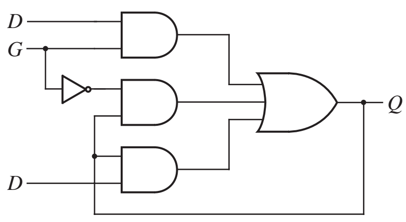


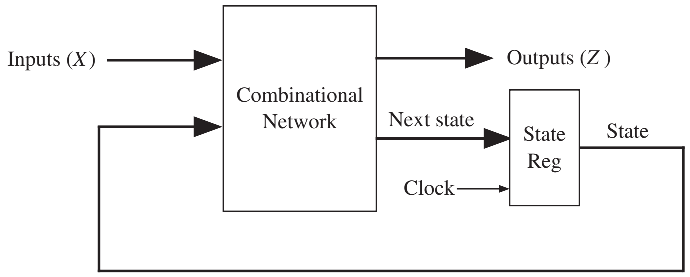
$Q^+ = DG + G'Q + (DQ)$

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 outputs depend on both the present state and the present inputs. In a Moore circuit, the outputs depend only on the present state. A general model of a Mealy sequential circuit consists of a combinational circuit, which generates the outputs and the next state, and a state register, which holds the present state (see 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 appear 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.

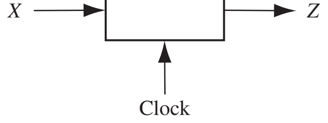
FIGURE 1-17: General Model of Mealy Sequential Machine



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

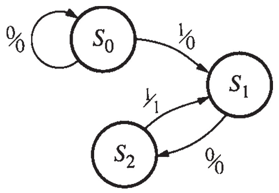


The circuit will examine a string of 0s and 1s applied to the X input and generate an output $Z=1$ only when the input sequence is 101. The input X can change only between clock pulses. The output $Z=1$ coincides with the last 1 in 101. The circuit does not reset when a 1 output occurs. A typical input sequence and the corresponding output sequence are shown here:

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

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 indicate that the first two inputs of the desired sequence (10) 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 0s 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

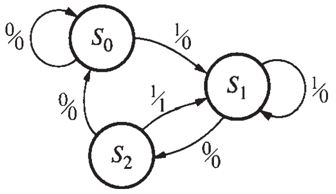


TABLE 1-3: State Table for Sequence Detector

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: (i) one-hot state assignment and (ii) 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 3 K-maps are shown in Figure 1-21.

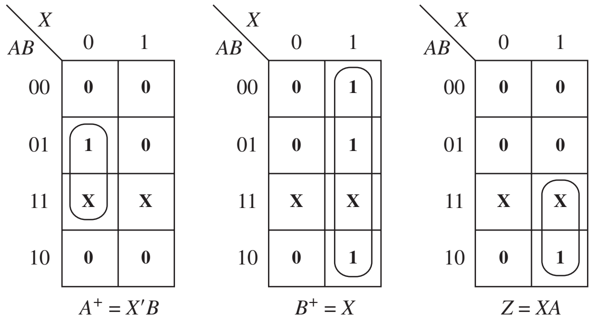
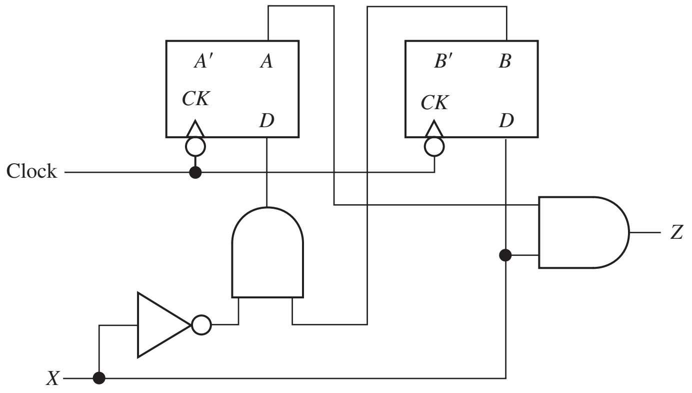


FIGURE 1-21: K-Maps for Next States and Output of Sequence Detector

The next step is deriving the flip-flop inputs to obtain the next states shown in Figure 1-21. 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-flop A, $D_A = A^+$ and $D_B = B^+$. The resulting circuit is shown in Figure 1-22.

FIGURE 1-22: Circuit for Mealy Sequence Detector



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 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,

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

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. Here, however, the bits arrive sequentially, one bit at a time. Hence this code converter must be implemented sequentially.

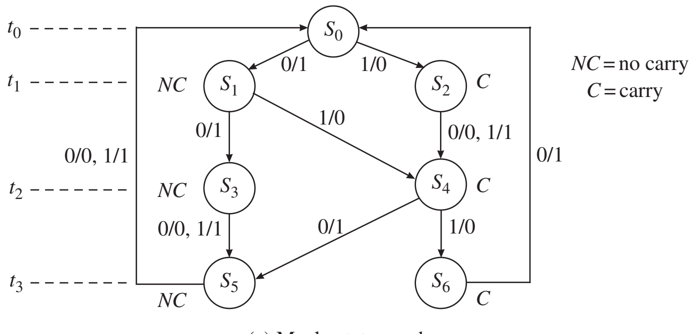
Let us now construct a state graph for the code converter (Figure 1-23(a)). Let us designate the start state as S_0. The first bit arrives, and one needs to add 1 to this bit, as it is the least significant bit (LSB) of 0011, the number to be added to the BCD digit to obtain the excess-3 code.

At t_0, we add 1 to the LSB, 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: State Graph and Table for Code Converter



(a) Mealy state graph

	NS		Z	
PS	X=0	X=1	X=0	X=1
S_0	S_1	S_2	1	0
S_1	S_3	S_4	1	0
S_2	S_4	S_4	0	1
S_3	S_5	S_5	0	1
S_4	S_5	S_6	1	0
S_5	S_0	S_0	0	1
S_6	S_0	–	1	–

(b) State table

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 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 *Fundamentals of Logic Design,* 7th ed., Cengage Learning, 2014 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 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 1-24(a) gives an assignment map, which satisfies the guidelines, and the corresponding transition table (Figure 1-24(b)). 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.

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

$Q_2Q_3 \backslash Q_1$	0	1
00	S_0	S_1
01		S_2
11	S_5	S_3
10	S_6	S_4

(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

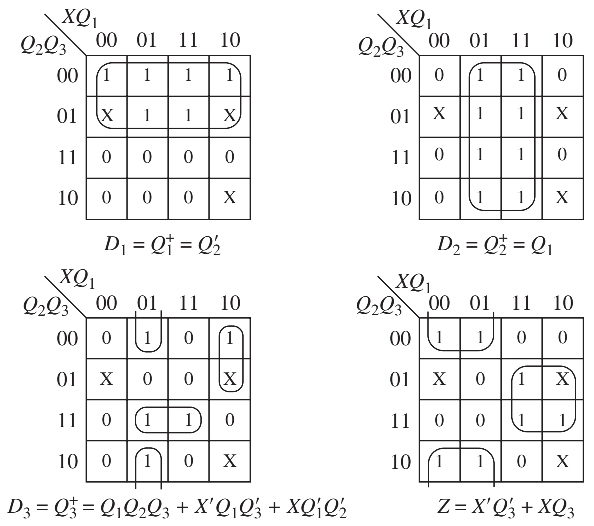


FIGURE 1-25: Karnaugh Maps for Figure 1-23

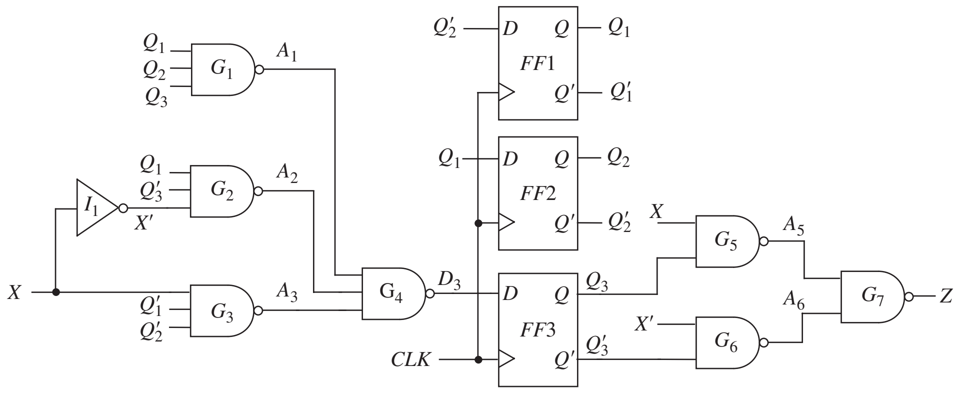


FIGURE 1-26: Realization of Code Converter

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.

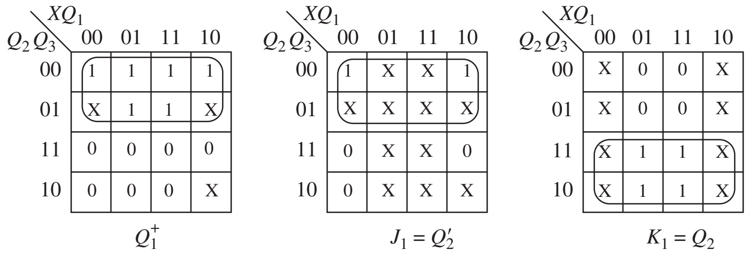
TABLE 1-6 Excitation Table for a J-K Flip-Flop

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.)

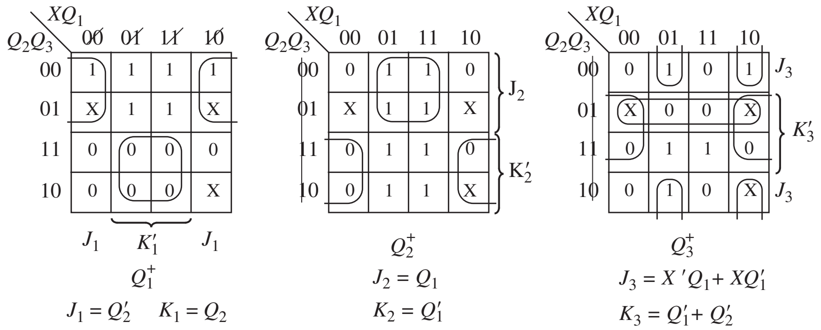
Figure 1-27 shows the derivation of J-K flip-flops for the state table of Figure 1-23 using the state assignment of Figure 1-24. First, we derive the J-K input equations for flip-flop Q_1 using the Q_1^+ map as the starting point. From the preceding table, whenever Q_1 is 0, $J = Q_1^+$ and $K = X$. Therefore, we can fill in the

$Q_1 = 0$ half of the J_1 map the same as with $Q_1{}^+$ 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 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 consists of 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 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 *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 $Q = 1$.

FIGURE 1-27: 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 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 ($2^{n-1} < m \le 2^n$), *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 Design of a Moore Sequential Circuit

In a Moore circuit, the outputs depend only on the present state. Moore machines are typically easier to design and debug than 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.

Moore Machine Design Example 1: Sequence Detector

As an example of a simple Moore circuit, let us design a sequence detector. The circuit will examine a string of 0s and 1s applied to the X input and generate an output $Z=1$ only when the input sequence is 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 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 0s. The sequence 100 resets the circuit to S_0. A sequence 1010 takes the circuit back to S_2 because another 1 input would cause Z to become 1 again.

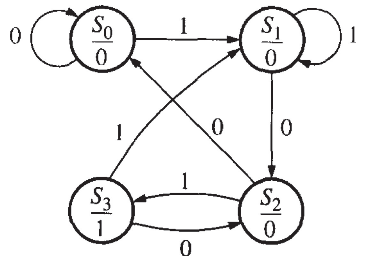


FIGURE 1-28: State Graph of the Moore Sequence Detector

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 following transition table is obtained.

TABLE 1-8: Transition Table for Moore Sequence Detector

	A^+B^+		
AB	$X = 0$	$X = 1$	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). The transition table can be used to write the next state maps and inputs to the flip-flops can be derived.

Moore Machine Design Example 2: NRZ to Manchester Code Converter

As another example of designing a more complex 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 as a 0 for the second half. Thus, the Manchester encoded bit always changes in the middle of the bit time.

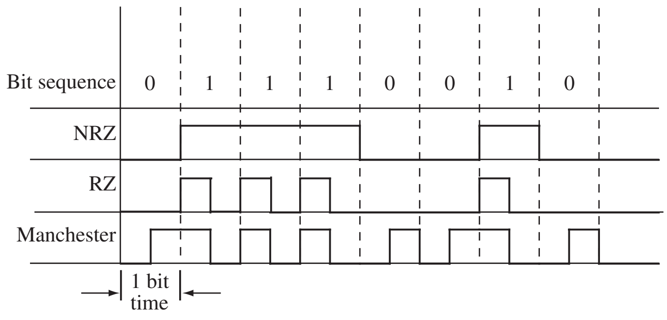
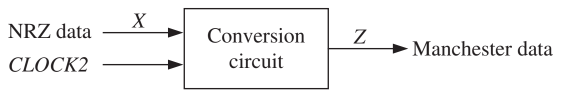


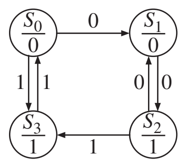
FIGURE 1-29: Coding Schemes for Serial Data Transmission

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



(a) Conversion circuit



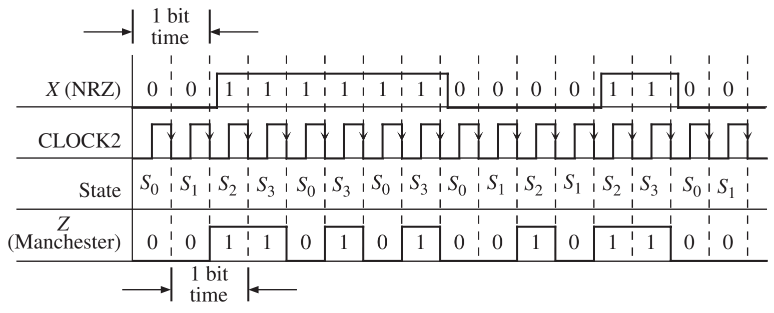
(b) State graph

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 1-31 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

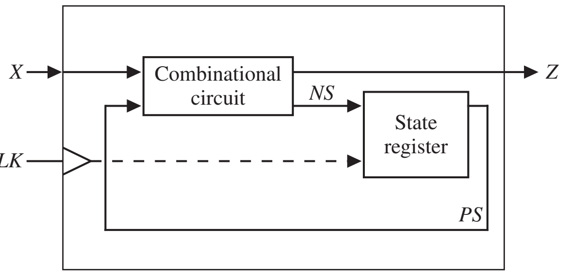


2.15.1 Modeling a Sequential Machine

In this section, we discuss several ways of writing Verilog descriptions for sequential machines. Let us assume that we have to write a **behavioral** model for a Mealy sequential circuit represented by the state table in Figure 2-51 (one may note that this is the BCD to Excess-3 code converter designed in Chapter 1). A block diagram of this state machine is also shown in Figure 2-51. This view of the circuit can be used to write its entity description. Please note that the current state and next state are not visible externally.

