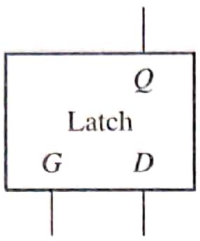
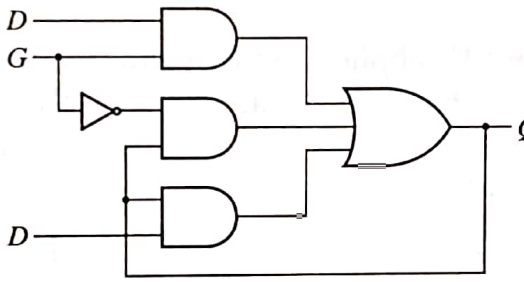


G	D	Q	Q^+
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Figure 1-15: *Transparent D Latch*



$Q^+ = DG + G'Q + (DQ)$

Figure 1-16: *Implementation of D Latch*

1.7 MEALY SEQUENTIAL CIRCUIT DESIGN

There are two basic types of sequential circuits: Mealy and Moore. In a Mealy circuit, the output depend on both the present state and the present inputs. In a Moore circuit, the outputs depend onl on the present state. A general model of a Mealy sequential circuit consists of a combinational circui which generates the outputs and the next state, and a state register, which holds the present state (se

Figure 1-17). The state register normally consists of D flip-flops. The normal sequence of events is (1) the X inputs change to a new value; (2) after a delay, the corresponding Z outputs and next state appears at the output of the combinational circuit; and (3) the next state is clocked into the state register and the state changes. The new state feeds back into the combinational circuit and the process is repeated.

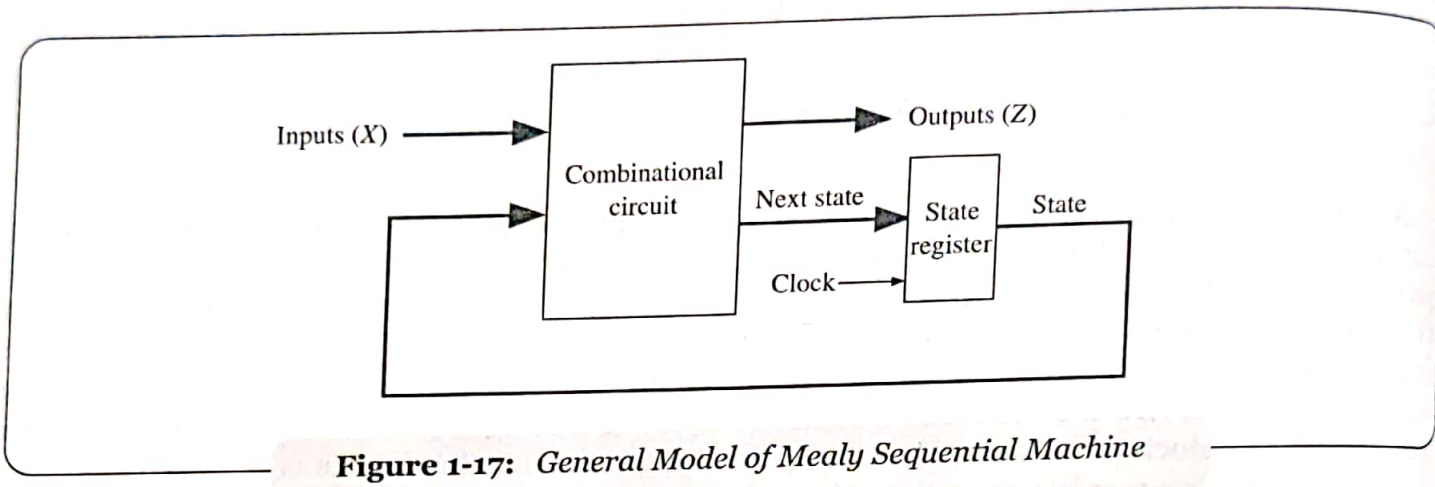


Figure 1-17: *General Model of Mealy Sequential Machine*

1.7.1 Mealy Machine Design Example 1: Sequence Detector

To illustrate the design of a clocked Mealy sequential circuit, let us design a sequence detector. The circuit has the form indicated in the block diagram in Figure 1-18.

