nously set to 960. Create a MAX+PLUS II simulation to verify the operation of the design.

9.9 Shift Register Counters

9.56 Write the VHDL code for a ring counter of generic width and instantiate it as an 8-bit ring counter. List the sequence of states in a table, assuming the counter is initially cleared, and create a simulation to verify the circuit's operation. Include a clear input (synchronous).

9.57 Construct the count sequence table of a 5-bit Johnson counter, assuming the counter is initially cleared. What changes must be made to the decoder part of the circuit in Figure 9.84 (p. 446) if it is to decode the 5-bit Johnson counter?

9.58 A control sequence has ten steps, each activated by a logic HIGH. Use MAX+PLUS II to design a counter and decoder in each of the following configurations to produce the required sequence: binary counter, ring counter, and Johnson counter. You may use a Graphic Design File or VHDL. Create a simulation for each counter and decoder.

9.59 Use the MAX+PLUS II Graphic Editor to design a 4-bit ring counter that can be asynchronously initialized to $Q_3Q_2Q_1Q_0 = 1000$ by using only the clear inputs of its flip-flops. No presets allowed. *Hint:* use a circuit with a "double twist" in the data path.

ANSWERS TO SECTION REVIEW PROBLEMS

Section 9.1

9.1 A mod-24 UP counter goes from 00000 to 10111 (0 to 23). This requires 5 outputs. The counter is a truncated sequence since its modulus is less than $2^5 = 32$.

Section 9.2

9.2 1001, 0000

Section 9.3

9.3 JK flip-flops: $J_3K_3 = X0$, $J_2K_2 = 1X$, $J_1K_1 = X1$, $J_0K_0 = X1$
D flip-flops: $D_3 = 1$, $D_2 = 1$, $D_1 = 0$, $D_0 = 0$

Section 9.4

9.4
```
If (clock'EVENT AND clock = '0') THEN
        count := count + 1;
END IF;
```

Section 9.5

9.5 The completed timing diagram is shown in Figure 9.93.

Section 9.6

9.6 Asynchronous clear: PROCESS (clock, clear); Synchronous clear: PROCESS (clock)

Section 9.7

9.7 JK flip-flops can be used in the shift register of Figure 9.58. The Q output of any stage connects to the J input of the next stage and the $\overline{Q}$ output of any stage connects to the K input of the next. The ***serial_in*** input connects directly to the J input of the first flip-flop. ***Serial_in*** is applied to K of the first flip-flop through an inverter (NOT gate).

Section 9.8

9.8 A shift register output is defined as a port of mode BUFFER because this mode allows a signal to be fed back into the PLD matrix and reused as an input to another part of the circuit.

Section 9.9

Binary: 5 flip-flops, 24 5-inputs NANDs; Ring: 24 flip-flops, no

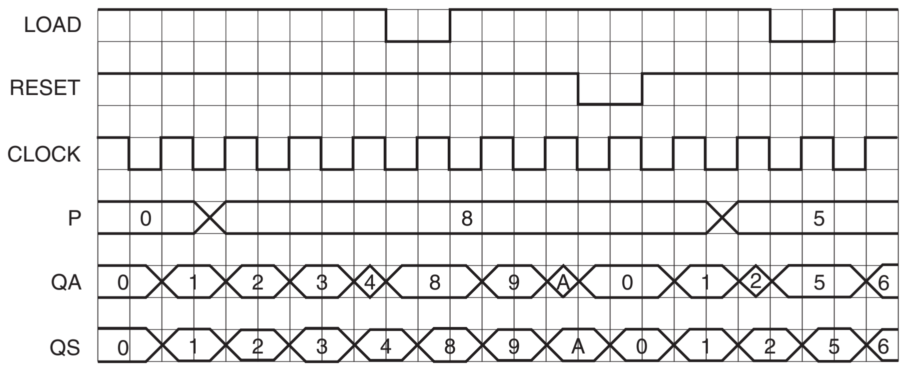


FIGURE 9.93
Answer to Section Review Problem 9.5

CHAPTER 10

State Machine Design

OUTLINE

CHAPTER OBJECTIVES

Upon successful completion of this chapter you will be able to:

- Describe the components of a state machine.
- Distinguish between Moore and Mealy implementations of state machines.
- Draw the state diagram of a state machine from a verbal description.
- Use the "classical" (state table) method of state machine design to determine the Boolean equations of the state machine.
- Translate the Boolean equations of a state machine into a Graphic Design File in Altera's MAX+PLUS II software.
- Write VHDL code to implement state machines.
- Create simulations in MAX+PLUS II to verify the function of a state machine design.
- Determine whether the output of a state machine is vulnerable to asynchronous changes of input.
- Design state machine applications, such as a switch debouncer, a single-pulse generator, and a traffic light controller.

10.1 State Machines

KEY TERMS

State machine A synchronous sequential circuit, consisting of a sequential logic section and a combinational logic section, whose outputs and internal flip-flops progress through a predictable sequence of states in response to a clock and other input signals.

Moore machine A state machine whose output is determined only by the sequential logic of the machine.

Mealy machine A state machine whose output is determined by both the sequential logic and the combinational logic of the machine.

State variables The variables held in the flip-flops of a state machine that determine its present state. *The number of state variables in a machine is equivalent to the number of flip-flops.*

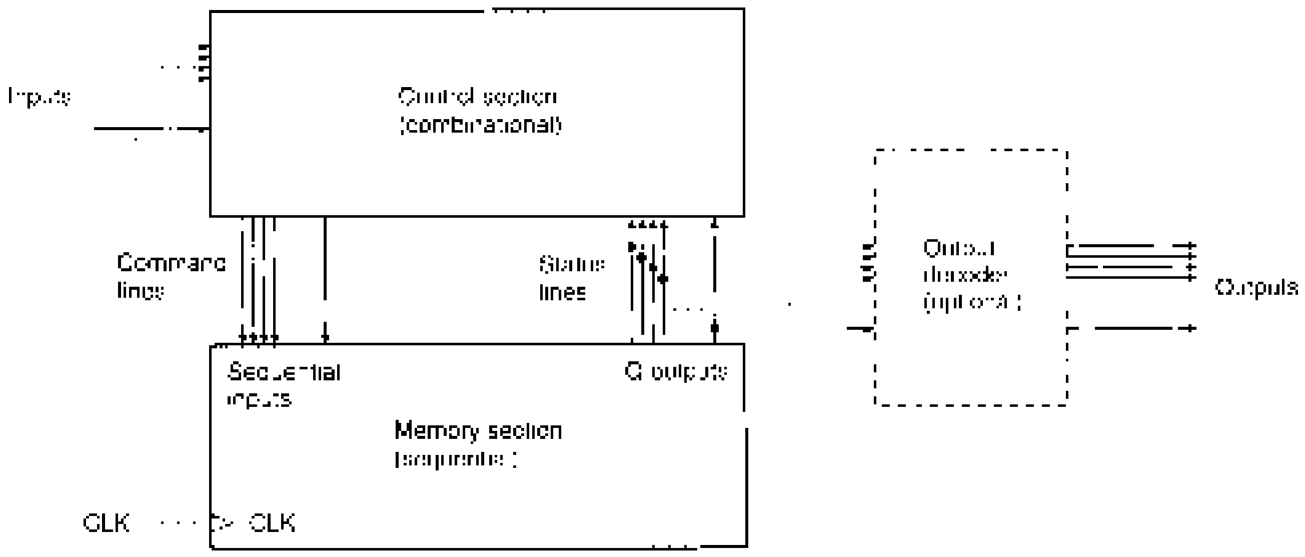


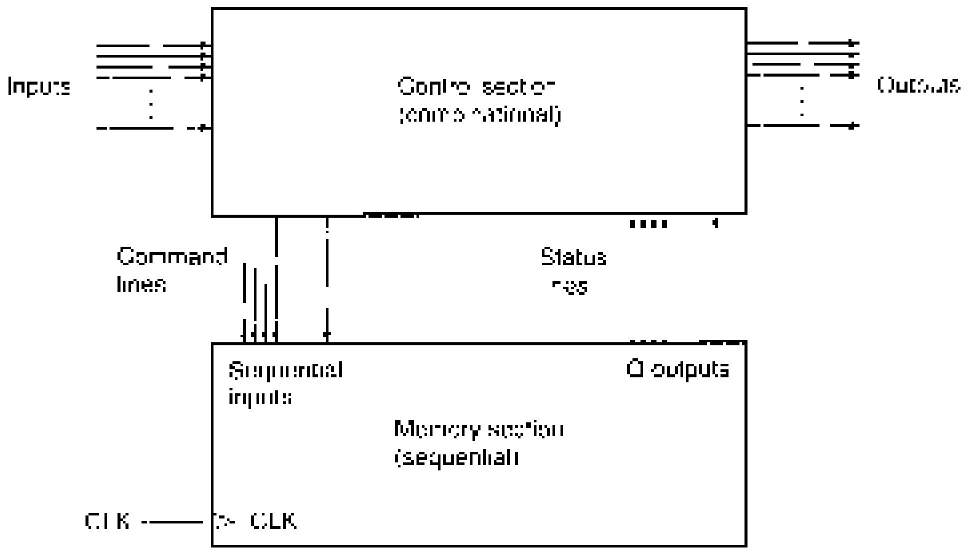
FIGURE 10.1
Moore-Type State Machine

The synchronous counters and shift registers we examined in Chapter 9 are examples of a larger class of circuits known as **state machines.** As described for synchronous counters in Section 9.2, a state machine consists of a memory section that holds the present state of the machine and a control section that determines the machine's next state. These sections communicate via a series of command and status lines. Depending on the type of machine, the outputs will either be functions of the present state only or of the present and next states.

Figure 10.1 shows the block diagram of a **Moore machine.** The outputs of a Moore machine are determined solely by the present state of the machine's memory section. The output may be directly connected to the *Q* outputs of the internal flip-flops, or the *Q* outputs might pass through a decoder circuit. The output of a Moore machine is synchronous to the system clock, since the output can only change when the machine's internal **state variables** change.

The block diagram of a **Mealy machine** is shown in Figure 10.2. The outputs of the Mealy machine are derived from the combinational (control) section of the machine, as

FIGURE 10.2
Mealy-Type State Machine



well as the sequential (memory) part of the machine. Therefore, the outputs can change asynchronously when the combinational circuit inputs change out of phase with the clock. (When we say that the outputs change asynchronously, we generally do not mean a change via a function such as asynchronous reset that directly affects the machine's flip-flops.)

SECTION 10.1 REVIEW PROBLEM

10.1 What is the main difference between a Moore-type state machine and a Mealy-type state machine?

10.2 State Machines with No Control Inputs

KEY TERMS

Bubble A circle in a state diagram containing the state name and values of the state variables.

A state machine can be designed using a classical technique, similar to that used to design a synchronous counter. We can also use a VHDL design method. We will design several state machines, using both classical and VHDL techniques.

As an example of these techniques, we will design a state machine whose output depends only on the clock input: a 3-bit counter with a Gray code count sequence. A 3-bit Gray code, shown in Table 10.1, changes only one bit between adjacent codes and is therefore not a binary-weighted sequence.

Table 10.1 3-bit Gray Code Sequence

$Q_2Q_1Q_0$
000
001
011
010
110
111
101
100

Gray code is often used in situations where it is important to minimize the effect of single-bit errors. For example, suppose the angle of a motor shaft is measured by a detected code on a Gray-coded shaft encoder, shown in Figure 10.3. The encoder indicates a 3-bit number for each of eight angular positions by having three concentric circular segments for each code. A dark band indicates a 1 and a transparent band indicates a 0, with the MSB as the outermost band. The dark or transparent bands are detected by three sensors that detect

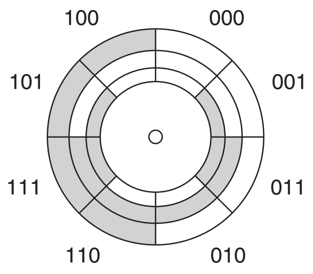


FIGURE 10.3
Gray Code on a Shaft Encoder

light shining through a transparent band. (A real shaft encoder has more bits to indicate an angle more precisely. For example, a shaft encoder that measures an angle of one degree would require nine bits, since there are 360 degrees in a circle and $2^8 \leq 360 \leq 2^9$.)

For most positions on the encoder, the error of a single bit results in a positional error of only one eighth of the circle. This is not true with binary coding, where single bit errors can give larger positional errors. For example if the positional decoder reads 100 instead of 000, this is a difference of 4 in binary. The same codes differ by only one position in Gray code.

Classical Design Techniques

We can summarize the classical design technique for a state machine, as follows:

1. Define the problem.
2. Draw a state diagram.
3. Make a state table that lists all possible present states and inputs and the next state and output state for each present state/input combination. *List the present states and inputs in* ***binary order.***
4. Use flip-flop excitation tables to determine at what states the flip-flop synchronous inputs must be to make the circuit go from each present state to its next state. *The next state variables are functions of the inputs and present state variables.*
5. Write the output value for each present state/input combination. *The output variables are functions of the inputs and present state variables.*
6. Simplify the Boolean expression for each output and synchronous input.
7. Use the Boolean expressions found in step 6 to draw the required logic circuit.

Let us follow this procedure to design a 3-bit Gray code counter. We will modify the procedure to account for the fact that there are no inputs other than the clock and no outputs that must be designed apart from the counter itself.

1. *Define the problem.* Design a counter whose outputs progress in the sequence defined in Table 10.1.
2. *Draw a state diagram.* The state diagram is shown in Figure 10.4. In addition to the values of state variables shown in each circle (or **bubble**), we also indicate a state name, such as s0, s1, s2, and so on. This name is independent of the value of state variables. We use numbered states (s0, s1, . . .) for convenience, but we could use any names we wanted to.

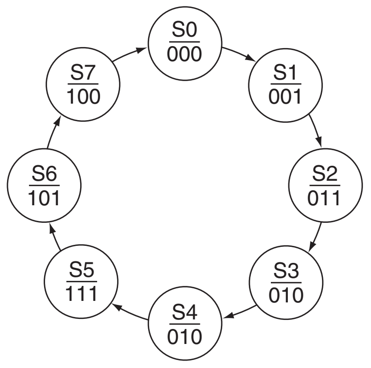


FIGURE 10.4
State Diagram for a 3-bit Gray Code Counter

3. *Make a state table.* The state table, based on D flip-flops, is shown in Table 10.2. *Since there are eight unique states in the state diagram, we require three state variables (2^3 = 8), and hence three flip-flops.* Note that the present states are in binary-weighted order, even though the count does not progress in this order. In such a case, it is essential to have an accurate state diagram, from which we derive each next state. For example, if

Table 10.2 State Table for a 3-bit Gray Code Counter

Present State	Next State	Synchronous Inputs
$Q_2Q_1Q_0$	$Q_2Q_1Q_0$	$D_2D_1D_0$
000	001	001
001	011	011
010	110	110
011	010	010
100	000	000
101	100	100
110	111	111
111	101	101

the present state is 010, the next state is not 011, as we would expect, but 110, which we derive by examining the state diagram.

Why list the present states in binary order, rather than the same order as the output sequence? By doing so, we can easily simplify the equations for the *D* inputs of the flip-flops by using a series of Karnaugh maps. This is still possible, but harder to do, if we list the present states in order of the output sequence.

4. *Use flip-flop excitation tables to determine at what states the flip-flop synchronous inputs must be to make the circuit go from each present state to its next state.* This is not necessary if we use *D* flip-flops, since *Q* follows *D*. The *D* inputs are the same as the next state outputs. For JK or T flip-flops, we would follow the same procedure as for the design of synchronous counters outlined in Chapter 9.
5. *Simplify the Boolean expression for each synchronous input.* Figure 10.5 shows three Karnaugh maps, one for each *D* input of the circuit.