FIGURE 2-51: State Table and Block Diagram of Sequential Machine

	NS		*Z*	
PS	$X = 0$	$X = 1$	$X = 0$	$X = 1$
S_0	S_1	S_2	1	0
S_1	S_3	S_4	1	0
S_2	S_4	S_4	0	1
S_3	S_5	S_5	0	1
S_4	S_5	S_6	1	0
S_5	S_0	S_0	0	1
S_6	S_0	—	1	—



There are several ways to model this sequential machine. One approach would be to use two **always** blocks to represent the two parts of the circuit. One **always** block models the combinational part of the circuit and generates the next state information and outputs. The other **always** block models the state register and updates the state at the appropriate edge of the clock. Figure 2-52 illustrates such a model for this Mealy machine. The first always block represents the combinational circuit. At the behavioral level, we will represent the state and next state of the circuit by integer signals initialized to 0. Please remember that this initialization is meaningful only for simulations. Since the circuit outputs, *Z* and *Nextstate*, can change when either the *State* or *X* changes, the sensitivity list includes both *State* and *X*. The case statement tests the value of *State*, and depending on the value of *X*, *Z* and *Nextstate* are assigned new values. The second always block represents the state register. Whenever the rising edge of the clock occurs, *State* is updated to the value of *Nextstate*, so *CLK* appears in the sensitivity list. The second always block will simulate correctly if written as

```
always @ (CLK)          // State Register
begin                   // rising edge of clock (simulation)
  State <= Nextstate;
end
```

but in order to synthesize with edge-triggered flip-flops, the `event expression` `(posedge)` or `negedge` must be used, as in

```
always @ (posedge CLK)          // State Register
begin                           // (synthesis)
  State <= Nextstate;           // rising edge of clock
end
```

In Figure 2-52, *State* is defined as 3-bit register. If *State* is defined as an integer, some synthesis tools may synthesize it to be a 32-bit register, but most modern synthesis tools will figure out that only values less than 7 are used and will synthesize it to a 3-bit register. The statement **`default`** is not actually needed when the outputs

FIGURE 2-52: Behavioral Model for Excess-3 Code Converter

```
// This is a behavioral model of a Mealy state machine (Figure 2-51)
// based on its state table. The output (Z) and next state are
// computed before the active edge of the clock. The state change
// occurs on the rising edge of the clock.

module Code_Converter(X, CLK, Z);

input X, CLK;
output Z;

reg   Z;
reg   [2:0] State;
reg   [2:0] Nextstate;

initial
begin
  State = 0;
  Nextstate = 0;
end

always @(State or X)
begin                           // Combinational Circuit
case(State)
  0 : begin
          if(X == 1'b0)
          begin
               Z = 1'b1;
               Nextstate = 1;
          end
          else
          begin
               Z = 1'b0;
               Nextstate = 2;
          end
  end
  1 : begin
          if(X == 1'b0)
          begin
               Z = 1'b1;
               Nextstate = 3;
          end
          else
          begin
               Z = 1'b0;
               Nextstate = 4;
          end
  end
  2 : begin
          if(X == 1'b0)
          begin
               Z = 1'b0;
```

```
                    Nextstate = 4;
                end
                else
                begin
                    Z = 1'b1;
                    Nextstate = 4;
                end
    end
    3 : begin
                if(X == 1'b0)
                begin
                    Z = 1'b0;
                    Nextstate = 5;
                end
                else
                begin
                    Z = 1'b1;
                    Nextstate = 5;
                end
    end
    4 : begin
                if(X == 1'b0)
                begin
                     Z = 1'b1;
                     Nextstate = 5;
                 end
                 else
                 begin
                     Z = 1'b0;
                     Nextstate = 6;
                 end
    end
    5 : begin
                 if(X == 1'b0)
                 begin
                     Z = 1'b0;
                     Nextstate = 0;
                 end
                 else
                 begin
                     Z = 1'b1;
                     Nextstate = 0;
                 end
    end
    6 : begin
                 if(X == 1'b0)
                 begin
                     Z = 1'b1;
                     Nextstate = 0;
                 end
                 else
```

```
            begin
                Z = 1'b0;
                Nextstate = 0;
            end
            end
  default : begin
  // should not occur
  end
endcase
end
  always @(posedge CLK)      // State Register
  begin                      // rising edge of clock
  State <= Nextstate;
  end
endmodule
```

and next states of all possible values of *State* are explicitly specified. In this case, however, it should be included because one of the eight states is not specified as a case. The empty statement implies no action, which is appropriate, since the other values of *State* should never occur. In case statements, the *State* variable should be bound to a specific number of bits, or results will be unpredictable.

A simulator command file that can be used to test Figure 2-52 is as follows:

```
add wave CLK X State Nextstate Z
force CLK 0 0, 1 100 -repeat 200
force X 0 0, 1 350, 0 550, 1 750, 0 950, 1 1350
run 1600
```

The first command specifies the signals that are to be included in the waveform output. The next command defines a clock with a period of 200 ns. *CLK* is 0 at time 0 ns, is 1 at time 100 ns, and repeats every 200 ns. In a command of the form

```
force signal_name v1 t1, v2 t2, ...
```

`signal_name` gets the value `v1` at time `t1`, the value `v2` at time `t2`, and so forth. *X* is 0 at time 0 ns, changes to 1 at time 350 ns, changes to 0 at time 550 ns, and so on. The *X* input corresponds to the sequence 0010 1001, and only the times at which *X* changes are specified. Execution of the preceding command file produces the waveforms shown in Figure 2-53.

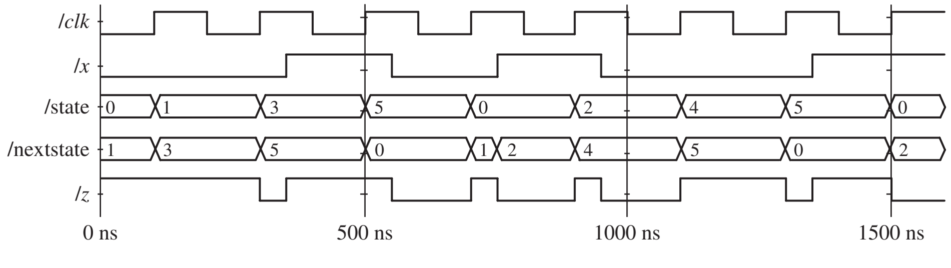


FIGURE 2-53: Simulator Output for Excess-3 Code Converter

In Chapter 1, we manually designed this state machine (Figure 1-26). This circuitry contained three flip-flops, four 3-input NAND gates, two 3-input NAND gates, and one inverter. The behavioral model of Figure 2-52 may not result in exactly that circuit. In fact, when we synthesized it using Xilinx ISE tools, we obtained a circuit that contains seven D-flip-flops, fifteen 2-input AND gates, three 2-input OR gates, and one 7-input OR gate. Apparently, the Xilinx synthesis tool may be using one-hot design by default, instead of encoded design. One-hot design is a popular approach for FPGAs, where flip-flops are abundant.

Figure 2-54 shows an alternative behavioral model for the code converter that uses a single **always** block instead of two **always** blocks. The next state is not computed explicitly, but instead the state register is updated directly to the proper next state value on the rising edge of the clock. Since *Z* can change whenever State or *X* changes, *Z* should not be computed in the clocked block. Instead, we have used a conditional assignment statement to compute *Z*. If *Z* were updated in the clocked block, then a flip-flop would be created to store *Z* and *Z* would be updated at the wrong time. In general, the two always block model for a state machine is preferable

FIGURE 2-54: Behavioral Model for Code Converter Using a Single Process

```
// This is a behavioral model of the Mealy state machine for BCD to
// Excess-3 Code Converter based on its state table. The state change
// occurs on the rising edge of the clock. The output is computed by a
// conditional assignment statement whenever State or Z changes.
module Code_Converter(X, CLK, Z);
input X, CLK;
output Z;
wire   X;           // Redundant declaration of X, CLK and Z.
wire   CLK;         // Input and output signals are wire data
wire   Z;           // type by default
reg    [2:0] State;
initial
begin
  State = 0;
end
always @(posedge CLK) begin
  case(State)
    0 : begin
      if(X == 1'b0)
        State <= 1;
      else
        State <= 2;
    end
    1 : begin
      if(X == 1'b0)
        State <= 3;
      else
        State <= 4;
    end
    2 :
      State <= 4;
    3 :
      State <= 5;
    4 : begin
```

```
      if(X == 1'b0)
        State <= 5;
      else
        State <= 6;
    end
    5 :
      State <= 0;
    6 :
      State <= 0;
    default : begin
    end
  endcase
end
  assign Z = (State == 0 && X == 1'b0) || (State == 1 && X == 1'b0) ||
  (State == 2 && X == 1'b1) || (State == 3 && X == 1'b1) ||
  (State == 4 && X == 1'b0) || (State == 5 && X == 1'b1) ||
  State == 6 ? 1'b1 : 1'b0;
endmodule
```

to the one always block model, since the former corresponds more closely to the hardware implementation which uses a combinational circuit and a state register.

Another way to model this Mealy machine is using the **data flow** approach (i.e., using equations). The data flow Verilog model of Figure 2-55 is based on the next state and output equations, which are derived in Chapter 1 (Figure 1-25). The flip-flops are updated in an always block that is sensitive to *CLK*. When the rising edge of the clock occurs, Q_1, Q_2, and Q_3 are all assigned new values. A 10 ns delay is included

FIGURE 2-55: Sequential Machine Model Using Equations

```
// The following is a description of the sequential machine of
// the BCD to Excess-3 code converter in terms of its next state
// equations obtained as in Figure 1-25. The following state assignment was
// used: S0-->0; S1-->4; S2-->5; S3-->7; S4-->6; S5-->3; S6-->2

module Code_Converter(X, CLK, Z);
   input   X;
   input   CLK;
   output  Z;

   reg     Q1;
   reg     Q2;
   reg     Q3;

   always @(posedge CLK)

   begin
     Q1 <= #10 (~Q2);
     Q2 <= #10 Q1;
     Q3 <= #10 (Q1 & Q2 & Q3) | ((~X) & Q1 & (~Q3)) |
            (X & (~Q1) & (~Q2));
   end

   assign #20 Z =  ((~X) & (~Q3)) | (X & Q3);

endmodule
```

to represent the propagation delay between the active edge of the clock and the change of the flip-flop outputs. Even though the statements in the always block are executed sequentially, Q_1, Q_2, and Q_3 are all scheduled to be updated at the same time, $T + \Delta$, where T is the time at which the rising edge of the clock occurred. Thus, the old value of Q_1 is used to compute Q_2^+, and the old values of Q_1, Q_2, and Q_3 are used to compute Q_3^+. The concurrent assignment statement for Z causes Z to be updated whenever a change in X or Q_3 occurs. The 20 ns delay represents two gate delays. One may note that in order to do Verilog modeling at this level, one needs to perform state assignments, derive next-state equations, etc. In contrast, at the behavioral level, the state table was sufficient to create the Verilog model.

Yet another approach to creating a Verilog model of the foregoing Mealy machine is to create a **structural** model describing the gates and flip-flops in the circuit. Figure 2-58 shows a structural Verilog representation of the circuit of Figure 1-26. One should note that the designer had to manually perform the design and obtain the gate level circuitry here in order to create a model as in Figure 2-58. Seven NAND gates, three D flip-flops, and one inverter are used in the design presented in Figure 1-26. When primitive components like gates and flip-flops are required, each of these components can be defined in a separate Verilog module. One could also use the built-in primitives predefined in Verilog. In this chapter, we are going to build our own building blocks. Predefined built-in primitives are described in Section 8.4. Depending on which CAD tools are used, the component modules must be included in the same file as the main Verilog description, or they must be inserted as separate files in a Verilog project. The code in Figure 2-58 requires component modules DFF, Nand3, Nand2, and Inverter. CAD tools might include modules with similar components. If such modules are used, one should use the exact component names and port-map statements that match the input-output signals of the component. The DFF module is as follows:

FIGURE 2-56: D Flipflop

```
// D Flip-Flop

module DFF(D, CLK, Q, QN);
   input   D;
   input   CLK;
   output  Q;
   output  QN;

   reg     Q;
   reg     QN;

   initial
   begin
     Q  = 1'b0;
     QN = 1'b1;
   end

   always @(posedge CLK)
      begin
         Q  <= #10 D;
         QN <= #10 (~D);
      end

endmodule
```

The Nand3 module is as follows:

FIGURE 2-57: 3-Input NAND Gate

```
// 3 input NAND gate

module Nand3(A1, A2, A3, Z);
   input   A1;
   input   A2;
   input   A3;
   output  Z;

   assign #10 Z = (~(A1 & A2 & A3));

endmodule
```

The Nand2 and Inverter modules are similar except for the number of inputs. We have assumed a 10-ns delay in each component, and this can easily be changed to reflect the actual delays in the hardware being used.

FIGURE 2-58: Structural Model of Sequential Machine

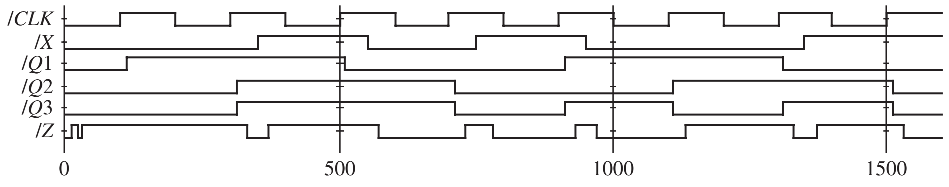
```
// The following is a STRUCTURAL Verilog description of
// the circuit to realize the BCD to Excess-3 code Converter.
// This circuit was illustrated in Figure 1-26.
// Uses components NAND3, NAND2, INVERTER and DFF
// The component modules can be included in the same file
// or they can be inserted as separate files.


module Code_Converter(X, CLK, Z);

input X, CLK;
output Z;
wire   X;
wire   CLK;
wire   Z;
wire   A1;
wire   A2;
wire   A3;
wire   A5;
wire   A6;
wire   D3;
wire   Q1;
wire   Q2;
wire   Q3;
wire   Q1N;
wire   Q2N;
wire   Q3N;
wire   XN;


Inverter I1(X, XN);
Nand3 G1(Q1, Q2, Q3, A1);
```

```
Nand3 G2(Q1, Q3N, XN, A2);
Nand3 G3(X, Q1N, Q2N, A3);
Nand3 G4(A1, A2, A3, D3);
DFF FF1(Q2N, CLK, Q1, Q1N);
DFF FF2(Q1, CLK, Q2, Q2N);
DFF FF3(D3, CLK, Q3, Q3N);
Nand2 G5(X, Q3, A5);
Nand2 G6(XN, Q3N, A6);
Nand2 G7(A5, A6, Z);

endmodule
```

Since *Q1*, *Q2*, and *Q3* are initialized to 0, the complementary flip-flop outputs (*Q1N*, *Q2N*, and *Q3N*) are initialized to 1. *G1* is a 3-input NAND gate with inputs *Q1*, *Q2*, *Q3*, and output *A1*. *FF1* is a D flip-flop with the *D* input connected to *Q2N*. Executing the simulator command file given below produces the waveforms of Figure 2-59, which are very similar to Figure 1-39.

```
add wave CLK X Q1 Q2 Q3 Z
force CLK 0 0, 1 100 -repeat 200
force X 0 0, 1 350, 0 550, 1 750, 0 950, 1 1350
run 1600
```

FIGURE 2-59: Waveforms for Code Converter



If you synthesize this structural description, you will certainly get exactly the same circuit that you had in mind. Now the circuit includes only three D-flip-flops, three 2-input NAND gates, and four 3-input NAND gates. Compare it with the seven D-flip-flops, fifteen 2-input AND gates, three 2-input OR gates, and one 7-input OR gate generated when Figure 2-52 was synthesized. When the designer specified all components and their interconnections, the synthesizer tool did not have to infer or "guess."

Synthesis with integers: Integers are generally treated as 32-bit quantities. While writing synthesizeable code, it is a good idea to use the appropriate number of bits for your variables as needed as opposed to using integers. One should use integers only where they are not explicitly synthesized, eg: looping variables. If integers are used for state variables in the state machine design presented in these pages, the synthesis tools we used could figure out that the state variable is actually only 3-bits long. However, some tools may generate 32-bit registers.

