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-

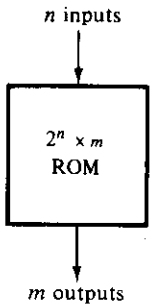


FIGURE 5-21
ROM block diagram

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

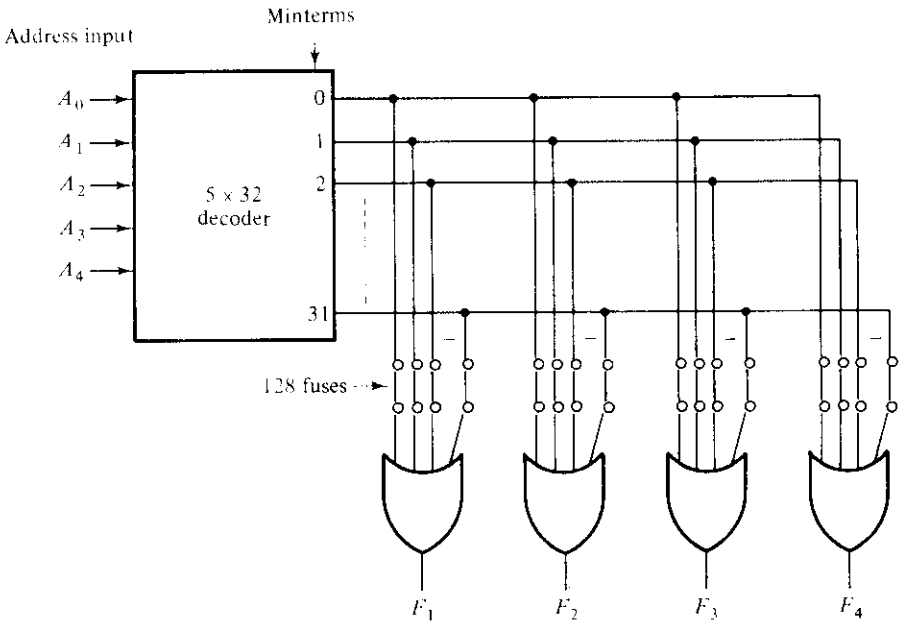


FIGURE 5-22
Logic construction of a 32 × 4 ROM

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.

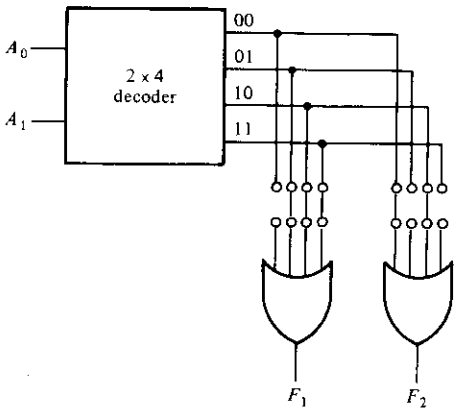
Combinational Logic Implementation

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.

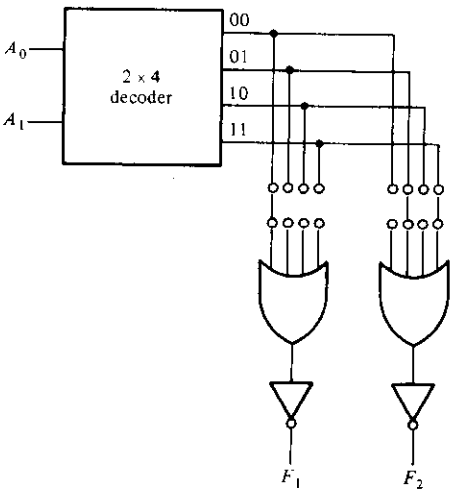
Let us clarify the process with a specific example. The truth table in Fig. 5-23(a) specifies a combinational circuit with two inputs and two outputs. The Boolean functions can be expressed in sum of minterms:

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.

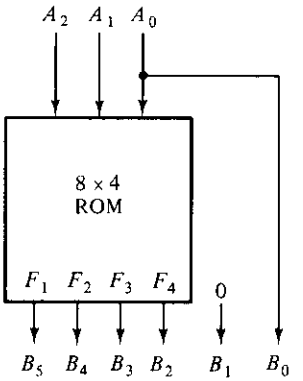
Example 5-3

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.