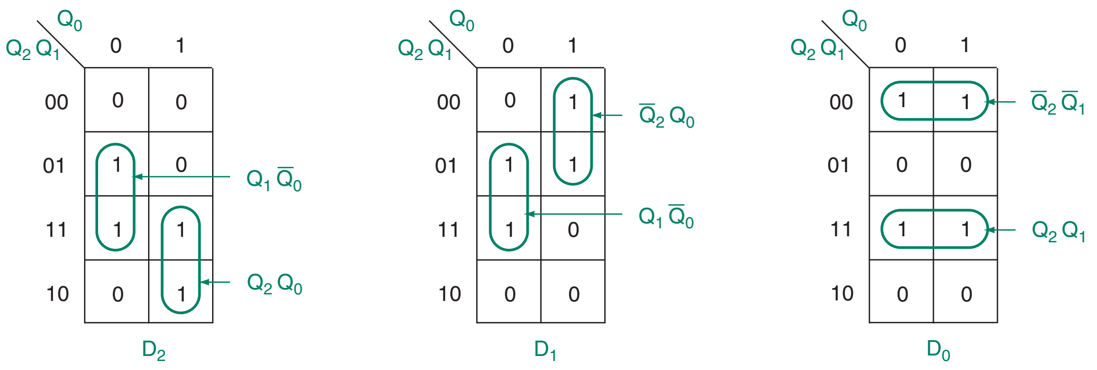


FIGURE 10.5
Karnaugh Maps for 3-bit Gray Code Counter

The K-maps yield three Boolean equations:

$$D_2 = Q_1\overline{Q}_0 + Q_2Q_0$$
$$D_1 = Q_1\overline{Q}_0 + \overline{Q}_2Q_0$$
$$D_0 = \overline{Q}_2\,\overline{Q}_1 + Q_2Q_1$$

6. *Draw the logic circuit for the state machine.* Figure 10.6 shows the circuit for a 3-bit Gray code counter, drawn as a Graphic Design File in MAX+PLUS II. A simulation for this circuit is shown in Figure 10.7, with the outputs shown as individual waveforms and as a group with a binary value.

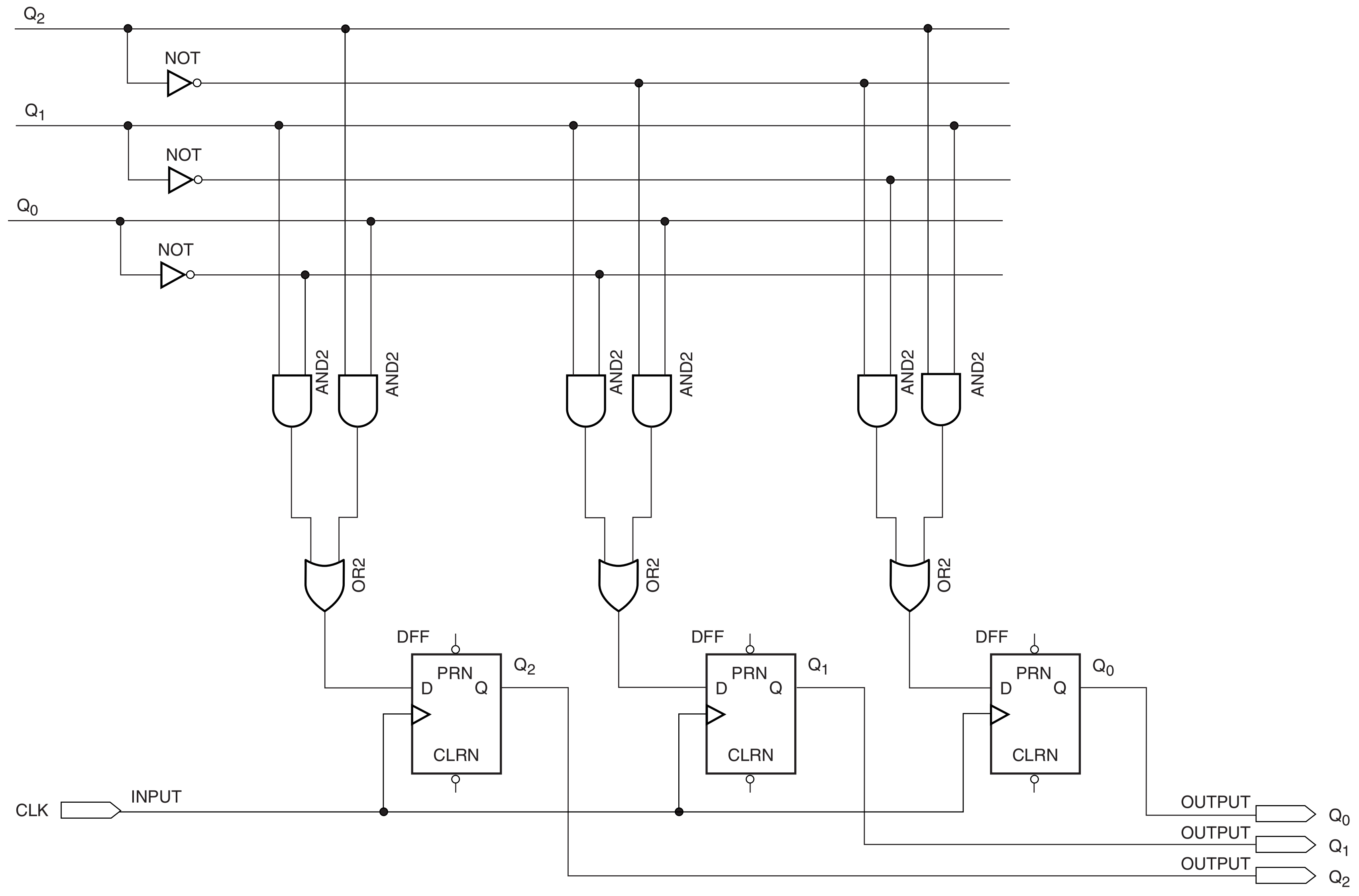


FIGURE 10.6
Logic Diagram of a 3-bit Gray Code Counter

gray_ct3.gof
gray_ct3.scf

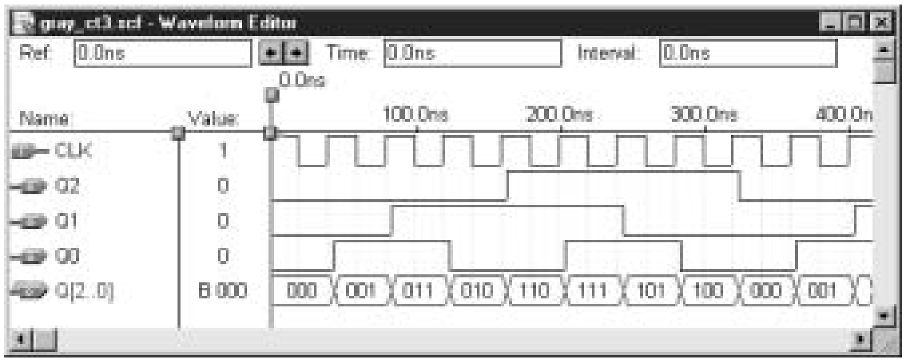


FIGURE 10.7
Simulation of a 3-bit Gray Code Counter (from Graphic Design File)

VHDL Design of State Machines

KEY TERMS

Enumerated type A user-defined type in VHDL in which all possible values of a named identifier are listed in a type definition statement.

State machines can be defined in VHDL within a CASE statement. The VHDL code below illustrates the principle, using the 3-bit Gray code counter as an example.

gray_ct1.vhd

```
-- gray_ct1.vhd
-- 3-bit Gray code counter
-- (state machine with decoded outputs)

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY gray_ct1 IS
    PORT(
        clk : IN    STD_LOGIC;
        q   : OUT   STD_LOGIC_VECTOR(2 downto 0));
END gray_ct1;

ARCHITECTURE a OF gray_ct1 IS
    TYPE STATE_TYPE IS (s0, s1, s2, s3, s4, s5, s6, s7);
    SIGNAL state: STATE_TYPE;
BEGIN
    PROCESS (clk)
    BEGIN
        IF clk'EVENT AND clk = '1' THEN
            CASE state IS
                WHEN s0 =>
                    state <= s1;
                WHEN s1 =>
                    state <= s2;
                WHEN s2 =>
                    state <= s3;
                WHEN s3 =>
                    state <= s4;
                WHEN s4 =>
                    state <= s5;
                WHEN s5 =>
                    state <= s6;
                WHEN s6=>
                    state <= s7;
                WHEN s7 =>
                    state <= s0;
        END CASE;
    END IF;
END PROCESS;

WITH state SELECT
    q <= "000" WHEN s0,
          "001" WHEN s1,
          "011" WHEN s2,
          "010" WHEN s3,
          "110" WHEN s4,
          "111" WHEN s5,
          "101" WHEN s6,
          "100" WHEN s7;
END a;
```

Recall that the format of a CASE statement is:

```
CASE __expression IS
    WHEN__constant_value =>
        __statement;
        __statement;
    WHEN__constant_value =>
        __statement;
        __statement;
    WHEN OTHERS =>
        __statement;
        __statement;
END CASE;
```

The keyword **expression** in the CASE statement refers to a signal called **state** that we define to represent the state variables within the machine. For each possible value of **state,** we make an assignment indicating the next state of the machine. For example, the clause `(WHEN s0 => (state <= s1));` indicates a transition from state s0 to state s1. The actual output values of the counter are assigned in a selected signal assignment statement after the PROCESS statement.

Notice that the signal **state** can have one of eight different values, from s0 to s7. Until now, we have seen signals with values such as `'1'` (BIT or STD_LOGIC types), `"011"` (BIT_VECTOR or STD_LOGIC_VECTOR types), or 7 (INTEGER types). The signal **state** is of type STATE_TYPE, which is a user-defined **enumerated type.** An enumerated type is simply a list of all values a signal, variable, or port of that type is allowed to have.

For example, we could define a type called DIRECTION with four values, with the statement:

```
TYPE DIRECTION IS (up, down, left, right);
```

We could then define a signal called **position** of type DIRECTION:

```
SIGNAL position: DIRECTION:
```

An IF statement or other construct could then assign one of the four defined values of type DIRECTION to the signal called **position:**

```
IF (x='0' and y='0') THEN
   position <= down;
ELSIF (x='0' and y='1') THEN
   position <= left;
ELSIF (x='1' and y='0') THEN
   position <= up;
ELSE
   position <= right;
END IF;
```

Thus the named identifier **position** of type DIRECTION can take on only the four values specified in the enumerated type definition.

An alternative way to encode the 3-bit counter is to include output assignments within the body of the CASE statement. Each case then has more than one statement, as indicated in the following VHDL code.

gray_ct2.vhd

```
-- gray_ct2.vhd
-- 3-bit Gray code counter
-- (outputs defined within states)

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY gray_ct2 IS
   PORT(
      clk : IN   STD_LOGIC;
      q   : OUT  STD_LOGIC_VECTOR(2 downto 0));
END gray_ct2;

ARCHITECTURE a OF gray_ct2 IS
   TYPE STATE_TYPE IS (s0, s1, s2, s3, s4, s5, s6, s7);
   SIGNAL state: STATE_TYPE;
BEGIN
   PROCESS (clk)
   BEGIN
      IF clk'EVENT AND clk = '1' THEN
         CASE state IS
            WHEN s0 =>
               state <= s1;
               q <= "001";
            WHEN s1 =>
               state <= s2;
               q <= "011";
            WHEN s2 =>
               state <= s3;
               q <= "010";
            WHEN s3 =>
               state <= s4;
               q <= "110";
            WHEN s4 =>
               state <= s5;
               q <= "111";
            WHEN s5 =>
               state <= s6;
               q <= "101";
            WHEN s6 =>
               state <= s7;
               q <= "100";
            WHEN s7 =>
```

```
                    state <= s0;
                    q <= "000";
                END CASE;
            END IF;
        END PROCESS;
END a;
```

The above VHDL code is identical to that of the previous example, except for the way the outputs are assigned.

SECTION 10.2 REVIEW PROBLEM

10.2 Write the Boolean equations for the *J* and *K* inputs of the flip-flops in a 3-bit Gray code counter based on JK flip-flops.

10.3 State Machines with Control Inputs

KEY TERMS

Control input A state machine input that directs the machine from state to state.

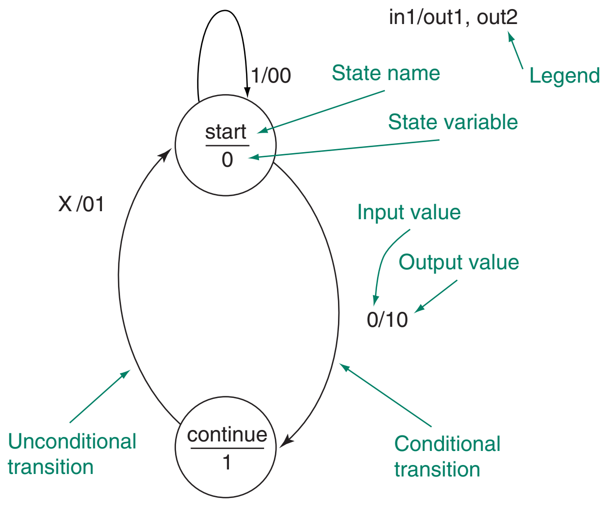
Conditional transition A transition between states of a state machine that occurs only under specific conditions of one or more control inputs.

Unconditional transition A transition between states of a state machine that occurs regardless of the status of any control inputs.

As an extension of the techniques used in the previous section, we will examine the design of state machines that use **control inputs,** as well as the clock, to direct their operation. Outputs of these state machines will not necessarily be the same as the states of the machine's flip-flops. As a result, this type of state machine requires a more detailed state diagram notation, such as that shown in Figure 10.8.

The state machine represented by the diagram in Figure 10.8 has two states, and thus

FIGURE 10.8
State Diagram Notation



requires only one state variable. Each state is represented by a bubble (circle) containing the state name and the value of the state variable. For example, the bubble containing the notation $\frac{\textbf{start}}{\textbf{0}}$ indicates that the state called **start** corresponds to a state variable with a value of 0. Each state must have a unique value for the state variable(s).

Transitions between states are marked with a combination of input and output values corresponding to the transition. The inputs and outputs are labeled **in1, in2, . . . , in*x*/out1, out2, . . . ,out*x*.** The inputs and outputs are sometimes simply indicated by the value of each variable for each transition. In this case, a legend indicates which variable corresponds to which position in the label.

For example, the legend in the state diagram of Figure 10.8 indicates that the inputs and outputs are labeled in the order **in1/out1, out2**. Thus if the machine is in the **start** state and the input **in1** goes to 0, there is a transition to the state **continue**. During this transition, **out1** goes to 1 and **out2** goes to 0. This is indicated by the notation 0/10 beside the transitional arrow. This is called a **conditional transition** because the transition depends on the state of **in1.** The other possibility from the **start** state is a no-change transition, with both outputs at 0, if **in1** = 1. This is shown as 1/00.

If the machine is in the state named **continue,** the notation X/01 indicates that the machine makes a transition back to the **start** state, regardless of the value of **in1,** and that **out1** = 0 and **out2** = 1 upon this transition. Since the transition always happens, it is called an **unconditional transition.**

What does this state machine do? We can determine its function by analyzing the state diagram, as follows.

1. There are two states, called **start** and **continue.** The machine begins in the **start** state and waits for a LOW input on **in1.** As long as **in1** is HIGH, the machine waits and the outputs **out1** and **out2** are both LOW.
2. When **in1** goes LOW, the machine makes a transition to **continue** in one clock pulse. Output **out1** goes HIGH.
3. On the next clock pulse, the machine goes back to **start.** The output **out2** goes HIGH and **out1** goes back LOW.
4. If **in1** is HIGH, the machine waits for a new LOW on **in1.** Both outputs are LOW again. If **in1** is LOW, the cycle repeats.

In summary, the machine waits for a LOW input on **in1,** then generates a pulse of one clock cycle duration on **out1,** then on **out2.** A timing diagram describing this operation is shown in Figure 10.9.

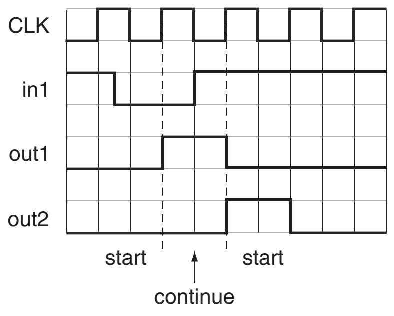


FIGURE 10.9
Ideal Operation of State Machine in Figure 10.8

Classical Design of State Machines with Control Inputs

We can use the classical design technique of the previous section to design a circuit that implements the state diagram of Figure 10.8.