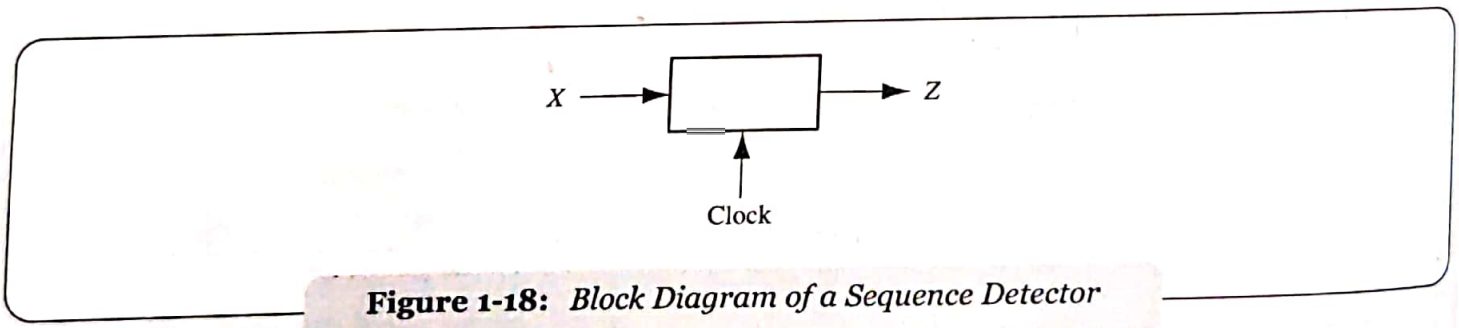


Figure 1-18: *Block Diagram of a Sequence Detector*

The circuit will examine a string of 0's and 1's applied to the X input and generate an output $Z = 1$ only when the input sequence ends in 1 0 1. The input X can change only between clock pulses. The output $Z = 1$ coincides with the last 1 in 1 0 1. The circuit does not reset when a 1 output occurs. A typical input sequence and the corresponding output sequence are

$$X = 0\ 0\ 1\ 1\ 0\ 1\ 1\ 0\ 0\ 1\ 0\ 1\ 0\ 1\ 0\ 0$$
$$Z = 0\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 1\ 0\ 0$$

Let us construct a *state graph* for this sequence detector. We will start in a reset state designated S_0. If a 0 input is received, we can stay in state S_0 as the input sequence we are looking for does not start with 0. However, if a 1 is received, the circuit should go to a new state. Let us denote that state as S_1. When in S_1, if we receive a 0, the circuit must change to a new state (S_2) to remember that the first two inputs of the desired sequence (1 0) have been received. If a 1 is received in state S_2, the desired input sequence is complete and the output should be a 1. The output will be produced as a Mealy output and will coincide with the last 1 in the detected sequence. Since we are designing a Mealy circuit, we are not going to go to a new state that indicates the sequence 101 has been received. When we receive a 1 in S_2, we cannot go to the start state since the circuit is not supposed

to reset with every detected sequence. But the last 1 in a sequence can be the first 1 in another sequence; hence, we can go to state S_1. The partial state graph at this point is indicated in Figure 1-19.

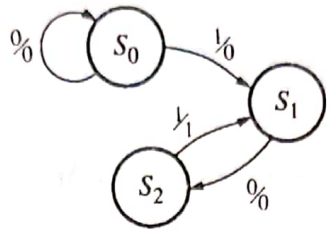


Figure 1-19: *Partial State Graph of the Sequence Detector*

When a 0 is received in state S_2, we have received two 0's in a row and must reset the circuit to state S_0. If a 1 is received when we are in S_1, we can stay in S_1 because the most recent 1 can be the first 1 of a new sequence to be detected. The final state graph is shown in Figure 1-20. State S_0 is the starting state, state S_1 indicates that a sequence ending in 1 has been received, and state S_2 indicates that a sequence ending in 10 has been received. Converting the state graph to a state table yields Table 1-3. In row S_2 of the table, an output of 1 is indicated for input 1.

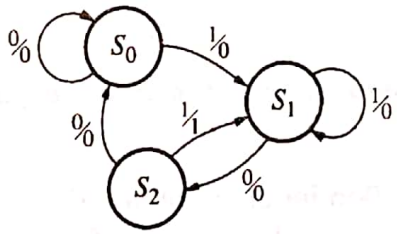


Figure 1-20: *Mealy State Graph for Sequence Detector*

Table 1-3: *State Table for Sequence Detector*

Present State	Next State $X = 0$	Next State $X = 1$	Present Output $X = 0$	Present Output $X = 1$
S_0	S_0	S_1	0	0
S_1	S_2	S_1	0	0
S_2	S_0	S_1	0	1

Next, *state assignment* is performed, whereby specific flip-flop values are associated with specific states. There are two techniques to perform state assignment (1) one-hot state assignment and (2) encoded state assignment. In one-hot state assignment, one flip-flop is used for each state. Hence three flip-flops will be required if this circuit is to be implemented using the one-hot approach. In encoded state assignment, just enough flip-flops to have a unique combination for each state are sufficient. Since we have three states, we need at least two flip-flops to represent all states. We will use encoded state assignment in this design. Let us designate the two flip-flops as A and B. Let the flip-flop states $A = 0$ and $B = 0$ correspond to state S_0; $A = 0$ and $B = 1$ correspond to state S_1; and $A = 1$ and $B = 0$ correspond to state S_2. Now, the transition table of the circuit can be written as in Table 1-4.

Table 1-4: *Transition Table for Sequence Detector*

	A^+B^+		Z	
AB	$X = 0$	$X = 1$	$X = 0$	$X = 1$
00	00	01	0	0
01	10	01	0	0
10	00	01	0	1

From this table, we can plot the K-maps for the next states and the output Z. The next states are typically represented by A^+ and B^+. The three K-maps are shown in Figure 1-21.