Those who have developed C code with assembly inlining may feel some similarity to the phenomenon occurring here. By inlining the assembly code, one can precisely describe what microprocessor instruction sequence you want to be

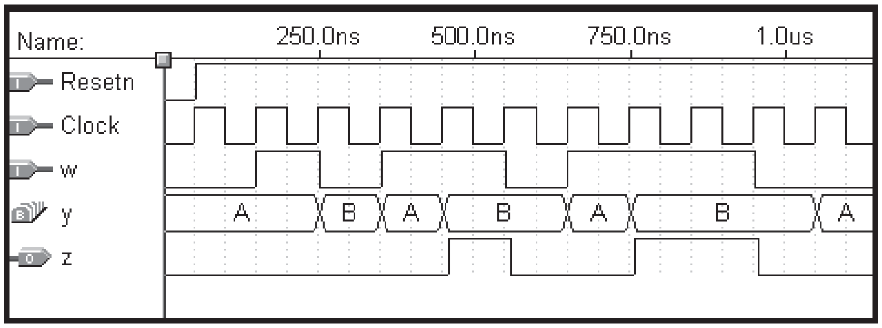


Figure 8.37 Simulation results for the Mealy machine.

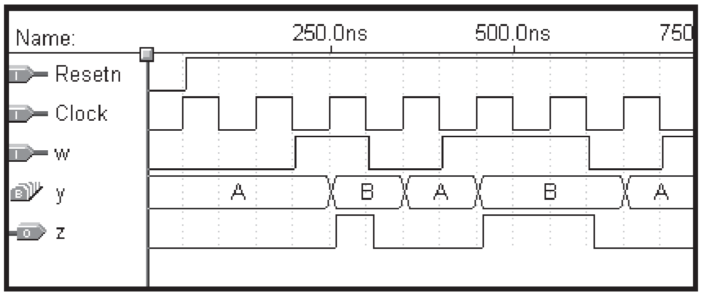


Figure 8.38 Potential problem with asynchronous inputs to a Mealy FSM.

8.5 SERIAL ADDER EXAMPLE

We will now present another simple example that illustrates the complete design process. In Chapter 5 we discussed the addition of binary numbers in detail. We explained several schemes that can be used to add two n-bit numbers in parallel, ranging from carry-ripple to carry-lookahead adders. In these schemes the speed of the adder unit is an important design parameter. Fast adders are more complex and thus more expensive. If speed is not of great importance, then a cost-effective option is to use a *serial adder*, in which bits are added a pair at a time.

8.5.1 MEALY-TYPE FSM FOR SERIAL ADDER

Let $A = a_{n-1}a_{n-2}\cdots a_0$ and $B = b_{n-1}b_{n-2}\cdots b_0$ be two unsigned numbers that have to be added to produce $Sum = s_{n-1}s_{n-2}\cdots s_0$. Our task is to design a circuit that will perform serial addition, dealing with a pair of bits in one clock cycle. The process starts by adding bits a_0 and b_0. In the next clock cycle, bits a_1 and b_1 are added, including a possible carry from the bit-position 0, and so on. Figure 8.39 shows a block diagram of a possible implementation. It includes three shift registers that are used to hold *A*, *B*, and *Sum* as the computation proceeds. Assuming that the input shift registers have parallel-load capability, as depicted in Figure 7.19, the addition task begins by loading the values of *A* and *B* into these registers. Then in each clock cycle, a pair of bits is added by the adder FSM, and at the end of the cycle the resulting sum bit is shifted into the *Sum* register. We will use positive-edge-triggered flip-flops in which case all changes take place soon after the positive edge of the clock, depending on the propagation delays within the various flip-flops. At this time the contents of all three shift registers are shifted to the right; this shifts the existing sum bit into *Sum*, and it presents the next pair of input bits a_i and b_i to the adder FSM.

Now we are ready to design the required FSM. This cannot be a combinational circuit because different actions will have to be taken, depending on the value of the carry from the previous bit position. Hence two states are needed: let *G* and *H* denote the states where the carry-in values are 0 and 1, respectively. Figure 8.40 gives a suitable state diagram, defined as a Mealy model. The output value, *s*, depends on both the state and the present value of the inputs *a* and *b*. Each transition is labeled using the notation ab/s, which indicates the value of *s* for a given valuation *ab*. In state *G* the input valuation 00 will produce $s = 0$, and the FSM will remain in the same state. For input valuations 01 and 10, the output will

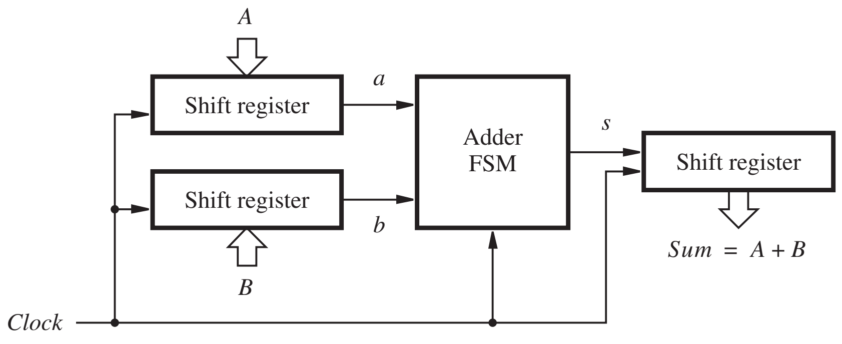


Figure 8.39 Block diagram for the serial adder.

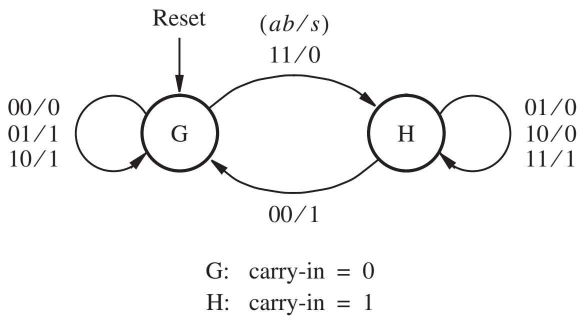


Figure 8.40 State diagram for the serial adder FSM.

be $s = 1$, and the FSM will remain in G. But for 11, $s = 0$ is generated, and the machine moves to state H. In state H valuations 01 and 10 cause $s = 0$, while 11 causes $s = 1$. In all three of these cases, the machine remains in H. However, when the valuation 00 occurs, the output of 1 is produced and a change into state G takes place.

The corresponding state table is presented in Figure 8.41. A single flip-flop is needed to represent the two states. The state assignment can be done as indicated in Figure 8.42. This assignment leads to the following next-state and output equations

$$Y = ab + ay + by$$
$$s = a \oplus b \oplus y$$

Comparing these expressions with those for the full-adder in section 5.2, it is obvious that y is the carry-in, Y is the carry-out, and s is the sum of the full-adder. Therefore, the adder FSM box in Figure 8.39 consists of the circuit shown in Figure 8.43. The flip-flop can be cleared by the *Reset* signal at the start of the addition operation.

The serial adder is a simple circuit that can be used to add numbers of any length. The structure in Figure 8.39 is limited in length only by the size of the shift registers.

8.5.2 MOORE-TYPE FSM FOR SERIAL ADDER

In the preceding example we saw that a Mealy-type FSM nicely meets the requirement for implementing the serial adder. Now we will try to achieve the same objective using a Moore-type FSM. A good starting point is the state diagram in Figure 8.40. In a Moore-type FSM, the output must depend only on the state of the machine. Since in both states, G and H, it is possible to produce two different outputs depending on the valuations of the inputs

Present state	Next state				Output s			
	ab = 00	01	10	11	00	01	10	11
G	G	G	G	H	0	1	1	0
H	G	H	H	H	1	0	0	1

Figure 8.41 State table for the serial adder FSM.

Present state	Next state				Output			
	ab = 00	01	10	11	00	01	10	11
y	Y				s			
0	0	0	0	1	0	1	1	0
1	0	1	1	1	1	0	0	1

Figure 8.42 State-assigned table for Figure 8.41.

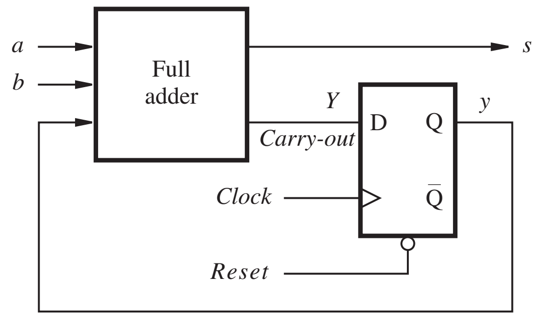


Figure 8.43 Circuit for the adder FSM in Figure 8.39.

a and *b*, a Moore-type FSM will need more than two states. We can derive a suitable state diagram by splitting both *G* and *H* into two states. Instead of *G*, we will use G_0 and G_1 to denote the fact that the carry is 0 and that the sum is either 0 or 1, respectively. Similarly, instead of *H*, we will use H_0 and H_1. Then the information in Figure 8.40 can be mapped into the Moore-type state diagram in Figure 8.44 in a straightforward manner.

The corresponding state table is given in Figure 8.45 and the state-assigned table in Figure 8.46. The next-state and output expressions are

$$\begin{aligned} Y_1 &= a \oplus b \oplus y_2 \\ Y_2 &= ab + ay_2 + by_2 \\ s &= y_1 \end{aligned}$$

The expressions for Y_1 and Y_2 correspond to the sum and carry-out expressions in the full-adder circuit. The FSM is implemented as shown in Figure 8.47. It is interesting to observe that this circuit is very similar to the circuit in Figure 8.43. The only difference is that in the Moore-type circuit, the output signal, *s*, is passed through an extra flip-flop and thus delayed by one clock cycle with respect to the Mealy-type sequential circuit. Recall that we observed the same difference in our previous example, as depicted in Figures 8.26 and 8.27.

A key difference between the Mealy and Moore types of FSMs is that in the former a change in inputs reflects itself immediately in the outputs, while in the latter the outputs do not change until the change in inputs forces the machine into a new state, which takes place one clock cycle later. We encourage the reader to draw the timing diagrams for the circuits in Figures 8.43 and 8.47, which will exemplify further this key difference between the two types of FSMs.

8.5.3 VERILOG CODE FOR THE SERIAL ADDER

The serial adder can be described in Verilog by writing code for the shift registers and the adder FSM. We will first design the shift register and then use it as a subcircuit in the serial adder.

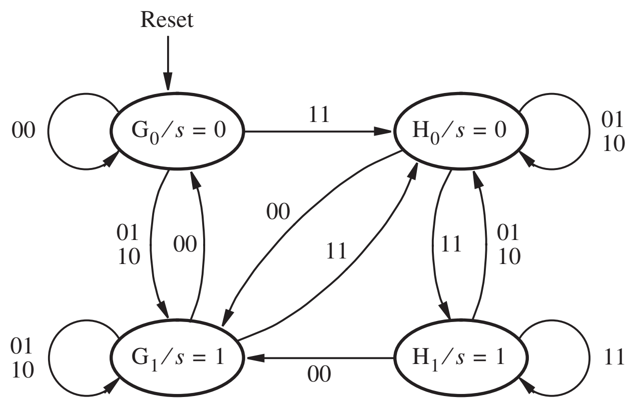


Figure 8.44 State diagram for the Moore-type serial adder FSM.

Present state	Next state				Output
	ab = 00	01	10	11	s
G_0	G_0	G_1	G_1	H_0	0
G_1	G_0	G_1	G_1	H_0	1
H_0	G_1	H_0	H_0	H_1	0
H_1	G_1	H_0	H_0	H_1	1

Figure 8.45 State table for the Moore-type serial adder FSM.

Present state	Next state				Output
	ab = 00	01	10	11	
y_2y_1	Y_2Y_1				s
0 0	0 0	0 1	0 1	1 0	0
0 1	0 0	0 1	0 1	1 0	1
1 0	0 1	1 0	1 0	1 1	0
1 1	0 1	1 0	1 0	1 1	1

Figure 8.46 State-assigned table for Figure 8.45.

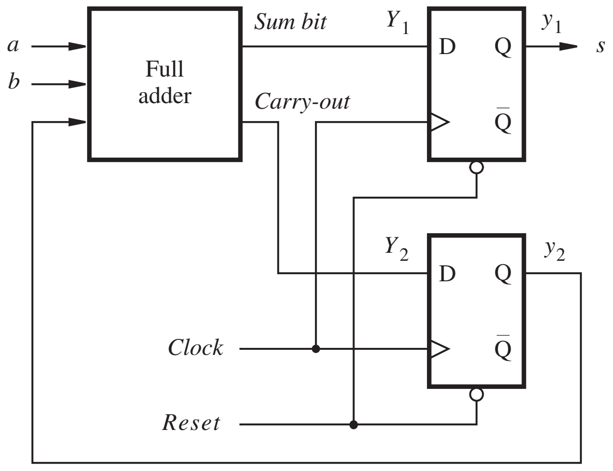


Figure 8.47 Circuit for the Moore-type serial adder FSM.

Shift Register Subcircuit

Figure 7.55 gives Verilog code for an *n*-bit shift register. In the serial adder it is beneficial to have the ability to prevent the shift register contents from changing when an active clock edge occurs. Figure 8.48 gives the code for a shift register named *shiftrne*, which has an enable input, E. When $E = 1$, the shift register behaves in the same way as the one in Figure 7.55. Setting $E = 0$ prevents the contents of the shift register from changing. The E input is usually called the *enable* input. It is useful for many types of circuits, as we will see in Chapter 10.

Complete Code

The code for the serial adder is shown in Figure 8.49. It instantiates three shift registers for the inputs *A* and *B* and the output *Sum*. The shift registers are loaded with parallel data when the circuit is reset. The state diagram for the adder FSM is described by two **always** blocks, using the style of code in Figure 8.36. In addition to the components of the serial adder shown in Figure 8.39, the Verilog code includes a down-counter to determine when the adder should be halted because all *n* bits of the required sum are present in the output shift register. When the circuit is reset, the counter is loaded with the number of bits in the serial adder, *n*. The counter counts down to 0, and then stops and disables further changes in the output shift register.

The code in Figure 8.49 implements a serial adder for eight-bit numbers. The wires Q*A* and Q*B* correspond to the parallel outputs of the shift registers with inputs *A* and *B* in Figure 8.39. The variable *s* represents the output of the adder FSM.

In Figure 8.39 the shift registers for inputs *A* and *B* do not use a serial input or an enable input. However, the *shiftrne* component, which is used for all three shift registers, includes these ports and so signals must be connected to them. The enable input for the two shift registers can be connected to logic value 1. The value shifted into the serial input does not

```
module shiftrne (R, L, E, w, Clock, Q);
   parameter n = 8;
   input [n−1:0] R;
   input L, E, w, Clock;
   output [n−1:0] Q;
   reg [n−1:0] Q;
   integer k;

   always @(posedge Clock)
      if (L)
         Q <= R;
      else if (E)
      begin
         for (k = n−1; k > 0; k = k−1)
            Q[k−1] <= Q[k];
         Q[n−1] <= w;
      end

endmodule
```

Figure 8.48 Code for a left-to-right shift register with an enable input.

matter, so it can be connected to either 1 or 0; we have chosen to connect it to 0. The shift registers are loaded in parallel by the *Reset* signal. We have chosen to use an active-high reset signal for the circuit. The output shift register does not need a parallel data input, so all 0s are connected to this input.

The first **always** block describes the state transitions and the output of the adder FSM in Figure 8.40. The output definition follows from observing in Figure 8.40 that when the FSM is in state G, the sum is $s = a \oplus b$, and when in state H, the sum is $s = \overline{a \oplus b}$. The second **always** block implements the flip-flop y and provides synchronous reset when $Reset = 1$.

