CHAPTER 3

DESIGNING WITH PROGRAMMABLE LOGIC DEVICES

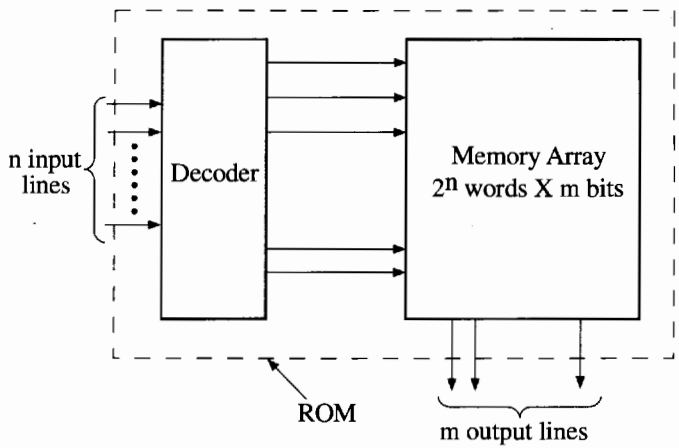
This chapter introduces the use of programmable logic devices (PLDs) in digital design. Read-only memories (ROMs), programmable logic arrays (PLAs), and programmable array logic devices (PALs) are discussed, and then more complex PLDs are introduced. Use of these devices allows us to implement complex logic functions, which require many gates and flip-flops, with a single IC chip. More complex PLDs and programmable gate arrays are described in Chapter 6.

This chapter contains several VHDL examples. Sequential networks using a ROM and using a PLA are described in VHDL. Two design examples, a traffic-light controller and a keypad scanner, also use VHDL. The latter example introduces the use of a testbench, which is written in VHDL, to test the VHDL code for the keypad scanner design.

3.1 READ-ONLY MEMORIES

A read-only memory (ROM) consists of an array of semiconductor devices interconnected to store an array of binary data. Once binary data is stored in the ROM, it can be read out whenever desired, but the data that is stored cannot be changed under normal operating conditions. A ROM that has n input lines and m output lines (Figure 3-1) contains an array of 2^n words, and each word is m bits long. The input lines serve as an address to select one of the 2^n words. Conceptually, a ROM consists of a decoder and a memory array. When a pattern of n 0s and ls is applied to the decoder inputs, exactly one of the 2^n decoder outputs is 1. This decoder output line selects one of the words in the memory array, and the bit pattern stored in this word is transferred to the memory output lines. A $2^n \times m$ ROM can realize m functions of n variables, since it can store a truth table with 2^n rows and m columns.

Figure 3-1 Basic ROM Structure



Basic types of ROMs include mask-programmable ROMs and erasable programmable ROMs (usually called EPROMs). At time of manufacture, the data array is permanently stored in a mask-programmable ROM. This is accomplished by selectively including or omitting the switching elements at the row-column intersections of the memory array. This requires preparation of a special "mask," which is used during fabrication of the integrated circuit. Preparation of this mask is expensive, so use of mask-programmable ROMs is economically feasible only if a large quantity (typically several thousand or more) are required with the same data array. If only a small quantity of ROMs are required with a given data array, EPROMs may be used.

Modification of the data stored in a ROM is often necessary during the developmental phases of a digital system, so EPROMS are used instead of mask-programmable ROMs. EPROMs use a special charge-storage mechanism to enable or disable the switching elements in the memory array. A PROM programmer is used to provide appropriate voltage pulses to store electronic charges in the memory array locations. The data stored in this manner is generally permanent until erased using an ultraviolet light. After erasure, a new set of data can be stored in the EPROM. The electrically erasable PROM (or EEPROM) is a more recent development. It is similar to the EPROM, except that erasure is accomplished using electrical pulses instead of ultraviolet light. An EEPROM can be erased and reprogrammed only a limited number of times, typically 100 to 1000 times. Flash memories are similar to EEPROMs, except that they use a different charge-storage mechanism. They usually have built-in programming and erase capability so that data can be written to the flash memory while it is in place in a circuit without the need for a separate programmer.

A sequential network can easily be designed using a ROM and flip-flops. Referring to the general model of a Mealy sequential network given in Figure 1-16, the combinational part of the sequential network can be realized using a ROM. The ROM can be used to realize the output functions and the next-state functions. The state of the network can then be stored in a register of D flip-flops and fed back to the input of the ROM. Use of D flip-

flops is preferable to J-K flip-flops, since use of 2-input flip-flops would require increasing the number of outputs from the ROM. The fact that the D flip-flop input equations would generally require more gates than the J-K equations is of no consequence, since the size of the ROM depends only on the number of inputs and outputs and not on the complexity of the equations being realized. For this reason, the state assignment used is also of little importance, and generally a state assignment in straight binary order is as good as any.

We will realize the sequential machine of Figure 1-17 using a ROM and three D flip-flops (see Figure 3-2). Table 3-1 gives the truth table for the ROM, which implements the transition table of Figure 1-18(b) with the don't cares replaced by 0s. Since the ROM has four inputs, it contains $2^4 = 16$ words. In general, a Mealy sequential network with i inputs, j outputs, and k state variables can be realized using k D flip-flops and a ROM with $i + k$ inputs (2^{i+k} words) and $j + k$ outputs.

Figure 3-2 Realization of a Mealy Sequential Network with a ROM

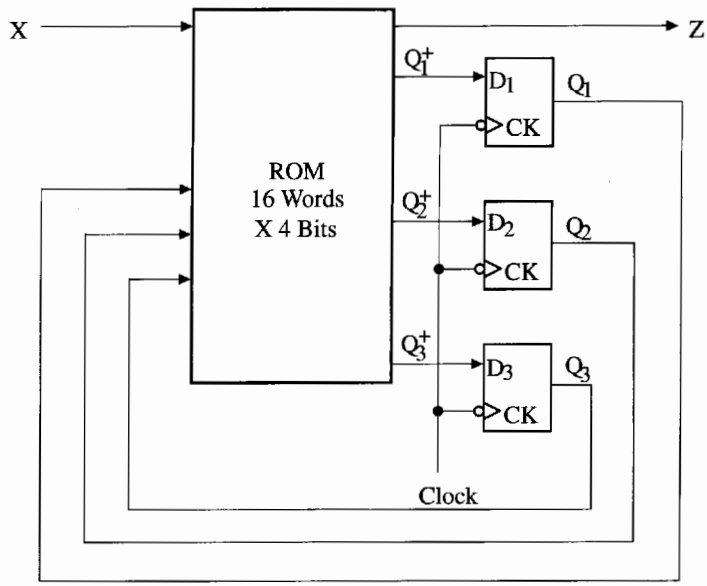


Table 3-1 ROM Truth Table

Q_1	Q_2	Q_3	X	Q_1^+	Q_2^+	Q_3^+	Z
0	0	0	0	1	0	0	1
0	0	0	1	1	0	1	0
0	0	1	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	1
0	1	0	1	0	0	0	0
0	1	1	0	0	0	0	0
0	1	1	1	0	0	0	1
1	0	0	0	1	1	1	1
1	0	0	1	1	1	0	0
1	0	1	0	1	1	0	0
1	0	1	1	1	1	0	1
1	1	0	0	0	1	1	1
1	1	0	1	0	1	0	0
1	1	1	0	0	1	1	0
1	1	1	1	0	1	1	1

The VHDL code for the ROM realization (Figure 3-3) is similar to that of Figure 2-13, except the process that determines the next state and output has been replaced by code that reads the ROM. The state register is represented by Q, which is a 3-bit vector (Q_1, Q_2, Q_3), and the next state of this register is *Qplus*. In VHDL, a ROM can be represented by a constant one-dimensional array of bit vectors. In this example, a type statement is used to declare type ROM as an array of 16 words of 4-bit vectors. A constant declaration specifies the contents of the ROM named *FSM_ROM*. The input to the *FSM_ROM* is Q concatenated with X. Since the index of an array must be an integer, the *vec2int* function is called to convert $Q\&X$ to an integer. The variable *ROMValue* is set equal to the ROM output, and then *ROMValue* is split into *Qplus* and Z. The state register Q is updated after the rising edge of the clock.

