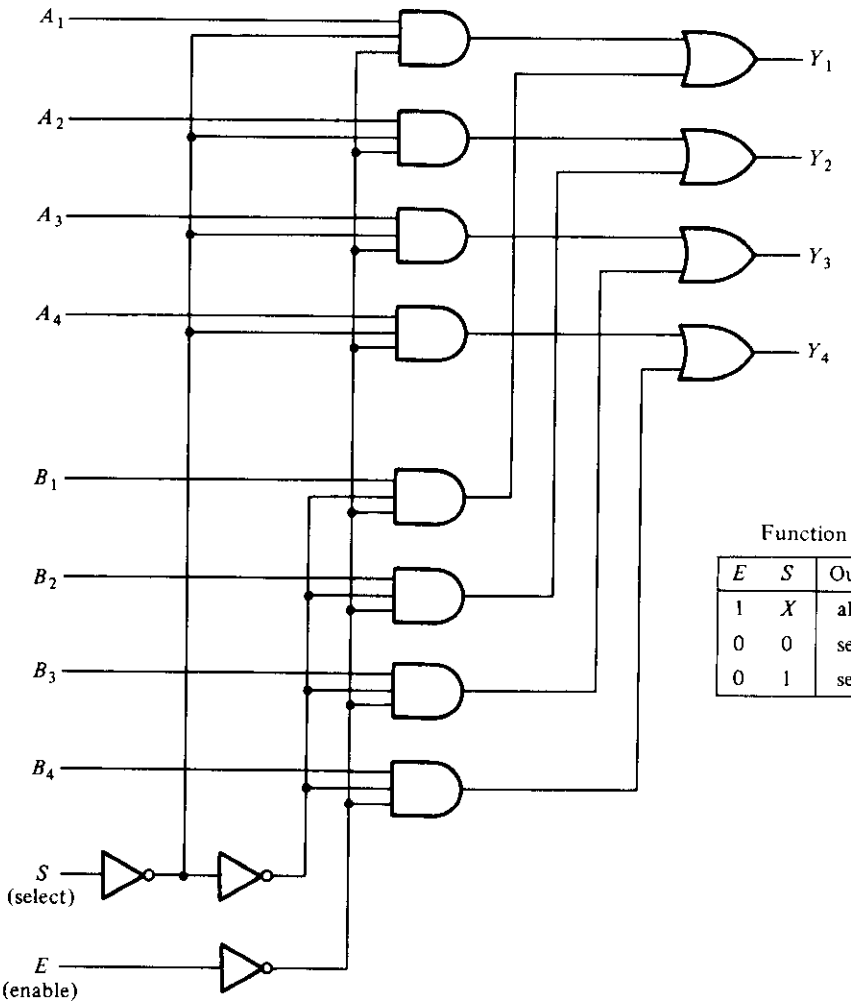
FIGURE 5-16
A 4-to-1-line multiplexer

makes their outputs equal to 0. The OR gate output is now equal to the value of I_2, thus providing a path from the selected input to the output. A multiplexer is also called a *data selector*, since it selects one of many inputs and steers the binary information to the output line.

The AND gates and inverters in the multiplexer resemble a decoder circuit and, indeed, they decode the input-selection lines. In general, a 2^n-to-1-line multiplexer 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 to provide the 1-line output. The size of a multiplexer is specified by the number 2^n of its input lines and the single output line. It is then implied that it also contains n selection lines. A multiplexer is often abbreviated as MUX.

As in decoders, multiplexer ICs may have an *enable* input to control the operation of the unit. When the enable input is in a given binary state, the outputs are disabled, and when it is in the other state (the enable state), the circuit functions as a normal multiplexer. The enable input (sometimes called *strobe*) can be used to expand two or more multiplexer ICs to a digital multiplexer with a larger number of inputs.

In some cases, two or more multiplexers are enclosed within one IC package. The selection and enable inputs in multiple-unit ICs may be common to all multiplexers. As an illustration, a quadruple 2-to-1-line multiplexer IC is shown in Fig. 5-17. It has four multiplexers, each capable of selecting one of two input lines. Output Y_1 can be selected



Function table

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

FIGURE 5-17
Quadruple 2-to-1-line multiplexer

to be equal to either A_1 or B_1. Similarly, output Y_2 may have the value of A_2 or B_2, and so on. One input selection line, S, suffices to select one of two lines in all four multiplexers. The control input E enables the multiplexers in the 0 state and disables them in the 1 state. Although the circuit contains four multiplexers, we may think of it as a circuit that selects one in a pair of 4-input lines. As shown in the function table, the unit is selected when $E = 0$. Then, if $S = 0$, the four A inputs have a path to the outputs. On the other hand, if $S = 1$, the four B inputs are selected. The outputs have all 0's when $E = 1$, regardless of the value of S.

Boolean-Function Implementation

It was shown in the previous section that a decoder can be used to implement a Boolean function by employing an external OR gate. A quick reference to the multiplexer of Fig. 5-16 reveals that it is essentially a decoder with the OR gate already available. The minterms out of the decoder to be chosen can be controlled with the input lines. The minterms to be included with the function being implemented are chosen by making their corresponding input lines equal to 1; those minterms not included in the function are disabled by making their input lines equal to 0. This gives a method for implementing any Boolean function of n variables with a 2^n-to-1 multiplexer. However, it is possible to do better than that.

If we have a Boolen function of $n + 1$ variables, we take n of these variables and connect them to the selection lines of a multiplexer. The remaining single variable of the function is used for the inputs of the multiplexer. If A is this single variable, the inputs of the multiplexer are chosen to be either A or A' or 1 or 0. By judicious use of these four values for the inputs and by connecting the other variables to the selection lines, one can implement any Boolean function with a multiplexer. In this way, it is possible to generate any function of $n + 1$ variables with a 2^n-to-1 multiplexer.

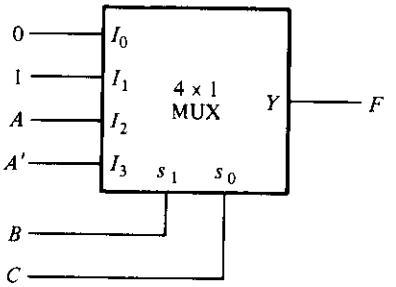
To demonstrate this procedure with a concrete example, consider the function of three variables:

$$F(A, B, C) = \Sigma(1, 3, 5, 6)$$

The function can be implemented with a 4-to-1 multiplexer, as shown in Fig. 5-18. Two of the variables, B and C, are applied to the selection lines in that order, i.e., B is connected to s_1 and C to s_0. The inputs of the multiplexer are 0, 1, A, and A'. When $BC = 00$, output $F = 0$ since $I_0 = 0$. Therefore, both minterms $m_0 = A'B'C'$ and $m_4 = AB'C'$ produce a 0 output, since the output is 0 when $BC = 00$ regardless of the value of A. When $BC = 01$, output $F = 1$, since $I_1 = 1$. Therefore, both minterms $m_1 = A'B'C$ and $m_5 = AB'C$ produce a 1 output, since the output is 1 when $BC = 01$ regardless of the value of A. When $BC = 10$, input I_2 is selected. Since A is connected to this input, the output will be equal to 1 only for minterm $m_6 = ABC'$, but not for minterm $m_2 = A'BC'$, because when $A' = 1$, then $A = 0$, and since $I_2 = 0$, we have $F = 0$. Finally, when $BC = 11$, input I_3 is selected. Since A' is connected to this input, the output will be equal to 1 only for minterm $m_3 = A'BC$, but not for $m_7 = ABC$. This information is summarized in Fig. 5-18(b), which is the truth table of the function we want to implement.

This discussion shows by analysis that the multiplexer implements the required function. We now present a general procedure for implementing any Boolean function of n variables with a 2^{n-1}-to-1 multiplexer.

First, express the function in its sum of minterms form. Assume that the ordered sequence of variables chosen for the minterms is $ABCD \ldots$, where A is the leftmost variable in the ordered sequence of n variables and $BCD \ldots$ are the remaining $n - 1$ variables. Connect the $n - 1$ variables to the selection lines of the multiplexer, with B connected to the high-order selection line, C to the next lower selection line, and so on



(a) Multiplexer implementation

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

(b) Truth table

	I_0	I_1	I_2	I_3
A'	0	①	2	③
A	4	⑤	⑥	7
	0	1	A	A'

(c) Implementation table

FIGURE 5-18
Implementing $F(A, B, C) = \sum (1, 3, 5, 6)$ with a multiplexer

down to the last variable, which is connected to the lowest-order selection line s_0. Consider now the single variable A. Since this variable is in the highest-order position in the sequence of variables, it will be complemented in minterms 0 to $(2^n/2) - 1$, which comprise the first half in the list of minterms. The second half of the minterms will have their A variable uncomplemented. For a three-variable function, A, B, C, we have eight minterms. Variable A is complemented in minterms 0 to 3 and uncomplemented in minterms 4 to 7.