1. *Define the problem.* Implement a digital circuit that generates a pulse on each of two outputs, as described above. For this implementation, let us use JK flip-flops for the state logic. If we so chose, we could also use D or T flip-flops.
2. *Draw a state diagram.* The state diagram is shown in Figure 10.8.

Table 10.3 State Table for State Diagram in Figure 10.8

Present State	Input	Next State	Sync. Inputs	Outputs	
Q	*in1*	*Q*	*JK*	*out1*	*out2*
0	0	1	1X	1	0
0	1	0	0X	0	0
1	0	0	X1	0	1
1	1	0	X1	0	1

3. *Make a state table.* The state table is shown in Table 10.3. The combination of present state and input are listed in binary order, thus making Table 10.3 into a truth table for the next state and output functions. Since there are two states, we require one state variable, *Q*. The next state of *Q*, a function of the present state and the input **in1**, is determined by examining the state diagram. (Thus, if you are in state 0, the next state is 1 if **in1** = 0 and 0 if **in1** = 1. If you are in state 1, the next state is always 0.)

Table 10.4 JK Flip-Flop Excitation Table

Transition	*JK*
0→0	0X
0→1	1X
1→0	X1
1→1	X0

4. *Use flip-flop excitation tables to determine at what states the flip-flop synchronous inputs must be to make the circuit go from each present state to its next state.* Table 10.4 shows the flip-flop excitation table for a JK flip-flop. The synchronous inputs are derived from the present-to-next state transitions in Table 10.4 and entered into Table 10.3. (Refer to the synchronous counter design process in Chapter 9 for more detail about using flip-flop excitation tables.)
5. *Write the output values for each present state/input combination.* These can be determined from the state diagram and are entered in the last two columns of Table 10.3.
6. *Simplify the Boolean expression for each output and synchronous input.* The following equations represent the next state and output logic of the state machine:

$$J = \overline{Q} \cdot \overline{in1} + Q \cdot \overline{in1} = \overline{in1}$$
$$K = 1$$
$$out1 = \overline{Q} \cdot \overline{in1}$$
$$out2 = Q \cdot \overline{in1} + Q \cdot in1 = Q$$

7. *Use the Boolean expressions found in step 6 to draw the required logic circuit.*

Figure 10.10 shows the circuit of the state machine drawn as a MAX+PLUS II Graphic Design File. Since **out1** is a function of the control section and the memory section of the machine, we can categorize the circuit as a Mealy machine. (All counter circuits that we have previously examined have been Moore machines since their outputs are derived solely from the flip-flop outputs of the circuit.)

Since the circuit is a Mealy machine, it is vulnerable to asynchronous changes of output due to asynchronous input changes. This is shown in the simulation waveforms of Figure 10.11.

state_x2a.gdf
state_x2a.scf

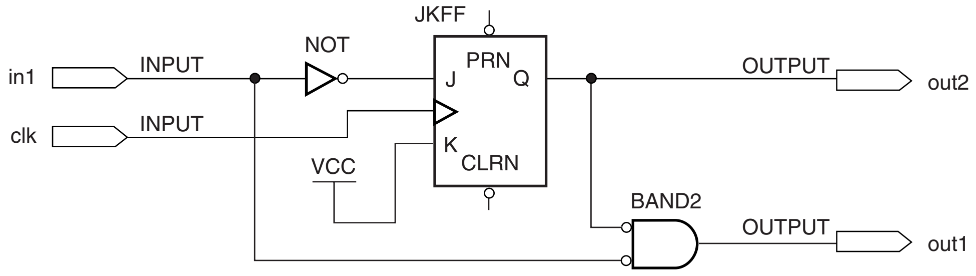


FIGURE 10.10
Implementation of State Machine of Figure 10.8

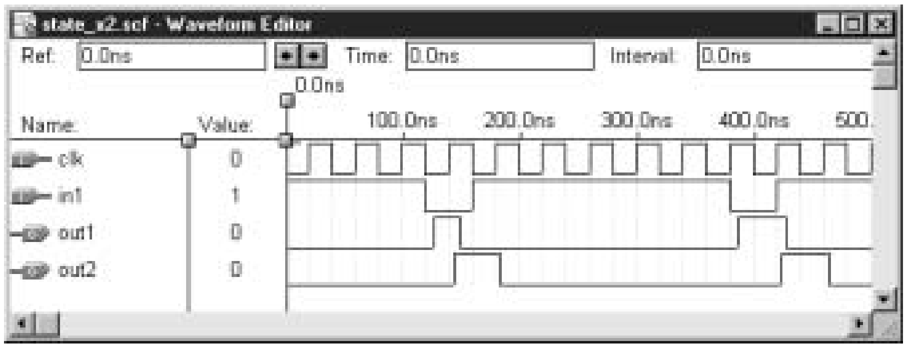


FIGURE 10.11
Simulation of State Machine Circuit of Figure 10.10

Ideally, **out1** should not change until the first positive clock edge after **in1** goes LOW. However, since **out1** is derived from a combinational output, it will change as soon as **in1** goes LOW, after allowing for a short propagation delay. Also, since **out2** is derived directly from a flip-flop and **out1** is derived from the same flip-flop via a gate, **out1** stays HIGH for a short time after **out2** goes HIGH. (The extra time represents the propagation delay of the gate.)

If output synchronization is a problem (and it may not be), it can be fixed by adding a synchronizing D flip-flop to each output, as shown in Figure 10.12.

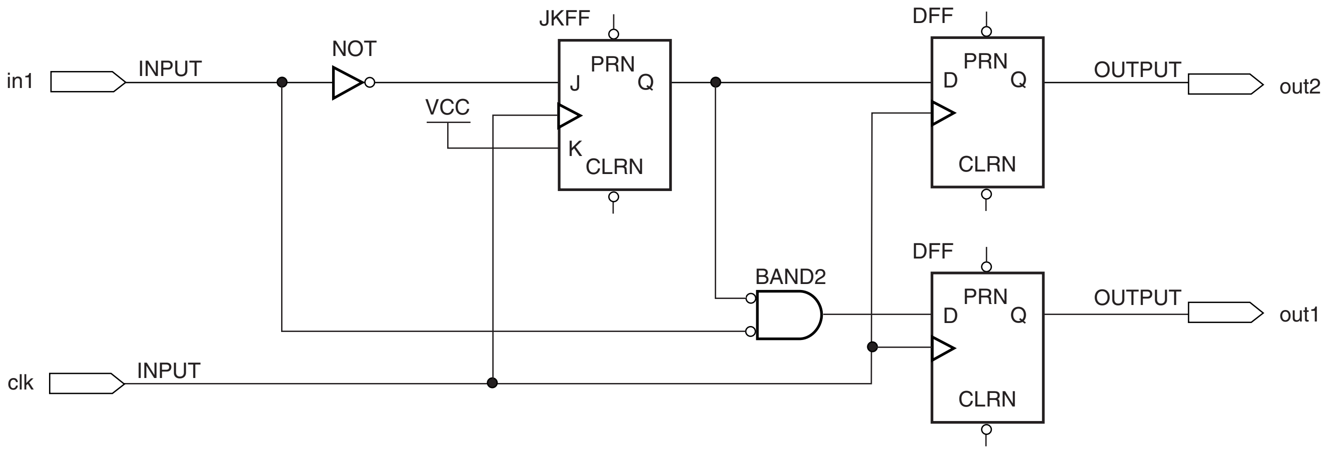


FIGURE 10.12
State Machine with Synchronous Outputs

state_x3a.gdf
state_x3a.scf

The state variable is stored as the state of the JK flip-flop. This state is clocked through a D flip-flop to generate **out2** and combined with **in1** to generate **out1** via another flip-flop. The simulation for this circuit, shown in Figure 10.13, indicates that the two outputs are synchronous with the clock, but delayed by one clock cycle after the state change.

VHDL Implementation of State Machines with Control Inputs

The VHDL code for a state machine with one or more control inputs is similar to that for a machine with no control inputs. The machine states are still defined using a CASE statement, but a case representing a conditional transition will contain an IF statement.

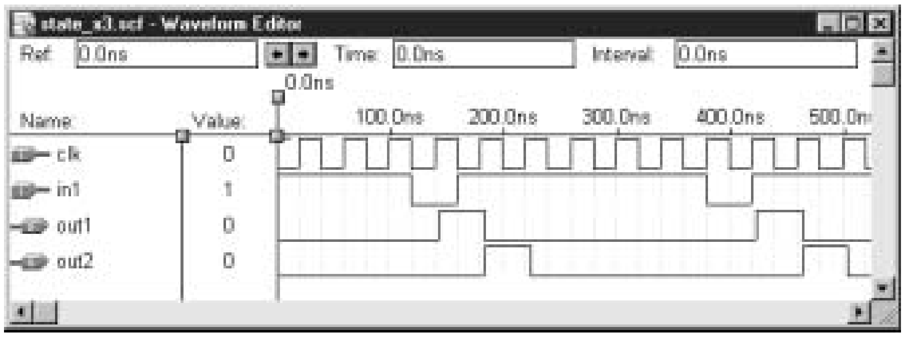


FIGURE 10.13
Simulation of the State Machine of Figure 10.12

The VHDL code for the state machine implemented above is as follows.

state_x1.vhd
state_x1.scf

```
-- state_x1.vhd
-- state machine example 1
-- Two states, one input, two outputs
-- Generates a pulse on one output, then the next
-- after receiving a LOW on the input

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY state_x1 IS
    PORT(
        clk, in1    : IN   STD_LOGIC;
        out1, out2  : OUT  STD_LOGIC);
END state_x1;

ARCHITECTURE a OF state_x1 IS
    TYPE PULSER IS (start, continue);
    SIGNAL sequence: PULSER;
BEGIN
    PROCESS (clk)
    BEGIN
        IF clk'EVENT AND clk = '1' THEN
            CASE sequence IS
                WHEN start =>
                    IF in1 = '1' THEN
                        sequence <= start; -- no change if in1 = 1
                        out1 <= '0';
                        out2 <= '0';
                    ELSE
                         sequence <= continue; -- proceed if in1 = 0
                         out1 <= '1';          -- pulse on out1
                         out2 <= '0';
                    END IF;
                WHEN continue =>
                    sequence <= start;
                    out1 <= '0';
                    out2 <= '1';              -- pulse on out2
            END CASE;
        END IF;
    END PROCESS;
END a;
```

The transition from **start** is conditional, so the case for **start** contains an IF statement that defines the possible state transitions and their associated output states. The transition from **continue** is unconditional, so no IF statement is needed in the corresponding case.

Figure 10.14 shows the simulation for the VHDL design entity, **state_x1.vhd**. The values of the state variable, **sequence,** are also shown in the simulation. This gives us a ready indication of the machine's state (**start** or **continue**).

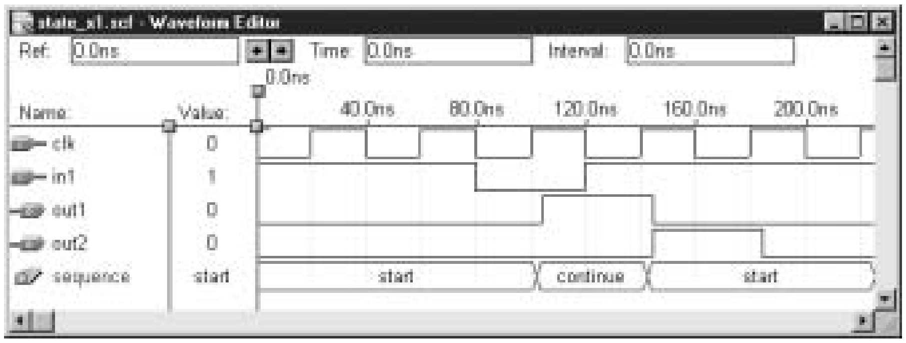


FIGURE 10.14
Simulation of the State Machine in VHDL Entity state_x1

The design of the state machine is such that if the input **in1** is held LOW beyond the end of one pulse cycle, the cycle will repeat, as shown in the simulation of Figure 10.15.

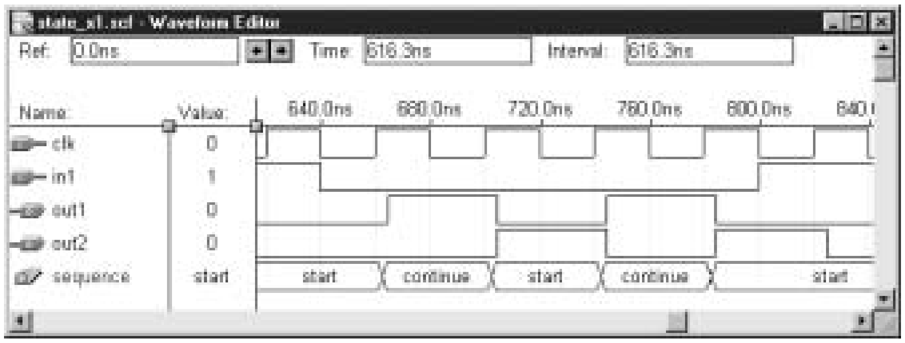


FIGURE 10.15
Simulation of VHDL State Machine Showing a Repeated Output Cycle

EXAMPLE 10.1

Application

A state machine called a single-pulse generator operates as follows:

1. The circuit has two states: **seek** and **find,** an input called **sync** and an output called **pulse**.
2. The state machine resets to the state **seek.** If **sync** = 1, the machine remains in **seek** and the output, **pulse,** remains LOW.
3. When **sync** = 0, the machine makes a transition to **find**. In this transition, **pulse** goes HIGH.
4. When the machine is in state **find** and **sync** = 0, the machine remains in **find** and **pulse** goes LOW.
5. When the machine is in **find** and **sync** = 1, the machine goes back to **seek** and **pulse** remains LOW.

Use classical state machine design techniques to design the circuit for the single-pulse generator, using D flip-flops for the state logic. Use MAX+PLUS II to draw the state machine circuit. Create a simulation to verify the design operation. Briefly describe what this state machine does.

Solution Figure 10.16 shows the state diagram derived from the description of the state machine. The state table is shown in Table 10.5. Since Q follows D, the D input is the same as the next state of Q.

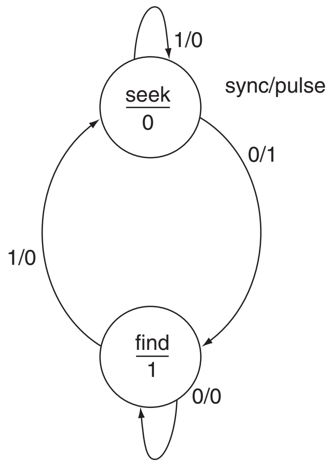


FIGURE 10.16
Example 10.1
State Diagram for a Single-pulse Generator

Table 10.5 State Table for Single-Pulse Generator

Present State	Input	Next State	Sync. Input	Output
Q	*sync*	*Q*	*D*	*pulse*
0	0	1	1	1
0	1	0	0	0
1	0	1	1	0
1	1	0	0	0

pulse1.gdf
pulse1.scf

The next-state and output equations are:

$$D = \overline{Q} \cdot \overline{sync} + Q \cdot \overline{sync} = \overline{sync}$$
$$pulse = \overline{Q} \cdot \overline{sync}$$

Figure 10.17 shows the state machine circuit derived from the above Boolean equations. The simulation for this circuit is shown in Figure 10.18. The simulation shows that the circuit generates one pulse when the input **sync** goes LOW, regardless of the length of time that **sync** is LOW. The circuit could be used in conjunction with a debounced pushbutton to produce exactly one pulse, regardless of how long the pushbutton was held down. Figure 10.19 shows such a circuit.