Figure 3-3 ROM Realization of Figure 1-17

```
library BITLIB;
use BITLIB.bit_pack.all;

entity ROM1_2 is
  port(X,CLK: in bit;
       Z: out bit);

end ROM1_2;
architecture ROM1 of ROM1_2 is
signal Q, Qplus: bit_vector(1 to 3) := "000";
type ROM is array (0 to 15) of bit_vector(3 downto 0);
constant FSM_ROM:  ROM :=
                ("1001","1010","0000","0000",
                 "0001","0000","0000","0001",
                 "1111","1100","1100","1101",
                 "0111","0100","0110","0111");
begin
  process(Q,X)              -- determines the next state and output
  variable ROMValue: bit_vector(3 downto 0);
  begin
    ROMValue := FSM_ROM(vec2int(Q & X));    -- read ROM output
    Qplus <= ROMValue(3 downto 1);
    Z <= ROMValue(0);
  end process;

process(CLK)
  begin
    if CLK='1' then Q <= Qplus; end if;    -- update state register
  end process;
end ROM1;
```

3.2 PROGRAMMABLE LOGIC ARRAYS (PLAS)

A programmable logic array (PLA) performs the same basic function as a ROM. A PLA with n inputs and m outputs (Figure 3-4) can realize m functions of n variables. The internal organization of the PLA is different from that of the ROM. The decoder is replaced with an AND array that realizes selected product terms of the input variables. The OR array ORs together the product terms needed to form the output functions.

Figure 3-4 Programmable Logic Array Structure

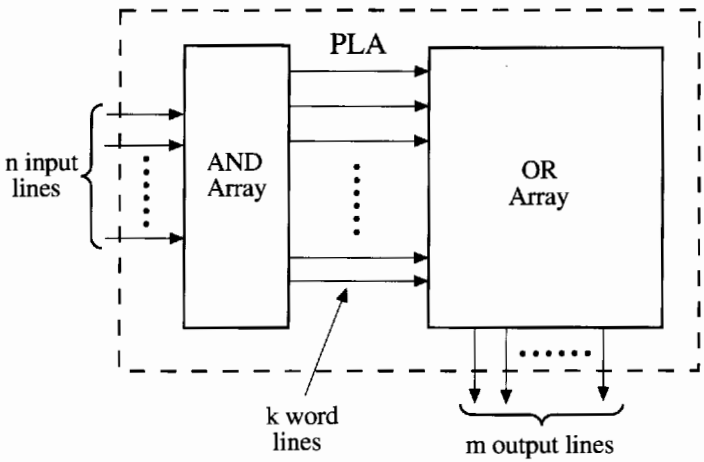


Figure 3-5 shows an nMOS PLA that realizes the following functions:

$$F_0 = \sum m(0, 1, 4, 6) = A'B' + AC' \quad \textbf{(3-1)}$$

$$F_1 = \sum m(2, 3, 4, 6, 7) = B + AC'$$

$$F_2 = \sum m(0, 1, 2, 6) = A'B' + BC'$$

$$F_3 = \sum m(2, 3, 5, 6, 7) = AC + B$$

Internally, the PLA uses NOR-NOR logic, but the added input and output inverting buffers make it equivalent to AND-OR logic. Logic gates are formed in the array by connecting nMOS switching transistors between the column lines and the row lines. Figure 3-6 shows the implementation of a two-input NOR gate. The transistors act as switches, so if the gate input is a logic 0, the transistor is off. If the gate input is a logic 1, the transistor provides a conducting path to ground. If $X_1 = X_2 = 0$, both transistors are off, and the pull-up resistor brings the Z output to a logic 1 level (+V). If either X_1 or X_2 is 1, the corresponding transistor is turned on, and $Z = 0$. Thus, $Z = (X_1 + X_2)' = X_1'X_2'$, which corresponds to a NOR gate. The part of the PLA array that realizes F_0 is equivalent to the NOR-NOR gate structure shown in Figure 3-7. After canceling the extra inversions, this reduces to an AND-OR structure. The AND-OR array shown in Figure 3-8 is thus equivalent to the nMOS PLA structure of Figure 3-5.