List the inputs of the multiplexer and under them list all the minterms in two rows. The first row lists all those minterms where A is complemented, and the second row all the minterms with A uncomplemented, as shown in Fig. 5-18(c). Circle all the minterms of the function and inspect each column separately.

If the two minterms in a column are not circled, apply 0 to the corresponding multiplexer input.

If the two minterms are circled, apply 1 to the corresponding multiplexer input.

If the bottom minterm is circled and the top is not circled, apply A to the corresponding multiplexer input.

If the top minterm is circled and the bottom is not circled, apply A' to the corresponding multiplexer input.

This procedure follows from the conditions established during the previous analysis.

Figure 5-18(c) shows the implementation table for the Boolean function

$$F(A, B, C) = \Sigma(1, 3, 5, 6)$$

from which we obtain the multiplexer connections of Fig. 5-18(a). Note that B must be connected to s_1 and C to s_0.

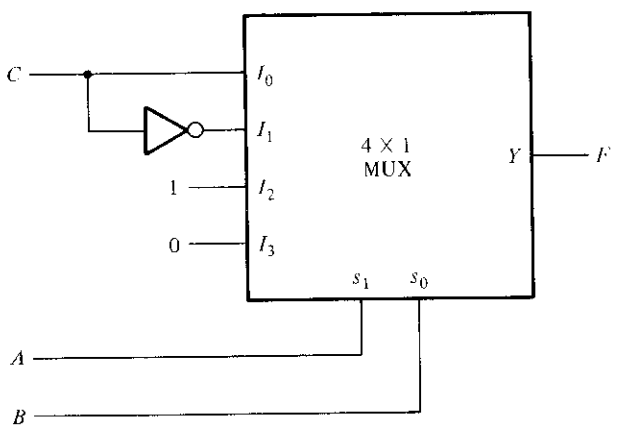
It is not necessary to choose the leftmost variable in the ordered sequence of a variable list for the data inputs of the multiplexer. In fact, any one of the variables can be chosen for the inputs, provided we modify the multiplexer implementation table. Moreover, it is possible to derive the multiplexer circuit directly from the truth table. Consider, for example, the following three-variable Boolean function:

$$F(A, B, C) = \Sigma(1, 2, 4, 5)$$

We wish to implement the function with a multiplexer, but in this case, we will connect variables A and B to selection inputs s_1 and s_0, respectively, and use the rightmost vari-

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

FIGURE 5-19
Implementing $F(A, B, C) = \sum (1, 2, 4, 5)$ with a multiplexer

able C for the data inputs of the multiplexer. Figure 5-19(a) is the truth table of the function. The table is divided into sections, with each section having identical values for variables A and B. We note that when $AB = 00$, output F is the same as input C. When $AB = 01$, F is the same as C'. When $AB = 10$, $F = 1$, and when $AB = 11$, $F = 0$. The multiplexer circuit of Fig. 5-19(b) can be derived directly from the truth table without the need of an implementation table. However, if an implementation table is desired, it must be modified to take into account the relationship between the minterms and the inputs of the multiplexer. As seen from the truth table, variable C is complemented in the even-numbered minterms 0, 2, 4, and 6, and uncomplemented in the odd-numbered minterms 1, 3, 5, and 7. The arrangement of the two rows in the implementation table must be as shown in Fig. 5-19(c). By circling the minterms of the function and using the rules stated before, we obtain the multiplexer inputs for implementing the function.

In a similar fashion, it is possible to choose any other variable of the function for the multiplexer data inputs. In any case, all input variables except one are applied to the selection inputs of the multiplexer. The remaining single variable, or its complement, or 0, or 1, is then applied to the data inputs of the multiplexer.

Example 5-2

Implement the following function with a multiplexer:

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

This is a four-variable function and, therefore, we need a multiplexer with three selection lines and eight inputs. We choose to apply variables B, C, and D to the selection lines. The implementation table is then as shown in Fig. 5-20. The first half of the

	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

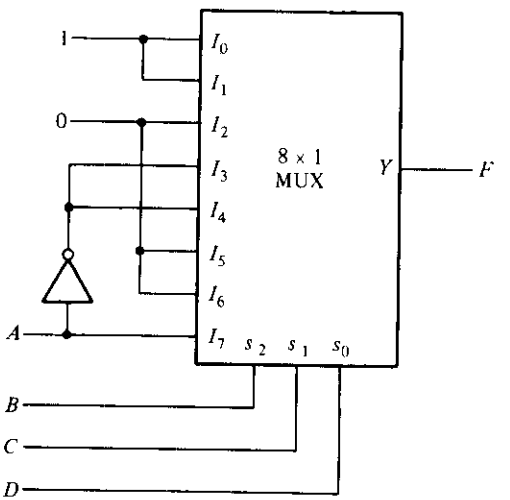


FIGURE 5-20
Implementing $F(A, B, C, D) = \sum (0, 1, 3, 4, 8, 9, 15)$

minterms are associated with A' and the second half with A. By circling the minterms of the function and applying the rules for finding values for the multiplexer inputs, we obtain the implementation shown. ■

Let us now compare the multiplexer method with the decoder method for implementing combinational circuits. The decoder method requires an OR gate for each output function, but only one decoder is needed to generate all minterms. The multiplexer method uses smaller-size units but requires one multiplexer for each output function. It would seem reasonable to assume that combinational circuits with a small number of outputs should be implemented with multiplexers. Combinational circuits with many output functions would probably use fewer ICs with the decoder method.

Although multiplexers and decoders may be used in the implementation of combinational circuits, it must be realized that decoders are mostly used for decoding binary information and multiplexers are mostly used to form a selected path between multiple sources and a single destination.

5-7 READ-ONLY MEMORY (ROM)

We saw in Section 5-5 that a decoder generates the 2^n minterms of the n input variables. By inserting OR gates to sum the minterms of Boolean functions, we were able to generate any desired combinational circuit. A read-only memory (ROM) is a device that includes both the decoder and the OR gates within a single IC package. The connections between the outputs of the decoder and the inputs of the OR gates can be specified for each particular configuration. The ROM is used to implement complex combinational circuits within one IC package or as permanent storage for binary information.

A ROM is essentially a memory (or storage) 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. ROMs come with special internal electronic fuses that can be "programmed" for a specific configuration. Once the pattern is established, it stays within the unit even when power is turned off and on again.

A block diagram of a ROM is shown in Fig. 5-21. It consists of n input lines and m output lines. Each bit combination of the input variables is called an *address*. Each bit combination that comes out of the output lines is called a *word*. The number of bits per word is equal to the number of output lines, m. An address is essentially a binary number that denotes one of the minterms of n variables. The number of distinct addresses possible with n input variables is 2^n. An output word can be selected by a unique address, and since there are 2^n distinct addresses in a ROM, there are 2^n distinct words that are said to be stored in the unit. The word available on the output lines at any given time depends on the address value applied to the input lines. A ROM is charac-

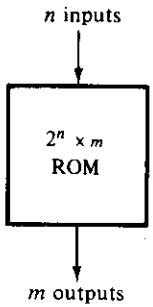


FIGURE 5-21
ROM block diagram

terized by the number of words 2^n and the number of bits per word m. This terminology is used because of the similarity between the read-only memory and the random-access memory, which is presented in Section 7-7.

Consider a 32 × 8 ROM. The unit consists of 32 words of 8 bits each. This means that there are eight output lines and that there are 32 distinct words stored in the unit, each of which may be applied to the output lines. The particular word selected that is presently available on the output lines is determined from the five input lines. There are only five inputs in a 32 × 8 ROM because $2^5 = 32$, and with five variables, we can specify 32 addresses or minterms. For each address input, there is a unique selected word. Thus, if the input address is 00000, word number 0 is selected and it appears on the output lines. If the input address is 11111, word number 31 is selected and applied to the output lines. In between, there are 30 other addresses that can select the other 30 words.