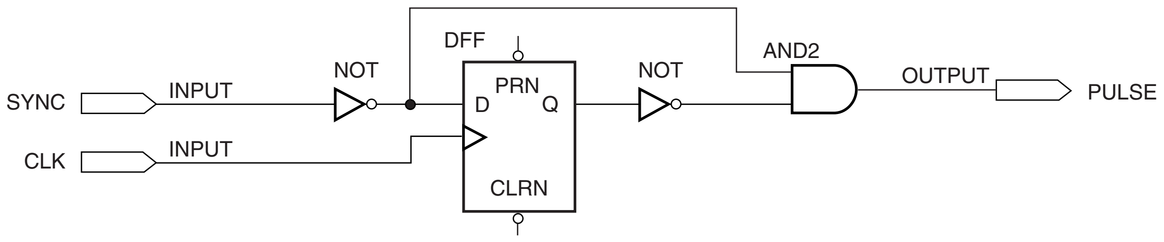


FIGURE 10.17
Example 10.1
Single-pulse Generator

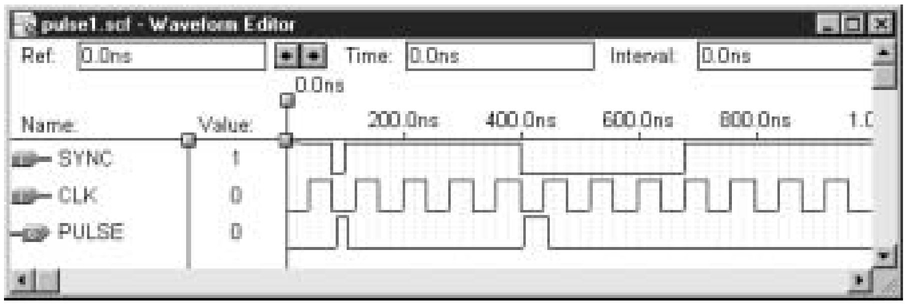


FIGURE 10.18
Example 10.1
Simulation of a Single-pulse Generator (from GDF)

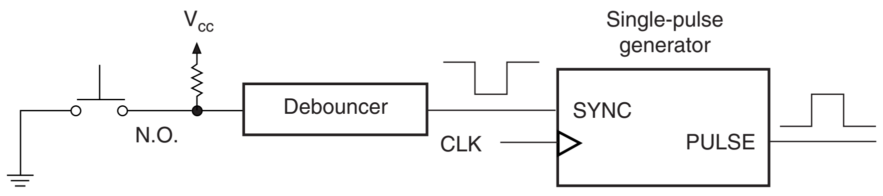


FIGURE 10.19
Example 10.1
Single-pulse Generator Used with a Debounced Pushbutton

EXAMPLE 10.2

The state machine of Example 10.1 is vulnerable to asynchronous input changes. How do we know this from the circuit schematic and from the simulation waveform? Modify the circuit to eliminate the asynchronous behavior and show the effect of the change on a simulation of the design. How does this change improve the design?

Solution The output, **pulse,** in the state machine of Figure 10.17 is derived from the state flip-flop and the combinational logic of the circuit. The output can be affected by a change that is purely combinational, thus making the output asynchronous. This is demonstrated on the first pulse of the simulation in Figure 10.18, where **pulse** momentarily goes HIGH between clock edges. Since no clock edge was present when either the input, **sync,** changed or when **pulse** changed, the output pulse must be due entirely to changes in the combinational part of the circuit.

The circuit output can be synchronized to the clock by adding an output flip-flop, as in Figure 10.20. A simulation of this circuit is shown in Figure 10.21. With the synchronized output, the output pulse is always the same width: one clock period. This gives a more predictable operation of the circuit.

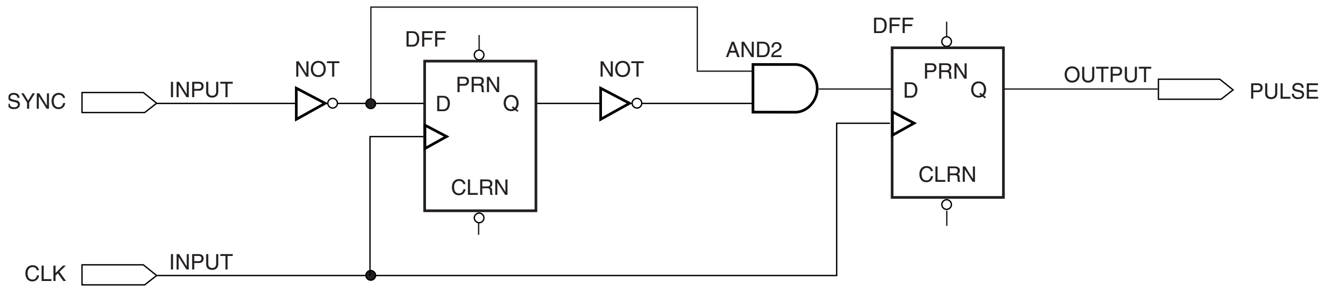


FIGURE 10.20
Example 10.2
Single-pulse Generator with Synchronous Output

pulse1a.gdf
pulse1a.scf

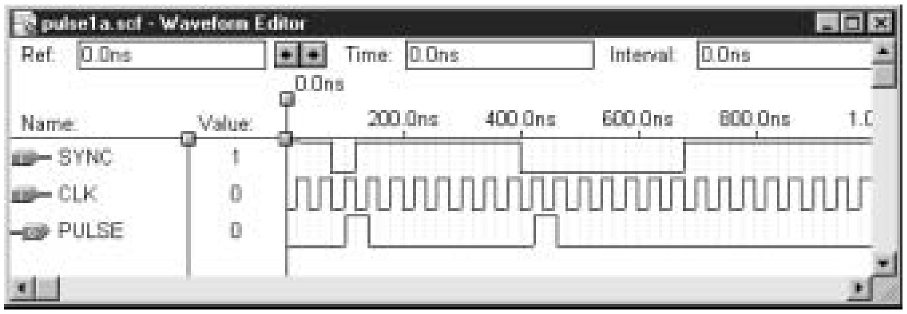


FIGURE 10.21
Example 10.2
Simulation of a Single-pulse Generator with Synchronous Output (from GDF)

EXAMPLE 10.3

Write the VHDL code for a design entity that implements the single-pulse generator, as described in Example 10.1. Create a simulation that verifies the operation of the design.

Solution The required VHDL code is given here in the design entity **sngl_pls**.

sngl_pls.vhd
sngl_pls.scf

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY sngl_pls IS
    PORT(
        clk, sync : IN STD_LOGIC;
        pulse     : OUT STD_LOGIC);
END sngl_pls;

ARCHITECTURE pulser OF sngl_pls IS
    TYPE PULSE_STATE IS (seek, find);
    SIGNAL status: PULSE_STATE;
BEGIN
    PROCESS (clk, sync)
    BEGIN
        IF (clk'EVENT and clk = '1') THEN
            CASE status IS
                WHEN seek =>   IF (sync = '1') THEN
                                    status <= seek;
                                    pulse <= '0';
                               ELSE
                                    status <= find;
                                    pulse <= '1';
                               END IF;
                WHEN find =>   IF (sync = '1') THEN
                                    status <= seek;
                                    pulse <= '0';
                               ELSE
                                    status <= find;
                                    pulse <= '0';
                               END IF;
            END CASE;
        END IF;
    END PROCESS;
END pulser;
```

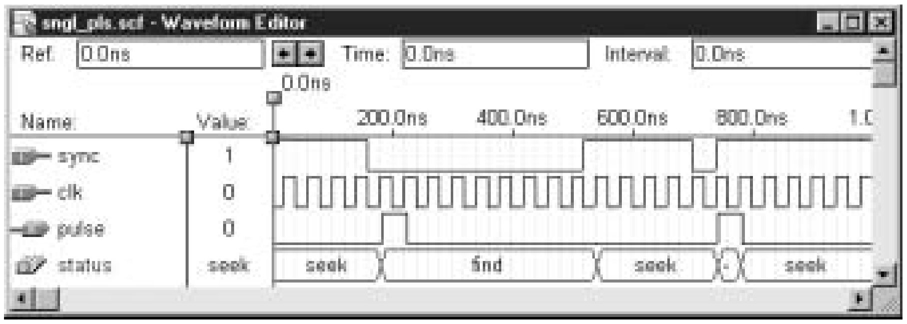


FIGURE 10.22
Example 10.3
Simulation of a Single-pulse Generator (VHDL)

The simulation of the VHDL design entity **sngl_pls** is shown in Figure 10.22

SECTION 10.3 REVIEW PROBLEM

10.3 Briefly explain why the single-pulse circuit in Figure 10.20 has a flip-flop on its output.

10.4 Switch Debouncer for a Normally Open Pushbutton Switch

www.electronictech.com

KEY TERMS

Form A contact	A normally open contact on a switch or relay.
Form B contact	A normally closed contact on a switch or relay.
Form C contact	A pair of contacts, one normally open and one normally closed, that operate with a single action of a switch or relay.

A useful interface function is implemented by a digital circuit that removes the mechanical bounce from a pushbutton switch. The easiest way to debounce a pushbutton switch is with a NAND latch, as shown in Figure 10.23.

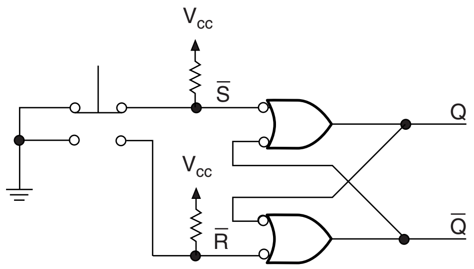


FIGURE 10.23
NAND Latch as a Switch Debouncer

The latch eliminates switch bounce by setting or resetting on the first bounce of a switch contact and ignoring further bounces. The limitation of this circuit is that the input switch must have **Form C contacts**. That is, the switch has normally open, normally closed, and common contacts. This is so that the switch resets the latch when pressed (i.e., when the normally open contact closes) and sets the latch when released (normally closed contact recloses). Each switch position activates an opposite latch function.

If the only available switch has a single set of contacts, such as the normally open **(Form A)** pushbuttons on the Altera UP-1 Education Board, a different debouncer circuit must be used. We will look at two solutions using VHDL: one based on an existing device (the Motorola MC14490 Contact Bounce Eliminator) and another that implements a state machine solution to the contact bounce problem.

Switch Debouncer Based on a 4-bit Shift Register

The circuit in Figure 10.24 is based on the same principle as the Motorola MC14490 Contact Bounce Eliminator, adapted for use in an Altera CPLD, such as the EPM7128S or the EPF10K20 on the Altera UP-1 Education Board.

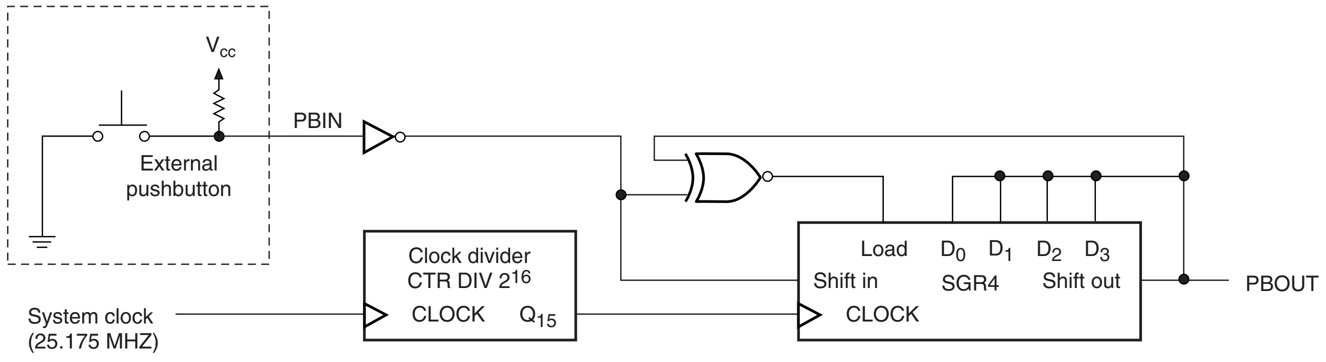


FIGURE 10.24
Switch Debouncer Based on a 4-bit Shift Register

The heart of the debouncer circuit in Figure 10.24 is a 2-bit comparator (an Exclusive NOR gate) and a 4-bit serial shift register, with active-HIGH synchronous LOAD. The XNOR gate compares the shift register serial input and output. When the shift register input and output are *different,* the input data are serially shifted through the register. When input and output of the shift register are *the same,* the binary value at the serial output is parallel-loaded back into all bits of the shift register.

Figure 10.25 shows the timing of the debouncer circuit with switch bounces on both make and break phases of the switch contact. The line labeled **4-bit delay** refers to the shift register flip-flop outputs. Pushbutton input is **pb_in**, debounced output is **pb_out** and **clk** is the UP-1 system clock, divided by 2^{16}. (Time values in Figure 10.25 are not to scale and should be disregarded.)

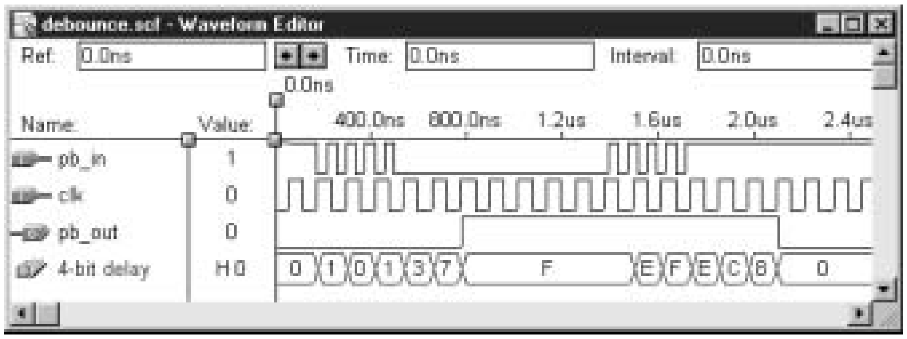

debounce.scf - Waveform Editor
Ref: 0.0ns
Time: 0.0ns
Interval: 0.0ns
0.0ns
Name:
Value:
400.0ns 800.0ns 1.2us 1.6us 2.0us 2.4us
pb_in 1
clk 0
pb_out 0
4-bit delay H0
0 1 0 1 3 7 F E F E C 8 0

FIGURE 10.25
Simulation of the Shift Register-Based Debouncer

Assume the shift register is initially filled with 0s. The pushbutton rest state is HIGH. As shown in Figure 10.24, the pushbutton input value is inverted and applied to the shift register input. Therefore, before the switch is pressed, both input and output of the shift register are LOW. Since they are the same, the XNOR output is HIGH, which keeps the shift register in *LOAD* mode and the LOW at **pb_out** is reloaded to the register on every positive clock edge.

When the switch is pressed, it will bounce, as shown above the second, third, and fourth clock pulses on Figure 10.25. Just before the second clock pulse, **pb_in** is LOW. This makes the shift register input and output different, so a 1 is shifted in. (Recall that **pb_in** is at the opposite logic level to the shift register input.) On the next clock pulse, **pb_in** has bounced HIGH again. The shift register input and output are now the same, so the output value, 0, is loaded in parallel to all flip-flops of the shift register. On the fifth pulse, **pb_in** is stable at logic LOW. Since the shift register input is now HIGH and the output is LOW, the HIGH is shifted through the register. We see this by **4-bit delay** increasing in value: 0, 1, 3, 7, F, which in binary is equivalent to 0000, 0001, 0011, 0111, 1111. At this point, the input and output are now the same and the output value, 1, is parallel-loaded into the register on each clock pulse.

A similar process occurs when the waveform goes back to the HIGH state. When the input goes HIGH, a LOW is shifted into the shift register. If the input bounces back LOW, the shift register is parallel-loaded with HIGHs and the process starts over. When **pb_in** is stable at a HIGH level, a LOW is shifted through the register, resulting in the hexadecimal sequence F, E, C, 8, 0, which is equivalent to the binary values 1111, 1110, 1100, 1000, 0000.

To produce an output change, the shift register input and output must remain different for at least four clock pulses. This implies that the input is stable for that period of time. If the input and output are the same, this could mean one of two things. Either the input is stable and the shift register flip-flops should be kept at a constant state or the input has bounced back to its previous level and the shift register should be reinitialized. In either case, the output value should be parallel loaded back into the shift register. Serial shifting should only occur if there has been an input change.

The debouncer in Figure 10.24 is effective for removing bounce that lasts for no more than 4 clock periods. Since switch bounce is typically about 10 ms in duration, the clock should have a period of about 2.5 ms. At 25.175 MHz (a clock period of about 40 ns), the Altera UP-1 system clock is much too fast.

If we divide the oscillator frequency by 65536 ($= 2^{16}$) using a 16-bit counter, we obtain a clock waveform for the debouncer with a period of 2.6 ms. Four clock periods (10.2 ms) are sufficient to take care of switch bounce.

We can use VHDL to synthesize the switch debouncer by instantiating a counter and shift register from the Altera Library of Parameterized Modules and connecting them together with internal signals. The VHDL code is as follows.

debounce.vhd
debounce.scf

```
-- debounce.vhd
-- Switch Debouncer for a Form A contact, based on a 4-bit shift
-- register.  Function is similar to a Motorola MC14490 Contact
-- Bounce Eliminator.