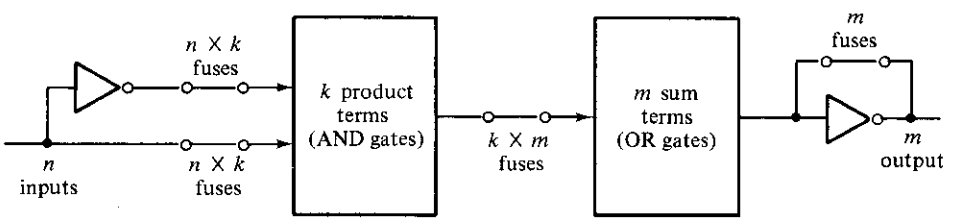


FIGURE 5-25
PLA block diagram

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.

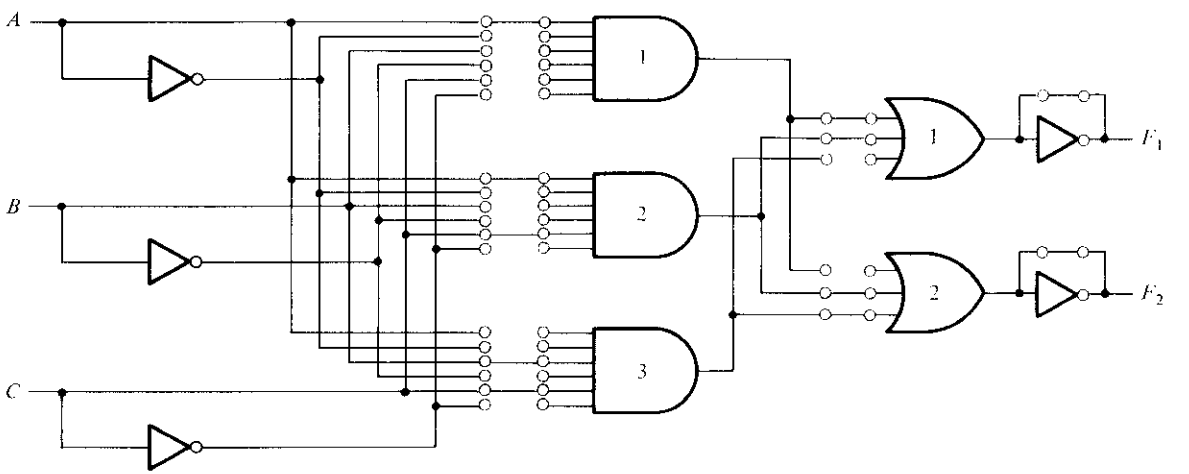
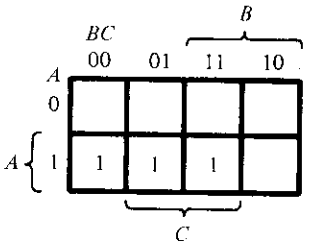


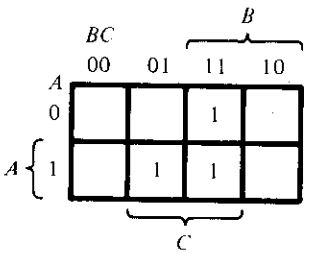
FIGURE 5-26
PLA with three inputs, three product terms, and two outputs; it implements the combinational circuit specified in Fig. 5-27

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

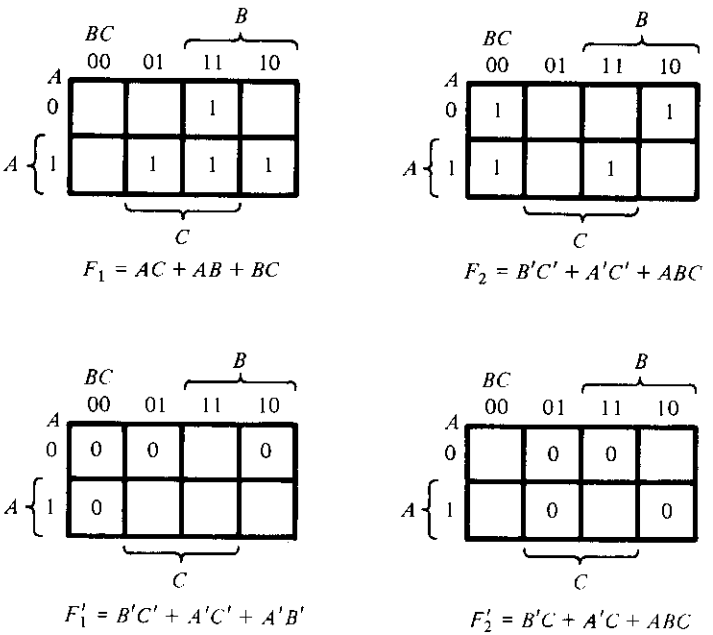
A combinational circuit is defined by the functions

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

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