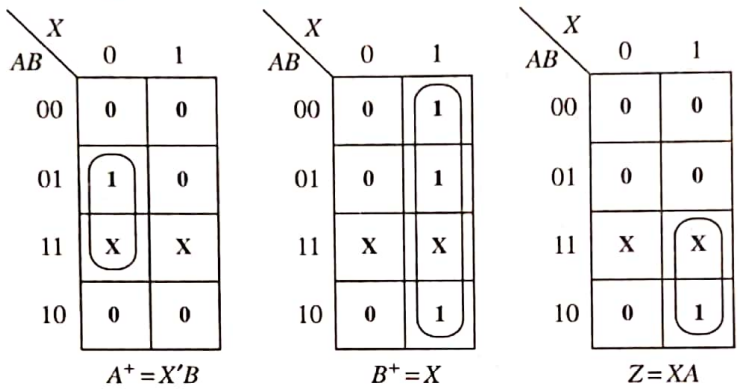


Figure 1-21: *K-Maps for Next States and Output of Sequence Detector*

The next step is deriving the flip-flop inputs to obtain the desired next states. If D flip-flops are used, one simply needs to give the expected next state of the flip-flop to the flip-flop input. So, for flip-flops A and B, $D_A = A^+$ and $D_B = B^+$. The resulting circuit is shown in Figure 1-22.

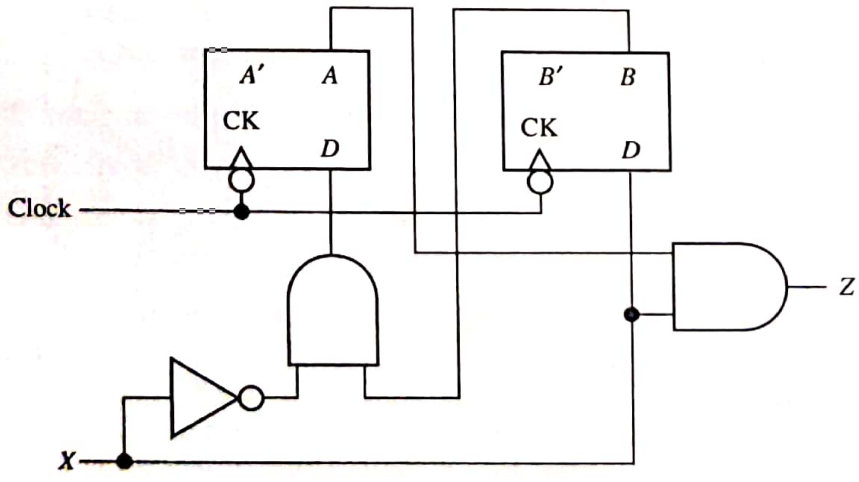


Figure 1-22: *Circuit for Mealy Sequence Detector*

1.7.2 Mealy Machine Design Example 2: BCD to Excess-3 Code Converter

As an example of a more complex Mealy sequential circuit, we will design a serial code converter that converts an 8-4-2-1 binary-coded-decimal (BCD) digit to an excess-3-coded decimal digit. The input (X) will arrive serially with the least significant bit (LSB) first. The outputs will be generated serially as well. Table 1-5 lists the desired inputs and outputs at times t_0, t_1, t_2, and t_3. After receiving four inputs, the circuit should reset to its initial state, ready to receive another BCD digit.

Table 1-5: *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

The excess-3 code is formed by adding 0011 to the BCD digit. For example,

$$\begin{array}{r} 0\ 1\ 0\ 0 \\ +\ 0\ 0\ 1\ 1 \\ \hline 0\ 1\ 1\ 1 \end{array} \qquad \begin{array}{r} 0\ 1\ 0\ 1 \\ +\ 0\ 0\ 1\ 1 \\ \hline 1\ 0\ 0\ 0 \end{array}$$

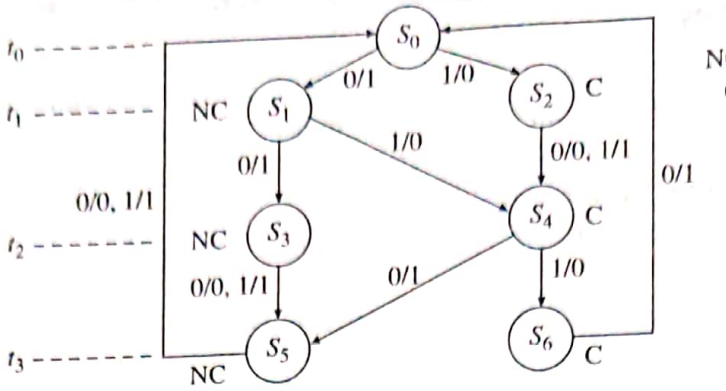
If all of the BCD bits are available simultaneously, this code converter can be implemented as a combinational circuit with four inputs and four outputs. However, here the bits arrive sequentially, one bit at a time. Hence we must implement this code converter sequentially.