Figure 3-5 PLA with 3 Inputs, 5 Product Terms, and 4 Outputs

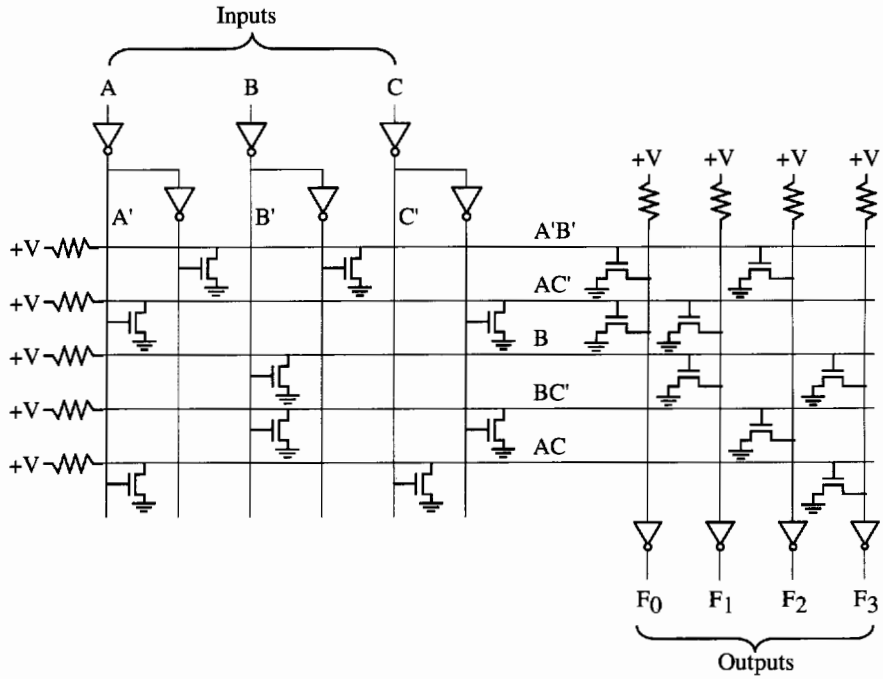


Figure 3-6 nMOS NOR Gate

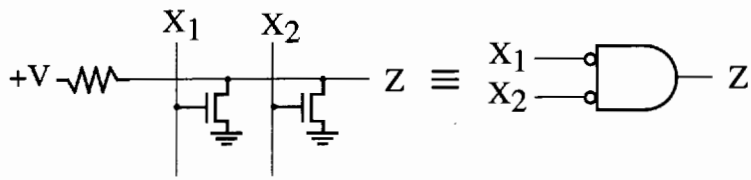


Figure 3-7 Conversion of NOR-NOR to AND-OR

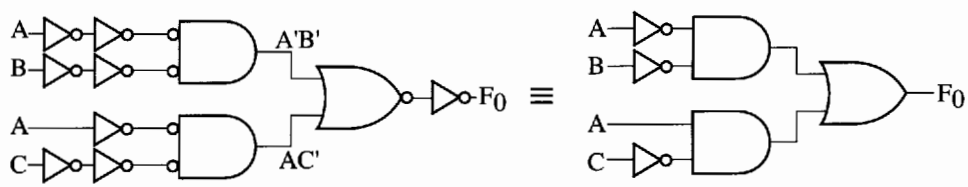
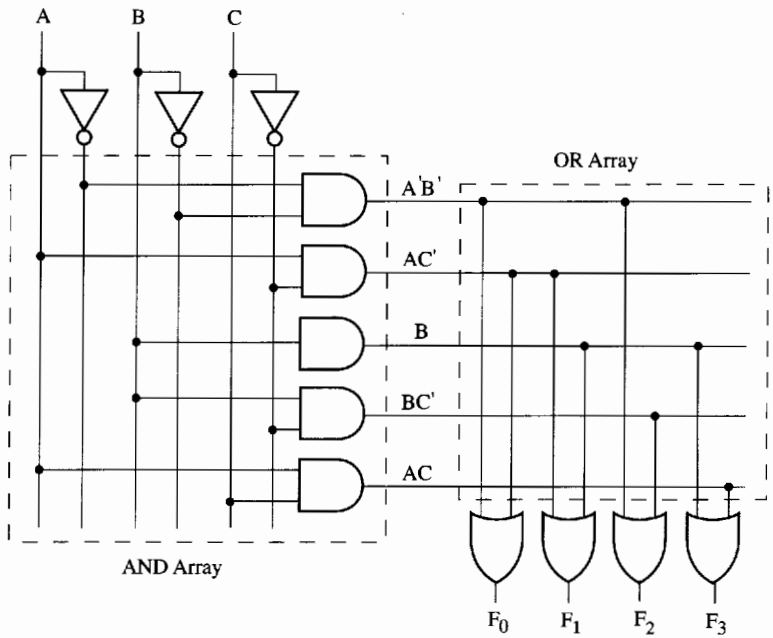


Figure 3-8 AND-OR Array Equivalent to Figure 3-5



The contents of a PLA can be specified by a modified truth table. Table 3-2 specifies the PLA in Figure 3-5. The input side of the table specifies the product terms. The symbols 0, 1, and – indicate whether a variable is complemented, not complemented, or not present in the corresponding product term. The output side of the table specifies which product terms appear in each output function. A 1 or 0 indicates whether a given product term is present or not present in the corresponding output function. Thus, the first row of Table 3-2 indicates that the term $A'B'$ is present in output functions F_0 and F_2, and the second row indicates that AC' is present in F_0 and F_1.

Table 3-2 PLA Table for Figure 3-5

Product Term	Inputs			Outputs			
	A	B	C	F_0	F_1	F_2	F_3
$A'B'$	0	0	–	1	0	1	0
AC'	1	–	0	1	1	0	0
B	–	1	–	0	1	0	1
BC'	–	1	0	0	0	1	0
AC	1	–	1	0	0	0	1

Next we will realize the following functions using a PLA:

$$F_1 = \sum m(2, 3, 5, 7, 8, 9, 10, 11, 13, 15) \quad \textbf{(3-2)}$$

$$F_2 = \sum m(2, 3, 5, 6, 7, 10, 11, 14, 15)$$

$$F_3 = \sum m(6, 7, 8, 9, 13, 14, 15)$$

If we minimize each function separately, the result is

$$F_1 = bd + b'c + ab' \quad \textbf{(3-3)}$$

$$F_2 = c + a'bd$$

$$F_3 = bc + ab'c' + abd$$

If we implement these reduced equations in a PLA, a total of eight different product terms (including c) are required.

Instead of minimizing each function separately, we want to minimize the total number of rows in the PLA table. In this case, the number of terms in each equation is not important, since the size of the PLA does not depend on the number of terms. Equations (3-3) are plotted on the Karnaugh maps shown in Figure 3-9. Since the term $ab'c'$ is already needed for F_3, we can use it in F_1 instead of ab', since the other two 1s in ab' are covered by the $b'c$ term. This eliminates the need to use a row of the PLA table for ab'. Since the terms $a'bd$ and abd are needed in F_2 and F_3, respectively, we can replace bd in F_1 with $a'bd + abd$. This eliminates the need for a row to implement bd. Since $b'c$ and bc are used in F_1 and F_3, respectively, we can replace c in F_3 with $b'c + bc$. The resulting equations (3-4) correspond to the reduced PLA table (Table 3-3). Instead of using Karnaugh maps to reduce the number of rows in the PLA, the Espresso algorithm can be used. This complex algorithm is described in *Logic Minimization Algorithms for VLSI Synthesis* by Brayton [10].

Figure 3-9 Multiple-Output Karnaugh Maps

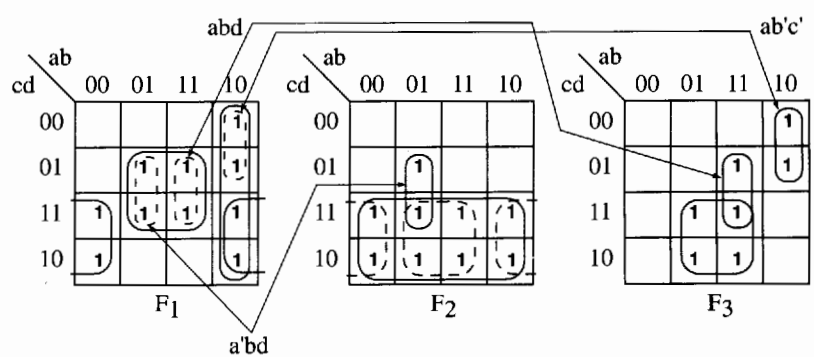


Table 3-3 Reduced PLA Table

a	b	c	d	F_1	F_2	F_3
0	1	–	1	1	1	0
1	1	–	1	1	0	1
1	0	0	–	1	0	1
–	0	1	–	1	1	0
–	1	1	–	0	1	1