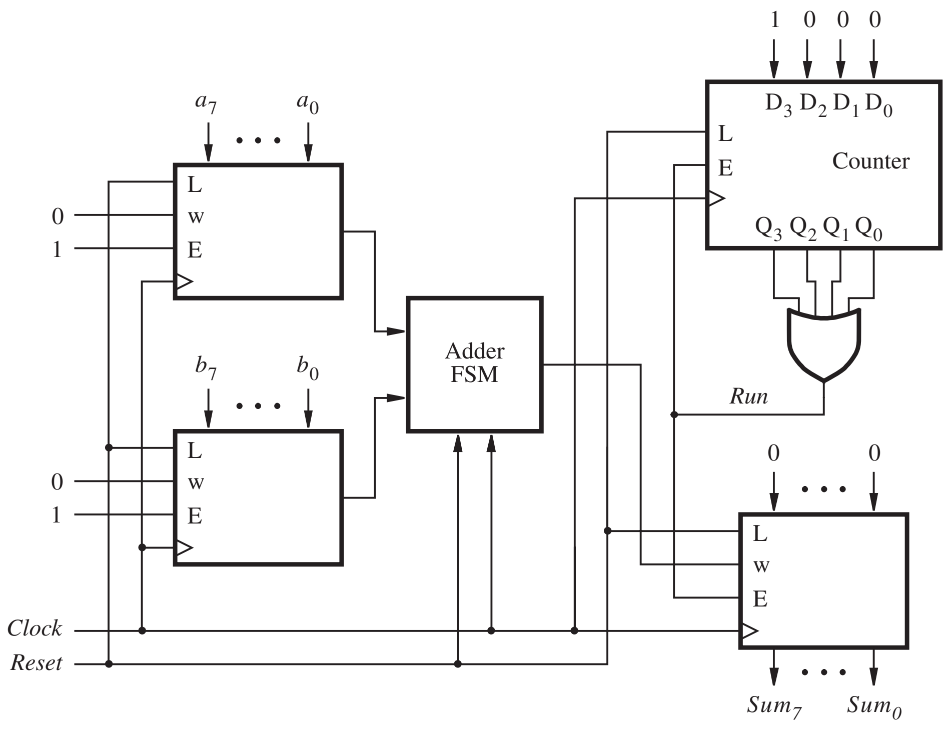
The enable input for the output shift register is named *Run*. It is derived from the outputs of the down-counter specified in the third **always** block. When $Reset = 1$, *Count* is initialized to the value 8. Then as long as $Run = 1$, *Count* is decremented in each clock cycle. *Run* is set to 0 when *Count* is equal to 0, which is detected by using the reduction OR operator.

Synthesis and Simulation of the Verilog Code

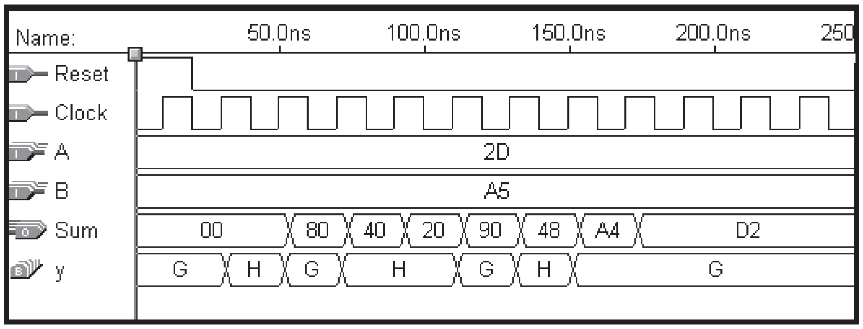
The results of synthesizing a circuit from the code in Figure 8.49 are illustrated in Figure 8.50*a*. The outputs of the counter are ORed to provide the *Run* signal, which enables clocking of both the output shift register and the counter. A sample of a timing simulation for the circuit is shown in Figure 8.50*b*. The circuit is first reset, resulting in the values of A and B being loaded into the input shift registers, and the value 8 loaded into the

```
module serial_adder (A, B, Reset, Clock, Sum);
   input [7:0] A, B;
   input Reset, Clock;
   output [7:0] Sum;
   reg [3:0] Count;
   reg s, y, Y;
   wire [7:0] QA, QB, Sum;
   wire Run;
   parameter G = 0, H = 1;

   shiftrne shift_A (A, Reset, 1, 0, Clock, QA);
   shiftrne shift_B (B, Reset, 1, 0, Clock, QB);
   shiftrne shift_Sum (0, Reset, Run, s, Clock, Sum);

   // Adder FSM
   // Output and next state combinational circuit
   always @(QA or QB or y)
      case (y)
         G: begin
                s = QA[0] ^ QB[0];
                if (QA[0] & QB[0])   Y = H;
                else   Y = G;
            end
         H: begin
                s = QA[0] ~^ QB[0];
                if (~QA[0] & ~QB[0])   Y = G;
                else   Y = H;
            end
         default:   Y = G;
      endcase

   // Sequential block
   always @(posedge Clock)
      if (Reset)   y <= G;
      else   y <= Y;

   // Control the shifting process
   always @(posedge Clock)
      if (Reset)   Count = 8;
      else if (Run)   Count = Count − 1;
   assign Run = |Count;

endmodule
```

Figure 8.49 Verilog code for the serial adder.



(a) Circuit



(b) Simulation Results

Figure 8.50 Synthesized serial adder.

down-counter. After each clock cycle one pair of bits of the input numbers is added by the adder FSM, and the sum bit is shifted into the output shift register. After eight clock cycles the output shift register contains the correct sum, and shifting is halted by the *Run* signal becoming equal to 0.

8.6 STATE MINIMIZATION

Our introductory examples of finite state machines were so simple that it was easy to see that the number of states that we used was the minimum possible to perform the required function. When a designer has to design a more complex FSM, it is likely that the initial attempt will result in a machine that has more states than is actually required. Minimizing the number of states is of interest because fewer flips-flops may be needed to represent the states and the complexity of the combinational circuit needed in the FSM may be reduced.

If the number of states in an FSM can be reduced, then some states in the original design must be equivalent to other states in their contribution to the overall behavior of the FSM. We can express this more formally in the following definition.

Definition 8.1 – *Two states S_i and S_j are said to be equivalent if and only if for every possible input sequence, the same output sequence will be produced regardless of whether S_i or S_j is the initial state.*

It is possible to define a minimization procedure that searches for any states that are equivalent. Such a procedure is very tedious to perform manually, but it can be automated for use in CAD tools. We will not pursue it here, because of its tediousness. However, to provide some appreciation of the impact of state minimization, we will present an alternative approach, which is much more efficient but not quite as broad in scope.

Instead of trying to show that some states in a given FSM are equivalent, it is often easier to show that some states are definitely **not** equivalent. This idea can be exploited to define a simple minimization procedure.

8.6.1 PARTITIONING MINIMIZATION PROCEDURE

Suppose that a state machine has a single input w. Then if the input signal $w = 0$ is applied to this machine in state S_i and the result is that the machine moves to state S_u, we will say that S_u is a 0-*successor* of S_i. Similarly, if $w = 1$ is applied in the state S_i and it causes the machine to move to state S_v, we will say that S_v is a 1-*successor* of S_i. In general, we will refer to the successors of S_i as its k-successors. When the FSM has only one input, k can be either 0 or 1. But if there are multiple inputs to the FSM, then k represents the set of all possible combinations (valuations) of the inputs.

From Definition 8.1 it follows that if the states S_i and S_j are equivalent, then their corresponding k-successors (for all k) are also equivalent. Using this fact, we can formulate a minimization procedure that involves considering the states of the machine as a set and then breaking the set into *partitions* that comprise subsets that are definitely not equivalent.

6 IMPLEMENTATION FABRICS

The hardware of a digital system is implemented using integrated circuits connected together on printed circuit boards. In this chapter, we describe the range of integrated circuits that are used for digital systems. We also discuss some of the important characteristics of integrated circuits and printed circuit boards that give rise to constraints on our designs.

6.1 INTEGRATED CIRCUITS

The history of digital logic circuits predates the invention of integrated circuits. Early digital systems were constructed using discrete switching components, such as relays, vacuum tubes, and transistors. However, the ability to manufacture a complete circuit on the surface of a silicon wafer brought about a tremendous cost reduction. Invention of the integrated circuit is credited to Jack Kilby at Texas Instruments in 1958. The techniques were refined by several developers, and the market for ICs grew rapidly during the 1960s. As digital ICs became commodity parts, adoption of digital logic circuits became widespread.

It is instructive to review the history of development of digital IC technology for two reasons. First, we sometimes need to deal with *legacy systems*, that is, systems designed some time ago but that are still in operation and needing maintenance. Where obsolete parts are unavailable, we need to design replacement circuits to keep the system operating. Hence, we need to understand the operation of legacy components and the constraints under which they operate. Second, we need to realize that circuit technology is continually evolving. It's not sufficient for us to learn how to design using current components, since they will be obsolete at some stage in the future. Instead, we need to understand technology evolution and trends, so that we can "future proof" our designs. Understanding history is important for projecting into the future.

In this section, we will review the history of digital logic components and survey the components that are available to us now. We will also consider the trends affecting near-term evolution of these components and implementation fabrics.

6.1.1 INTEGRATED CIRCUIT MANUFACTURE

Implementation fabrics for digital systems are based on integrated circuits (ICs), which are manufactured on the surface of a wafer of pure crystalline silicon using a sequence of photographic and chemical process steps. A number of identical rectangular ICs are manufactured together, and then broken apart for individual packaging. Hence, we often use the name *silicon chip* to refer to a piece of a silicon on which an IC is manufactured.

In preparation for chip manufacture, a cylindrical ingot of silicon is formed (Figure 6.1, left) and then sawn into wafers less than a millimeter thick and finely polished (Figure 6.1, right). Early wafers were 50mm in diameter, but, since then, improvements in manufacturing processes have yielded successively larger wafer sizes. Now, chips can be manufactured on 300mm diameter wafers. This allows more chips to be manufactured at once, and reduces the waste at the edges.

The process of manufacturing a circuit on the wafer surface involves a number of steps that change the properties of certain areas of the surface silicon, or add a surface layer of some material in certain areas. There are several kinds of processing steps that can be applied to selected areas of the wafer, including

- Ion implantation: exposing the surface to a plasma of impurity ions that diffuse into the silicon, thus altering its electrical properties in controlled ways.
- Etching: chemically eroding an underlying film of material that has been deposited onto the surface. Films include insulating materials, such as silicon dioxide; semiconducting materials, such as polycrystalline silicon (also known as polysilicon); and conducting materials, such as aluminum and copper.

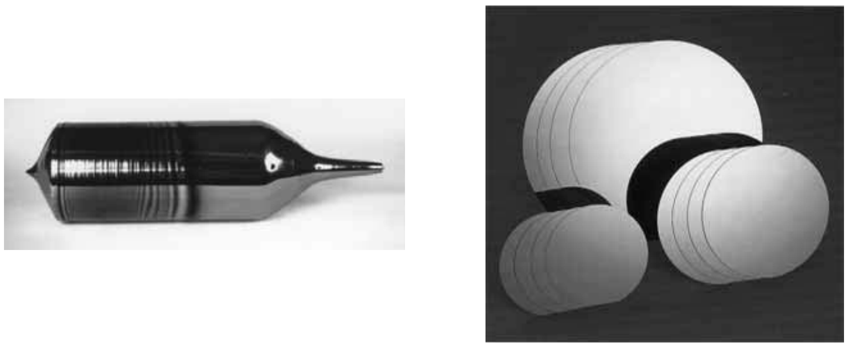


FIGURE 6.1 An ingot of crystalline silicon (left), and sawn wafers (right).

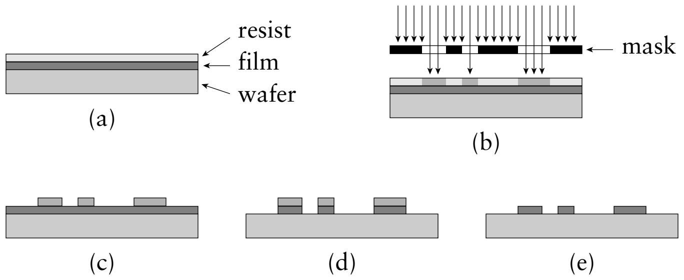


FIGURE 6.2 Steps in photolithographic etching: (a) the wafer and film coated with resist; (b) exposure through a photomask; (c) the resist developed; (d) etching of the underlying film; (e) the remaining resist stripped.

The key to selecting which areas are affected is *photolithography*, which means using a photographic process to draw on the surface (see Figure 6.2, showing selective etching of a film). The surface is coated with a thin layer of *photoresist*, a chemical whose resistance to chemical reaction is changed by exposure to light. The surface is then exposed to light through a mask that has opaque and transparent areas in the pattern of features to be drawn. The resist is then developed, dissolving either the exposed areas or the unexposed areas, depending on the kind of resist. The areas that are now uncoated can be processed, and then the remaining resist is stripped off.

Manufacturing circuits requires several different masks to form the circuit layers, as illustrated in Figure 6.3. MOS transistors are formed in the bottom layers with channel areas containing diffused impurity ions and gates formed from polysilicon lines. Wiring is formed in higher layers using etched metal conductors. A complete wafer contains between a few hundred to a few thousand circuits, depending on the individual circuit size, as shown in Figure 6.4.

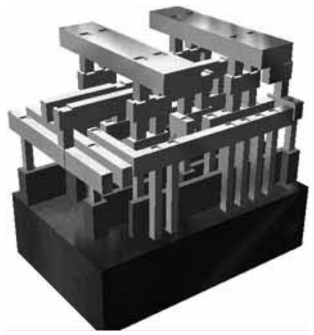


FIGURE 6.3 Graphical representation of the layers of an IC.

Once the circuits on a wafer have been manufactured, they must be tested to determine which ones work and which fail due to defects. Small defects can be introduced into an individual IC by stray particles obscuring light during photolithography, by impurities occurring in chemical process steps, or by particles impinging on wafers during handling in the manufacturing process. IC foundries are meticulous in their cleanliness, using chemicals of high purity and operating in special clean rooms. Nonetheless, stray particles and impurities cannot be completely avoided. A defect can prevent an IC on a wafer from working. The *yield* is the proportion of manufactured ICs that work. Since a whole wafer-lot of ICs is manufactured together, the cost of the discarded defective ICs must be amortized over those that work. Larger ICs have an increased chance of being defective, so it is important for designers to constrain IC area to reduce cost.

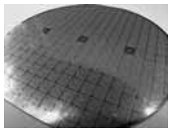


FIGURE 6.4 A complete wafer containing multiple ICs.

After testing the ICs on a wafer, the wafer is broken into individual chips, which are then packaged. We will describe the different kinds of packaging in Section 6.3. Since the packaging process may introduce further defects, the packaged chips are further tested. They can also be graded according to maximum operating speed, so that higher performance chips can be sold at a premium price.

A number of parameters of the manufacturing process determine the minimum size of transistors and wires in an IC, and thus the size of a complete IC. One of the main parameters is the photolithography resolution, that is, the smallest feature that can accurately be drawn and processed. Much of the improvement in IC technology is attributable to advances in photolithography, including use of higher-resolution masks and shorter wavelengths of light. Reducing the feature size has a number of benefits. It results in smaller chips for a given function, thus reducing cost. It allows more circuitry to be placed on a chip of a given size, thus increasing functionality. It also reduces circuit delays, allowing higher operating speed. Feature size, along with several other parameters, have been improving exponentially as manufacturing technology has matured. Currently, use of extreme ultraviolet light allows us to manufacture circuits with 90nm feature sizes. Further improvements will allow for 65nm and smaller sizes. These trends are expected to continue for some time yet. The publication *Exponential Trends in the Integrated Circuit Industry* (see Section 6.6) summarizes the trends.

6.1.2 SSI AND MSI LOGIC FAMILIES

While many early ICs were developed for specific applications, in 1961 Texas Instruments introduced a family of logic components that designers could use as building blocks for larger circuits. Three years later, they introduced the 5400 and 7400 families of TTL (transistor-transistor logic) ICs that became the basis of logic design for many years. The 5400 family components were manufactured for high-reliability military applications, requiring operation over large temperature ranges, whereas the 7400 family components were for commercial and industrial applications. Other IC manufacturers also provided compatible components, thus making the 7400 family a *de facto* standard.