Let us now construct a state graph for the code converter (Figure 1-23(a)). Let us designate the start state as S_0. The first bit arrives and we need to add 1 to this bit, as it is the LSB of 0011, the number to be added to the BCD digit to obtain the excess-3 code. 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). Let us use S_1 to indicate no carry after the first addition, and S_2 to indicate a carry of 1 after the addition to the LSB.

At t_1, we add 1 to the next bit, so if there is no carry from the first addition (state S_1), $X = 0$ gives $Z = 0 + 1 + 0 = 1$ and no carry (state S_3), and $X = 1$ gives $Z = 1 + 1 + 0 = 0$ and a carry (state S_4). If there is a carry from the first addition (state S_2), then $X = 0$ gives $Z = 0 + 1 + 1 = 0$ and a carry (S_4), and $X = 1$ gives $Z = 1 + 1 + 1 = 1$ and a carry (S_4).

At t_2, 0 is added to X, and transitions to S_5 (no carry) and S_6 are determined in a similar manner. At t_3, 0 is again added to X, and the circuit resets to S_0.

Figure 1-23(b) gives the corresponding state table. At this point, we should verify that the table has a minimum number of states before proceeding (see Section 1–9). Then state assignment must be performed. Since this state table has seven states, three flip-flops will be required to realize the table in encoded state assignment. In the one-hot approach, one flip-flop is used for each state. Hence seven flip-flops will be required if this circuit is to be implemented using the one-hot approach. The next step is to make a state assignment that relates the flip-flop states to the states



(a) Mealy state graph

PS	NS $X=0$	NS $X=1$	Z $X=0$	Z $X=1$
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 1-23: *State Graph and Table for Code Converter*

in the table. In the sequence detector example, we simply did a straight binary state assignment. Here we are going to look for an optimal assignment. 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 may be preferred. In recent years, with the abundance of transistors on silicon chips, the emphasis on optimal state assignment has been reduced.

In order to reduce the amount of logic required, we will make a state assignment using the following guidelines (see Roth, *Fundamentals of Logic Design*, 5th Ed. [Thomson Brooks/Cole, 2004] for details):

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 1's 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 1-24(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 1-25. Figure 1-26 shows the realization of the code converter using NAND gates and D flip-flops.

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

Figure 1-24: *State Assignment for BCD to Excess-3 Code Converter*

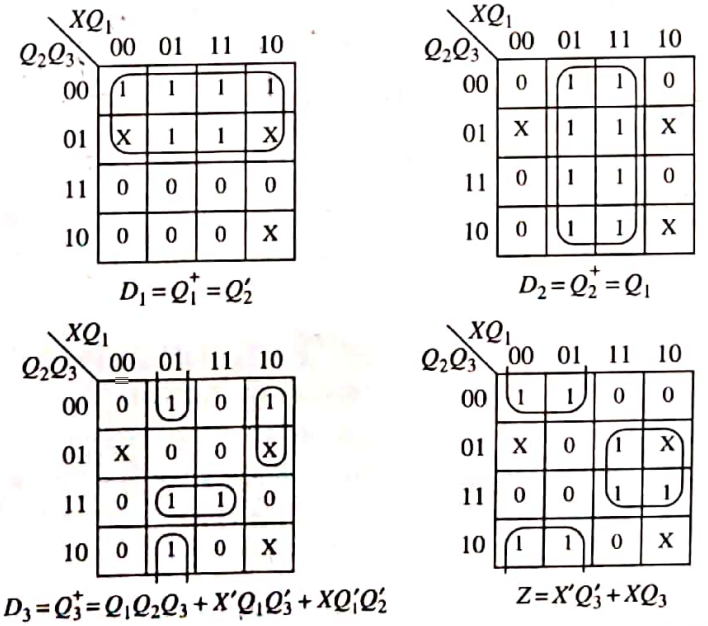


Figure 1-25: *Karnaugh Maps for Code Converter*

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 Table 1-6, also known as the excitation table. This table is derived from the truth table in Figure 1-12.

Figure 1-27 shows derivation of J-K flip-flop input equations for the state table of Figure 1-23 using the state assignment of Figure 1-24. First, we derive the J-K input equations for flip-flop Q_1