The number of addressed words in a ROM is determined from the fact that n input lines are needed to specify 2^n words. A ROM is sometimes specified by the total number of bits it contains, which is $2^n \times m$. For example, a 2048-bit ROM may be organized as 512 words of 4 bits each. This means that the unit has four output lines and nine input lines to specify $2^9 = 512$ words. The total number of bits stored in the unit is $512 \times 4 = 2048$.

Internally, the ROM is a combinational circuit with AND gates connected as a decoder and a number of OR gates equal to the number of outputs in the unit. Figure 5-22 shows the internal logic construction of a 32 × 4 ROM. The five input variables are decoded into 32 lines by means of 32 AND gates and 5 inverters. Each output of the decoder represents one of the minterms of a function of five variables. Each one of the 32 addresses selects one and only one output from the decoder. The address is a 5-bit number applied to the inputs, and the selected minterm out of the decoder is the one marked with the equivalent decimal number. The 32 outputs of the decoder are connected through fuses to each OR gate. Only four of these fuses are shown in the diagram, but actually each OR gate has 32 inputs and each input goes through a fuse that can be blown as desired.

The ROM is a two-level implementation in sum of minterms form. It does not have to be an AND-OR implementation, but it can be any other possible two-level minterm

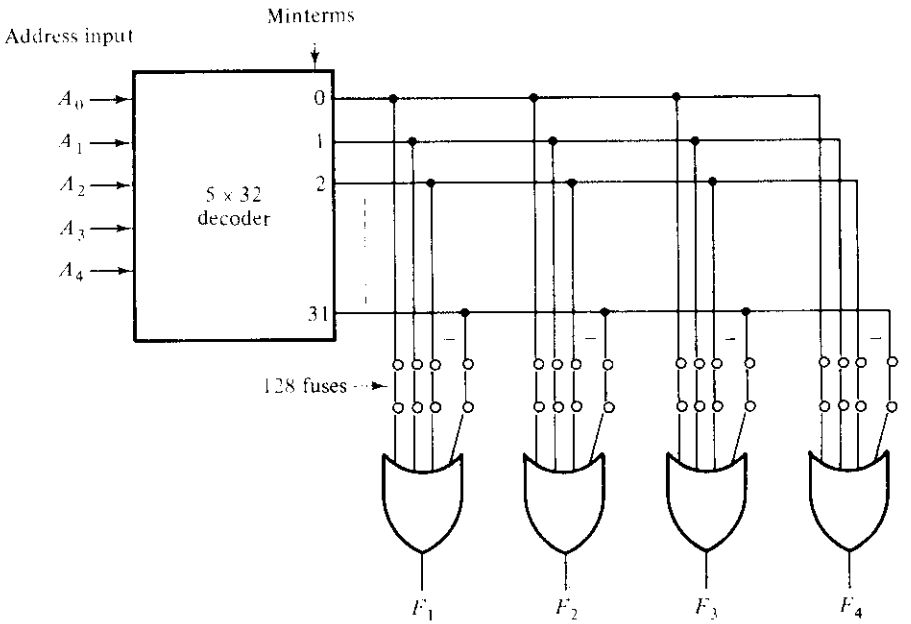


FIGURE 5-22
Logic construction of a 32 × 4 ROM

implementation. The second level is usually a wired-logic connection (see Section 3-7) to facilitate the blowing of fuses.

ROMs have many important applications in the design of digital computer systems. Their use for implementing complex combinational circuits is just one of these applications. Other uses of ROMs are presented in other parts of the book in conjunction with their particular applications.

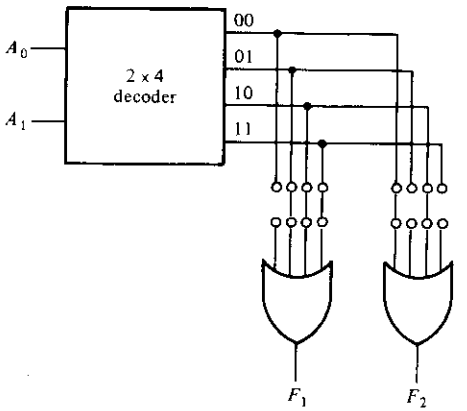
Combinational Logic Implementation

From the logic diagram of the ROM, it is clear that each output provides the sum of all the minterms of the n input variables. Remember that any Boolean function can be expressed in sum of minterms form. By breaking the links of those minterms not included in the function, each ROM output can be made to represent the Boolean function of one of the output variables in the combinational circuit. For an n-input, m-output combinational circuit, we need a $2^n \times m$ ROM. The blowing of the fuses is referred to as *programming* the ROM. The designer need only specify a ROM program table that gives the information for the required paths in the ROM. The actual programming is a hardware procedure that follows the specifications listed in the program table.

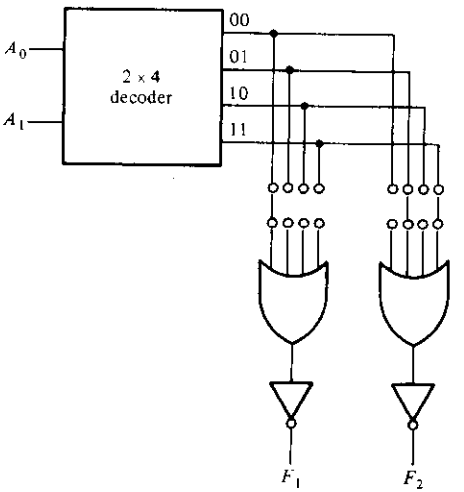
Let us clarify the process with a specific example. The truth table in Fig. 5-23(a) specifies a combinational circuit with two inputs and two outputs. The Boolean functions can be expressed in sum of minterms:

A_1	A_0	F_1	F_2
0	0	0	1
0	1	1	0
1	0	1	1
1	1	1	0

(a) Truth table



(b) ROM with AND-OR gates



(c) ROM with AND-OR-INVERT gates

FIGURE 5-23
Combinational-circuit implementation with a 4 × 2 ROM

$$F_1(A_1, A_0) = \Sigma(1, 2, 3)$$
$$F_2(A_1, A_0) = \Sigma(0, 2)$$

When a combinational circuit is implemented by means of a ROM, the functions must be expressed in sum of minterms or, better yet, by a truth table. If the output functions are simplified, we find that the circuit needs only one OR gate and an inverter. Obviously, this is too simple a combinational circuit to be implemented with a ROM. The advantage of a ROM is in complex combinational circuits. This example merely demonstrates the procedure and should not be considered in a practical situation.

The ROM that implements the combinational circuit must have two inputs and two outputs; so its size must be 4 × 2. Figure 5-23(b) shows the internal construction of such a ROM. It is now necessary to determine which of the eight available fuses must be blown and which should be left intact. This can be easily done from the output functions listed in the truth table. Those minterms that specify an output of 0 should not have a path to the output through the OR gate. Thus, for this particular case, the truth table shows three 0's, and their corresponding fuses to the OR gates must be blown. It

is obvious that we must assume here that an open input to an OR gate behaves as a 0 input.

Some ROM units come with an inverter after each of the OR gates and, as a consequence, they are specified as having initially all 0's at their outputs. The programming procedure in such ROMs requires that we open the paths of the minterms (or addresses) that specify an output of 1 in the truth table. The output of the OR gate will then generate the complement of the function, but the inverter placed after the OR gate complements the function once more to provide the normal output. This is shown in the ROM of Fig. 5-23(c).

The previous example demonstrates the general procedure for implementing any combinational circuit with a ROM. From the number of inputs and outputs in the combinational circuit, we first determine the size of ROM required. Then we must obtain the programming truth table of the ROM; no other manipulation or simplification is required. The 0's (or 1's) in the output functions of the truth table directly specify those fuses that must be blown to provide the required combinational circuit in sum of minterms form.

In practice, when one designs a circuit by means of a ROM, it is not necessary to show the internal gate connections of fuses inside the unit, as was done in Fig. 5-23. This was shown there for demonstration purposes only. All the designer has to do is specify the particular ROM (or its designation number) and provide the ROM truth table, as in Fig. 5-23(a). The truth table gives all the information for programming the ROM. No internal logic diagram is needed to accompany the truth table.