The components in the 7400 family are numbered according to the logic functions they provide. For example, a 7400 component provides four NAND gates, a 7427 provides three NOR gates, and a 7474 provides two D flip-flops. Since these components integrate a relatively small number of circuit elements, they are referred to as *small-scale integrated* (SSI) components. As manufacturing techniques improved, larger circuits could be integrated, leading to what we now call *medium-scale integrated* (MSI) components. Examples include the 7490 4-bit counter, and the 7494 4-bit shift register. The boundary between SSI and MSI is somewhat arbitrary. For example, it's not clear whether a 7442 BCD decoder is SSI or MSI.

In addition to extending the range of functions available within the 7400 family, manufacturers developed alternative versions of the components with different internal circuitry and correspondingly different electrical characteristics. One variation reduced the power consumed by components, at the cost of reduced switching speed. Components in this family are identified by inclusion of the letter "L" in the part number, for example, 74L00 and 74L74. Another variation used Schottky diodes within the internal circuits to reduce switching delays, albeit at the expense of increased power consumption. These components include the letter "S" in the part number, for example, 74S00 and 74S74. One of the most popular variations, the 74LS00 family, combined the lower-power circuits with Schottky diodes to yield a good compromise between power and speed. Later variations included the 74F00 "fast" family components and the 74ALS00 "advanced low-power Schottky" family.

One of the problems with TTL circuits is that they use bipolar transistors, which have relatively high power consumption even when not switching. In previous chapters, we have described an alternative circuit structure, CMOS, which uses field-effect transistors. It was originally developed around the same time as TTL. One of the earliest CMOS logic families was the 4000 family, which provided SSI and MSI functions, but with much lower power consumption. They could also operate over a much larger power supply range (3V−15V), compared to TTL's nominal 5V, but were much slower and had logic levels that were incompatible with TTL components. Hence, they did not gain widespread use.

Later, in the 1980s, some manufacturers introduced a new family of CMOS logic components, the 74HC00 family, that were compatible with TTL components. They provided the same functionality, but with lower power consumption and comparable speed. Subsequent variations, such as the 74AHC00 family, offered improved speed and electrical characteristics.

One important characteristic of CMOS circuits is that the power consumption and speed are dependent on the power-supply voltage. By reducing the voltage, as well as by making the internal circuit features smaller, speed is increased and power consumption is reduced. These considerations led the electronics industry to agree on a lower standard power supply voltage of 3.3V. Manufacturers subsequently developed component families to operate at the lower voltage with reduced logic thresholds (74LVC00 family) or with TTL-compatible thresholds (74LVT00 family). They also developed advanced variations, such as the 74ALVC00 and 74ALVT00 family.

As a result of these evolutionary steps, we now have numerous logic families from different manufacturers, with an alphabet soup of letters between the "74" prefix and the number that denotes the logic function. Each family has different trade-offs in power consumption, speed and logic-level thresholds. The data books published by the manufacturers document the characteristics of each family. As designers, we need to understand the power, speed and compatibility constraints of an application and choose components from a family that meets the constraints.

Another aspect of evolution of these families is a change in the logic functions provided. Early components (generally those with smaller numbers) provided gates and simple combinational and sequential functions from which more complex systems could be built. However, during the 1970s, IC technology developed to the level of *large-scale integration* (LSI), at which it became feasible to provide a small computer, a *microprocessor*, on a single IC. Embedded systems using microprocessors became more cost-effective in many applications than systems constructed from SSI and MSI components. The 7400 family components were then commonly used as *glue logic*, that is, simple logic circuits for interconnecting LSI components. As a consequence, new functions added to the later logic families were more oriented toward glue and interconnection functions, such as multibit tristate drivers and registers. These more recent components in the CMOS logic families are the only ones we would consider for new designs, with older components and TTL families being used only for maintenance of legacy systems. Programmable logic devices (described in Section 6.2) and ASICs have almost completely supplanted other families.

EXAMPLE 6.1 Use the following components to design a 4-digit decimal counter with a 7-segment LED display: two 74LS390 dual decade counters, four 74LS47 BCD to 7-segment decoders, four 7-segment displays, plus any additional gates required.

SOLUTION The 74LS390 component contains two counters, each as shown in Figure 6.5. Internally, the counter consists of a single-bit counter clocked on the falling edge of CP0, and a 3-bit divide-by-five counter clocked on the falling edge of CP1. A decade (divide-by-ten) counter can be formed by using the single-bit counter for the least significant bit and connecting the Q0 output externally to the CP1 input. When Q0 changes from 1 to 0, it causes the more significant bits to count up. The MR input to the counter is a master reset input. When 1, it forces the counter outputs to 0000.

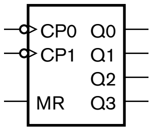


FIGURE 6.5 A symbol for each of the decade counters in a 74LS390 component.

We can cascade the 74LS390 decade counters together, using the outputs of each decade to generate a clock for the next decade. The outputs of a given decade changing from 1001 (the binary code for 9) to 0000 should cause the next decade to count up. The only time this occurs is when Q3 and Q0 of the given decade both change to 0, so we can use an AND gate to generate the clock for the next decade, as shown in Figure 6.6.

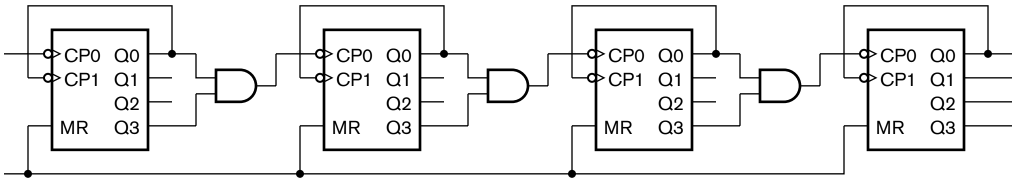


FIGURE 6.6 Four 74LS390 decade counters cascaded to make a 4-digit counter.

The 74LS47 component is shown in Figure 6.7. The inputs A through D are the BCD value, with A being the least significant bit and D the most significant bit. The segment outputs a through g are active-low, allowing them to drive the cathodes of a 7-segment display. When the lamp test input, LT, is low, it turns all segments on. We can tie it to a high logic level in this application, since we don't need to use it. The ripple-blank input (RBI) and ripple-blank output (RBO) are use to turn off any leading zero digits in the displayed value. When the RBI input to a decoder is low and the BCD value is 0000, all of the segments are turned off and RBO is driven low. We tie the RBI input of the most significant digit low, and chain the RBO of all digits to the RBI of the next digit (except the least significant digit, which we always want to display something).

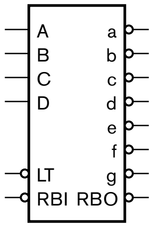


FIGURE 6.7 A symbol for the 74LS47 BCD to 7-segment decoder.

Our complete circuit for the counter with display is shown in Figure 6.8. The two 74LS390 components are connected as shown in Figure 6.6 using three AND gates. These gates can be implemented using three of the four AND gates in a 74LS08 component. Each counter output drives a 74LS47 decoder, which in turn drives a 7-segment LED display. The resistors are required to limit the current flowing in each segment LED. The value of the resistor depends on the required display brightness. Information on current versus brightness can be found from manufacturer's data sheets. If we assume 2mA is sufficient, that the decoder output has a low-level voltage of 0.4V at 2mA, and that the voltage drop across the segment LED is 1.6V, we need the resistor to drop $5.0 - 1.6 - 0.4 = 3.0$V at 2mA. Thus, a 1.5kΩ resistor will suffice.

6.1.3 APPLICATION-SPECIFIC INTEGRATED CIRCUITS (ASICS)

The developmewnt of IC technology beyond the LSI level led to *very large scale integrated* (VLSI) circuits. At that stage, it became clear that the industry would soon run out of superlatives to prefix to "LSI." Thus, the term VLSI came to refer more to the way in which ICs were designed than the number of transistors they carried. The term now usually means the detailed circuit design of integrated circuits, as opposed to system-level design. The widespread availability of CAD tools for VLSI design and the growth of the IC manufacturing service industry has now made it practical to develop ICs for a wide range of applications. We use the term *application-specific integrated circuit*, or *ASIC*, to refer to an IC

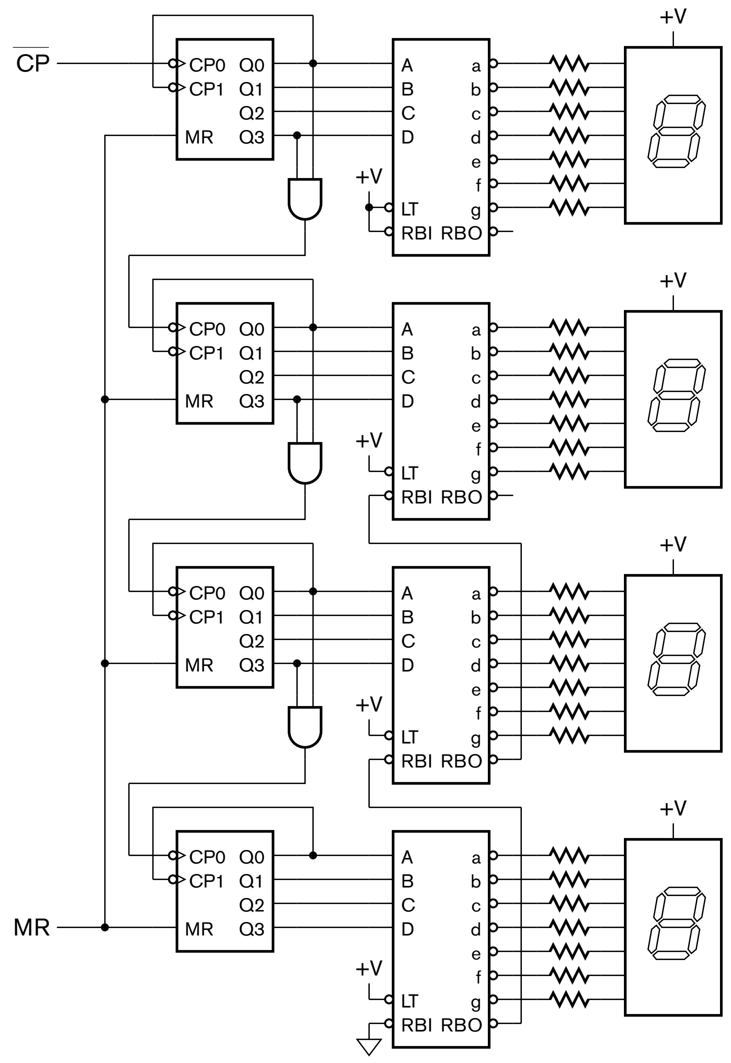


FIGURE 6.8 The complete circuit for the 4-digit counter with display.

manufactured for a particular application. This is not to say that an ASIC is necessarily manufactured just for one customer or project. Rather, it is designed to meet a particular set of requirements, and so contains circuits customized for those requirements. It may be designed for a particular end product provided by one manufacturer, for example, a portable music player, a toy, an automobile, a piece of military equipment, or an industrial machine. Alternatively, it may be designed for use in a range of products provided by manufacturers in a particular market segment. These kinds of ASIC are sometimes called *application-specific standard products*, or *ASSPs*, since they are treated as a standard part within the market segment, but are not of use outside that segment. Examples include ICs for cell phones, which are used by a number of competing cell-phone manufacturers, but which are not of use in, say, automobile control circuits.

One of the main reasons we would develop an ASIC for a product is that, being customized for that application, it has lower cost per IC than a programmable component such as an FPGA (see Section 6.2). However, in order to achieve that level of customization, we need to invest much more design and verification effort. We must amortize the *non-recurring engineering* (NRE) cost over all of the product units sold. Hence, it only makes sense to use an ASIC if our product sales volume is sufficiently large. The amortized NRE cost per unit should be less than the cost difference between an ASIC and a programmable part. This, of course, assumes that it is feasible to use a programmable component. If the application requires a level of performance that cannot be achieved with an FPGA, then an ASIC or an ASSP is our only real option, and the higher NRE is a necessary part of the product cost.

There are two main design and manufacturing techniques for ASICs, differing in the degree of customization for the application. We will describe them briefly in this section, deferring in-depth discussion to advanced references on VLSI design. First, *fully custom* integrated circuits involve detailed design of all of the transistors and connections in an ASIC. This allows the most effective use of the hardware resources on an IC and yields higher performance, but has high NRE cost and requires advanced VLSI design expertise within the design team. As a consequence, fully custom ASICs are usually only designed for high-volume products, such as CPUs and ICs for consumer appliances. Second, *standard cell* ASICs involve selection of basic cells, such as gates and flip-flops, from a library to form the circuit. The cells have been previously designed by an IC manufacturer or an ASIC vendor, and are used by the synthesis tool during the design process to implement the design. The value of this approach is that the NRE cost for each ASIC design is significantly reduced, since the cost of designing the cell library is amortized over a number of ASIC designs. The compromise is that the ASIC may not be as dense or have the performance of a fully custom ASIC.

KNOWLEDGE TEST QUIZ

1. What is *photolithography* in IC manufacture?
2. How do IC area and defect density on a wafer affect IC cost?
3. What do the "L" and "S" in the part name 74LS47 stand for?
4. What is meant by the term *glue logic*?
5. What do the terms *ASIC* and *ASSP* stand for?
6. Would it make sense to design an ASIC for a customized building security system to be installed in a new office building? Why (or why not)?

7. Similarly, would it make sense to design an ASIC for an engine control system in a car? Why (or why not)?

6.2 PROGRAMMABLE LOGIC DEVICES

The components in SSI and MSI families and ASICs all have fixed functions, determined by the logic circuit for each component. *Programmable logic devices* (PLDs), on the other hand, can be programmed after manufacture to have different functions. In this section, we will look at the evolution of PLDs, leading to FPGAs that are in widespread use today.

6.2.1 PROGRAMMABLE ARRAY LOGIC

One of the first successful families of PLDs was introduced in the late 1970s by Monolithic Memories, Inc., and called *programmable array logic* (PAL) components. These components were an evolution of earlier PLDs, but were simpler to use in many applications. A simple representative component in the family is the PAL16L8, whose circuit is shown in Figure 6.9. The component has 10 pins that are inputs, 2 pins that are outputs, and 6 pins that are both inputs and outputs. This gives a total of 16 inputs and 8 outputs (hence the name "16L8"). The symbol at each input in Figure 6.9 represents a gate that is a combination of a buffer and an inverter. Thus, the vertical signals carry all of the input signals and their negations. The area in the dashed box is the *programmable AND array* of the PAL. Each horizontal signal in the array represents a p-term of the inputs, suggested by the AND-gate symbol at the end of the line. (Recall that a p-term, or product term, is the logical AND of a number of signals; see Section 2.1.1.) In the unprogrammed state, there is a wire called a *fusible link*, or *fuse*, at each intersection of a vertical and horizontal signal wire, connecting those signal wires. The PAL component can be programmed by blowing some of the fuses to break their connections, and leaving other fuses intact. This is done by a special programming instrument before the component is inserted into the final system.

On the diagram of Figure 6.9, we draw an X at the intersection of a vertical and a horizontal signal to represents an intact fuse. An intersection without an X means that the intersecting signals are not connected. So, for example, the horizontal signal numbered 0 has connections to the vertical signals numbered 24 and 31, which are the signals I8 and $\overline{\text{I10}}$. Some of the p-terms are connected to the enable control signals for the inverting tristate output drivers. Others are connected to the 7-input OR gates. So, for each output, we can form the AND-OR-INVERT function of inputs, with up to 7 p-terms involved. In the circuit shown in Figure 6.9, output O1 implements the function $\overline{\text{I1} \cdot \text{I2} + \text{I3} \cdot \overline{\text{I10}}}$, with the output enabled by the condition $\text{I8} \cdot \overline{\text{I10}}$.