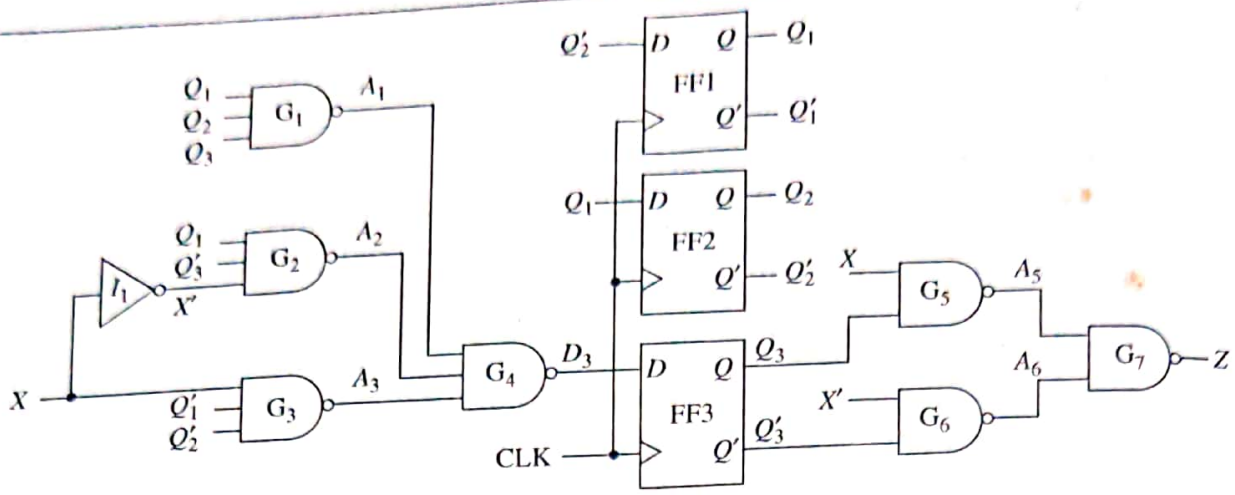
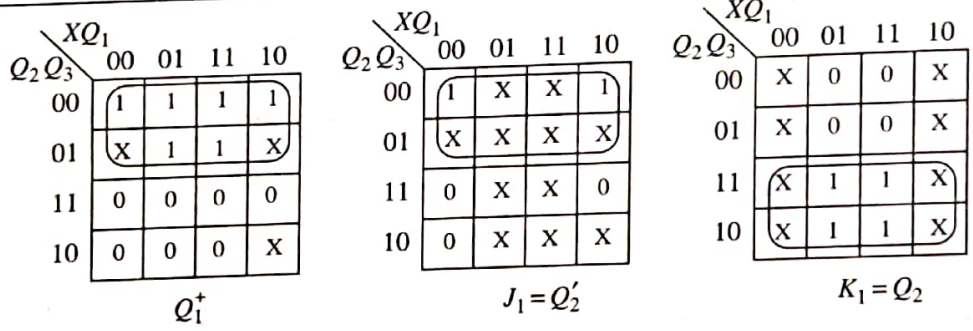


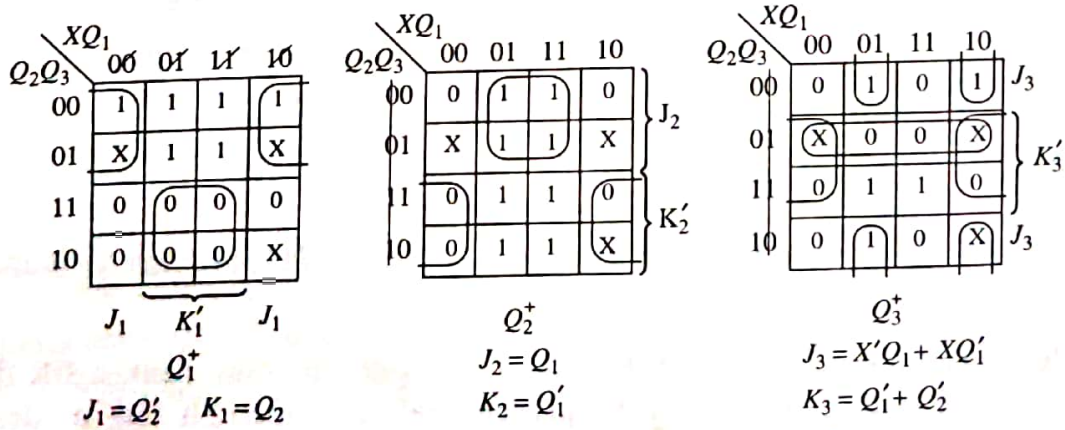
Figure 1-26: *Realization of Code Converter*

Table 1-6: *Excitation Table for a J-K Flip-Flop*

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.)



(a) Derivation using separate J–K maps



(b) Derivation using the shortcut method

Figure 1-27: *Derivation of J-K Input Equations*

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 X's. 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 X's 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 J's and K's directly from the Q^+ maps, as illustrated in Figure 1-27(b). This shortcut method is based on the following: If $Q = 0$, then $J = Q^+$, so loop the 1's on the $Q = 0$ half of the map to get J. If $Q = 1$, then $K = (Q^+)'$, so loop the 0's 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 J's and K's 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 $Q = 1$.

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

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. This is the encoded state assignment technique. Alternately, a one-hot assignment with m flip-flops can be used.
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 computer-aided design (CAD) program.

1.8 Moore Sequential Circuit Design

In a Moore circuit, the outputs depend only on the present state. Moore machines are typically easier to design and debug compared to Mealy machines, but they often contain more states than equivalent Mealy machines. In Moore machines, there are no outputs that happen during the transition. The outputs are associated entirely to the state.