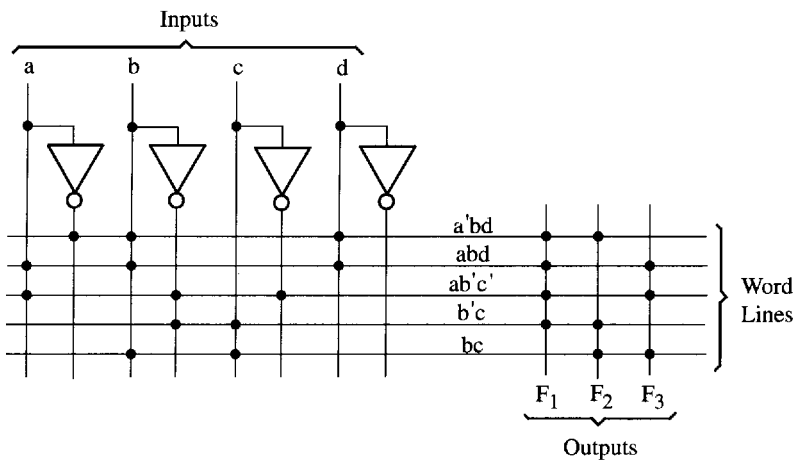
$$F_1 = a'bd + abd + ab'c' + b'c \quad (3\text{-}4)$$

$$F_2 = a'bd + b'c + bc$$

$$F_3 = abd + ab'c' + bc$$

Equations (3-4) have only 5 different product terms, so the PLA table has only five rows. This is a significant improvement over equations (3-3), which require 8 product terms. Figure 3-10 shows the corresponding PLA structure, which has 4 inputs, 5 product terms, and 3 outputs. A dot at the intersection of a word line and an input or output line indicates the presence of a switching element in the array.

Figure 3-10 PLA Realization of Equations (3-4)



A PLA table is significantly different than a truth table for a ROM. In a truth table each row represents a minterm; therefore, exactly one row will be selected by each combination of input values. The 0s and ls of the output portion of the selected row determine the corresponding output values. On the other hand, each row in a PLA table represents a general product term. Therefore, zero, one, or more rows may be selected by each combination of input values. To determine the value of F for a given input combination, the values of F in the selected rows of the PLA table must be ORed together. The following examples refer to the PLA table of Table 3-3. If $abcd = 0001$, no rows are selected; and all F_i's are 0. If $abcd = 1001$, only the third row is selected, and $F_1F_2F_3 = 101$. If $abcd = 0111$, the first and fifth rows are selected. Therefore, $F_1 = 1 + 0 = 1$, $F_2 = 1 + 1 = 1$, and $F_3 = 0 + 1 = 1$.

Next we realize the sequential machine of Figure 1-17 using a PLA and three D flip-flops. The network structure is the same as Figure 3-2, except that the ROM is replaced by a PLA. The required PLA table, based on the equations given in Figure 1-19, is Table 3-4.

Table 3-4 PLA Table

Product Term	Q_1	Q_2	Q_3	X	Q_1^+	Q_2^+	Q_3^+	Z
Q_2'	–	0	–	–	1	0	0	0
Q_1	1	–	–	–	0	1	0	0
$Q_1Q_2Q_3$	1	1	1	–	0	0	1	0
$Q_1Q_3'X'$	1	–	0	0	0	0	1	0
$Q_1'Q_2'X$	0	0	–	1	0	0	1	0
$Q_3'X'$	–	–	0	0	0	0	0	1
Q_3X	–	–	1	1	0	0	0	1

Reading the output of a PLA in VHDL is somewhat more difficult than reading the ROM output. Since the input to the PLA can match several rows, and the outputs from those rows must be ORed together, a function is required to sequentially scan the PLA array and determine the PLA output.

In Figure 3-11, we represent the PLA as a two-dimensional array of type PLAmtrx and use a constant declaration to specify the contents of the *FSM_PLA*. The function *PLAout* is called to determine the output of *FSM_PLA* when the input is *Q&X*, and this output is assigned to the variable *PLAValue*. After splitting *PLAValue* into *Qplus* and *Z*, *Q* is updated on the rising edge of the clock. The PLAout function is explained in Section 8.5.

Because of the –'s in the PLA table input section, multiple-valued logic is required for the PLA computation. We could define a VHDL type with elements 0, 1, and –; however, it is preferable to use a predefined type. We have used IEEE std_logic type, which is defined in the IEEE library. The std_logic type has nine values, including '0', '1', and 'X'. When we inserted the PLA table into the VHDL code, we used 'X' to represent a dash (–). We placed the type declaration for *PLAmtrx* and the function *PLAout* in the *MVLLIB* (multivalued logic library) and also make use of the IEEE standard logic package. Details of using the multivalued logic and the standard logic package are discussed in Chapter 8.

Figure 3-11 PLA Realization of Figure 1-17

```
library ieee;
use ieee.std_logic_1164.all;              -- IEEE standard logic package
library MVLLIB;                           -- includes PLAmtrx type and
use MVLLIB.mvl_pack.all;                  -- PLAout function

entity PLA1_2 is
  port(X,CLK: in std_logic;
       Z: out std_logic);
end PLA1_2;

architecture PLA of PLA1_2 is
signal Q, Qplus: std_logic_vector(1 to 3) := "000";
constant FSM_PLA:  PLAmtrx(0 to 6, 7 downto 0) :=
                ("X0XX1000",
                 "1XXX0100",
                 "111X0010",
                 "1X000010",
                 "00X10010",
                 "XX000001",
                 "XX110001");
begin
  process(Q,X)
  variable PLAValue: std_logic_vector(3 downto 0);
  begin
    PLAValue := PLAout(FSM_PLA,Q & X);      -- read PLA output
    Qplus <= PLAValue(3 downto 1);
    Z <= PLAValue(0);
  end process;

  process(CLK)
  begin
    if CLK='1' then Q <= Qplus; end if;    -- update state register
  end process;
end PLA;
```

3.3 PROGRAMMABLE ARRAY LOGIC (PALS)

The PAL (programmable array logic) is a special case of the programmable logic array in which the AND array is programmable and the OR array is fixed. The basic structure of the PAL is the same as the PLA shown in Figure 3-4. Because only the AND array is programmable, the PAL is less expensive than the more general PLA, and the PAL is easier to program. For this reason, logic designers frequently use PALs to replace individual logic gates when several logic functions must be realized.

Figure 3-12 Combinational PAL Segment

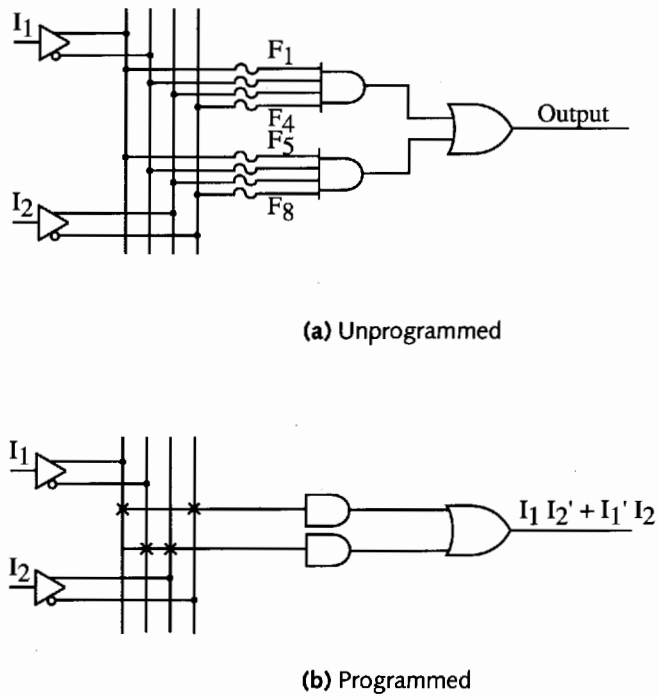
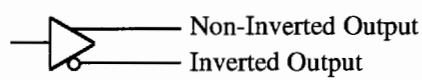
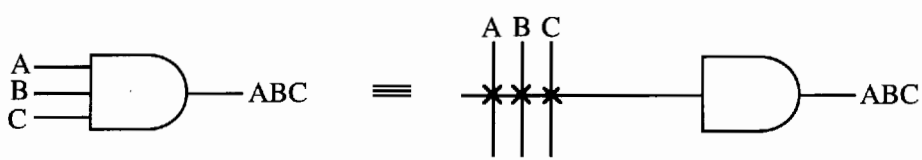