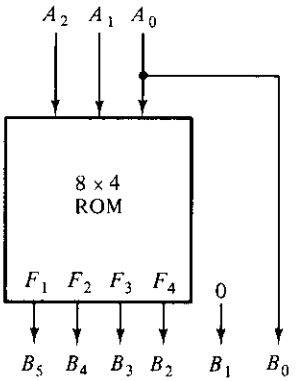
Example 5-3

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

The first step is to derive the truth table for the combinational circuit. In most cases, this is all that is needed. In some cases, we can fit a smaller truth table for the ROM by using certain properties in the truth table of the combinational circuit. Table 5-5 is the

TABLE 5-5
Truth Table for Circuit of Example 5-3

Inputs			Outputs						
A_2	A_1	A_0	B_5	B_4	B_3	B_2	B_1	B_0	Decimal
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



(a) Block diagram

A_2	A_1	A_0	F_1	F_2	F_3	F_4
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	1
0	1	1	0	0	1	0
1	0	0	0	1	0	0
1	0	1	0	1	1	0
1	1	0	1	0	0	1
1	1	1	1	1	0	0

(b) ROM truth table

FIGURE 5-24
ROM implementation of Example 5-3

truth table for the combinational circuit. Three inputs and six outputs are needed to accommodate all possible numbers. We note that output B_0 is always equal to input A_0; so there is no need to generate B_0 with a ROM since it is equal to an input variable. Moreover, output B_1 is always 0, so this output is always known. We actually need to generate only four outputs with the ROM; the other two are easily obtained. The minimum-size ROM needed must have three inputs and four outputs. Three inputs specify eight words, so the ROM size must be 8 × 4. The ROM implementation is shown in Fig. 5-24. The three inputs specify eight words of four bits each. The other two outputs of the combinational circuit are equal to 0 and A_0. The truth table in Fig. 5-24 specifies all the information needed for programming the ROM, and the block diagram shows the required connections. ■

Types of ROMs

The required paths in a ROM may be programmed in two different ways. The first is called *mask programming* and is done by the manufacturer during the last fabrication process of the unit. The procedure for fabricating a ROM requires that the customer fill out the truth table the ROM is to satisfy. The truth table may be submitted on a special form provided by the manufacturer. More often, it is submitted in a computer input medium in the format specified on the data sheet of the particular ROM. The manufacturer makes the corresponding mask for the paths to produce the 1's and 0's according to the customer's truth table. This procedure is costly because the vendor charges the customer a special fee for custom masking a ROM. For this reason, mask programming is economical only if large quantities of the same ROM configuration are to be manufactured.

For small quantities, it is more economical to use a second type of ROM called a *programmable read-only memory*, or PROM. When ordered, PROM units contain all 0's (or all 1's) in every bit of the stored words. The fuses in the PROM are blown by application of current pulses through the output terminals. A blown fuse defines one binary state and an unbroken link represents the other state. This allows the user to program the unit in the laboratory to achieve the desired relationship between input addresses and stored words. Special units called *PROM programmers* are available commercially to facilitate this procedure. In any case, all procedures for programming ROMs are *hardware* procedures even though the word *programming* is used.

The hardware procedure for programming ROMs or PROMs is irreversible and, once programmed, the fixed pattern is permanent and cannot be altered. Once a bit pattern has been established, the unit must be discarded if the bit pattern is to be changed. A third type of unit available is called *erasable PROM*, or EPROM. EPROMs can be restructured to the initial value (all 0's or all 1's) even though they have been changed previously. When an EPROM is placed under a special ultraviolet light for a given period of time, the shortwave radiation discharges the internal gates that serve as contacts. After erasure, the ROM returns to its initial state and can be reprogrammed. Certain ROMs can be erased with electrical signals instead of ultraviolet light, and these are called *electrically erasable PROMs*, or EEPROMs.

The function of a ROM can be interpreted in two different ways. The first interpretation is of a unit that implements any combinational circuit. From this point of view, each output terminal is considered separately as the output of a Boolean function expressed in sum of minterms. The second interpretation considers the ROM to be a storage unit having a fixed pattern of bit strings called *words*. From this point of view, the inputs specify an *address* to a specific stored word, which is then applied to the outputs. For example, the ROM of Fig. 5-24 has three address lines, which specify eight stored words as given by the truth table. Each word is four bits long. This is the reason why the unit is given the name *read-only memory*. *Memory* is commonly used to designate a storage unit. *Read* is commonly used to signify that the contents of a word specified by an address in a storage unit is placed at the output terminals. Thus, a ROM is a memory unit with a fixed word pattern that can be read out upon application of a given address. The bit pattern in the ROM is permanent and cannot be changed during normal operation.

ROMs are widely used to implement complex combinational circuits directly from their truth tables. They are useful for converting from one binary code to another (such as ASCII to EBCDIC and vice versa), for arithmetic functions such as multipliers, for display of characters in a cathode-ray tube, and in many other applications requiring a large number of inputs and outputs. They are also employed in the design of control units of digital systems. As such, they are used to store fixed bit patterns that represent the sequence of control variables needed to enable the various operations in the system. A control unit that utilizes a ROM to store binary control information is called a *microprogrammed control unit*.

5-8 PROGRAMMABLE LOGIC ARRAY (PLA)

A combinational circuit may occasionally have don't-care conditions. When implemented with a ROM, a don't-care condition becomes an address input that will never occur. The words at the don't-care addresses need not be programmed and may be left in their original state (all 0's or all 1's). The result is that not all the bit patterns available in the ROM are used, which may be considered a waste of available equipment.

Consider, for example, a combinational circuit that converts a 12-bit card code to a 6-bit internal alphanumeric code (see end of Section 1-7). The input card code consists of 12 lines designated by 0, 1, 2, . . . , 9, 11, 12. The size of the ROM for implementing the code converter must be 4096 × 6, since there are 12 inputs and 6 outputs. There are only 47 valid entries for the card code; all other input combinations are don't-care conditions. Thus, only 47 words of the 4096 available are used. The remaining 4049 words of ROM are not used and are thus wasted.

For cases where the number of don't-care conditions is excessive, it is more economical to use a second type of LSI component called a *programmable logic array*, or PLA. A PLA is similar to a ROM in concept; however, the PLA does not provide full decoding of the variables and does not generate all the minterms as in the ROM. In the PLA, the decoder is replaced by a group of AND gates, each of which can be programmed to generate a product term of the input variables. The AND and OR gates inside the PLA are initially fabricated with fuses among them. The specific Boolean functions are implemented in sum of products form by blowing appropriate fuses and leaving the desired connections.

A block diagram of the PLA is shown in Fig. 5-25. It consists of n inputs, m outputs, k product terms, and m sum terms. The product terms constitute a group of k AND gates and the sum terms constitute a group of m OR gates. Fuses are inserted between all n inputs and their complement values to each of the AND gates. Fuses are also provided between the outputs of the AND gates and the inputs of the OR gates. Another set of fuses in the output inverters allows the output function to be generated either in the AND-OR form or in the AND-OR-INVERT form. With the inverter fuse in place, the inverter is bypassed, giving an AND-OR implementation. With the fuse blown, the inverter becomes part of the circuit and the function is implemented in the AND-OR-INVERT form.

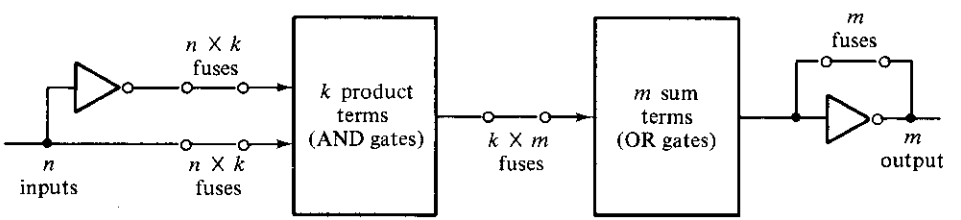


FIGURE 5-25
PLA block diagram

The size of the PLA is specified by the number of inputs, the number of product terms, and the number of outputs (the number of sum terms is equal to the number of outputs). A typical PLA has 16 inputs, 48 product terms, and 8 outputs. The number of programmed fuses is $2n \times k + k \times m + m$, whereas that of a ROM is $2^n \times m$.

Figure 5-26 shows the internal construction of a specific PLA. It has three inputs, three product terms, and two outputs. Such a PLA is too small to be available commercially; it is presented here merely for demonstration purposes. Each input and its complement are connected through fuses to the inputs of all AND gates. The outputs of the AND gates are connected through fuses to each input of the OR gates. Two more fuses are provided with the output inverters. By blowing selected fuses and leaving others intact, it is possible to implement Boolean functions in their sum of products form.