1.8.1 Moore Machine Design Example 1: Sequence Detector

As an example, let us design the sequence detector of Section 1.7.1 using the Moore Method. The circuit will examine a string of 0's and 1's applied to the X input and generate an output $Z = 1$ only when the input sequence ends in 101. The input X can change only between clock pulses. The circuit does not reset when a 1 output occurs.

As in the Mealy machine example, we start in a reset state designated S_0 in Figure 1-28. If a 0 input is received, we can stay in state S_0 as the input sequence we are looking for does not start with 0. However, if a 1 is received, the circuit goes to a new state, S_1. When in S_1, if we receive a 0, the circuit must change to a new state (S_2) to remember that the first two inputs of the desired sequence (10) have been received. If a 1 is received in state S_2, the circuit should go to a new state to indicate

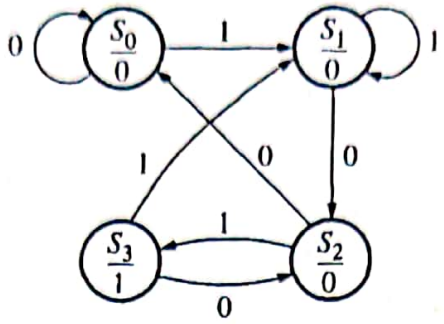


Figure 1-28: *State Graph of the Moore Sequence Detector*

that the desired input sequence is complete. Let us designate this new state as S_3. In state S_3, the output must have a value of 1. The outputs in states S_0, S_1 and S_2 must be 0's. The sequence 100 resets the circuit to S_0. A sequence 1010 takes the circuit back to S_2 because another 1 input should cause Z to become 1 again.

The state table corresponding to the circuit is given by Table 1-7. Note that there is a single column for output because the output is determined by the present state and does not depend on X. Note that this sequence detector requires one more state than the Mealy sequence detector in Table 1-3, which detects the same input sequence.

Table 1-7: *State Table for Sequence Detector*

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

Because there are four states, two flip-flops are required to realize the circuit. Using the state assignment $AB = 00$ for S_0, $AB = 01$ for S_1, $AB = 11$ for S_2, and $AB = 10$ for S_3, the transition table shown in Table 1-8 is obtained.

Table 1-8: *Transition Table for Moore Sequence Detector*

AB	A^+B^+ $X = 0$	A^+B^+ $X = 1$	Z
00	00	01	0
01	11	01	0
11	00	10	0
10	11	01	1

The output function $Z = AB'$. Note that Z depends only on the flip-flop states and is independent of X, while for the corresponding Mealy machine, Z was a function of X. (It was equal to AX in Figure 1-21.) The transition table can be used to write the next state maps and inputs to the flip-flops can be derived.

1.8.2 Moore Machine Design Example 2: NRZ to Manchester Code Converter

As another 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 1-29 shows three 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. In contrast, 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.

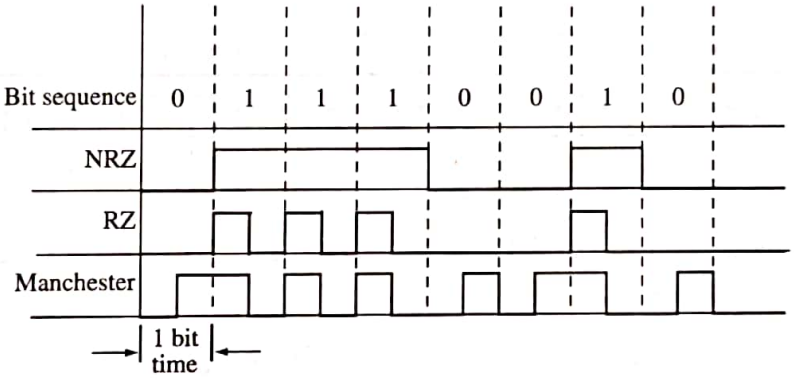
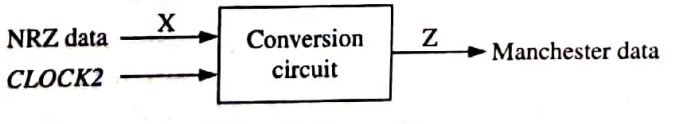
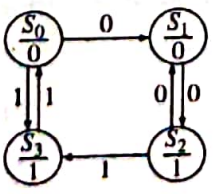


Figure 1-29: *Coding Schemes for Serial Data Transmission*

We will design a Moore sequential circuit that converts an NRZ-coded bit stream to a Manchester-coded bit stream (Figure 1-30). 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



(a) Conversion circuit



(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 1-30: *Moore Circuit for NRZ-to-Manchester Conversion*

01 and 10. When a 0 is received, the circuit 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 circuit 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 a 0 output and the circuit is ready to receive another 00 or 11 sequence. When in S_2, if a 00 sequence is received, the circuit can go to S_1 and then back to S_2. If a 11 sequence is received in S_2, the circuit 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 1-31 shows the timing chart for the Moore circuit. Note that the Manchester output is shifted one clock time with respect to the NRZ input. This shift occurs because a Moore circuit cannot respond to an input until the active edge of the clock occurs. This is in contrast to a Mealy circuit, for which the output can change after the input changes and before the next clock.