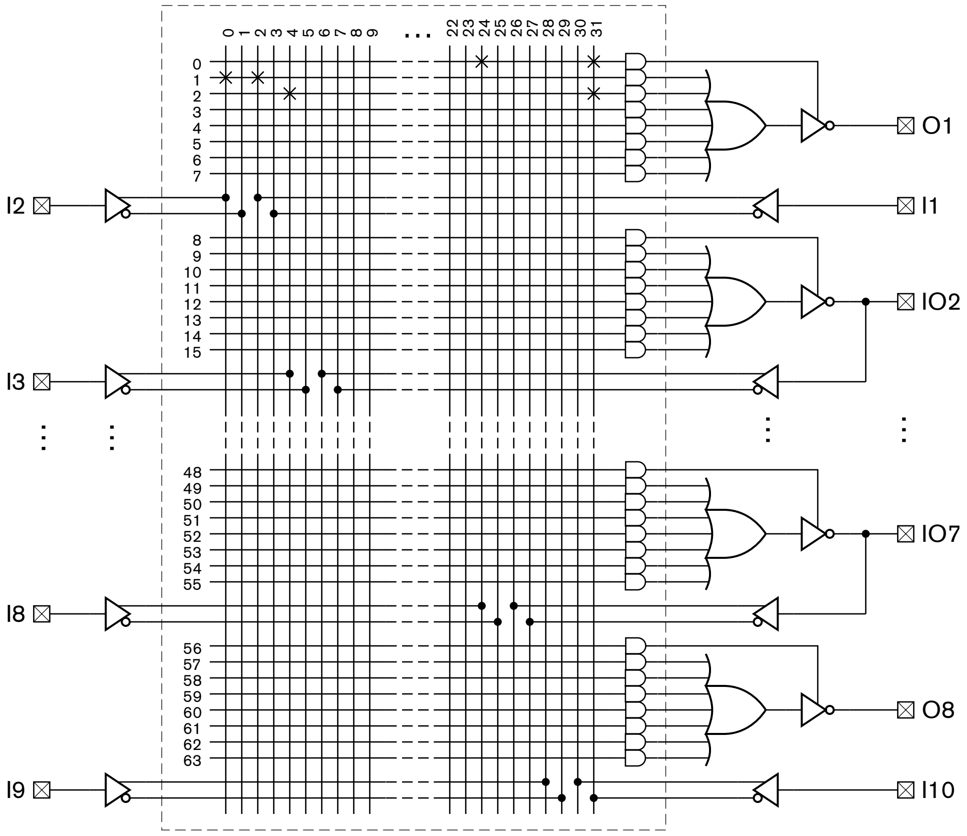


FIGURE 6.9 The internal circuit of a PAL16L8 component.

A PAL component such as the PAL16L8 can be programmed to implement a variety of combinational functions. Other PAL components also include registers, allowing us to implement simple sequential circuits. As an example, the output circuit of a PAL16R8 component is shown in Figure 6.10. The feedback from the register output to the programmable AND array is useful for implementation of FSMs. Even if a circuit is very simple, requiring only a handful of gates and flip-flops, there is often

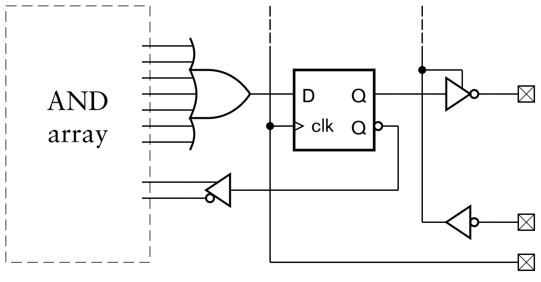


FIGURE 6.10 Registered output circuit of a PAL16R8.

a benefit in cost and reliability gained from combining several separate packages into one PAL package. We typically describe the functions to be implemented in a PAL component in terms of Boolean equations expressed in an HDL. We then use a synthesis tool to transform the equations into a gate-level circuit, which we can verify using the same testbench that we used to verify the Boolean-equation model. Finally, we use physical-design CAD tools to transform the gate-level model into a *fuse map*, that is, a file used by the programming instrument to determine which fuses to blow. If our functions are too complex to be expressed using the resources available in a given PAL component, we would either need to use a larger component, or divide the functions across several components.

As manufacturers developed PLD technology, they found it more convenient to provide fewer generic components in a family, rather than a larger number of variants of a given organizational theme. Most of the variation in earlier families arose in the resources provided for the outputs, for example, whether outputs were inverting or not, whether registers were provided, and whether outputs could be fed back as inputs. In contrast, generic array logic (GAL) components provide *output logic macrocells* (OLMCs) that replace the combinations of OR gates, registers and tristate drivers in PAL output circuits. Each OLMC includes circuit elements together with programmable multiplexers, allowing the output functionality to be determined as part of programming the component. As an example, Figure 6.11 shows the internal circuit organization of the GAL22V10 component, now manufactured by Lattice Semiconductor Corporation, and Figure 6.12 shows the OLMC circuit for each section. The OLMC has p-term inputs from an AND array with the same organization as that of a PAL component. The number of p-terms ranges from 8 for some sections to 16 for others. The output of the OR gate connects to a D flip-flop that has clock, asynchronous reset

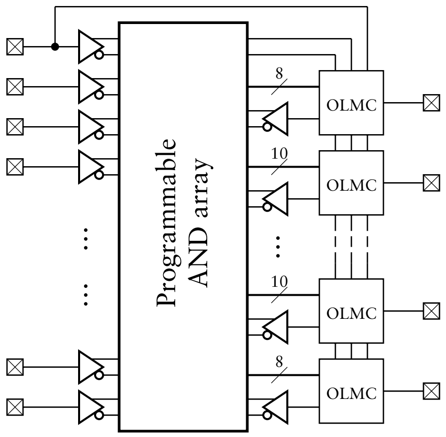


FIGURE 6.11 Internal circuit organization of a GAL220V10 component.

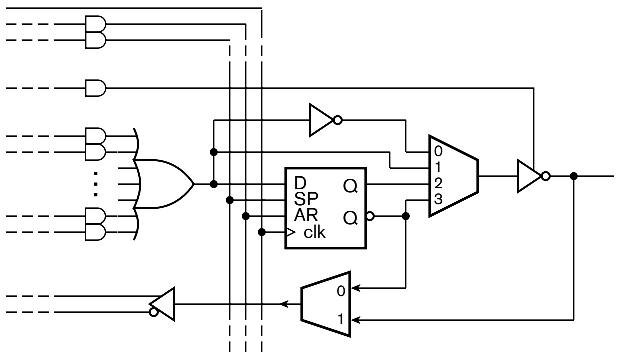


FIGURE 6.12 Output logic macrocell of a GAL22V10 component.

and synchronous preset signals in common with other OLMCs in the component. The select inputs of the multiplexers are set by programming the component. The four-input multiplexer allows selection of registered or combinational output, either inverting or noninverting. The two-input multiplexer allows either registered or combinational feedback, or, if the output driver is in the high-impedance state, direct input from the component pin. By appropriate programming, a GAL component can emulate any of the PAL components, including the PAL16L8 and PAL16R8 components we have shown here.

In modern designs, we would typically use PLDs such as GAL components for simple combinational glue logic, and for relatively simple sequential circuits. As with PAL components, we would describe the required functionality in terms of Boolean equations and use a CAD tool to determine the programming for the component. The circuitry of the original GAL families was based on similar technology to EPROMs, allowing them to be programmed, erased using ultraviolet light, and subsequently reprogrammed. Current components can be erased electrically and programmed *in situ* in the final circuit.

EXAMPLE 6.2 Design a priority encoder that has 16 inputs, I[0:15]; a four-bit encoded output, Z[3:0]; and a valid output that is 1 when any input is 1. Input I[0] has the highest priority, and I[15] the lowest priority. The design is to be implemented in a GAL22V10 component.

SOLUTION The Boolean equations for the encoder, expressed in Verilog, are

```
assign win[0] = I[0];
assign win[1] = I[1] & ~I[0];
assign win[2] = I[2] & ~I[1] & ~I[0];
...
```

(*continued*)

```
assign win[15] = I[15] & ~I[14] & ~I[13] & ... & ~I[0];

assign Z[3] = win[15] | win[14] | win[13] | win[12]
                | win[11] | win[10] | win[9] | win[8];
assign Z[2] = win[15] | win[14] | win[13] | win[12]
                | win[7] | win[6] | win[5] | win[4];
assign Z[1] = win[15] | win[14] | win[11] | win[10]
                | win[7] | win[6] | win[3] | win[2];
assign Z[0] = win[15] | win[13] | win[11] | win[9]
                | win[7] | win[5] | win[3] | win[1];

assign valid = I[15] | I[14] | I[13] | ... | I[0];
```

Each of the win elements can be implemented as a p-term in a row of the GAL AND array. Each Z output is thus the OR of 8 p-terms. Since each OLMC in a GAL22V10 component has at least 8 p-term inputs, these equations will fit in any of the sections.

The valid output is the OR of 16 inputs, so it could fit in either of the two sections that have 16 p-term inputs to the OLMC. However, we can rewrite the equation for valid using the DeMorgan law (see Section 2.1.2) as

```
assign valid = ~(~I[15] & ~I[14] & ~I[13] & ... & ~I[0]);
```

By programming the OLMC for the valid output to negate the OR result, we can use just one p-term. This allows us to place the valid output in any section of the GAL component rather than only in those sections that have 16 OR-gate inputs. The flexibility afforded by this transformation reduces the constraints on choice of output pins for the component, and may thus simplify connection of the component in a larger circuit.

6.2.2 COMPLEX PLDS

A further evolution of PLDs, tracking advances in integrated circuit technology, led to the development of so-called *complex programmable logic devices* (CPLDs). We can think of a CPLD as incorporating multiple PAL structures, all interconnected by a programmable network of wires, as shown in Figure 6.13. (This gives a general idea of CPLD organization. The actual organization varies between components provided by different manufacturers.) Each of the PAL structures consists of an AND array and a number of embedded macrocells (M/Cs in the figure). The macrocells contain OR gates, mutiplexers and flip-flops, allowing choice among combinational or registered connections to other elements within the

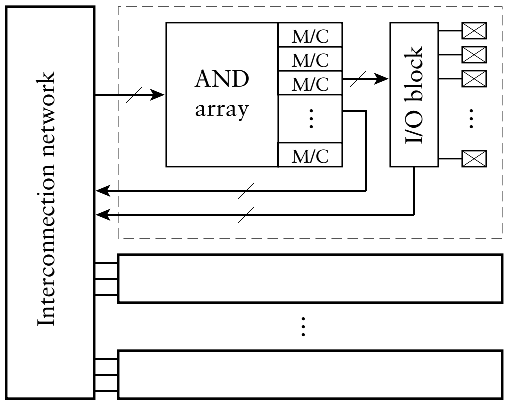


FIGURE 6.13 The internal organization of a CPLD.

component, with or without logical negation, choice of initialization for flip-flops, and so on. They are essentially expanded forms of the simple macrocell shown in Figure 6.12, but without the direct connections to external pins. Instead, the external pins are connected to an I/O block, which allows selection among macrocell outputs to drive each pin. The network interconnecting the PAL structures allows each PAL to use feedback connections from other PALs as well as inputs from external pins.

As well as providing more circuit resources than simple PLDs, modern CPLDs are typically programmed differently. Rather than using EPROM-like technology, they use SRAM cells to store configuration bits that control connections in the AND-OR arrays and the select inputs of multiplexers. Configuration data is stored in nonvolatile flash RAM within the CPLD chip, and is transferred into the SRAM when power is applied. Separate pins are provided on the chip for writing to the flash RAM, even while the chip is connected in the final system. Thus, designs using CPLDs can be upgraded by reprogramming the configuration information.

Manufacturers provide a range of CPLDs, varying in the number of internal PAL structures and input/output pins. A large CPLD may contain the equivalent of tens of thousands of gates and hundreds of flip-flops, allowing for implementation of quite complex circuits. Whereas it might be feasible to manually determine the programming for a simple PLD, it would be quite intractable to do so for a CPLD. Hence, we would use CAD tools to synthesize a design from an HDL model and to map the design to the resources provided by a CPLD.

6.2.3 FIELD-PROGRAMMABLE GATE ARRAYS

As we saw in the last section, manufacturers were able to provide larger programmable implementation fabrics by replicating the basic PAL structure on a chip. However, there is a limit to how far this structure can be expanded. For large designs, mapping the circuit onto CPLD resources becomes very difficult and results in inefficient use of the resources provided by the chip. For this reason, manufacturers turned to an alternate programmable circuit structure, based on smaller programmable cells to implement logic and storage functions, combined with an interconnection network whose connections could be programmed. They named such structures *field-programmable gate arrays* (FPGAs), since they could be thought of as arrays of gates whose interconnection could be programmed "in the field," away from the factory where the chips were made. Given the relative complexity of the components, it was not expected that designers would implement circuits for FPGAs manually. Instead, manufacturers provided CAD tools to allow designs expressed in an HDL to be synthesized, mapped, placed and routed automatically, though with designer intervention if necessary. Since their introduction, FPGAs have grown in capacity and performance, and are now one of the main implementation fabrics for designs, particularly where product volumes do not warrant custom integrated circuits.

Most FPGAs available today are organized along the lines shown in Figure 6.14. They include an array of logic blocks that can be programmed to implement simple combinational or sequential logic functions; input/output (I/O) blocks that can be programmed to be registered or nonregistered, as well as implementing various specifications for voltage levels, loading and timing; embedded RAM blocks; and a programmable interconnection network. The more recent FPGAs also include special circuits for clock generation and distribution. The specific organization, as well as the names used for the blocks, varies between manufacturers and FPGA families.

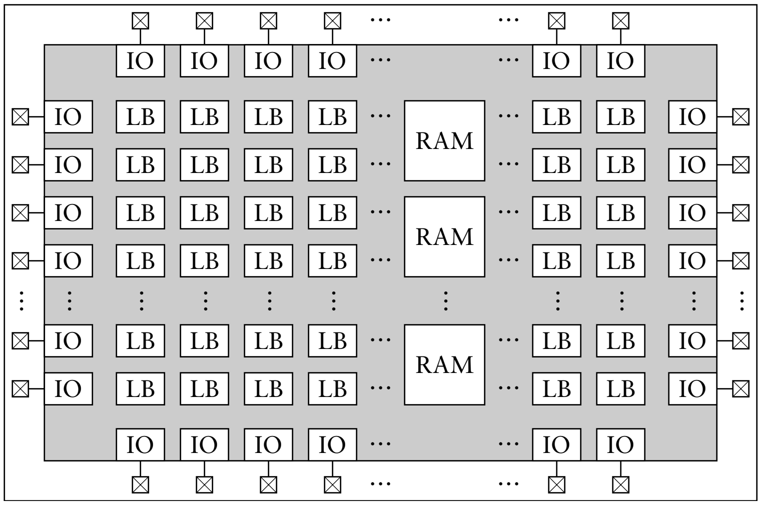


FIGURE 6.14 The internal organization of an FPGA consisting of logic blocks (LB), input/output blocks (IO), embedded RAM blocks (RAM) and programmable interconnections (shown in gray).

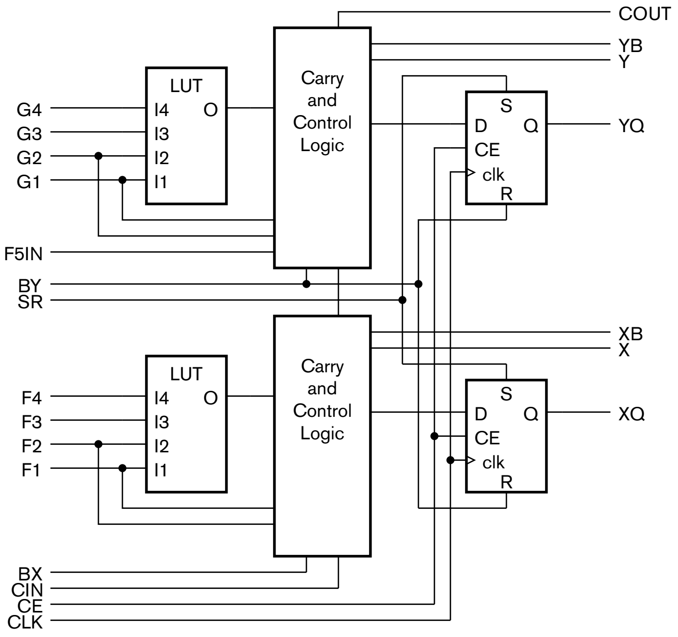


FIGURE 6.15 The circuit of a slice of a Xilinx Spartan-II FPGA logic block.