As with a ROM, the PLA may be mask-programmable or field-programmable. With a mask-programmable PLA, the customer must submit a PLA program table to the manufacturer. This table is used by the vendor to produce a custom-made PLA that has the required internal paths between inputs and outputs. A second type of PLA available is called a *field-programmable logic array*, or FPLA. The FPLA can be programmed by the user by means of certain recommended procedures. Commercial hardware programmer units are available for use in conjunction with certain FPLAs.

PLA Program Table

The use of a PLA must be considered for combinational circuits that have a large number of inputs and outputs. It is superior to a ROM for circuits that have a large number of don't-care conditions. The example to be presented demonstrates how a PLA is programmed. Bear in mind when going through the example that such a simple circuit will not require a PLA because it can be implemented more economically with SSI gates.

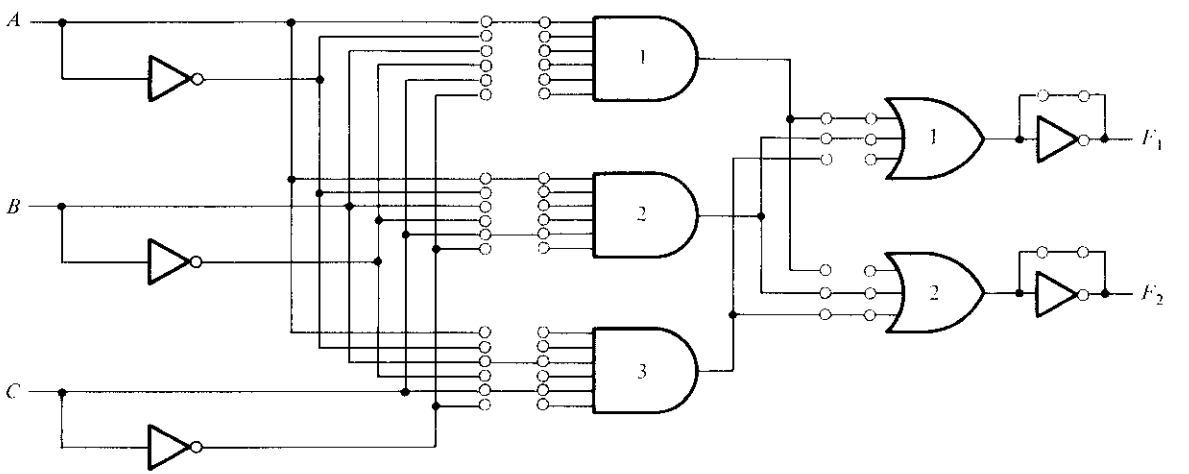
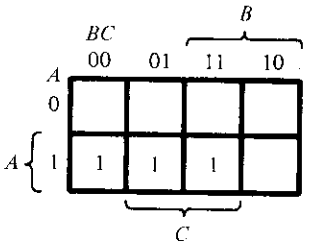


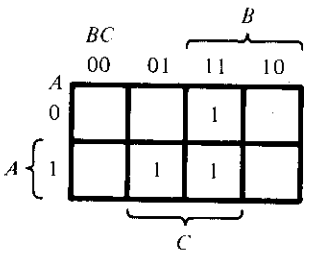
FIGURE 5-26
PLA with three inputs, three product terms, and two outputs; it implements the combinational circuit specified in Fig. 5-27

A	B	C	F_1	F_2
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	0	1
1	0	0	1	0
1	0	1	1	1
1	1	0	0	0
1	1	1	1	1

(a) Truth table



$F_1 = AB' + AC$



$F_2 = AC + BC$

(b) Map simplification

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

(c) PLA program table

FIGURE 5-27
Steps required in PLA implementation

Consider the truth table of the combinational circuit, shown in Fig. 5-27(a). Although a ROM implements a combinational circuit in its sum of minterms form, a PLA implements the functions in their sum of products form. Each product term in the expression requires an AND gate. Since the number of AND gates in a PLA is finite, it is necessary to simplify the function to a minimum number of product terms in order to minimize the number of AND gates used. The simplified functions in sum of products are obtained from the maps of Fig. 5-27(b):

$$F_1 = AB' + AC$$

$$F_2 = AC + BC$$

There are three distinct product terms in this combinational circuit: AB', AC, and BC. The circuit has three inputs and two outputs; so the PLA of Fig. 5-26 can be used to implement this combinational circuit.

Programming the PLA means that we specify the paths in its AND-OR-NOT pattern. A typical PLA program table is shown in Fig. 5-27(c). It consists of three columns. The first column lists the product terms numerically. The second column specifies the required paths between inputs and AND gates. The third column specifies the paths between the AND gates and the OR gates. Under each output variable, we write a T (for true) if the output inverter is to be bypassed, and C (for complement) if the function is to be complemented with the output inverter. The Boolean terms listed at the left are not part of the table; they are included for reference only.

For each product term, the inputs are marked with 1, 0, or – (dash). If a variable in the product term appears in its normal form (unprimed), the corresponding input variable is marked with a 1. If it appears complemented (primed), the corresponding input variable is marked with a 0. If the variable is absent in the product term, it is marked with a dash. Each product term is associated with an AND gate. The paths between the inputs and the AND gates are specified under the column heading *inputs*. A 1 in the input column specifies a path from the corresponding input to the input of the AND gate that forms the product term. A 0 in the input column specifies a path from the corresponding complemented input to the input of the AND gate. A dash specifies no connection. The appropriate fuses are blown and the ones left intact form the desired paths, as shown in Fig. 5-26. It is assumed that the open terminals in the AND gate behave like a 1 input.

The paths between the AND and OR gates are specified under the column heading *outputs*. The output variables are marked with 1's for all those product terms that formulate the function. In the example of Fig. 5-27, we have

$$F_1 = AB' + AC$$

so F_1 is marked with 1's for product terms 1 and 2 and with a dash for product term 3. Each product term that has a 1 in the output column requires a path from the corresponding AND gate to the output OR gate. Those marked with a dash specify no connection. Finally, a T (true) output dictates that the fuse across the output inverter remains intact, and a C (complement) specifies that the corresponding fuse be blown. The internal paths of the PLA for this circuit are shown in Fig. 5-26. It is assumed that an open terminal in an OR gate behaves like a 0, and that a short circuit across the output inverter does not damage the circuit.

When designing a digital system with a PLA, there is no need to show the internal connections of the unit, as was done in Fig. 5-26. All that is needed is a PLA program table from which the PLA can be programmed to supply the appropriate paths.

When implementing a combinational circuit with PLA, careful investigation must be undertaken in order to reduce the total number of distinct product terms, since a given PLA would have a finite number of AND terms. This can be done by simplifying each function to a minimum number of terms. The number of literals in a term is not important since we have available all input variables. Both the true value and the complement of the function should be simplified to see which one can be expressed with fewer product terms and which one provides product terms that are common to other functions.

Example 5-4

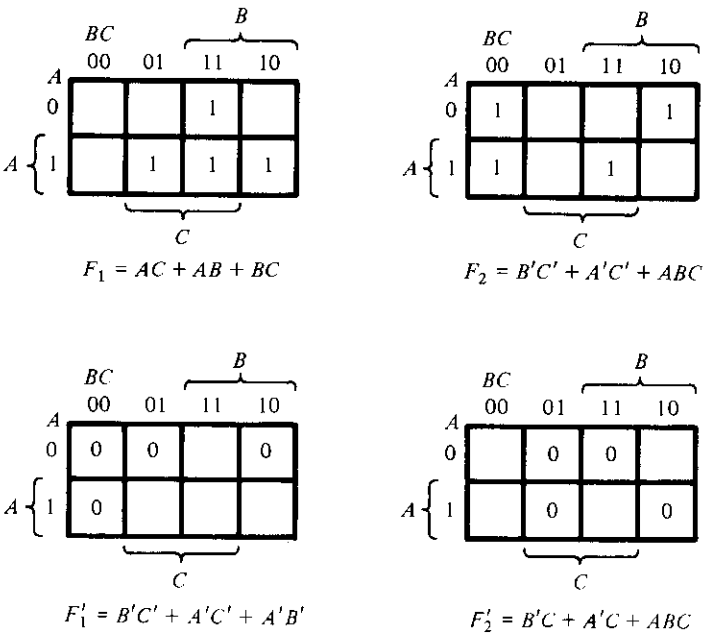
A combinational circuit is defined by the functions

$$F_1(A, B, C) = \Sigma(3, 5, 6, 7)$$
$$F_2(A, B, C) = \Sigma(0, 2, 4, 7)$$

Implement the circuit with a PLA having three inputs, four product terms, and two outputs.