-- Use modules from Library of Parameterized Modules (LPM):
--     LPM_SHIFTREG  (Shift Register)
--     LPM_COUNTER   (16-bit counter)

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
LIBRARY lpm;
USE lpm.lpm_components.ALL;
```

```
ENTITY debounce IS
    PORT(
        clk : IN STD_LOGIC;
        pb_in : IN STD_LOGIC;
        pb_out : OUT STD_LOGIC);
END debounce;

ARCHITECTURE debouncer OF debounce IS
-- Internal signals required to interconnect counter and shift
register
   SIGNAL srg_ser_out, srg_ser_in, srg_clk, srg_load : STD_LOGIC;
   SIGNAL srg_data    : STD_LOGIC_VECTOR(3 DOWNTO 0);
   SIGNAL ctr_q       : STD_LOGIC_VECTOR (15 DOWNTO 0);
BEGIN
-- Instantiate 16-bit counter
    clock_divider: lpm_counter
       GENERIC MAP (LPM_WIDTH    => 16)
       PORT MAP (clock      =>   clk,
                 q          =>   ctr_q(15 DOWNTO 0));

-- Instantiate 4-bit shift register
    four_bit_delay: lpm_shiftreg
        GENERIC MAP (LPM_WIDTH     => 4)
        PORT MAP (shiftin      => srg_ser_in,
                  clock        => srg_clk,
                  load         => srg_load,
                  data         => srg_data(3 downto 0),
                  shiftout     => srg_ser_out);

-- Shift register is clocked by counter output
-- (divides system clock by 2^16)
    srg_clk   <= ctr_q(15);

-- Undebounced pushbutton input to shift register
   srg_ser_in <=  not pb_in;

-- Shift register is parallel-loaded with output data if
-- shift register input and output are the same.
-- If input and output are different,
-- data are serial-shifted.
   srg_data(3)    <= srg_ser_out;
   srg_data(2)    <= srg_ser_out;
   srg_data(1)    <= srg_ser_out;
   srg_data(0)    <= srg_ser_out;
   pb_out         <= srg_ser_out;
   srg_load       <= not((not pb_in) xor srg_ser_out);
END debouncer;
```

Figure 10.26 shows a fairly easy way to test the switch debouncer. The debouncer output is used to clock an 8-bit counter whose outputs are decoded by two seven-segment decoders. (The decoders are VHDL files developed in a similar way to the seven-segment decoders in Chapter 5.)

Pin numbers are given for the EPM7128S CPLD on the Altera UP-1 circuit board. Since the clock and seven segment displays are hardwired on the Altera board, the only external connections required for the circuit are wires for the two pushbutton inputs, **reset** and **pb_in.**

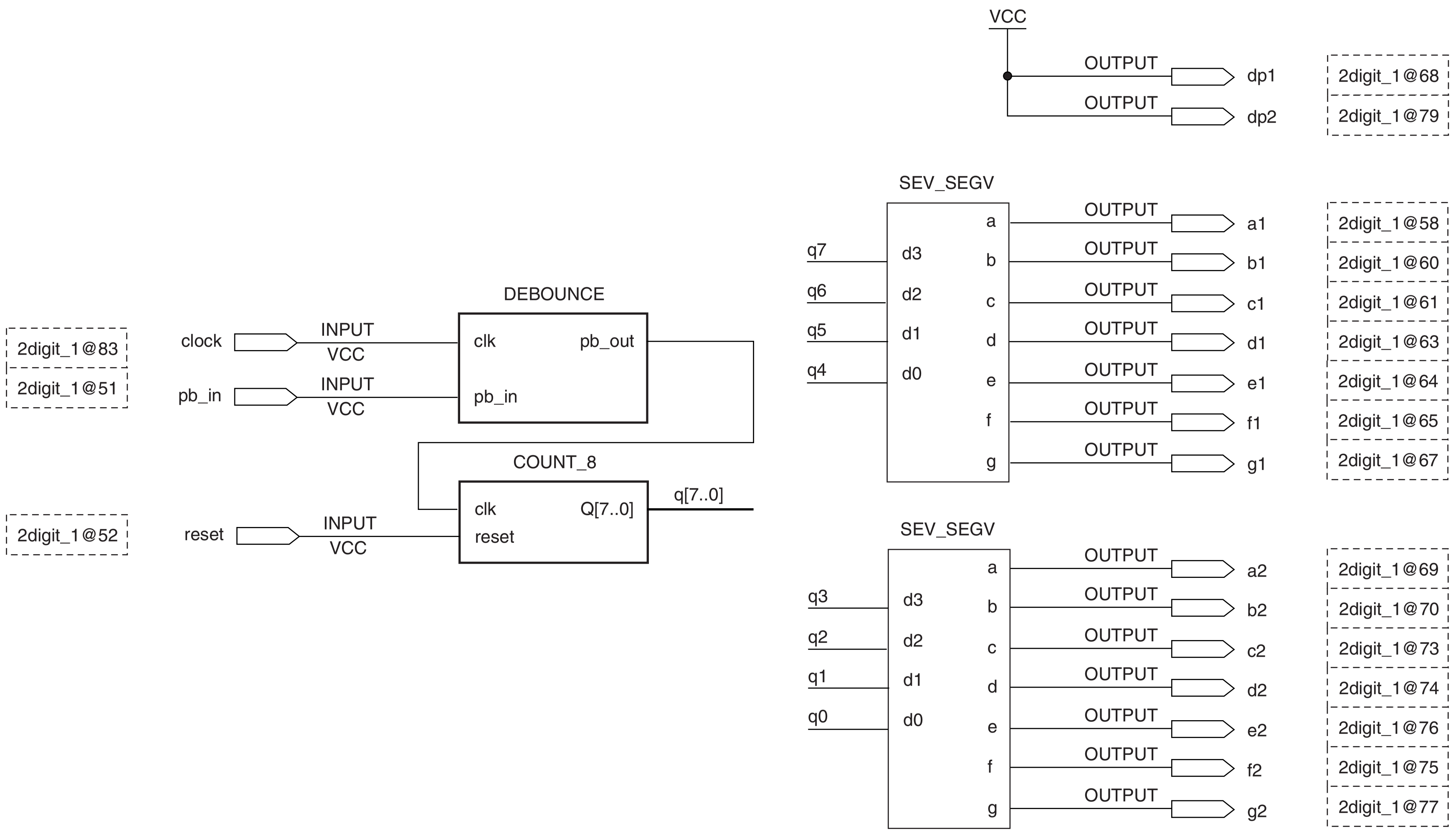


FIGURE 10.26
Test Circuit for a Switch Debouncer

If the debouncer is working properly, the seven-segment display should advance by one each time **pb_in** is pressed. If the debouncer is not working, the display will change by an unpredictable number with each switch press.

2digit.gdf
count_8.vhd
sev_segv.vhd

The component source files for the debouncer and test circuit components are supplied on the CD accompanying this book in the folder ***drive:*\Student Files\Chapter 10\**. To use these files, create a symbol for each one (**File** menu; **Project; Set Project to Current File**; then **File** menu; **Create Default Symbol)** and draw the Graphic Design File of Figure 10.26.

Alternatively, you can instantiate each file as a component in a VHDL design entity (all components are designed in VHDL) and connect them together with internal signals.

Behaviorally Designed Switch Debouncer

We can also design a switch debouncer by using a behavioral state machine description in VHDL. In order to do so, we need to define the operation of the circuit with a state diagram, as in Figure 10.27.

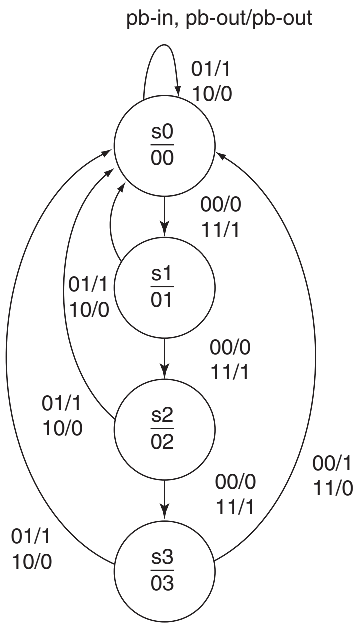


FIGURE 10.27
State Diagram for a Behaviorally Designed Switch Debouncer

Transitions between states are determined by comparing **pb_in** and **pb_out**. If they are the same (00 or 11), the machine advances to the next state; if they are different (01 or 10), the machine reverts to the initial state, s0. At any point in the state diagram (including state s3, the last state), the machine will reset if **pb_in** and **pb_out** are different, indicating a bounce on the input.

If **pb_in** and **pb_out** are the same for four clock pulses, the input is deemed to be stable. Only at this point will the output change to its opposite state.

NOTE

In the shift register–based debouncer, the circuit advanced to the next state if the shift register input and output were different and reset if they were the same. This might appear to be opposite to our behavioral description, but it is not if you look carefully. The shift register debouncer circuit inverts **pb_in** before applying the signal to the serial input of the shift register. Therefore, viewed from the circuit input and output terminals, rather than at the shift register input and output, the description is the same in both cases.

The VHDL code corresponding to the behavioral description of the switch debouncer is given next. The only output change is specified on the transition from state s3 to s0 when **pb_in** = **pb_out**. Since no change is allowed at any other time, no other output state needs to be specified.

dbc_behv.vhd
dbc_behv.scf

```
-- dbc_behv.vhd
-- Behavioral definition of a switch debouncer
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY dbc_behv IS
    PORT(
        clk, pb_in : IN STD_LOGIC;
        pb_out     : BUFFER STD_LOGIC);
END dbc_behv;