Figure 3-12(a) represents a segment of an unprogrammed PAL. The symbol



represents an input buffer with noninverted and inverted outputs. A buffer is used, since each PAL input must drive many AND gate inputs. When the PAL is programmed, the fusible links ($F_1, F_2, \ldots, F_8$) are selectively blown to leave the desired connections to the AND gate inputs. Connections to the AND gate inputs in a PAL are represented by ×s, as shown here:



As an example, we will use the PAL segment of Figure 3-12(a) to realize the function $I_1I_2' + I_1'I_2$. The ×s indicate that the I_1 and I_2' lines are connected to the first AND gate, and the I_1' and I_2 lines are connected to the other gate (see Figure 3-12(b)).

Typical combinational PALs have from 10 to 20 inputs and from 2 to 10 outputs, with 2 to 8 AND gates driving each OR gate. PALs are also available that contain D flip-flops with inputs driven from the programmable array logic. Such PALs provide a convenient way of realizing sequential networks. Figure 3-13 shows a segment of a sequential PAL. The D flip-flop is driven from an OR gate, which is fed by two AND gates. The flip-flop output is fed back to the programmable AND array through a buffer. Thus the AND gate inputs can be connected to A, A', B, B', Q, or Q'. The ×s on the diagram show the realization of the next-state equation

$$Q^+ = D = A'BQ' + AB'Q$$

The flip-flop output is connected to an inverting tristate buffer, which is enabled when EN = 1.

Figure 3-13 Segment of a Sequential PAL

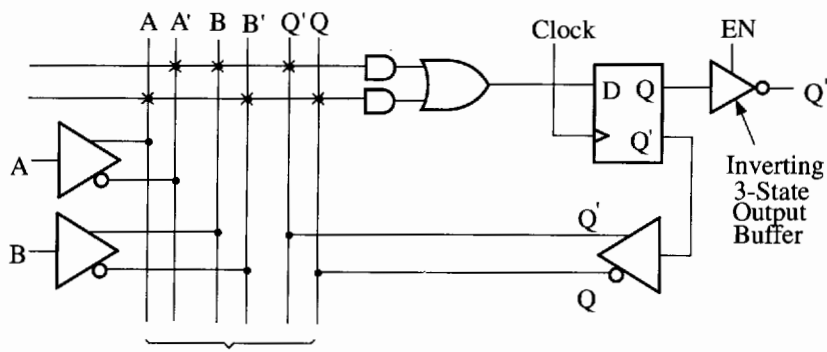
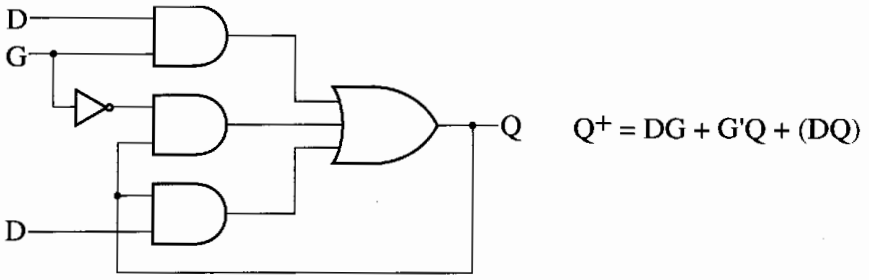


Figure 3-14 shows a logic diagram for a typical sequential PAL, the 16R4. This PAL has an AND gate array with 16 input variables, and it has 4 D flip-flops. Each flip-flop output goes through a tristate inverting buffer (output pins 14–17). One input (pin 11) is used to enable these buffers. The rising edge of a common clock (pin 1) causes the flip-flops to change state. Each D flip-flop input is driven from an OR gate, and each OR gate is fed from 8 AND gates. The AND gate inputs can come from the external PAL inputs (pins 2–9) or from the flip-flop outputs, which are fed back internally. In addition there are 4 input/output (I/O) terminals (pins 12, 13, 18, and 19), which can be used as either network outputs or as inputs to the AND gates. Thus, each AND gate can have a maximum of 16 inputs (8 external inputs, 4 inputs fed back from the flip-flop outputs, and 4 inputs from the I/O terminals). When used as an output, each I/O terminal is driven from an inverting tristate buffer. Each of these buffers is fed from an OR gate and each OR gate is fed from 7 AND gates. An eighth AND gate is used to enable the buffer.

When the 16R4 PAL is used to realize a sequential network, the I/O terminals are normally used for the Z outputs. Thus, a single 16R4 with no additional logic could realize a sequential network with up to 8 inputs, 4 outputs, and 16 states. Each next-state equation could contain up to 8 terms, and each output equation could contain up to 7 terms. As an

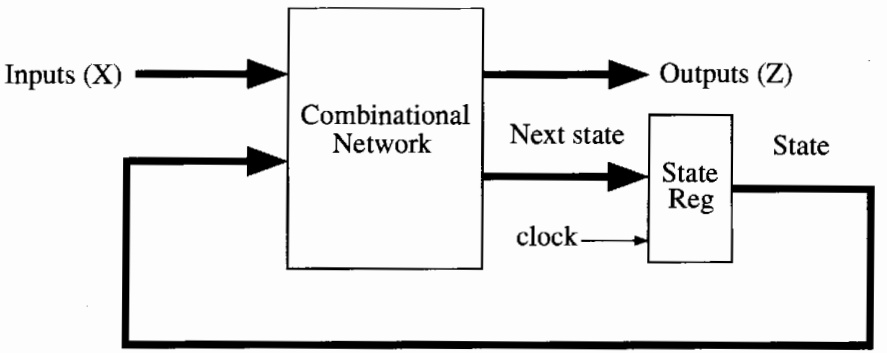
Figure 1-15 Implementation of D Latch



1.7 MEALY SEQUENTIAL NETWORK DESIGN

The two basic types of sequential networks are Mealy and Moore. In a Mealy network, the outputs depend on both the present state and the present inputs. In a Moore network, the outputs depend only on the present state. A general model of a Mealy sequential network consists of a combinational network, which generates the outputs and the next state, and a state register, which holds the present state (see Figure 1-16). The state register normally consists of D flip-flops. The normal sequence of events is (1) the *X* inputs are changed to a new value, (2) after a delay, the corresponding *Z* outputs and next state appear at the output of the combinational network, and (3) the next state is clocked into the state register and the state changes. The new state feeds back into the combinational network, and the process is repeated.