In many FPGA components, the basic elements within logic blocks are small 1-bit-wide asynchronous RAMs called *lookup tables* (LUTs). The LUT address inputs are connected to the inputs of the logic block. The content of an LUT determines the values of a Boolean function of the inputs, in much the same way as we discussed in Section 5.2.5. By programming the LUT content differently, we can implement any Boolean function of the inputs. The logic blocks also contain one or more flip-flops and various multiplexers and other logic for selecting data sources and for connecting data to adjacent logic blocks.

As an illustration, Figure 6.15 shows the circuit for a slice within a logic block of a Xilinx Spartan-II FPGA. The logic block contains two such slices, together with a small amount of additional logic. Each slice consists of two 4-input LUTs, each of which can be programmed to implement any function of the four inputs. The carry and control logic consists of circuitry to combine the LUT outputs, an XOR gate and an AND gate for implementing adders and multipliers, as well as multiplexers that can be used to implement a fast carry chain (see Section 3.1.2). Additional components, not shown in the figure, allow programming for various signals to be negated. A number of the connections within the control and carry logic are governed by the programming of the FPGA. The logic block contains SRAM cells for these programming bits.

In contrast to LUT-based logic blocks, which can implement relatively complex functions, some FPGAs have more fine-grained logic blocks. For example, the logic block of Actel ProASIC3 FPGAs contains just enough gates, multiplexers and switches to implement combinational functions of three inputs, or a flip-flop with set or reset. Since each logic block is smaller and simpler, CAD software that maps a design into the FPGA resources may find it easier to perform its task without leaving parts of logic blocks unused. However, a given design will require more logic blocks, and consequently denser interconnection between them. This may make the place and route software's task more difficult.

The I/O block of an FPGA is typically organized as shown in Figure 6.16, but with some variation between components from different vendors. The select inputs of the multiplexers are programmed to control whether the output is registered or combinational. The top flip-flop and multiplexer control the high-impedance state of the tristate driver that drives the pin as an output, and the middle flip-flop and multiplexer drive the output value. The output driver is programmable, allowing selection of logic levels (regular 5V TTL, low voltage TTL, or others) and control of the slew rate, that is, rate of voltage change at the output. (We will discuss why slew rate control is important in Section 6.4.) The input buffer is likewise programmable, allowing selection of threshold voltage and other characteristics. The pull-up and pull-down resistors are programmable, allowing them to be connected and their resistance to be selected. The reason for making all of these characteristics programmable is to allow the FPGA to be used in a wide range of systems that use different signaling standards between chips, and to

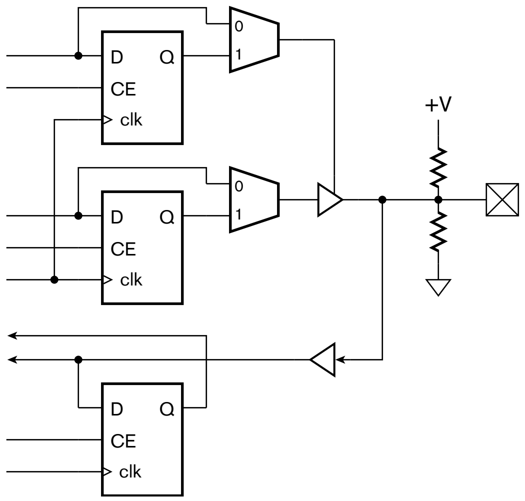


FIGURE 6.16 Typical organization of an FPGA I/O block.

accommodate the different drivers and loads to which different pins of an FPGA may be connected.

The RAM blocks in an FPGA provide for storage of information to be processed by the FPGA circuitry. As we shall see when we consider embedded computer systems in more detail, many applications require data to be input in blocks or streams, and for "chunks" of data to be processed at once. The RAM blocks can be used to store such chunks between processing steps. Also, when an embedded processor is implemented within an FPGA, RAM blocks provide a place to store the processor's instructions and the data upon which it operates. Typical modern FPGAs provide synchronous static RAM (SSRAM) blocks that can be programmed to be flow-through or pipelined, and that have two access ports that can be programmed to be read-only or read-write. The RAM blocks are each relatively small in capacity, but can be interconnected to form larger memories. Each block can be programmed to trade off the number of locations against the number of bits per location. For example, in a Xilinx Spartan-3 FPGA, each RAM block has a total of 18K bits of storage, which can be programmed to provide a 16K $\times$ 1-bit, 8K $\times$ 2-bit, 4K $\times$ 4-bit, 2K $\times$ 9-bit, 1K $\times$ 18-bit or 512 $\times$ 36-bit organization. (The 9-, 18- and 36-bit organizations can be used to provide a parity bit with each byte, or the extra bit per byte can be ignored.) The number of RAM blocks varies from 4 to 104 among different members of the Spartan-3 family. FPGAs from other vendors provide similar storage capacities and organizations.

Each of the various logic, I/O and RAM blocks on an FPGA connect to interconnection wires through programmable switches. The connections can be programmed so that a given input or output of a block can be connected (or not) to a wire that passes the block. The interconnections between logic blocks consist of a mix of short and long wires, and possibly wires of intermediate length, depending on the FPGA. Short wires connect nearby logic blocks, whereas long wires connect distant logic blocks or connect to a number of logic blocks distributed across the FPGA. It is the job of the place and route software to ensure that parts of the design are implemented in logic blocks in such a way that the interconnection resources can be programmed to "wire up" the design.

There are two forms of FPGA that differ in the way they are configured. The first form uses RAM cells to store the configuration information. The main advantage of this approach is that an FPGA can be programmed after the chip has been assembled into a system, without the need for any separate handling during manufacture. Furthermore, the system can be upgraded after delivery by storing new configuration information, rather than having to replace chips or other hardware. If the configuration is stored using volatile SRAM cells, it needs to be loaded each time power is applied to the system. Hence, the configuration needs to be stored in a separate nonvolatile memory, and additional circuits need to be included in the system to manage loading the configuration. The two main FPGA vendors, Xilinx and Altera, both use SRAM cells for their devices and provide specialized flash RAM devices for storing and configuring the FPGAs. Other vendors, such as Actel, provide FPGAs that use nonvolatile flash RAM cells for the configuration information. Such devices do not need the external components for storing or loading the configuration, thus reducing overall system complexity. However, the trade-off is a reduced maximum operating speed.

The second main form of FPGA uses *antifuses* to configure the device. An antifuse, as its name suggests, is a conductive connection that is formed during programming, as opposed to being blown. Since programming is done by forming a connection, no storage is needed, either inside the FPGA or externally. Moreover, the device is less susceptible to soft errors due to radiation (see Section 5.3). However, the device must be programmed separately before being installed in the final system. This requires additional manufacturing steps and handling, adding cost to the manufacturing process.

Platform FPGAs

It should be clear now that integrated circuit technology has developed continuously. This trend applies equally to FPGAs. As they have become denser and faster, it has become feasible to use them for applications requiring significant computational performance, such as audio and video processing and information encryption and decryption. In order to improve their usability for these kinds of applications, manufacturers have added specialized circuitry to the larger recent FPGAs, including processor cores, computer network transmitter/receivers and arithmetic circuits. Such FPGAs are often called *platform FGPAs*, meaning that the chip serves as a complete platform upon which a complex application can be implemented. Embedded software can run on the processor cores with instructions and data stored in block RAMs. The network connections can be used to communicate with other computers and devices, and the programmable logic and specialized arithmetic circuits can be used for high-performance data transformations required by the application. A minimal amount of circuitry is required externally to the FPGA, thus reducing the overall cost of the system.

Structured ASICs

Recently, manufacturers have developed a new kind of IC, called *structured ASICs*, that is midway between PLDs and standard-cell ASICs. A structured ASIC is an array of basic logic elements, like an FPGA. However, it is not programmable and omits the programmable interconnect. Moreover, the logic elements are generally very simple, comprising a collection of transistors that can be formed into logic gates and flip-flops. Whereas an FPGA is customized by loading a configuration program, a structured ASIC is customized by designing the top one or more layers of metal interconnection for the chip. Since the underlying logic elements and lower interconnection layers are fixed, the design effort and NRE cost for customization are much lower than those for a standard-cell ASIC. Further, since the structured ASIC is not programmable, just customized by a design and manufacturing process, the performance is potentially very close to that of a standard cell ASIC. Many observers expect that structured ASICs will become popular for complex medium- to high-volume applications over the next few years.

KNOWLEDGE TEST QUIZ

1. How does a programmable logic device differ from a fixed-function component?
2. What is a *fuse map*?
3. If crosses were drawn at the intersections (56, 28), (57, 0), (57, 7) and (58, 30) of the diagram in Figure 6.9, what logic function would be implemented?
4. Suppose the OLMC of Figure 6.12 is used for a state bit S2 of a finite-state machine. For each multiplexer, which input would be selected to make S2 available as an output and to feed it back for use in computing the next-state function?
5. What is the benefit of allowing a PLD in a system to be reprogrammed?
6. What are the purposes of logic blocks and I/O blocks in an FPGA?
7. What other blocks are included in an FPGA?
8. If an FPGA uses volatile SRAM cells to store configuration information, how is the configuration information stored and supplied to the FPGA?
9. What is an *antifuse*?
10. What distinguishes a platform FPGA from a simple FPGA?

6.3 PACKAGING AND CIRCUIT BOARDS

A single bare IC does not form a complete digital system. It needs to be packaged so that it can be connected to other ICs and components, including input and output displays for interacting with a user and connectors for cables for interacting with other systems. An IC is bonded into a package that serves several purposes. It protects the IC from moisture and airborne contaminants, it provides electrical connections, and it removes heat. There are numerous different kinds of IC package, each

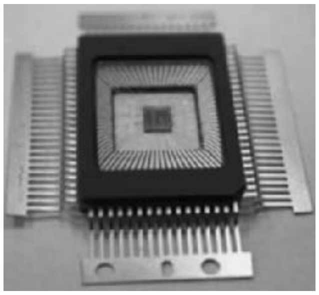


FIGURE 6.17 An IC in a package with bond-wires connecting to the lead frame.

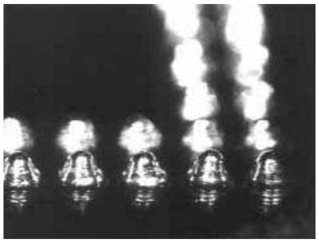


FIGURE 6.18 Connection bumps on a flip-chip IC.

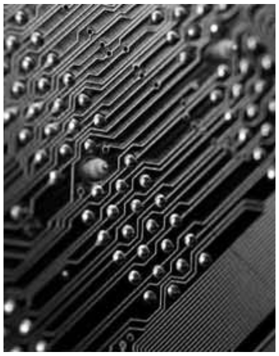


FIGURE 6.19 A through-hole PCB.

with different physical, electrical and thermal properties. The choice of package depends on the number of connections required and the environment in which the product is to operate, among other factors.

Within a package, the IC is affixed to the bottom of a cavity. Fine gold wires are connected from pads on the edge of the IC to points on the package's lead frame (see Figure 6.17), which is the metal framework leading to the external package pins. The cavity is then sealed to protect the IC and the wires. As IC technology has developed, the maximum number of pins has increased, as have operating speeds. For a high pin-count, high performance IC, using bond wires introduces mechanical problems and delays and degrades signals. Recent packages for these ICs have adopted flip-chip technology. The connection pads on the IC are covered in conductive material forming bumps (Figure 6.18). The IC is then flipped over and affixed to the substrate of the package, with the bumps in direct contact with substrate connection points. The connection points lead to the external pins of the package.

The packaged ICs and other components in a system are assembled together on a *printed circuit board* (PCB). This consists of layers of fiberglass or other insulating material separating layers of metal wiring. The metal is deposited in a layer on a fiberglass sheet, and then etched using a photolithographic process, similar to that used in manufacturing ICs. Several layers are sandwiched together. Small holes are drilled through the layers and coated with metal to form connections, called *vias*, between the layers. The completed PCB contains all the circuit wiring needed for the product.

One form of PCB, a *through-hole* PCB (Figure 6.19), includes additional metal-coated holes into which IC package pins are inserted. Solder, a metal alloy with a low melting point, is melted into the holes to form electrical connections between the pins and the PCB wiring. Products using this form of manufacture need ICs in *insertion-type* packages, such as those shown in Figure 6.20. Dual in-line packages (DIPs) have two rows of pins with 0.1-inch spacing. These were among the first IC packages to be introduced, being used for SSI and MSI components, but are less common now. They are relatively large and are limited in the number of pins they can provide, with a 48-pin DIP being about the largest practical size. ICs requiring more pins can be packaged in a pin-grid array (PGA) package, having up to 400 or more pins. However, these have largely been replaced by newer forms of package, and are now mainly used for ICs such as computer CPUs that are to be mounted in sockets so that they can be removed. One of the advantages of through-hole PCBs is that they can be manually assembled, since the component sizes are manageable. This is good for low-volume products, since the cost of setting up a manufacturing run is less than that for automated assembly. However, the move to ICs with higher pin counts has reduced the applicability of this technology.

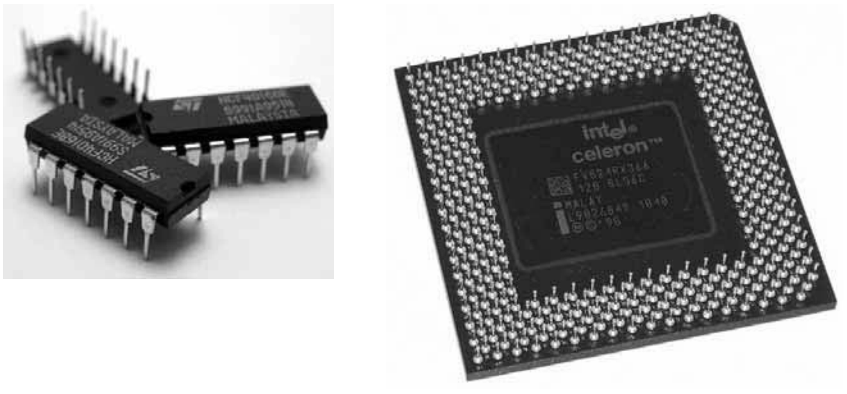


FIGURE 6.20 Insertion-type IC packages: DIPs (left) and PGAs (right).

The second form of PCB is a *surface-mount* PCB (Figure 6.21), so-called because components are mounted on the surface rather than being inserted in holes. This has the advantage of reduced manufacturing cost (for higher-volume products), finer feature sizes and increased circuit density. Surface mounting IC packages have pins or connection points that come into contact with a metal pad on the PCB. Solder paste is applied between each pin and pad and subsequently melted, forming the connection. There are numerous different surface mounting packages, some of which are shown in Figure 6.22. Quad flat-pack (QFP) packages have pins along all four sides, and are suitable for ICs with up to 200 or so pins. The spacing between pins varies from 1 mm for the packages with fewer pins, down to 0.65mm for the higher pin-count packages. Fine-pitch QFP packages allow increased pin count, up to nearly 400 pins, by reducing the pin spacing to 0.4mm. Given the delicacy of these pins, the packages are not suitable for manual handling and assembly. The most common package in use now for high pin-count ICs is the ball-grid array (BGA) package. Depending on the package size and the pin spacing, BGA

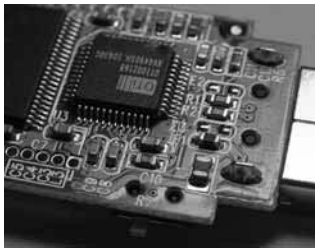


FIGURE 6.21 A surface-mount PCB.