The two functions are simplified in the maps of Fig. 5-28. Both the true values and the complements of the functions are simplified. The combinations that give a minimum number of product terms are

$$F_1 = (B'C' + A'C' + A'B')'$$
$$F_2 = B'C' + A'C' + ABC$$



PLA program table

	Product term	Inputs A	B	C	Outputs F_1	F_2
$B'C'$	1	–	0	0	1	1
$A'C'$	2	0	–	0	1	1
$A'B'$	3	0	0	–	1	–
ABC	4	1	1	1	–	1
					C	T	T/C

FIGURE 5-28
Solution to Example 5-4

This gives only four distinct product terms: $B'C'$, $A'C'$, $A'B'$, and ABC. The PLA program table for this combination is shown in Fig. 5-28. Note that output F_1 is the normal (or true) output even though a C is marked under it. This is because F_1' is generated *prior* to the output inverter. The inverter complements the function to produce F_1 in the output. ■

The combinational circuit for this example is too small for practical implementation with a PLA. It was presented here merely for demonstration purposes. A typical commercial PLA would have over 10 inputs and about 50 product terms. The simplification of Boolean functions with so many variables should be carried out by means of a tabulation method or other computer-assisted simplification method. This is where a computer program may aid in the design of complex digital systems. The computer program should simplify each function of the combinational circuit and its complement to a minimum number of terms. The program then selects a minimum number of distinct terms that cover all functions in their true or complement form.

5-9 PROGRAMMABLE ARRAY LOGIC (PAL)

Programmable logic devices have hundreds of gates interconnected through hundreds of electronic fuses. It is sometimes convenient to draw the internal logic of such devices in a compact form referred to as *array logic*. Figure 5-29 shows the conventional and array logic symbols for a multiple-input AND gate. The conventional symbol is drawn with multiple lines showing the fuses connected to the inputs of the gate. The corresponding array logic symbol uses a single horizontal line connected to the gate input and multiple vertical lines to indicate the individual inputs. Each intersection between a vertical line and the common horizontal line has a fused connection. Thus, in Fig. 5-29(b), the AND gate has four inputs connected through fuses. In a similar fashion, we can draw the array logic for the OR gate or any other type of multiple-input gate.

The programmable array logic (PAL) is a programmable logic device with a fixed OR array and a programmable AND array. Because only the AND gates are programmable, the PAL is easier to program, but is not as flexible as the PLA. Figure 5-30 shows the array logic configuration of a typical PAL. It has four inputs and four out-

(a) Conventional symbol (b) Array logic symbol

FIGURE 5-29
Two graphic symbols for an AND gate

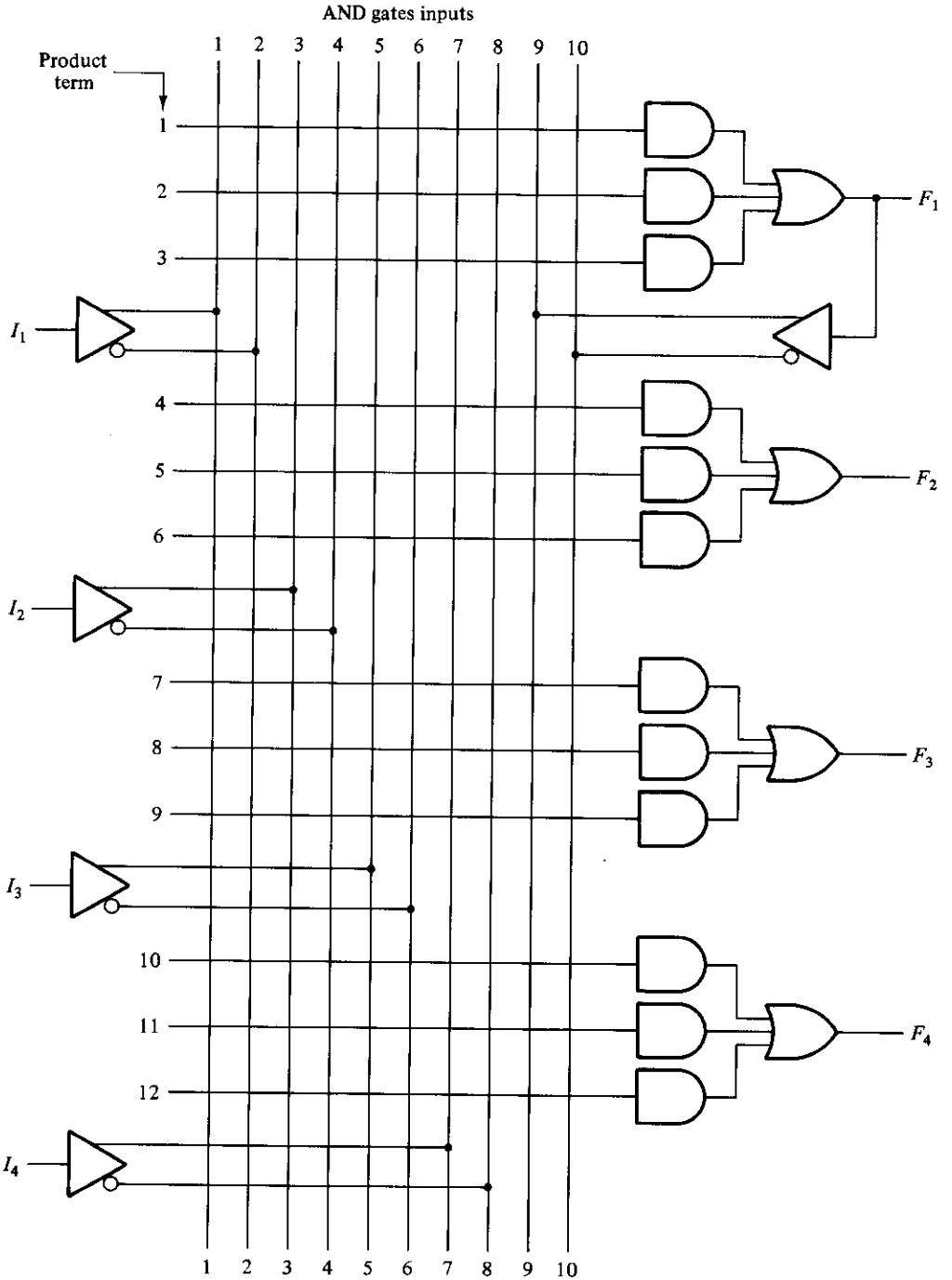


FIGURE 5-30
PAL with four inputs, four outputs, and three-wide AND-OR structure

puts. Each input has a buffer and an inverter gate. Note that the two gates are shown with one composite graphic symbol with normal and complement outputs. There are four sections in the unit, each being composed of a threewide AND-OR array. This is the term used to indicate that there are three programmable AND gates in each section and one fixed OR gate. Each AND gate has 10 fused programmable inputs. This is shown in the diagram by 10 vertical lines intersecting each horizontal line. The horizontal line symbolizes the multiple-input configuration of the AND gate. One of the outputs is connected to a buffer–inverter gate and then fed back into the inputs of the AND gates through fuses.

Commercial PAL devices contain more gates than the one shown in Fig. 5-30. A typical PAL integrated circuit may have eight inputs, eight outputs, and eight sections, each consisting of an eightwide AND-OR array. The output terminals are sometimes bidirectional, which means that they can be programmed as inputs instead of outputs if desired.

When designing with a PAL, the Boolean functions must be simplified to fit into each section. Unlike the PLA, a product term cannot be shared among two or more OR gates. Therefore, each function can be simplified by itself without regard to common product terms. The number of product terms in each section is fixed, and if the number of terms in the function is too large, it may be necessary to use two sections to implement one Boolean function.

As an example of using a PAL in the design of a combinational circuit, consider the following Boolean functions given in sum of minterms:

$$w(A, B, C, D) = \Sigma(2, 12, 13)$$

$$x(A, B, C, D) = \Sigma(7, 8, 9, 10, 11, 12, 13, 14, 15)$$

$$y(A, B, C, D) = \Sigma(0, 2, 3, 4, 5, 6, 7, 8, 10, 11, 15)$$

$$z(A, B, C, D) = \Sigma(1, 2, 8, 12, 13)$$

Simplifying the four functions to a minimum number of terms results in the following Boolean functions:

$$w = ABC' + A'B'CD'$$

$$x = A + BCD$$

$$y = A'B + CD + B'D'$$

$$z = ABC' + A'B'CD' + AC'D' + A'B'C'D$$

$$= w + AC'D' + A'B'C'D$$

Note that the function for z has four product terms. The logical sum of two of these terms is equal to w. By using w, it is possible to reduce the number of terms for z from four to three.