Figure 1-16 General Model of Mealy Sequential Machine



As an example of a Mealy sequential network, we will design a code converter that converts an 8-4-2-1 binary-coded-decimal (BCD) digit to an excess-3-coded decimal digit. The input (X) and output (Z) will be serial with the least significant bit first. Table 1-2 lists the desired inputs and outputs at times t_0, t_1, t_2, and t_3. After receiving four inputs, the network should reset to its initial state, ready to receive another BCD digit.

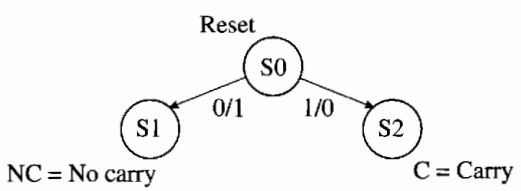
Table 1-2 Code Converter

X Input (BCD)				*Z* Output (excess -3)			
t_3	t_2	t_1	t_0	t_3	t_2	t_1	t_0
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

We now construct a state graph for the code converter (Figure 1-17(a)). The excess-3 code is formed by adding 0011 to the BCD digit. For example,

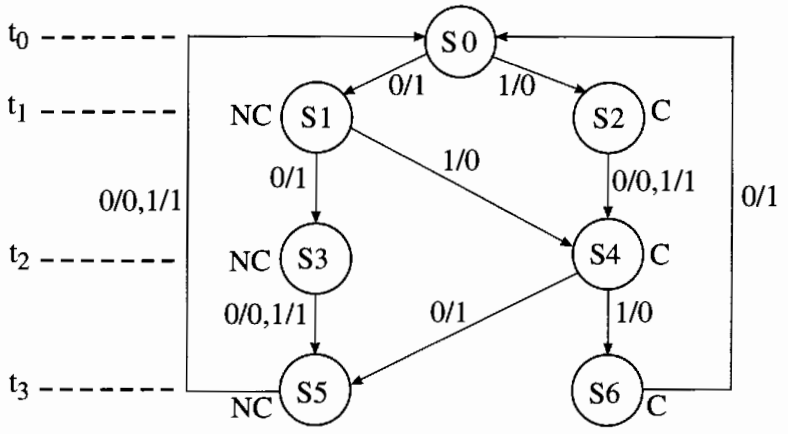
```
 0100        0101
+0011       +0011
 0111        1000
```

At t_0, we add 1 to the least significant bit, so if $X = 0$, $Z = 1$ (no carry), and if $X = 1$, $Z = 0$ (carry = 1). This leads to the following partial state graph:



S0 is the reset state, S1 indicates no carry after the first addition, and S2 indicates a carry of 1. At t_1, we add 1 to the next bit, so if there is no carry from the first addition (state S1), $X = 0$ gives $Z = 0 + 1 + 0 = 1$ and no carry (state S3), and $X = 1$ gives $Z = 1 + 1 + 0 = 0$ and a carry (state S4). If there is a carry from the first addition (state S2), then $X = 0$ gives $Z = 0 + 1 + 1 = 0$ and a carry (S4), and $X = 1$ gives $Z = 1 + 1 + 1 = 1$ and a carry (S4). At t_2, 0 is added to X, and transitions to S5 (no carry) and S6 are determined in a similar manner. At t_3, 0 is again added to X, and the network resets to S0.

Figure 1-17 State Graph and Table for Code Converter



(a) Mealy state graph

	NS		Z	
PS	X = 0	X = 1	X = 0	X = 1
S0	S1	S2	1	0
S1	S3	S4	1	0
S2	S4	S4	0	1
S3	S5	S5	0	1
S4	S5	S6	1	0
S5	S0	S0	0	1
S6	S0	–	1	–

(b) State table

Figure 1-17(b) gives the corresponding state table. (*Fundamentals of Logic Design,* pp. 429–430 gives an alternative way of deriving this state table.) At this point, we should verify that the table has a minimum number of states before proceeding (see Section 1-9). Since the state table has seven states, three flip-flops will be required to realize the table. The next step is to make a state assignment that relates the flip-flop states to the states in the table. The best state assignment to use depends on a number of factors. In many cases, we should try to find an assignment that will reduce the amount of required logic. For some types of programmable logic, a straight binary state assignment will work just as well as any other. For programmable gate arrays, a one-hot assignment (see Section 6.4) may be preferred.

In order to reduce the amount of logic required, we will make a state assignment using the following guidelines (see *Fundamentals of Logic Design,* p. 412):

I. States that have the same next state (NS) for a given input should be given adjacent assignments (look at the columns of the state table).
II. States that are the next states of the same state should be given adjacent assignments (look at the rows).
III. States that have the same output for a given input should be given adjacent assignments.

Using these guidelines tends to clump 1s together on the Karnaugh maps for the next state and output functions. The guidelines indicate that the following states should be given adjacent assignments:

I. (1, 2), (3, 4), (5, 6) (in the $X = 1$ column, S_1 and S_2 both have NS S_4; in the $X = 0$ column, S_3 and S_4 have NS S_5, and S_5 and S_6 have NS S_0)
II. (1, 2), (3, 4), (5, 6) (S_1 and S_2 are NS of S_0; S_3 and S_4 are NS of S_1; and S_5 and S_6 are NS of S_4)
III. (0, 1, 4, 6), (2, 3, 5)

Figure 1-18(a) gives an assignment map, which satisfies the guidelines, and the corresponding transition table. Since state 001 is not used, the next state and outputs for this state are don't cares. The next state and output equations are derived from this table in Figure 1-19. Figure 1-20 shows the realization of the code converter using NAND gates and D flip-flops.

Figure 1-18

Q_2Q_3 \ Q_1	0	1
00	S0	S1
01		S2
11	S5	S3
10	S6	S4

(a) Assignment map

	$Q_1^+ Q_2^+ Q_3^+$		Z	
$Q_1Q_2Q_3$	X = 0	X = 1	X = 0	X = 1
000	100	101	1	0
100	111	110	1	0
101	110	110	0	1
111	011	011	0	1
110	011	010	1	0
011	000	000	0	1
010	000	xxx	1	x
001	xxx	xxx	x	x

(b) Transition table

If J-K flip-flops are used instead of D flip-flops, the input equations for the J-K flip-flops can be derived from the next state maps. Given the present state flip-flop (Q) and the desired next state (Q^+), the J and K inputs can be determined from the following table, which was derived from the truth table in Figure 1-11:

Q	Q^+	J	K	
0	0	0	X	(No change in Q; J must be 0, K may be 1 to reset Q to 0.)
0	1	1	X	(Change to $Q = 1$; J must be 1 to set or toggle.)
1	0	X	1	(Change to $Q = 0$; K must be 1 to reset or toggle.)
1	1	X	0	(No change in Q; K must be 0, J may be 1 to set Q to 1.)