ARCHITECTURE debounce of dbc_behv IS
    TYPE sequence IS (s0, s1, s2, s3);
    SIGNAL state: sequence;
BEGIN
    PROCESS (clk, pb_in)
    BEGIN
        IF (clk‘EVENT and clk=‘1’) THEN
            CASE state IS
                WHEN s0=> IF (pb_in = pb_out) THEN
                                state  <= s1;
                            ELSE
                                state  <= s0;
                            END IF;
                WHEN s1=> IF (pb_in = pb_out) THEN
                                state  <= s2;
                            ELSE
                                state  <= s0;
                            END IF;
                WHEN S2=> IF (pb_in = pb_out) THEN
                                state  <= s3;
                            ELSE
                                state  <= s0;
                            END IF;
                WHEN s3=> IF (pb_in = pb_out) THEN
                                state  <= s0;
                                pb_out <= not pb_out;
                            ELSE
                                state  <= s0;
                            END IF;
                WHEN others => state  <= s0;
            END CASE;
        END IF;
    END PROCESS;
END debounce;
```

Figure 10.28 shows a simulation of the behaviorally-designed switch debouncer. State s1 through s3 are of too short a duration to show properly on the simulation, so further details of the simulation are shown in Figures 10.29 and 10.30.

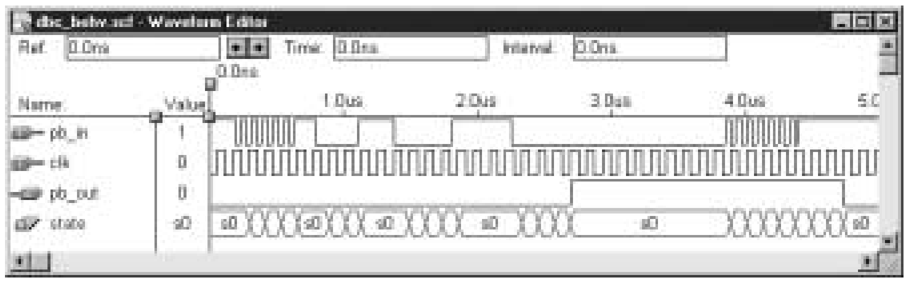


FIGURE 10.28
Simulation of a Behaviorally Designed Switch Debouncer

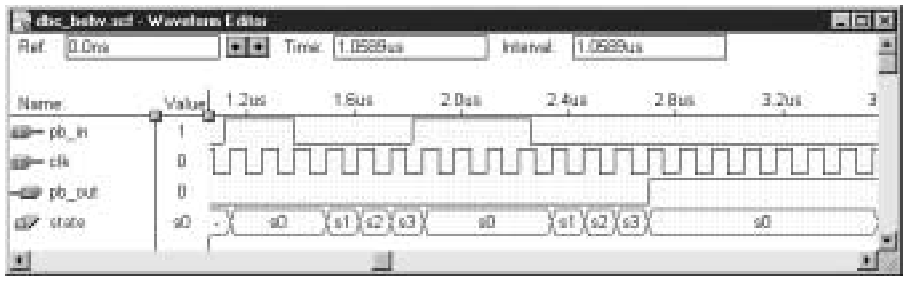


FIGURE 10.29
Simulation Detail (Behaviorally Designed Switch Debouncer)

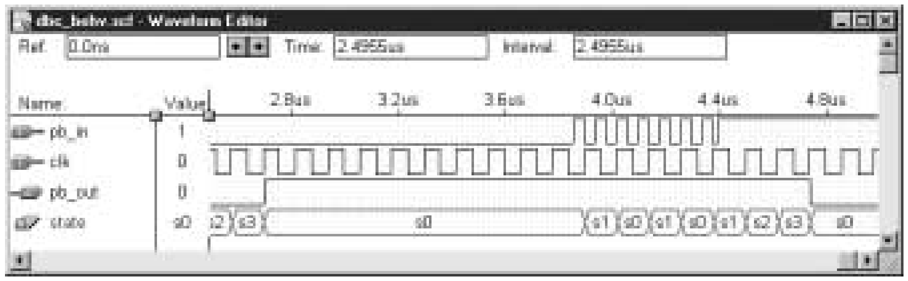


FIGURE 10.30
Simulation Detail (Behaviorally Designed Switch Debouncer)

Note that the behaviorally designed switch debouncer does not have a built-in clock divider. If we were to use the circuit on the Altera UP-1 board, we would need to include a divide-by-2^{16} counter to the circuit, as shown in Figure 10.31.

SECTION 10.4 REVIEW PROBLEM

10.4 What is the fastest acceptable clock rate for the shift register portion of the debouncer in Figure 10.24 if the pushbutton switch bounces for 15ms?

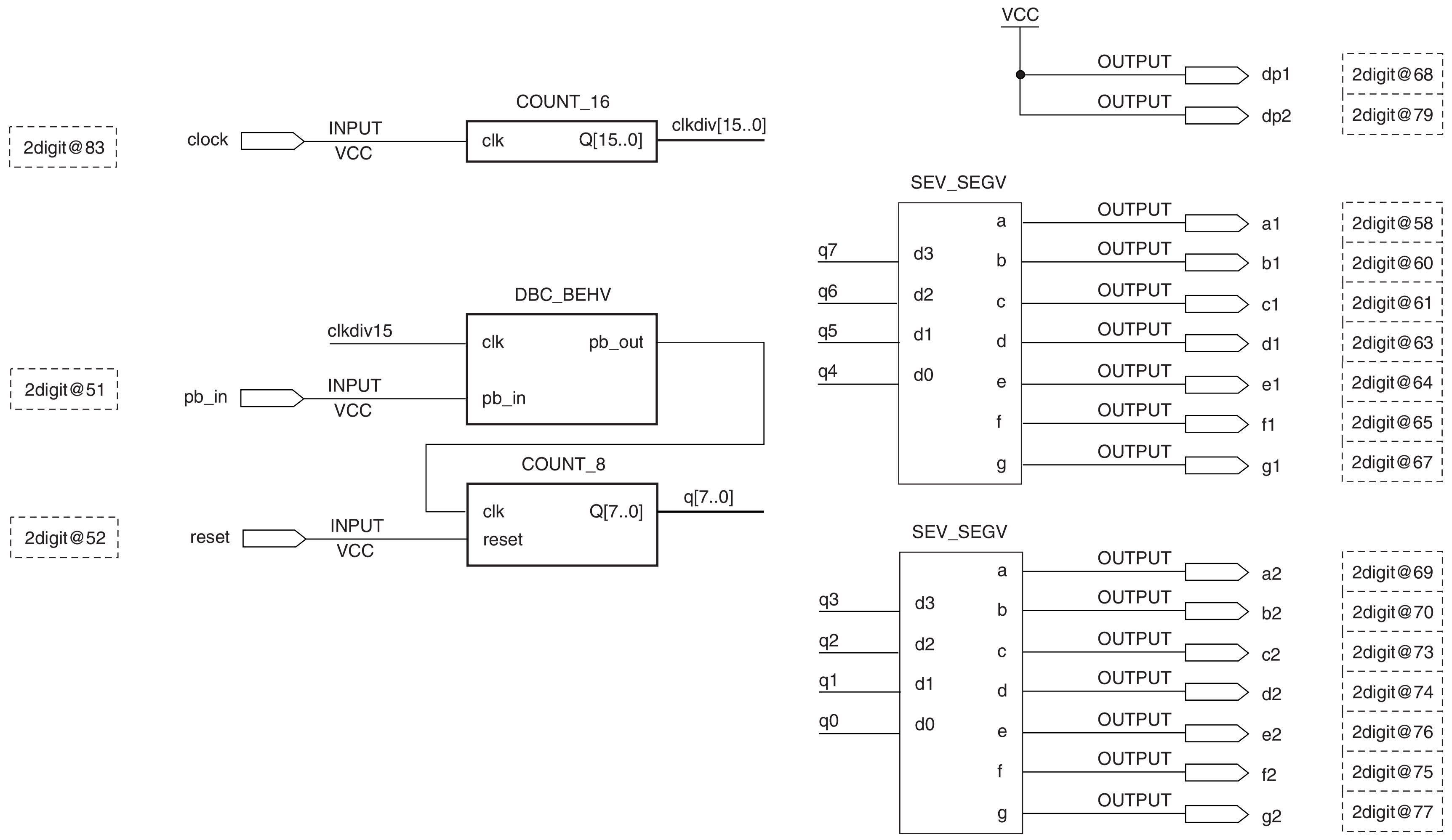


FIGURE 10.31
Using a Behaviorally Designed Debouncer with a 16-bit Clock Divider

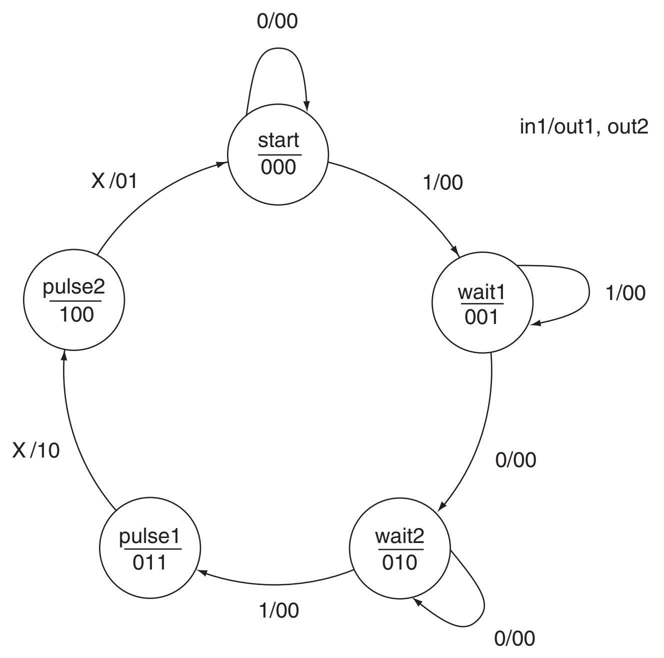


FIGURE 10.32
State Diagram for a Two-pulse Generator

10.5 Unused States in State Machines

In our study of counter circuits in Chapter 9, we found that when a counter modulus is not equal to a power of two there were unused states in the counter's sequence. For example, a mod-10 counter has six unused states, as the counter requires four bits to express ten states and the maximum number of 4-bit states is sixteen. The unused states (1010, 1011, 1100, 1101, 1110, and 1111) have to be accounted for in the design of a mod-10 counter.

The same is true of state machines whose number of states does not equal a power of two. For instance, a machine with five states requires three state variables. There are up to eight states available in a machine with three state variables, leaving three unused states. Figure 10.32 shows the state diagram of such a machine.

Unused states can be dealt with in two ways: they can be treated as don't care states, or they can be assigned specific destinations in the state diagram. In the latter case, the safest destination is the first state, in this case the state called **start.**

EXAMPLE 10.4

Redraw the state diagram of Figure 10.32 to include the unused states of the machine's state variables. Set the unused states to have a destination state of **start.** Briefly describe the intended operation of the state machine.

Solution Figure 10.33 shows the revised state diagram.

The machine begins in state **start** and waits for a HIGH on **in1**. The machine then makes a transition to **wait1** and stays there until **in1** goes LOW again. The machine goes to **wait2** and stays there until **in1** goes HIGH and then makes an unconditional transition to **pulse1** on the next clock pulse. Until this point, there is no change in either output.

The machine makes an unconditional transition to **pulse2** and makes **out1** go HIGH. The next transition, also unconditional, is to **start**, when **out1** goes LOW and **out2** goes HIGH. If **in1** is LOW, the machine stays in **start**. Otherwise, the cycle continues as above. In either case, **out2** goes LOW again.

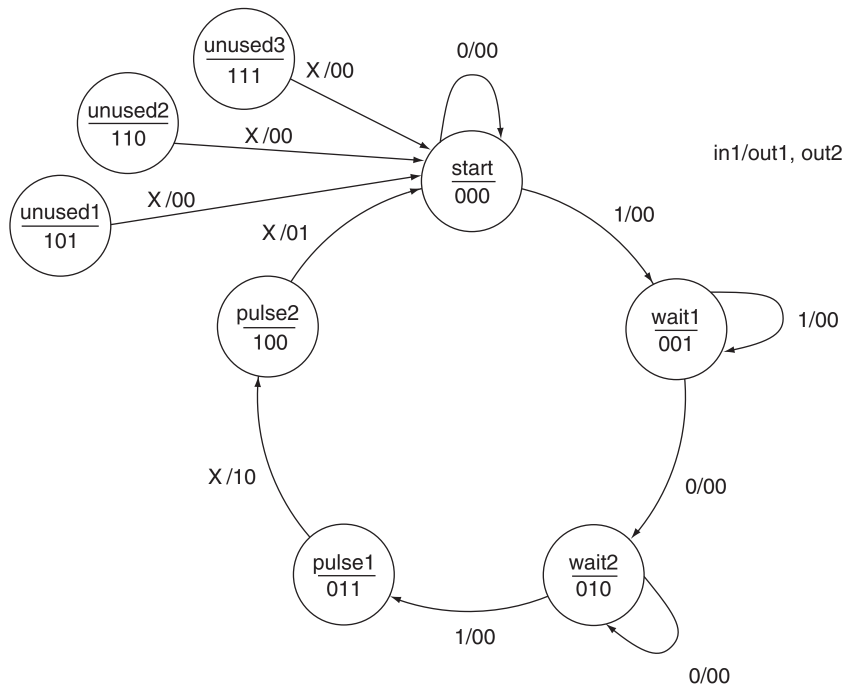


FIGURE 10.33
Example 10.4
State Diagram for Two-pulse Generator Showing Unused States

Thus the machine waits for a HIGH-LOW-HIGH input sequence and generates a pulse sequence on two outputs.

EXAMPLE 10.5

Use classical state machine design techniques to implement the state machine described in the modified state diagram of Figure 10.33. Draw the state machine as a Graphic Design File in Max+PLUS II and create a simulation to verify its function.

Solution Table 10.6 shows the state table of the state machine represented by Figure 10.33.

Table 10.6 State Table for State Machine of Figure 10.33

Present State	Input	Next State	Outputs	
$Q_2Q_1Q_0$	*in1*	$Q_2Q_1Q_0$	*out1*	*out2*
000	0	000	0	0
000	1	001	0	0
001	0	010	0	0
001	1	001	0	0
010	0	010	0	0
010	1	011	0	0
011	0	100	1	0
011	1	100	1	0
100	0	000	0	1
100	1	000	0	1
101	0	000	0	0
101	1	000	0	0
110	0	000	0	0
110	1	000	0	0
111	0	000	0	0
111	1	000	0	0

Figure 10.34 shows the Karnaugh maps used to simplify the next-state equations for the state variable flip-flops. The output equations can be simplified by inspection.

The next-state and output equations for the state machine are:

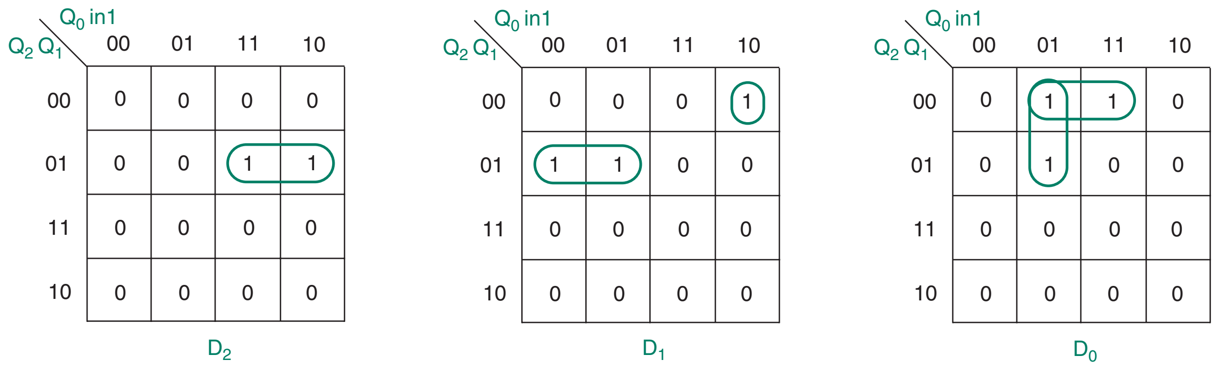


FIGURE 10.34
Example 10.5
K-Maps for Two-pulse Generator

$$
\begin{aligned}
D_2 &= \overline{Q_2}Q_1Q_0 \\
D_1 &= \overline{Q_2}Q_1\overline{Q_0} + \overline{Q_2}\,\overline{Q_1}Q_0\overline{in1} \\
D_0 &= \overline{Q_2}\,\overline{Q_0}in1 + \overline{Q_2}\,\overline{Q_1}in1 \\
out1 &= \overline{Q_2}Q_1Q_0 \\
out2 &= Q_2\overline{Q_1}\,\overline{Q_0}
\end{aligned}
$$

Figure 10.35 shows the Graphic Design File schematic for the state machine. Figure 10.36 shows the MAX+PLUS II simulation waveforms.

We can monitor the state variables in the MAX+PLUS II simulation file by adding a group of waveforms for the buried nodes q2, q1, and q0. These are shown on the simulation as q[2..0].Q, meaning the *Q* outputs of the flip-flops named q2, q1, q0.

To add the buried nodes, select **Enter Node from SNF** from the **Node** menu in the simulator window. In the dialog box shown in Figure 10.37, check the box that says **All**, and click on **List**. Select the nodes q2.Q, q1.Q, and q0.Q from the **Available Nodes and Groups** and transfer them to the **Selected Nodes and Groups**. Click**OK**. Select the three new waveforms and from the **Node** menu, select **Group**. Click **OK** in the resulting dialog box.

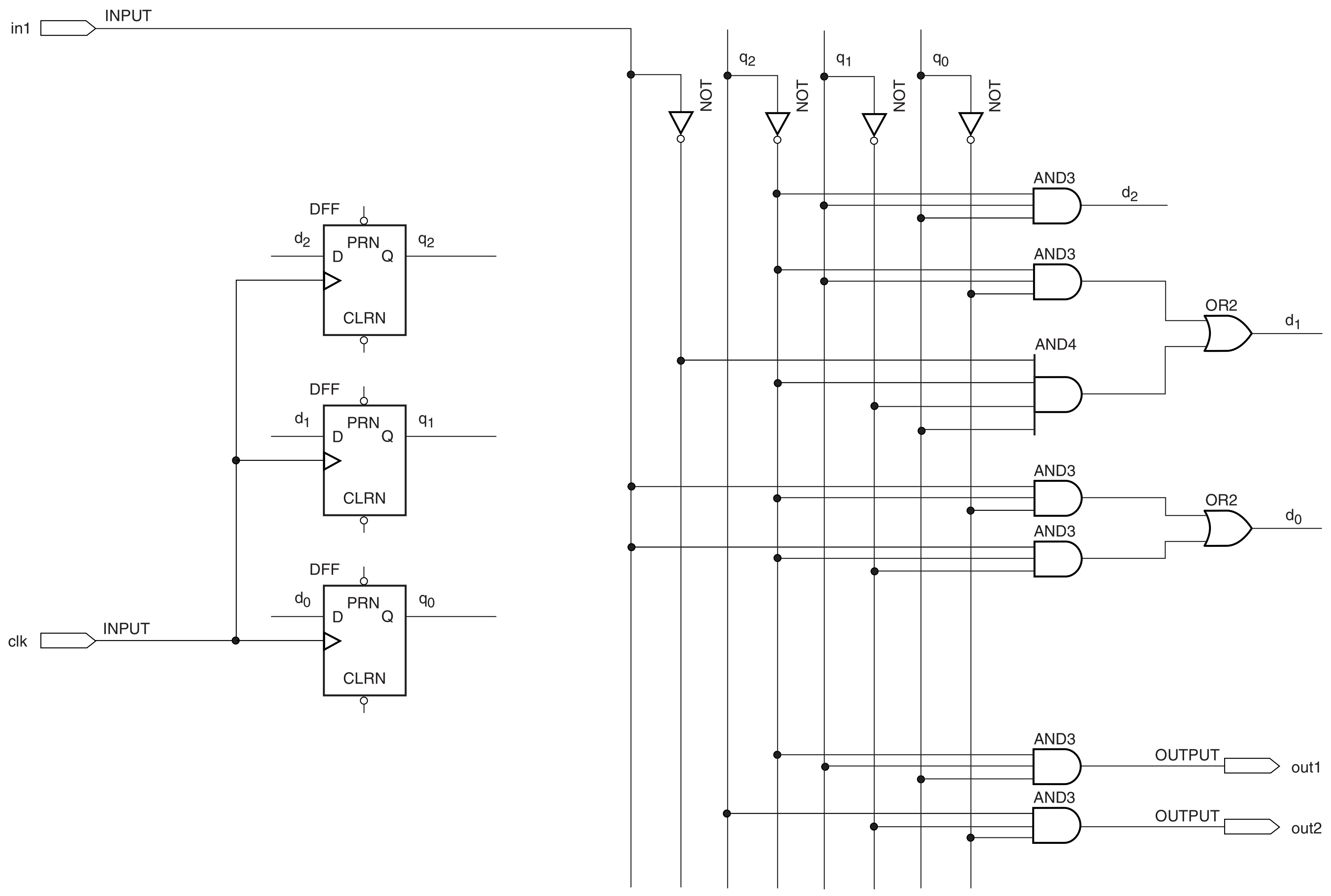


FIGURE 10.35
Example 10.5
Two-pulse Generator

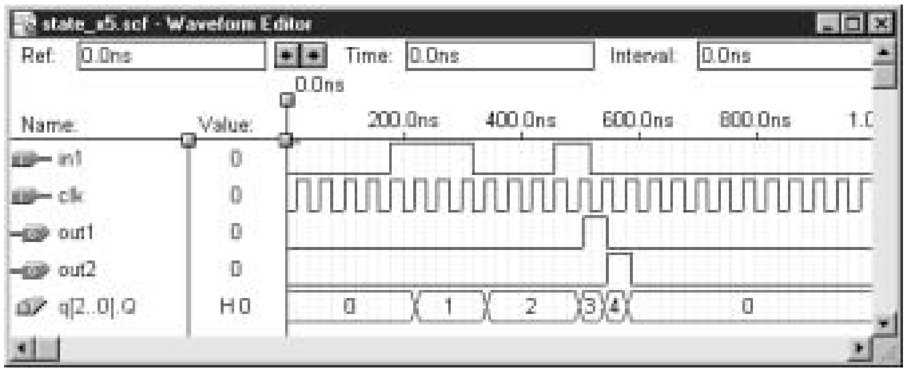


FIGURE 10.36
Example 10.5
Simulation of a Two-pulse Generator (GDF)

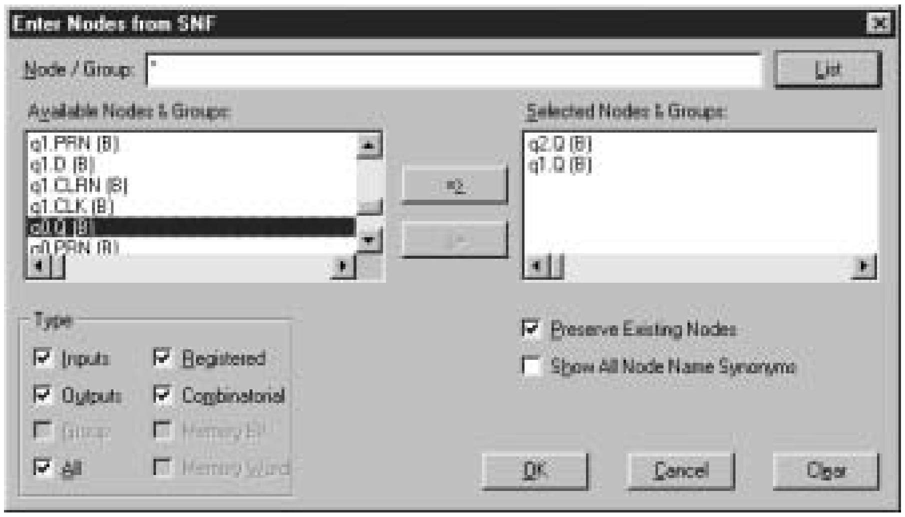


FIGURE 10.37
Adding Buried Nodes to a Simulation

EXAMPLE 10.6 Write the VHDL code required to implement the two-pulse generator described in Examples 10.4 and 10.5. Create a MAX+PLUS II simulation to verify the operation of the design. Based on your examination of the simulations for the VHDL design and the GDF design of the previous example, how do the two designs differ in their operation? What is the reason for the difference?

Solution The VHDL code for the state machine in design entity **two_pulse.vhd** follows. The unused states are accounted for in the **others** clause.