The PAL programming table is similar to the one used for the PLA except that only the inputs of the AND gates need to be programmed. Table 5-6 lists the PAL programming table for the four Boolean functions. The table is divided into four sections with three product terms in each to conform with the PAL of Fig. 5-30. The first two sections need only two product terms to implement the Boolean function. The last section for output z needs four product terms. Using the output from w, we can reduce the function to three terms.

The fuse map for the PAL as specified in the programming table is shown in Fig. 5-31. For each 1 or 0 in the table, we mark the corresponding intersection in the diagram with the symbol for an intact fuse. For each dash, we mark the diagram with blown fuses in both the true and complement inputs. If the AND gate is not used, we leave all its input fuses intact. Since the corresponding input receives both the true and complement of each input variable, we have $AA' = 0$ and the output of the AND gate is always 0.

As with all PLDs, the design with PALs is facilitated by using computer-aided design techniques. The blowing of internal fuses is a hardware procedure done with the help of special electronic instruments.

TABLE 5-6
PAL Programming Table

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	–	$+ A'B'C'D$

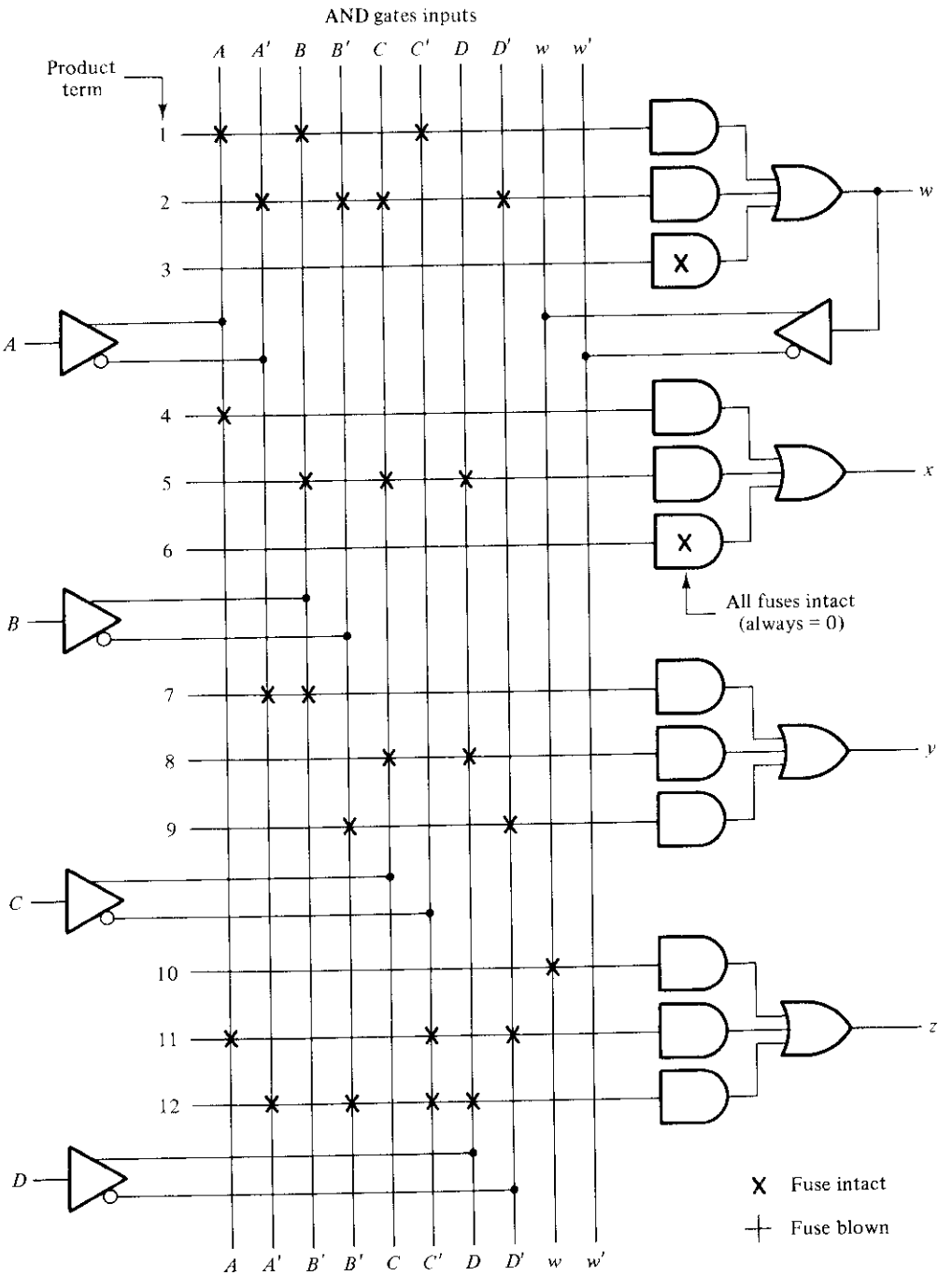


FIGURE 5-31
Fuse map for PAL as specified in Table 5-6

REFERENCES

1. BLAKESLEE, T. R., *Digital Design with Standard MSI and LSI*, 2nd Ed. New York: John Wiley, 1979.
2. SANDIGE, R. S., *Digital Concepts Using Standard Integrated Circuits*. New York: McGraw-Hill, 1978.
3. MANO, M. M., *Computer Engineering: Hardware Design*. Englewood Cliffs, NJ: Prentice-Hall, 1988.
4. MANO, M. M., *Computer System Architecture*, 2nd Ed. Englewood Cliffs, NJ: Prentice-Hall, 1982.
5. SHIVA, S. G., *Introduction to Logic Design*. Glenview, IL: Scott, Foresman, 1988.
6. *The TTL Logic Data Book*. Dallas: Texas Instruments, 1988.
7. TOCCI, R. J., *Digital Systems Principles and Applications*, 4th Ed. Englewood Cliffs, NJ: Prentice-Hall, 1988.
8. FLETCHER, W. I., *An Engineering Approach to Digital Design*. Englewood Cliffs, NJ: Prentice-Hall, 1979.
9. KITSON, B., *Programmable Array Logic Handbook*. Sunnyvale, CA: Advanced Micro Devices, 1983.
10. *Programmable Logic Data Manual*. Sunnyvale, CA: Signetics, 1986.
11. *Programmable Logic Data Book*. Dallas: Texas Instruments, 1988.

PROBLEMS

5-1 Construct a 16-bit parallel adder with four MSI circuits, each containing a 4-bit parallel adder. Use a block diagram with nine inputs and five outputs for each 4-bit adder. Show how the carries are connected between the MSI circuits.

5-2 Construct a BCD-to-excess-3-code converter with a 4-bit adder. Remember that the excess-3 code digit is obtained by adding three to the corresponding BCD digit. What must be done to change the circuit to an excess-3-to-BCD-code converter?

5-3 The adder-subtractor of Fig. 5-2(b) is used to subtract the following unsigned 4-bit numbers: 0110 − 1001 (6 − 9).

(a) What are the binary values in the nine inputs of the circuit?

(b) What are the binary values of the five outputs of the circuit? Explain how the output is related to the operation of 6 − 9.

5-4 The adder-subtractor circuit of Fig. 5-2(b) has the following values for mode input M and data inputs A and B. In each case, determine the values of the outputs: S_4, S_3, S_2, S_1, and C_5.

	M	A	B
(a)	0	0111	0110
(b)	0	1000	1001
(c)	1	1100	1000
(d)	1	0101	1010
(e)	1	0000	0001

5-5 (a) Using the AND-OR-INVERT implementation procedure described in Section 3-7, show that the output carry in a full-adder circuit can be expressed as

$$C_{i+1} = G_i + P_i C_i = (G_i' P_i + G_i' C_i')'$$

(b) IC type 74182 is a look-ahead carry generator MSI circuit that generates the carries with AND-OR-INVERT gates. The MSI circuit assumes that the input terminals have the complements of the G's, the P's, and of C_1. Derive the Boolean functions for the look-ahead carries C_2, C_3, and C_4 in this IC. (*Hint:* Use the equation-substitution method to derive the carries in terms of C_1'.)