Figure 1-19 Karnaugh Maps for Figure 1-17

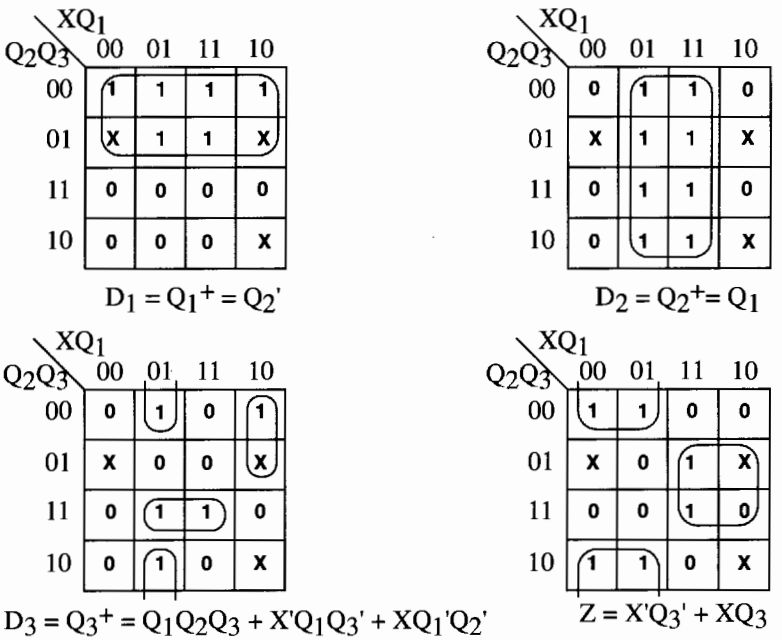


Figure 1-20 Realization of Code Converter

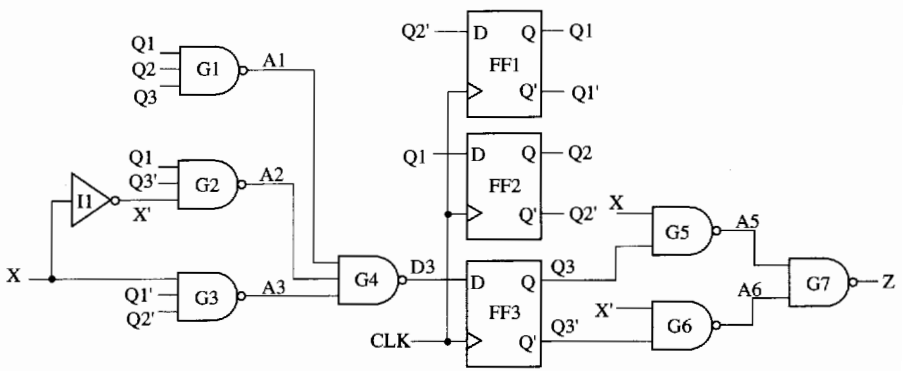
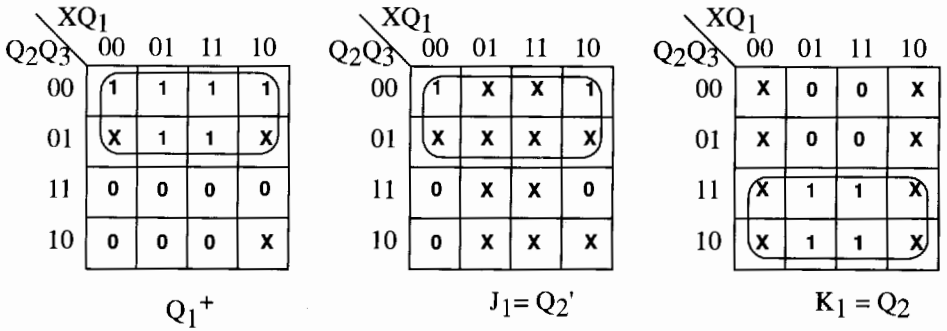
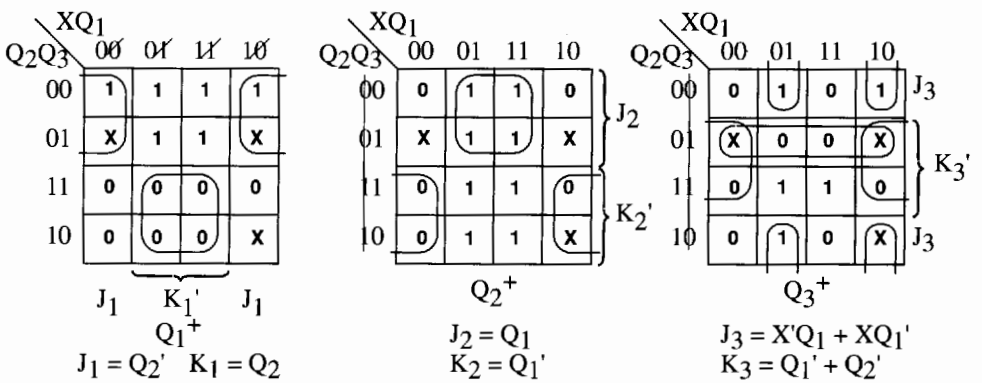


Figure 1-21 shows derivation of J-K flip-flops for the state table of Figure 1-17 using the state assignment of Figure 1-18. First, we derive the J-K input equations for flip-flop Q_1 using the Q_1^+ map as the starting point. From the preceding table, whenever Q_1 is 0, $J = Q_1^+$ and $K = X$. So, we can fill in the $Q_1 = 0$ half of the J_1 map the same as Q_1^+ and the $Q_1 = 0$ half of the K_1 map as all Xs. When Q_1 is 1, $J_1 = X$ and $K_1 = (Q_1^+)'$. So, we can fill in the $Q_1 = 1$ half of the J_1 map with Xs and the $Q_1 = 1$ half of the K_1 map with the complement of the Q_1^+. Since half of every J and K map is don't cares, we can avoid drawing separate J and K maps and read the Js and Ks directly from the Q^+ maps, as illustrated in Figure 1-21(b). This shortcut method is based on the following: If $Q = 0$, then $J = Q^+$, so loop the 1s on the $Q = 0$ half of the map to get J. If $Q = 1$, then $K = (Q^+)'$, so loop the 0s on the $Q = 1$ half of the map to get K. The J and K equations will be independent of Q, since Q is set to a constant value (0 or 1) when reading J and K. To make reading the Js and Ks off the map easier, we cross off the Q values on each map. In effect, using the shortcut method is equivalent to splitting the four-variable Q^+ map into two three-variable maps, one for $Q = 0$ and one for $Q = 1$.

Figure 1-21 Derivation of J-K Input Equations



(a) Derivation using separate J-K maps



(b) Derivation using the shortcut method

The following summarizes the steps required to design a sequential network:

1. Given the design specifications, determine the required relationship between the input and output sequences. Then find a state graph and state table.
2. Reduce the table to a minimum number of states. First eliminate duplicate rows by row matching; then form an implication table and follow the procedure in Section 1.9.
3. If the reduced table has m states ($2^{n-1} < m \leq 2^n$), n flip-flops are required. Assign a unique combination of flip-flop states to correspond to each state in the reduced table.
4. Form the transition table by substituting the assigned flip-flop states for each state in the reduced state tables. The resulting transition table specifies the next states of the flip-flops and the output in terms of the present states of the flip-flops and the input.
5. Plot next-state maps and input maps for each flip-flop and derive the flip-flop input equations. Derive the output functions.
6. Realize the flip-flop input equations and the output equations using the available logic gates.
7. Check your design using computer simulation or another method.