```
-- two_pulse.vhd
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY two_pulse IS
   PORT(
      clk, in1   : IN    STD_LOGIC;
      output     : OUT   STD_LOGIC_VECTOR (1 to 2));
END two_pulse;

ARCHITECTURE a OF two_pulse IS
   TYPE SEQUENCE IS (start, wait1, wait2, pulse1, pulse2);
   SIGNAL pulse_state : SEQUENCE;
BEGIN
   PROCESS(clk)
   BEGIN
      IF (clk'EVENT and clk = '1') THEN
         CASE pulse_state IS
```

two_pulse.vhd
two_pulse.scf
two_pulse.rpt

```
                    WHEN start =>
                       IF in1 = '0' THEN
                          pulse_state    <= start;
                          output     <= "00";
                       ELSIF in1 = '1' THEN
                          pulse_state <= wait1;
                          output     <= "00";
                       END IF;
                    WHEN wait1 =>
                       IF in1 = '0' THEN
                          pulse_state    <= wait2;
                          output     <= "00";
                       ELSIF in1 = '1' THEN
                          pulse_state <= wait1;
                          output     <= "00";
                       END IF;
                    WHEN wait2 =>
                       IF in1 = '0' THEN
                          pulse_state    <= wait2;
                          output     <= "00";
                       ELSIF in1 = '1' THEN
                          pulse_state <= pulse1;
                          output     <= "00";
                       END IF;
                    WHEN pulse1 =>
                       pulse_state   <= pulse2;
                       output     <= "10";
                    WHEN pulse2 =>
                       pulse_state   <= start;
                       output     <= "01";
                    WHEN others =>
                       pulse_state   <= start;
                       output     <= "00";
                 END CASE;
              END IF;
           END PROCESS;
        END a;
```

Figure 10.38 shows the MAX+PLUS II simulation of the state machine.

If you closely examine the simulation waveforms in Figures 10.36 and 10.38, you will note that the pulse outputs in Figure 10.38 (VHDL design) occur one clock cycle later than they do in Figure 10.36 (graphical design). This is because the VHDL compiler has synthesized each output with a D flip-flop, as we did for the single-pulse circuit in Figure

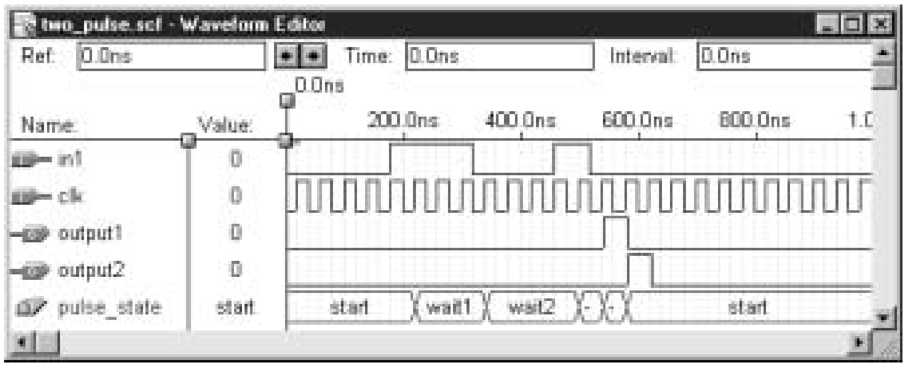


FIGURE 10.38
Example 10.6
Simulation of a Two-pulse Generator (VHDL)

10.20, in order to ensure synchronous output operation.(We can verify this by examining the EQUATIONS section of the project report file, **two_pulse.rpt**.) Since the outputs are both derived entirely from flip-flop outputs, this synthesis step is not strictly necessary to ensure that the outputs are synchronous with the clock.

SECTION 10.5 REVIEW PROBLEM

10.5 Is the state machine designed in Example 10.5 a Moore machine or a Mealy machine? Why?

10.6 Traffic Light Controller

A simple traffic light controller can be implemented by a state machine with a state diagram such as the one shown in Figure 10.39.

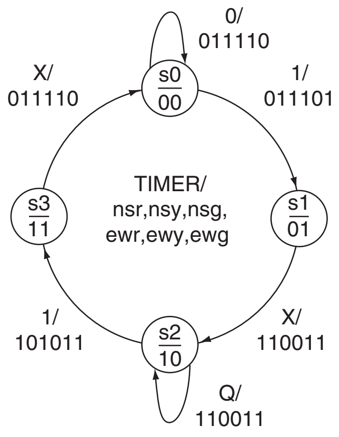


FIGURE 10.39
State Diagram of a Traffic Light Controller

The control scheme assumes control over a north-south road and an east-west road. The north-south lights are controlled by outputs called **nsr, nsy,** and **nsg** (north-south red, yellow, green). The east-west road is controlled by similar outputs called **ewr, ewy,** and **ewg.** A LOW controller output turns on a light. Thus an output 011110 corresponds to the north-south red and east-west green lights.

An input called *TIMER* controls the length of the two green-light cycles. When *TIMER* = 1, a transition from s0 to s1 or from s2 to s3 is possible (s0 represents the EW green; s2 the NS green). This transition accompanies a change from green to yellow on the active road. The light on the other road stays red. An unconditional transition follows, changing the yellow light to red on one road and the red light to green on the other.

The cycle can be set to any length by changing the signal on the *TIMER* input. (The yellow light will always be on for one clock pulse in this design.) For ease of observation, we will use a cycle of ten clock pulses. For either direction, the cycle consists of 4 clocks GREEN, 1 clock YELLOW, 5 clocks RED. This cycle can be generated by the MSB of a mod-5 counter, as shown in Figure 10.40. If we model the traffic controller using the Altera UP-1 board, we require a clock divider to slow down the 25.175 MHz clock to a rate of about 0.75 Hz, making it easy to observe the changes of lights. These blocks can all be instantiated in VHDL, which will be left as part of an exercise in the lab manual accompanying this book.

FIGURE 10.40
Traffic Control Demonstration Circuit for the Altera UP-1 Board

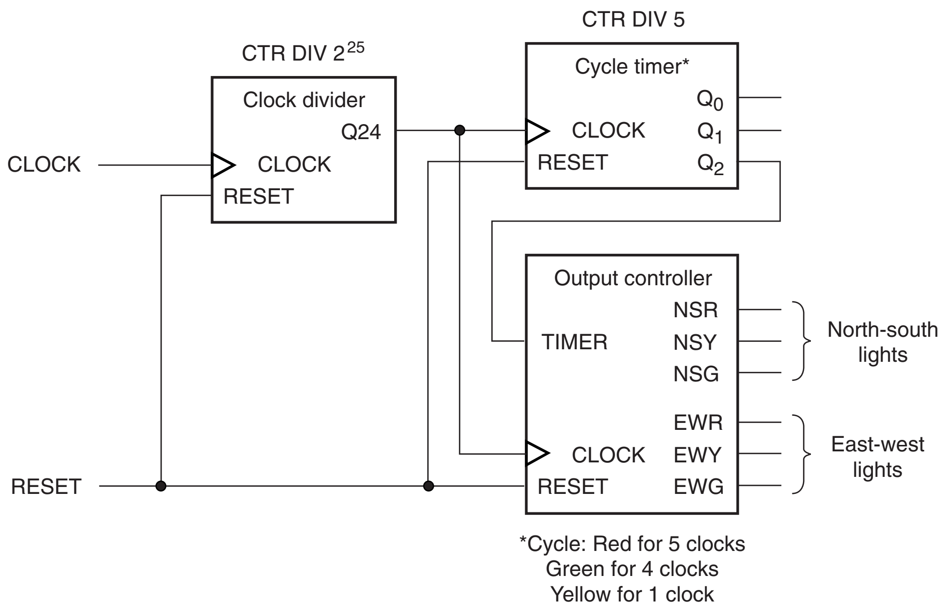


Figure 10.41 shows the simulation of the mod-5 counter that generates the *TIMER* control signal. The MSB goes HIGH for one clock period, then LOW for four. When applied to the *TIMER* input of the output controller, this signal directs the controller from state to state.

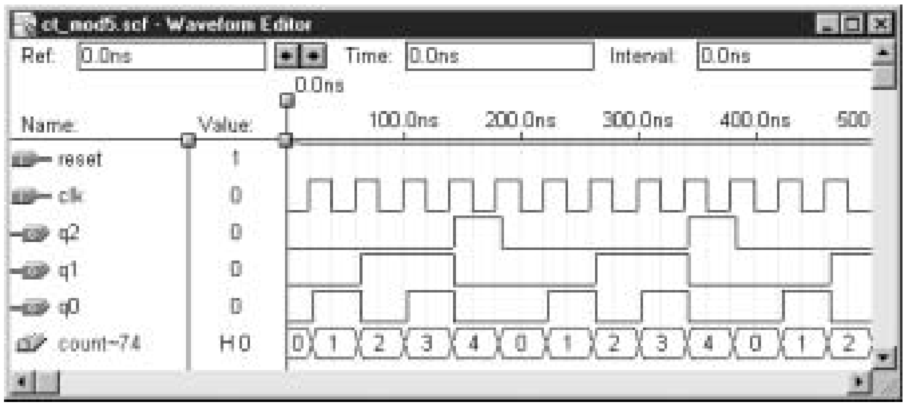


FIGURE 10.41
Simulation of a Mod-5 Counter

Figure 10.42 shows a simulation of the mod-5 counter and output controller. The north-south lights are red for five clock pulses (shown by 011 in the **north_south** waveform). At the same time, the east-west lights are green for four clock pulses (**east_west** = 110), followed by yellow for one clock pulse (**east_west** = 101). The cycle continues with an east-west red and north-south green and yellow.

According to the state diagram, the yellow light should happen on the transition where *TIMER* = 1. This corresponds to the point on the simulation waveforms where **count** = 4.

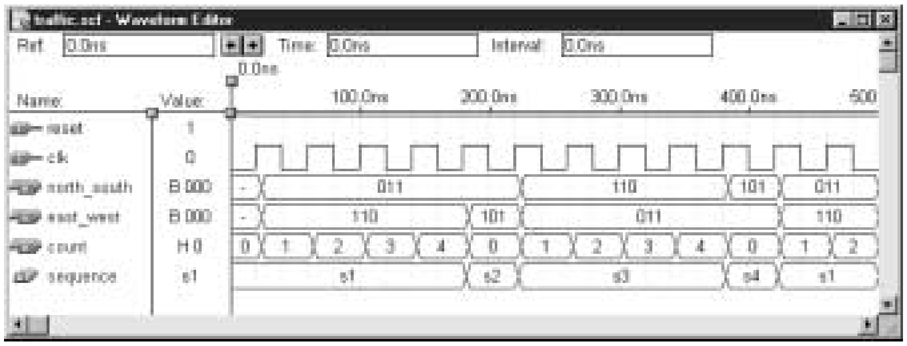


FIGURE 10.42
Simulation of a Traffic Light Controller

However, the yellow light does not come on until **count** = 0. This is because the MAX+PLUS II VHDL compiler synthesizes the controller outputs with synchronous outputs (flip-flops). As a result, the output states are delayed by one clock cycle. Since the relative lengths of the cycle proportions are preserved, this does not affect the operation of the controller.

SUMMARY

1. A state machine is a synchronous sequential circuit with a memory section (flip-flops) to hold the present state of the machine and a control section (gates) to determine the machine's next state.
2. The number of flip-flops in a state machine's memory section is the same as the number of state variables.
3. Two main types of state machine are the Moore machine and the Mealy machine.
4. The outputs of a Moore machine are entirely dependent on the states of the machine's flip-flops. Output changes will always be synchronous with the system clock.
5. The outputs of a Mealy machine depend on the states of the machine's flip-flops and the gates in the control section. A Mealy machine's outputs can change asynchronously, relative to the system clock.
6. A state machine can be designed in a classical fashion using the same method as in designing a synchronous counter, as follows:
    a. Define the problem and draw a state diagram.
    b. Construct a table of present and next states.
    c. Use flip-flop excitation tables to determine the flip-flop inputs for each state transition.
    d. Use Boolean algebra or K-maps to find the simplest Boolean expression for flip-flop inputs *(D, T,* or *JK)* in terms of outputs *(Q)*.
    e. Draw the logic diagram of the state machine.
7. The state names in a state machine can be named numerically (s0, s1, s2, . . .) or literally (start, idle, read, write), depending on the machine function. State names are independent of the values of the state variables.
8. A state machine can be defined in VHDL by using a CASE statement within a PROCESS to define the progression of states. The output values can be defined by a separate decoder construct or they can be assigned within each case of the CASE statement.
9. The possible values of the state variables of a machine are defined within an enumerated type definition. An enumerated type is a list of possible values that a port, variable, or signal of that type is allowed to have.
10. Notation for a state diagram includes a series of bubbles (circles) containing state names and values of state variables in the form $\dfrac{\textbf{state\_name}}{\textbf{state\_variable(s)}}$.
11. The inputs and outputs of a state machine are labeled **in1, in2, . . . , in*x*/out1, out2, . . . ,out*x***.
12. Transitions between states can be conditional or unconditional. A conditional transition happens only under certain conditions of a control input and is labeled with the relevant input condition. An unconditional transition happens under all conditions of input and is labeled with an X for each input variable.
13. Conditional transitions in a VHDL state machine are described by an IF statement within a particular case of the CASE statement that describes the machine.
14. Mealy machine outputs are susceptible to asynchronous output changes if a combinational input changes out of synchronization with the clock. This can be remedied by clocking each output through a separate synchronizing flip-flop.
15. A maximum of 2^n states can be assigned to a state machine that has n state variables. If the number of states is less than 2^n, the unused states must be accounted for. Either they can be treated as don't care states, or they can be assigned a specific destination state, usually the reset state.
16. In a VHDL implementation of a state machine, any unused states can be covered with an **others** clause in the CASE statement that defines the machine.

GLOSSARY

Conditional transition A transition between states of a state machine that occurs only under specific conditions of one or more control inputs.

Control input A state machine input that directs the operation of the machine from state to state.

Enumerated type A user-defined type in VHDL in which all possible values of a named identifier are listed in a type definition statement.

Form A contact A normally open contact on a switch or relay.

Form B contact A normally closed contact on a switch or relay.