5-6 (a) Redefine the carry propagate and carry generate as follows:

$$P_i = A_i + B_i$$

$$G_i = A_i B_i$$

Show that the output carry and output sum of a full-adder becomes

$$C_{i+1} = (C_i' G_i + P_i')' = G_i + P_i C_i$$

$$S_i = (P_i G_i') \oplus C_i$$

(b) The logic diagram of the first stage of a 4-bit parallel adder as implemented in IC type 74283 is shown in Fig. P5-6. Identify the P_i' and G_i' terminals as defined in part (a) and show that the circuit implements a full-adder circuit.

(c) Obtain the output carries C_3 and C_4 as functions of P_1', P_2', P_3', G_1', G_2', G_3', and C_1' in AND-OR-INVERT form, and draw the two-level look-ahead circuit for this IC. [*Hint:* Use the equation-substitution method as done in the text when deriving Fig. 5-4, but use the AND-OR-INVERT function given in part (a) for C_{i+1}.]

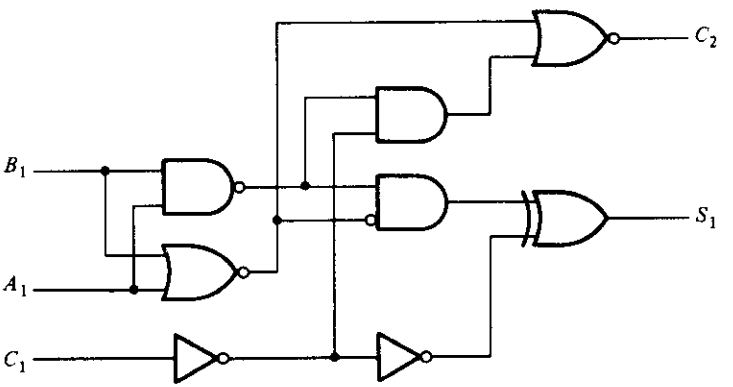


FIGURE P5-6
First stage of a parallel adder

5-7 Assume that the exclusive-OR gate has a propagation delay of 20 ns and that the AND or OR gates have a propagation delay of 10 ns. What is the total propagation delay time in the 4-bit adder of Fig. 5-5?

5-8 Derive the two-level Boolean expression for the output carry C_5 shown in the look-ahead carry generator of Fig. 5-5.

5-9 How many unused input combinations are there in a BCD adder?

5-10 Design a combinational circuit that generated the 9's complement of a BCD digit.

5-11 Construct a 4-digit BCD adder-subtractor using four BCD adders, as shown in Fig. 5-6, and four 9's complement circuits from Problem 5-10. Use block diagrams for each component, showing only inputs and outputs.

5-12 It is necessary to design a decimal adder for two digits represented in the excess-3 code. Show that the correction after adding the two digits with a 4-bit binary adder is as follows:

(a) The output carry is equal to the carry from the binary adder.
(b) If the output carry = 1, then add 0011.
(c) If the output carry = 0, then add 1101.

Construct the decimal adder with two 4-bit adders and an inverter.

5-13 Design a combinational circuit that compares two 4-bit numbers A and B to check if they are equal. The circuit has one output x, so that $x = 1$ if $A = B$ and $x = 0$ if A is not equal to B.

5-14 Design a BCD-to-decimal decoder using the unused combinations of the BCD code as don't-care conditions.

5-15 A combinational circuit is defined by the following three Boolean functions. Design the circuit with a decoder and external gates.

$$F_1 = x'y'z' + xz$$

$$F_2 = xy'z' + x'y$$

$$F_3 = x'y'z + xy$$

5-16 A combinational circuit is specified by the following three Boolean functions. Implement the circuit with a 3 × 8 decoder constructed with NAND gates (similar to Fig. 5-10) and three external NAND or AND gates. Use a block diagram for the decoder. Minimize the number of inputs in the external gates.

$$F_1(A, B, C) = \Sigma(2, 4, 7)$$

$$F_2(A, B, C) = \Sigma(0, 3)$$

$$F_3(A, B, C) = \Sigma(0, 2, 3, 4, 7)$$

5-17 Draw the logic diagram of a 2-to-4-line decoder with only NOR gates. Include an enable input.

5-18 Construct a 5 × 32 decoder with four 3 × 8 decoders with enable and one 2 × 4 decoder. Use block diagrams similar to Fig. 5-12.

5-19 Rearrange the truth table for the circuit of Fig. 5-10 and verify that it can function as a demultiplexer.

5-20 Design a 4-input priority encoder with inputs as in Table 5-4, but with input D_0 having the highest priority and input D_3 the lowest priority.

5-21 Specify the truth table of an octal-to-binary priority encoder. Provide an output V to indicate that at least one of the inputs is a 1. The input with the highest subscript number has the highest priority. What will be the value of the four outputs if inputs D_5 and D_3 are 1 and the other inputs are all 0's?

5-22 Draw the logic diagram of a dual 4-to-1-line multiplexer with common selection inputs and a common enable input.

5-23 Construct a 16 × 1 multiplexer with two 8 × 1 and one 2 × 1 multiplexers. Use block diagrams for the three multiplexers.

5-24 Implement the following Boolean function with an 8 × 1 multiplexer.

$$F(A, B, C, D) = \Sigma(0, 3, 5, 6, 8, 9, 14, 15)$$

5-25 Implement a full-adder with two 4 × 1 multiplexers.

5-26 Implement the Boolean function of Example 5-2 with an 8 × 1 multiplexer, but with inputs A, B, and C connected to selection inputs s_2, s_1, and s_0, respectively.

5-27 An 8 × 1 multiplexer has inputs A, B, and C connected to the selection inputs s_2, s_1, and s_0, respectively. The data inputs, I_0 through I_7, are as follows: $I_1 = I_2 = I_7 = 0$; $I_3 = I_5 = 1$; $I_0 = I_4 = D$; and $I_6 = D'$. Determine the Boolean function that the multiplexer implements.

5-28 Implement the following Boolean function with a 4 × 1 multiplexer and external gates. Connect inputs A and B to the selection lines. The input requirements for the four data lines will be a function of variables C and D. These values are obtained by expressing F as a function of C and D for each of the four cases when AB = 00, 01, 10, and 11. These functions may have to be implemented with external gates.

$$F(A, B, C, D) = \Sigma(1, 3, 4, 11, 12, 13, 14, 15)$$

5-29 Given a 32 × 8 ROM chip with an enable input, show the external connections necessary to construct a 128 × 8 ROM with four chips and a decoder.

5-30 A ROM chip of 4096 × 8 bits has two enable inputs and operates from a 5-volt power supply. How many pins are needed for the integrated-circuit package? Draw a block diagram and label all input and output terminals in the ROM.

5-31 Specify the size of a ROM (number of words and number of bits per word) that will accommodate the truth table for the following combinational circuit components:

(a) A binary multiplier that multiplies two 4-bit numbers.
(b) A 4-bit adder-subtractor; see Fig. 5-2(b).
(c) A quadruple 2-to-1-line multiplexers with common select and enable inputs; see Fig. 5-17.
(d) A BCD-to-seven-segment decoder with an enable input; see Problem 4-16.

5-32 Tabulate the truth table for an 8 × 4 ROM that implements the following four Boolean functions:

$$A(x, y, z) = \Sigma(1, 2, 4, 6,)$$

$$B(x, y, z) = \Sigma(0, 1, 6, 7)$$

$$C(x, y, z) = \Sigma(2, 6)$$

$$D(x, y, z) = \Sigma(1, 2, 3, 5, 7)$$

5-33 Tabulate the PLA programming table for the four Boolean functions listed in Problem 5-32. Minimize the number of product terms.

5-34 Derive the PLA programming table for the combinational circuit that squares a 3-bit number. Minimize the number of product terms. (See Fig. 5-24 for the equivalent ROM implementation.)

5-35 List the PLA programming table for the BCD-to-excess-3-code converter whose Boolean functions are simplified in Fig. 4-7.

5-36 Repeat Problem 5-35 using a PAL.

5-37 The following is a truth table of a 3-input, 4-output combinational circuit. Tabulate the PAL programming table for the circuit and mark the fuses to be blown in a PAL diagram similar to the one shown in Fig. 5-30.

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