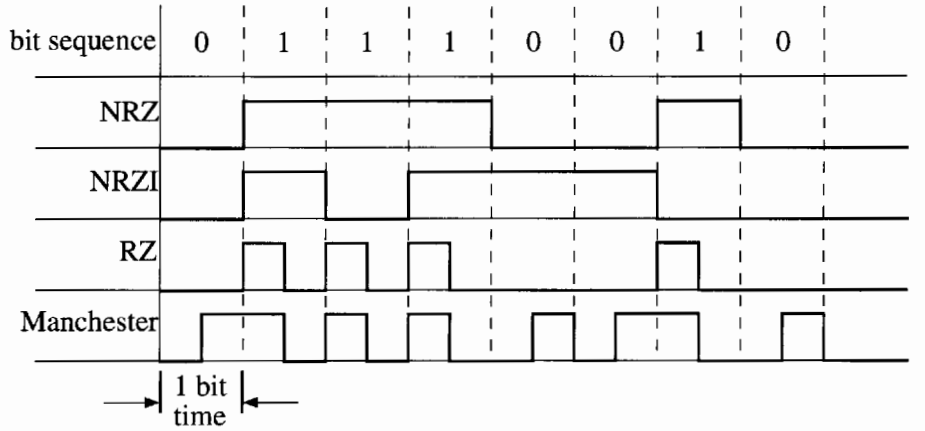
Steps 2 through 7 may be carried out using a suitable CAD program.

1.8 DESIGN OF A MOORE SEQUENTIAL NETWORK

As an example of designing a Moore sequential machine, we will design a converter for serial data. Binary data is frequently transmitted between computers as a serial stream of bits. Figure 1-22 shows four different coding schemes for serial data. The example shows transmission of the bit sequence 0, 1, 1, 1, 0, 0, 1, 0. With the NRZ (nonreturn-to-zero) code, each bit is transmitted for one bit time without any change. With the NRZI (nonreturn-to-zero-inverted) code, data is encoded by the presence or absence of transitions in the data signal. For each 0 in the original sequence, the bit transmitted is the same as the previous bit transmitted. For each 1 in the original sequence, the bit transmitted is the complement of the previous bit transmitted. For the RZ (return-to-zero) code, a 0 is transmitted as 0 for one full bit time, but a 1 is transmitted as a 1 for the first half of the bit time, and then the signal returns to 0 for the second half. For the Manchester code, a 0 is transmitted as 0 for the first half of the bit time and a 1 for the second half, but a 1 is transmitted as a 1 for the first half and a 0 for the second half. Thus, the Manchester encoded bit always changes in the middle of the bit time.

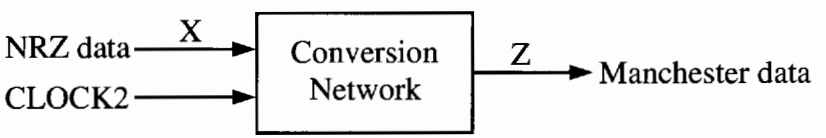
We will design a Moore sequential network that converts an NRZ-coded bit stream to a Manchester-coded bit stream (Figure 1-23). In order to do this, we will use a clock (*CLOCK2*) that is twice the frequency of the basic bit clock. If the NRZ bit is 0, it will be 0 for two *CLOCK2* periods, and if it is 1, it will be 1 for two *CLOCK2* periods. Thus, starting in the reset state (S_0), the only two possible input sequences are 00 and 11, and the corresponding output sequences are 01 and 10. When a 0 is received, the network goes to S_1 and outputs a 0; when the second 0 is received, it goes to S_2 and outputs a 1. Starting in S_0, if a 1 is received, the network goes to S_3 and outputs a 1, and when the second 1 is received, it must go to a state with a 0 output. Going back to S_0 is appropriate since S_0 has

Figure 1-22 Coding Schemes for Serial Data Transmission

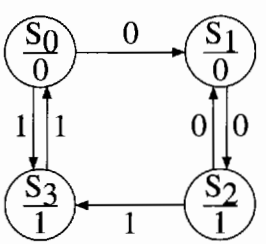


a 0 output and the network is ready to receive another 00 or 11 sequence. When in S_2, if a 00 sequence is received, the network can go to S_1 and back to S_2. If a 11 sequence is received in S_2, the network can go to S_3 and then back to S_0. The corresponding Moore state table has two don't cares, which correspond to input sequences that cannot occur.

Figure 1-23 Moore network for NRZ-to-Manchester Conversion



(a) Conversion network



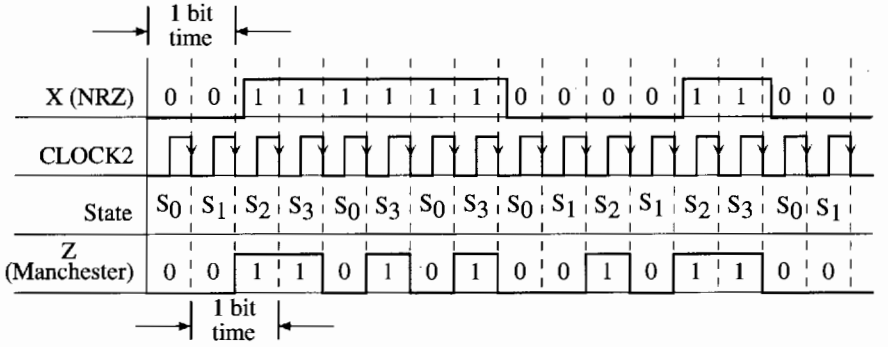
(b) State graph

Present State	Next State X = 0	Next State X = 1	Present Output (Z)
S_0	S_1	S_3	0
S_1	S_2	–	0
S_2	S_1	S_3	1
S_3	–	S_0	1

(c) State table

Figure 1-24 shows the timing chart for the Moore network. Note that the Manchester output is shifted one clock time with respect to the NRZ input. This shift occurs because a Moore network cannot respond to an input until the active edge of the clock occurs. This is in contrast to a Mealy network, for which the output can change after the input changes and before the next clock.

Figure 1-24 Timing for Moore Network



1.9 EQUIVALENT STATES AND REDUCTION OF STATE TABLES

The concept of equivalent states is important for the design and testing of sequential networks. Two states in a sequential network are said to be *equivalent* if we cannot tell them apart by observing input and output sequences. Consider two sequential networks, N_1 and N_2 (see Figure 1-25). N_1 and N_2 could be copies of the same network. N_1 is started in state s_i, and N_2 is started in state s_j. We apply the same input sequence, $\underline{X}$, to both networks and observe the output sequences, $\underline{Z}_1$ and $\underline{Z}_2$. (The underscore notation indicates a sequence.) If $\underline{Z}_1$ and $\underline{Z}_2$ are the same, we reset the networks to states s_i and s_j, apply a different input sequence, and observe $\underline{Z}_1$ and $\underline{Z}_2$. If the output sequences are the same for all possible input sequences, we say the s_i and s_j are equivalent ($s_i \equiv s_j$). Formally, we can define equivalent states as follows: $s_i \equiv s_j$ if and only if, for every input sequence $\underline{X}$, the output sequences $\underline{Z}_1 = \lambda_1(s_i, \underline{X})$ and $\underline{Z}_2 = \lambda_2(s_j, \underline{X})$ are the same. This is not a very practical way to test for state equivalence since, at least in theory, it requires input sequences of infinite length. In practice, if we have a bound on number of states, then we can limit the length of the test sequences.

Figure 1-25 Sequential Networks