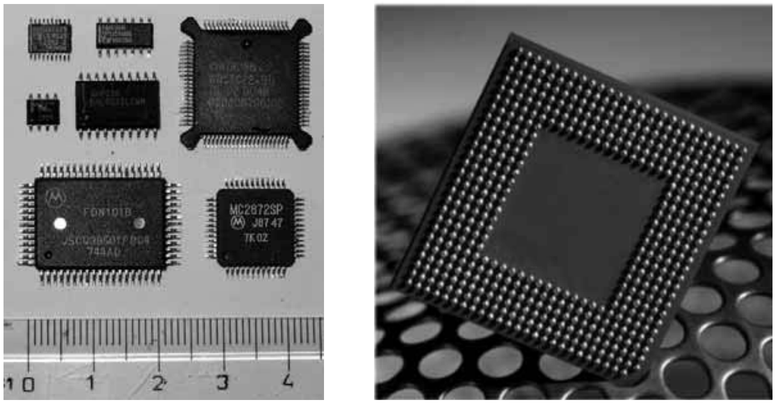


FIGURE 6.22 Surface mounting IC packages: QFPs (left) and BGA (right).

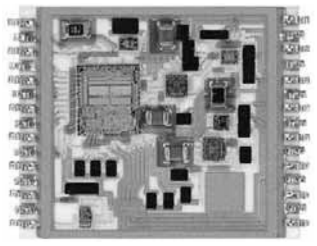


FIGURE 6.23 A multichip module.

packages can accommodate ICs with up to 1800 pins. Higher pin-count BGA packages are also being developed.

In recent times, high-density packaging techniques have been developed for use in products where space is constrained. A good example is a cell phone, in which small size and reduced weight are important marketing factors. Rather than placing each IC in a separate package and assembling several packages onto a PCB, *multichip modules* (MCMs) attach the bare chips to a ceramic substrate (see Figure 6.23). Interconnection wires and passive components (resistors and capacitors) are also printed or soldered onto the substrate. The complete module is then encapsulated with external connections made through package pins to a PCB. Even denser packaging can be achieved by building in three dimensions, rather than laying them out on a two-dimensional surface. For example, *chip stacking* involves placing two or more chips in a vertical stack. Connections can be made between adjacent chips by metal contacts, and between chips and the containing package by bond wires. Several flash memory manufacturers are using these techniques to provide high-capacity storage in very small packages. As demand for high-performance mobile devices increases, we can expect to see continued development of these high-density packaging techniques.

KNOWLEDGE TEST QUIZ

1. How does flip-chip IC packaging differ from previous packaging technologies?
2. What distinguishes surface-mount IC packages from insertion-type packages?
3. What is a *via* in a PCB?
4. For an IC with 1200 pins, what kind of package would most likely be used?

6.4 INTERCONNECTION AND SIGNAL INTEGRITY

When we introduced the digital abstraction in Chapter 1, we described signals as changing between low and high logic levels instantaneously. We emphasized, however, that this is an abstraction, and that real signals take time to change and time to propagate along signal wires. We have taken a relatively simple view of signal propagation between a source and a destination within a circuit. In practice, there are a number of complicating factors, particularly when the source and destination are in different ICs on a PCB. A signal change must propagate from the source driver, through the bond wire, package lead frame and pin of the source IC, along the PCB trace, through the pin, lead frame and bond wire of the destination IC, and into the receiver. Along this path, there are several influences that can cause distortion of the signal and introduce noise. The term *signal integrity* refers to the degree to which these effects are minimized. If we are using off-the-shelf ICs or PLDs, we do not have control over the path within the IC package. We must assume that the designers of the IC and package have done due diligence to maintain signal integrity. Alternatively, if we are implementing a design in an ASIC, we must take responsibility for signal integrity within the ASIC. Since this is a complex area, we largely defer it to an advanced reference on VLSI design, though many of the ideas that we discuss in this section do apply to ASIC design. In either case, using off-the-shelf parts or ASICs, we need to consider the effects of the PCB on signal integrity.

A change in a signal value causes a change in the current flowing through the PCB trace. This causes a change in the electric and magnetic fields around the trace. Propagation of those fields determines the speed of propagation of the signal change along the trace. In common PCB materials, the maximum propagation speed is approximately half the speed of light in a vacuum. Since the latter is $3 \times 10^8 \mathrm{ms}^{-1}$, we can use 150mm per nanosecond as a good rule of thumb for signal propagation along a PCB trace. For low speed designs and small PCBs, this element of total path delay is insignificant. However, for high-speed designs, particularly for signals on critical timing paths, it is significant. Two cases in point are the routing of clock signals and parallel bus signals. If a clock signal is routed through paths of different lengths to different ICs, we may introduce clock skew, in much the same way that we described in Section 4.4. Similarly, if different signals within a parallel bus are routed along paths of different lengths, changes in elements of the bus may not arrive concurrently, and may be incorrectly sampled at the destination's receiver. In these cases, it may be necessary to tune the timing of the system by adding to the length of some PCB traces to match propagation delays. CAD tools used for PCB layout offer features to help designers perform such tuning semiautomatically.

A major signal integrity issue in PCB design is *ground bounce*, which arises when one or more output drivers switch logic levels. During switching, both of the transistors in the driver's output stage are momentarily on, and transient current flows from the power supply to ground. Ideally, the power supply can source the transient current without distortion. In reality, however, there is inductance in both the power and the ground connections, as shown in Figure 6.24. The inductance causes voltage spikes in the power supply and ground on the IC. This can cause voltage spikes on other output drivers, possibly causing false transitions in the receivers to which they are connected. It can also cause transient shifting of the threshold voltage of receivers on the IC, causing false transitions at those receivers. The effect is particularly pronounced when multiple drivers switch concurrently, for example, when the value on a parallel bus changes, since the transient current is much greater.

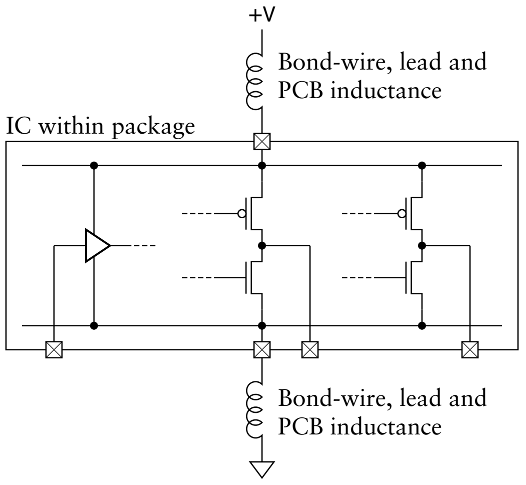

+V
Bond-wire, lead and PCB inductance
IC within package
Bond-wire, lead and PCB inductance

FIGURE 6.24 Inductance in the bond-wires, package leads and PCB connections for power and ground.

In order to reduce the effects of ground bounce, we can take a number of important measures. First, we can place *bypass capacitors* between power and ground at strategic places around a PCB. These capacitors hold a reserve of charge that can quickly supply the needs of switching drivers. A common rule of thumb is to place a capacitor close to each IC package. Values of 0.01μF to 0.1μF are common. Second, we can use separate PCB layers for the ground and power supply (Figure 6.26). This gives a low-inductance path for the power supply current and its ground return. It also has other benefits, mentioned below. Third, we can limit the rate of voltage change (the *slew rate*) and limit the drive current of the output drivers. These actions limit the rate of change of current, and so limit the inductive effect of the change. Components such as modern FPGAs have programmable output drivers that allow selection of slew rate and drive current limits. Of course, reducing the slew rate means that a signal takes longer to change from one logic level to the other, as illustrated in Figure 6.25. Hence, limiting slew rate may increase propagation delay through circuits, consequently requiring a reduction in clock rate. This is a case where a trade-off between speed of operation and noise immunity may be required. Finally, we can use differential signaling, discussed in Section 6.4.1, as a means of making the system more immune to noise induced by ground bounce.

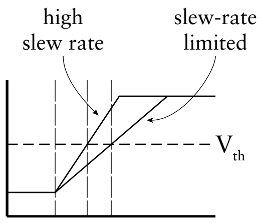


FIGURE 6.25 The effect of slew-rate limiting a signal. The signal takes longer to reach the threshold voltage V_{th}.

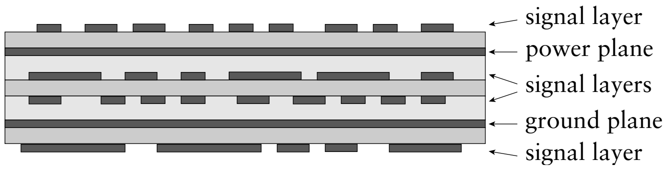


FIGURE 6.26 Cross section of a multilayer PCB with ground and power planes.

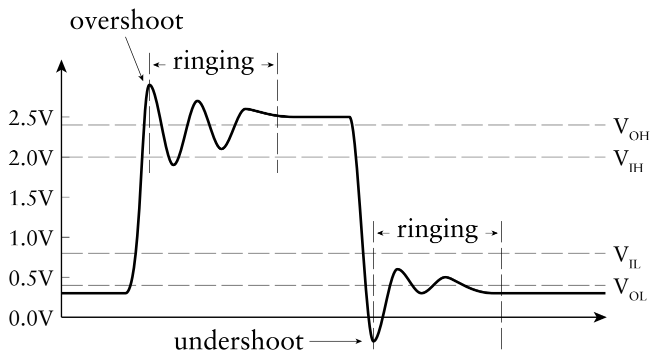


FIGURE 6.27 Overshoot, undershoot and ringing transmission-line effects.

Another signal integrity issue for high-slew rate signals is noise due to transmission-line effects. When the time for a transition between logic levels is similar to or shorter than the propagation delay along a signal path, the transition is affected by reflections at the driving and receiving ends of the path. A full analysis of the effects requires knowledge of the characteristic impedance of the path, as well as the source impedance of the driver and the terminating impedance of the receiver. Depending on the relationships between these values, the signal may suffer from partial transitions, overshoot, undershoot and ringing (Figure 6.27). The situation is made more complex if the signal wire is not a simple driver-to-receiver connection, but has multiple receivers along the path. PCB layout artifacts, such as vias and branching paths, also introduce further effects.

The main design techniques for managing transmission-line effects involve appropriate layout and proper termination of PCB traces. By running a trace of specific dimensions at a controlled distance between two ground or power planes in the PCB, we create a *stripline* transmission line with a controlled characteristic impedance. Where the transmission line effects are less critical, we can run a trace over just one plane, creating a *microstrip* transmission line. For critical signals, we can adopt circuit designs and layouts that avoid placing receivers along the PCB trace, or that group them together at the receiving end. Finally, we can include termination resistors to ensure proper matching of drivers and receivers to the characteristic impedance of the transmission line. In high-performance modern components, including FPGAs, the drivers include termination resistors on the IC. In other cases, we may need to include resistors as discrete components adjacent to IC pins.

As we mentioned earlier, transitions between logic levels on a signal cause electromagnetic fields to propagate around the PCB trace. Some of the field energy is radiated out from the system, and may impinge on other electronic systems, where it induces noise. This form of unwanted coupling is called *electromagnetic interference* (EMI). There are government and other regulations that limit the amount of EMI that a system may emit in various environments, since excessive EMI can be annoying (for example, if it interferes with your TV reception) or a safety hazard (for example, if it interferes with your aircraft navigation). Electromagnetic fields from an "aggressor" PCB trace can also impinge on adjacent traces, inducing *crosstalk* on the "victim" traces. The closer the traces and the longer their parallel paths, the more pronounced the crosstalk effect. As with other signal integrity issues, appropriate PCB design techniques, such as routing traces close to ground or power planes, can reduce EMI and crosstalk by containing the electromagnetic fields. Limiting slew rates of transitions also reduces the radiated energy, and so reduces EMI and crosstalk.

6.4.1 DIFFERENTIAL SIGNALING

The techniques for maintaining signal integrity that we have discussed so far are based on reducing the amount of interference induced on signal wires. Another technique, use of *differential signaling*, is based on the idea of reducing a system's susceptibility to interference. Rather than transmitting a bit of information as a single signal S, we transmit both the positive signal S_P and its negation S_N. At the receiving end, we sense the voltage difference between the two signals. If S_P − S_N is a positive voltage, then S is received as the value 1; if S_P − S_N is a negative voltage, then S is received as 0. This arrangement is illustrated in Figure 6.28. The assumption behind the differential signaling approach is that noise is induced equally on the wires for both S_P and S_N. Such common-mode noise is cancelled out when we sense the voltage difference. To show this, suppose a noise voltage V_N is induced equally on the two wires. At the receiver, we sense the voltage

$$(S\_P + V_N) - (S\_N + V_N) = S\_P + V_N - S\_N - V_N = S\_P - S\_N$$

For the assumption of common-mode noise induction to hold, differential signals must be routed along parallel paths on a PCB. While this might suggest a problem with crosstalk between the two traces, the fact that the signals are inverses of each other means that they both change at the same time, and crosstalk effects cancel out.

As well as rejecting common-mode noise, differential signaling also has the advantage that reduced voltage swings are needed for a given noise margin. Even though each of S_P and S_N switches between V_{OL} and V_{OH}, the differential swing at the receiver is between $V_{OL} - V_{OH}$ and $V_{OH} - V_{OL}$, that is, twice the swing of each individual signal. Reducing the voltage swing has multiple follow-on effects, including reduced switching current, reduced ground bounce, reduced EMI, and reduced crosstalk with other signals. Thus, use of differential signals can be very beneficial in high-speed designs.

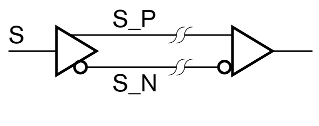


FIGURE 6.28 A differential driver and receiver.

approximation would be below the DAC output. The comparator output is stored as the most significant bit in the SAR, and remaining bits are shifted down one place. This gives the next approximation, $d_7$0111111, which is either one-quarter or three-quarters of the full-scale voltage, depending on d_7. During the next clock period, this next approximation is converted by the DAC and compared with the input voltage to yield the next most significant bit of the result and a refined approximation, $d_7d_6$011111. The process repeats over successive clock cycles, refining the approximation by one bit each cycle. When all bits of the result are determined, the SAR activates the done output, indicating that the complete result can be read.

The advantage of a successive approximation ADC over a flash ADC is that it requires significantly fewer analog components: just one comparator and a DAC. These components can be made to high precision, giving a high-precision ADC. 12-bit successive approximation ADCs, for example, are commonly available. The disadvantage, however, is that more time is required to convert a value. If the input signal changes by more than the precision of the ADC while the ADC is making successive approximation, we need to *sample and hold* the input. This requires a circuit that charges a capacitor to match the input voltage during a brief sampling interval, and then maintain the voltage on the capacitor while it is being converted. Another disadvantage of the successive approximation ADC is the amount of digital circuitry required to implement the SAR. However, that function could be implemented on an embedded processor, requiring just an output register to drive the DAC and an input bit from the comparator. The sequencing of successive approximations would then form part of the embedded software.

There are other forms of ADC apart from flash and successive approximation ADCs, each with advantages and disadvantages. Choice among them depends on the resolution, conversion speed and other factors dictated by the application. In practice, there is often a need to filter the analog input signal to ensure correct conversion to digital form. These considerations are beyond the scope of this book. More details can be found in books on digital signal processing mentioned in the Further Reading section.

8.1.2 OUTPUT DEVICES

Among the most common output devices are indicator lights that display on/off or true/false information. For example, an indicator might show whether a mode or operation is active, whether the system is busy, or whether an error condition has occurred. The simplest form of indicator is a single light-emitting diode (LED). It is low in cost, highly reliable, and easy to drive from a digital circuit, as Figure 8.7 shows. When the output from the driver is a low voltage, current flows through the LED,

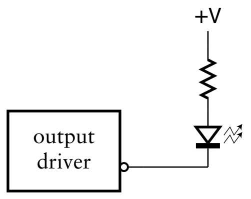


FIGURE 8.7 Output circuit for an LED indicator.