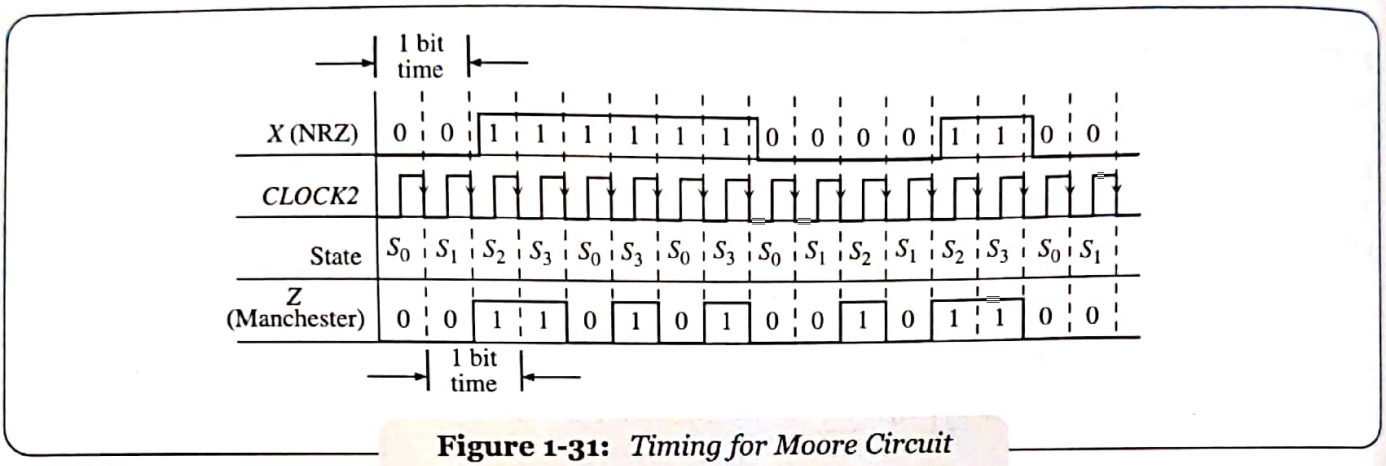


Figure 1-31: *Timing for Moore Circuit*

1.9 Equivalent States and Reduction of State Tables

The concept of equivalent states is important for the design and testing of sequential circuits. It helps to reduce the hardware consumed by circuits. Two states in a sequential circuit are said to be *equivalent* if we cannot tell them apart by observing input and output sequences. Consider two sequential circuits, N_1 and N_2 (see Figure 1-32). N_1 and N_2 could be copies of the same circuit. 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 circuits 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 circuits 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.

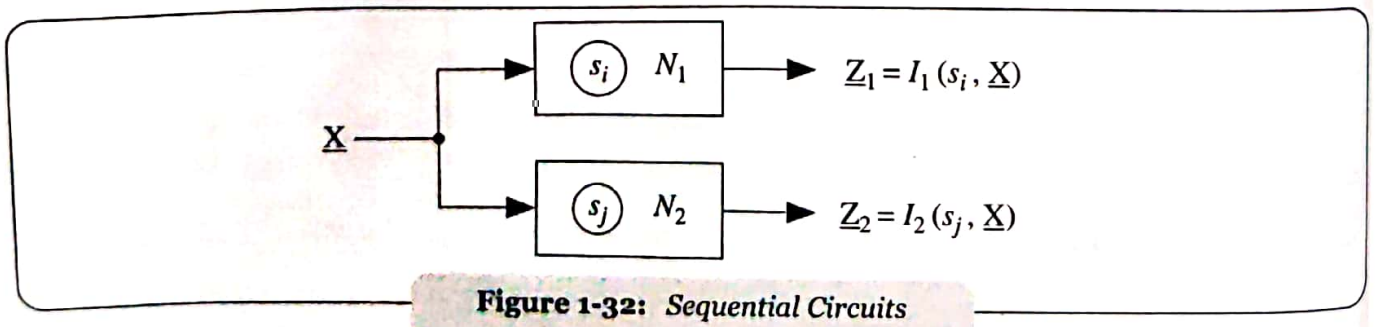


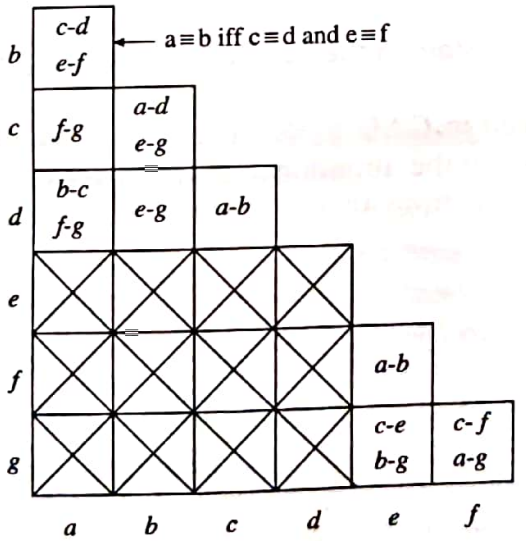
Figure 1-32: *Sequential Circuits*

A more practical way to determine state equivalence uses the state equivalence theorem: $s_i \equiv s_j$ if and only if for every single input X, the outputs are the same and the next states are equivalent. When using the definition of equivalence, we must consider all input sequences, but we do not need any information about the internal state of the system. When using the state equivalence theorem, we must look at both the output and next state, but we need to consider only single inputs rather than input sequences.

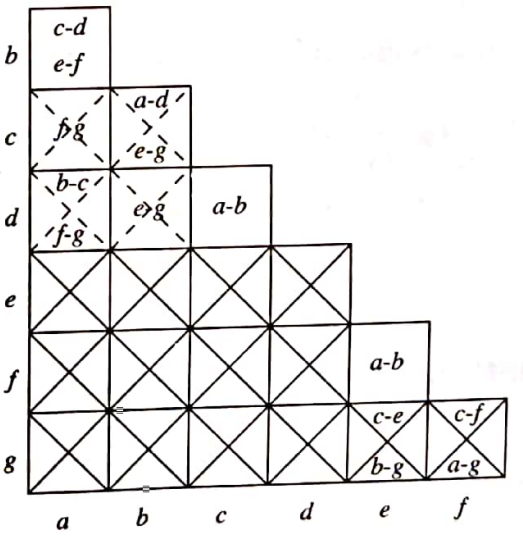
The table of Figure 1-33(a) can be reduced by eliminating equivalent states. First, observe that states a and h have the same next states and outputs when $X = 0$ and also when $X = 1$. Therefore, $a \equiv h$ so we can eliminate row h and replace h with a in the table. To determine if any of the remaining states are equivalent, we will use the state equivalence theorem. From the table, since the outputs for states a and b are the same, $a \equiv b$ if and only if $c \equiv d$ and $e \equiv f$. We say that c-d and e-f are

Present State	Next State $X=0$	1	Present Output $X=0$	1
a	*c*	*f*	0	0
b	*d*	*e*	0	0
c	*a*	*g*	0	0
d	*b*	*g*	0	0
e	*e*	*b*	0	1
f	*f*	*a*	0	1
g	*c*	*g*	0	1
~~*h*~~	~~*c*~~	~~*f*~~	~~0~~	~~0~~

(a) State table reduction by row matching



(b) Implication chart (first pass)



(c) After second and third passes

	$X=$	0	1	$X=$	0	1
a		*c*	*e*		0	0
c		*a*	*g*		0	0
e		*e*	*a*		0	1
g		*c*	*g*		0	1

(d) Final reduced table

Figure 1-33: ***State Table Reduction***

implied pairs for a-b. To keep track of the implied pairs, we make an *implication chart*, as shown in Figure 1-33(b). We place c-d and e-f in the square at the intersection of row a and column b to indicate the implication. Since states d and e have different outputs, we place an X in the d-e square to indicate that $d \neq e$. After completing the implication chart in this way, we make another pass through the chart. The e-g square contains c-e and b-g. Since the c-e square has an X, $c \neq e$, which implies $e \neq g$, so we X out the e-g square. Similarly, since $a \neq g$, we X out the f-g square. On the next pass through the chart, we X out all the squares that contain e-g or f-g as implied pairs (shown on the chart with dashed x's). In the next pass, no additional squares are X'ed out, so the process terminates. Since all the squares corresponding to non-equivalent states have been X'ed out, the coordinates of the remaining squares indicate equivalent state pairs. From the first column, $a \equiv b$; from third column, $c \equiv d$; and from the fifth column, $e \equiv f$.

The implication table method of determining state equivalence can be summarized as follows:

1. Construct a chart that contains a square for each pair of states.
2. Compare each pair of rows in the state table. If the outputs associated with states i and j are different, place an X in square i-j to indicate that $i \neq j$. If the outputs are the same, place the implied pairs in square i-j. (If the next states of i and j are m and n for some input x, then m-n is an implied pair.) If the outputs and next states are the same (or if i-j implies only itself), place a check ($\surd$) in square i-j to indicate that $i \equiv j$.
3. Go through the table square by square. If square i-j contains the implied pair m-n, and square m-n contains an X, then $i \neq j$, and an X should be placed in square i–j.
4. If any X's were added in step 3, repeat step 3 until no more X's are added.
5. For each square i-j that does not contain an X, $i \equiv j$.

If desired, row matching can be used to partially reduce the state table before constructing the implication table. Although we have illustrated this procedure for a Mealy table, the same procedure applies to a Moore table.

Two sequential circuits are said to be equivalent if every state in the first circuit has an equivalent state in the second circuit, and vice versa.

Optimization techniques such as this are incorporated in CAD tools. The importance of state minimization has slightly diminished in recent years due to the abundance of transistors on chips; however, it is still important to do obvious state minimizations to reduce the circuit's area and power.