Form C contact A pair of contacts, one normally open and one normally closed, that operate with a single action of a switch or relay.

Mealy machine A state machine whose output is determined by both the sequential logic and the combinational logic of the machine.

Moore machine A state machine whose output is determined only by the sequential logic of the machine.

State machine A synchronous sequential circuit, consisting of a sequential logic section and a combinational logic section, whose outputs and internal flip-flops progress through a predictable sequence of states in response to a clock and other input signals.

State variables The variables held in the flip-flops of a state machine that determine its present state.

Unconditional transition A transition between states of a state machine that occurs regardless of the status of any control inputs.

PROBLEMS

Problem numbers set in color indicate more difficult problems: those with underlines indicate most difficult problems.

Section 10.1 State Machines

10.1 Is the state machine in Figure 10.43 a Moore machine or a Mealy machine? Explain your answer.

10.2 Is the state machine in Figure 10.44 a Moore machine or a Mealy machine? Explain your answer.

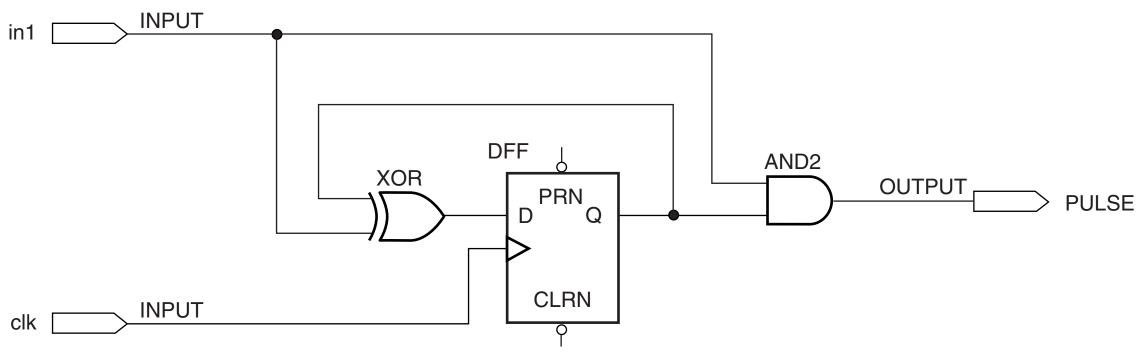


FIGURE 10.43
Problem 10.1
State Machine Circuit

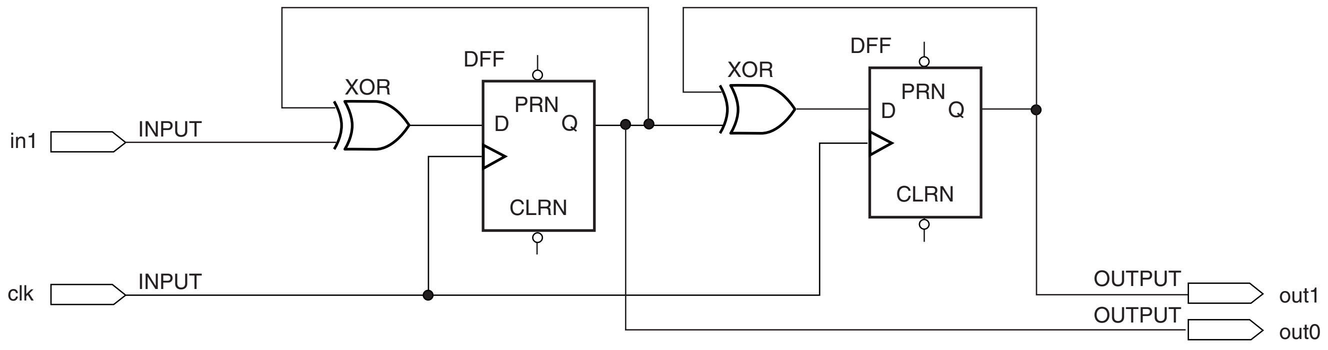


FIGURE 10.44
Problem 10.2
State Machine Circuit

Section 10.2 State Machines with No Control Inputs

10.3 A 4-bit Gray code sequence is shown in Table 10.7. Use classical design methods to design a counter with this sequence, using D flip-flops. Draw the resulting circuit diagram in a MAX+PLUS II Graphic Design File. Create a simulation to verify the circuit operation.

Table 10.7 4-bit Gray code sequence

$Q_3Q_2Q_1Q_0$
0000
0001
0011
0010
0110
0111
0101
0100
1100
1101
1111
1110
1010
1011
1001
1000

10.4 Use classical state machine design techniques to design a counter whose output sequence is shown in Table 10.8. (This is a divide-by-twelve counter in which the MSB output has a duty cycle of 50%.) Draw the state diagram, derive synchronous equations of the flip-flops, and draw the circuit implementation in MAX+PLUS II and create a simulation to verify the circuit's function.

Table 10.8 Counter Sequence for Problem 10.4

$Q_3Q_2Q_1Q_0$
0000
0001
0010
0011
0100
0101
1000
1001
1010
1011
1100
1101

10.5 Write the VHDL code required to implement a 4-bit Gray code counter. Create a simulation in MAX+PLUS II to verify the operation of the circuit.

10.6 Write the VHDL code required to implement a counter with the sequence shown in Table 10.8. Create a simulation in MAX+PLUS II to verify the operation of the circuit.

Section 10.3 State Machines with Control Inputs

10.7 Use classical state machine design techniques to find the Boolean next state and output equations for the state machine represented by the state diagram in Figure 10.45. Draw the state machine circuit as a Graphic Design File in MAX+PLUS II. Create a simulation file to verify the operation of the circuit. Briefly explain the intended function of the state machine.

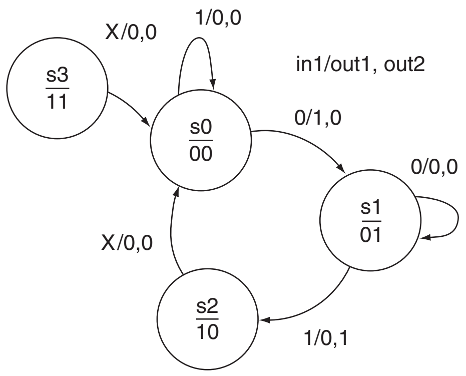


FIGURE 10.45
Problem 10.7
State Diagram

10.8 Referring to the simulation for the state machine in Problem 10.7, briefly explain why it is susceptible to asynchronous input changes. Modify the state machine circuit to eliminate the asynchronous behavior of the outputs. Create a MAX+PLUS II simulation to verify the function of the modified state machine.

10.9 Write the VHDL code required to implement the state machine in Problem 10.7. Create a simulation to verify the operation of the state machine.

10.10 A state machine is used to control an analog-to-digital converter, as shown in the block diagram of Figure 10.46.

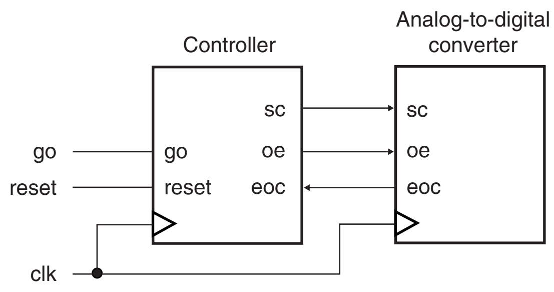


FIGURE 10.46
Problem 10.10
Analog-to-Digital Converter and Controller

The controller has four states, defined by state variables Q_1 and Q_0 as follows: **idle** (00), **start** (01), **waiting** (11), and **read** (10). There are two outputs: **sc** (Start Conversion; active-HIGH) and **oe** (Output Enable; active LOW). There are four inputs: clock, **go** (active-LOW) **eoc** (End of Conversion), and asynchronous reset (active LOW). The machine operates as follows:

a. In the **idle** state, the outputs are: sc = 0, oe = 1. The machine defaults to the **idle** state when the machine is reset.

b. Upon detecting a 0 at the **go** input, the machine makes a transition to the **start** state. In this transition, sc = 1, oe = 1.

c. The machine makes an unconditional transition to the **waiting** state; sc = 0, oe = 1. It remains in this state, with no output change, until input eoc = 1.

d. When eoc = 1, the machine goes to the **read** state; sc = 0, oe = 0.

e. The machine makes an unconditional transition to the **idle** state; sc = 0, oe = 1.

Use classical state machine design techniques to design the controller. Draw the required circuit in MAX+PLUS II and create a simulation to verify its operation. Is this machine vulnerable to asynchronous input change?

10.11 Use VHDL to implement the controller circuit of Problem 10.10. Create a simulation to verify its operation.

10.12 Write a VHDL file for a state machine that selects a 3-bit binary or Gray code count, depending on the state of an input called **gray.** If **gray** = **1,** count in Gray code. Otherwise count in binary. Create a simulation file that verifies the operation of the circuit, clearly showing the full Gray code count, binary count, and reset function.

Section 10.4 Switch Debouncer for a Normally Open Pushbutton Switch

10.13 Why is it not possible to debounce the pushbuttons on the Altera UP-1 board using a NAND latch?

10.14 Refer to the switch debouncer circuit in Figure 10.24 (p. 476). For how many clock periods must the input of the debouncer remain stable before the output can change?

10.15 What is the maximum switch bounce time that can be removed by the circuit of Figure 10.24 if the clock at the shift register is running at a rate of 480 Hz?

10.16 Briefly explain how the Exclusive NOR gate in the debounce circuit of Figure 10.24 determines if switch bounce has occurred.

10.17 Refer to the section on the behaviorally designed switch debouncer in Section 10.4. For how many clock periods must the input of the debouncer remain stable before the output can change? What is the maximum switch bounce time that can be removed by the circuit of Figure 10.24. if the state machine clock is running at a rate of 480 Hz?

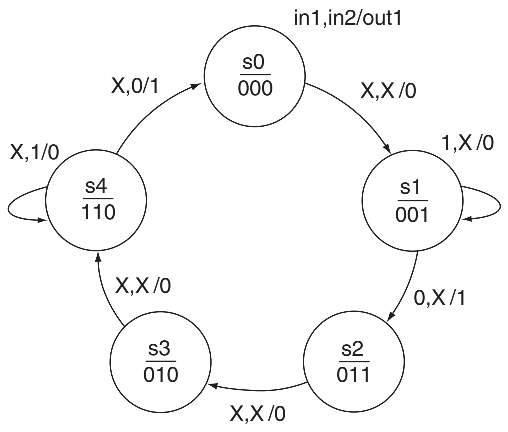


FIGURE 10.47
Problem 10.18
State Diagram

Section 10.5 Unused States in State Machines

10.18 Refer to the state diagram in Figure 10.47.

a. How many state variables are required to implement this state machine? Why?

b. How many unused states are there for this state machine? List the unused states.

c. Complete the partial timing diagram shown in Figure 10.48 to illustrate one complete cycle of the state machine represented by the state diagram of Figure 10.47.

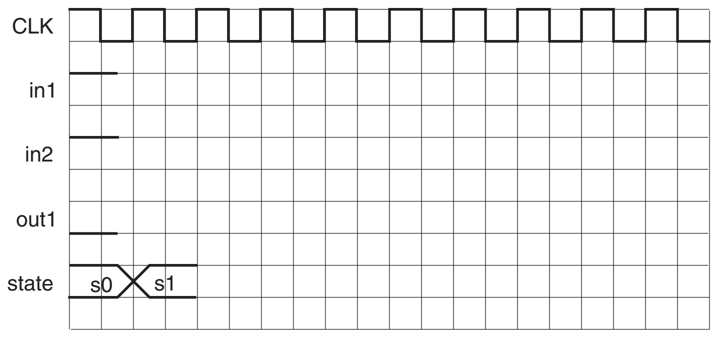


FIGURE 10.48
Problem 10.18
Partial Timing Diagram

10.19 Write the VHDL code required to implement the state machine described by the state diagram of Figure 10.47. Create a simulation file to verify the operation of the circuit.

10.20 Use classical state machine design techniques to design a state machine described by the state diagram of Figure 10.49. Briefly describe the intended operation of the circuit. Create a MAX+PLUS II simulation to verify the operation of the state machine design. Unused states may be treated as don't care states, but unspecified outputs should always be assigned to 0.

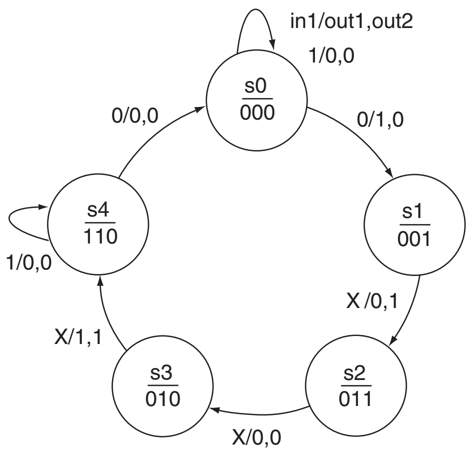


FIGURE 10.49
Problem 10.20
State Diagram

10.21 Determine the next state for each of the unused states of the state machine designed in Problem 10.20. Use this analysis to redraw the state diagram of Figure 10.49 so that it properly includes the unused states. (There is more than one right answer, depending on the result of the Boolean simplification process used in Problem 10.20.)

10.22 Write the VHDL code for the state machine described in Problem 10.20. Create a MAX+PLUS II simulation to verify the function of the state machine.

10.23 A state machine is used to control an analog-to-digital converter, as shown in the block diagram of Figure 10.46. (The following description is a modified version of the controller described in Problem 10.10.)

Five states are used: **idle, start, waiting1, waiting2,** and **read.** There are two outputs: **sc** (Start Conversion; active-HIGH) and **oe** (Output Enable; active HIGH). There are four inputs: clock, reset, **go,** and **eoc** (End of Conversion). The machine operates as follows:

a. In the **idle** state, the outputs are: sc = 0, oe = 0. The machine defaults to the **idle** state when asynchronously reset and remains there until go = 0.

b. When go = 0, the machine makes a transition to the **start** state. In this transition, sc = 1, oe = 0.

c. The machine makes an unconditional transition to the **waiting1** state; sc = 0, oe = 0. It remains in this state, with no output change, until input eoc = 0.

d. When eoc = 0, the machine goes to the **waiting2** state; sc = 0, oe = 0. It remains in this state, with no output change, until input eoc = 1.

e. The machine makes a transition to the **read** state when eoc = 1, sc = 0, oe = 1.

f. The machine makes an unconditional transition to the **idle** state; sc =, 0, oe = 0.

After reviewing the block diagram and the states just listed,

a. Draw the state diagram of the controller.

b. How many state variables are required for the controller described in this question?

10.24 Write the VHDL code for the state machine described in Problem 10.23. Create a simulation file to verify the function of the design.

ANSWERS TO SECTION REVIEW PROBLEMS

Section 10.1

10.1 A Moore state machine has outputs that depend only on the states of the flip-flops in the machine. A Mealy machine's outputs depend on the states of its flip-flops as well as the gates of the machine's control section. This can result in asynchronous output changes in the Mealy machine outputs.

Section 10.2

10.2

$$J_2 = Q_1\overline{Q_0}$$
$$K_2 = \overline{Q_1}\,\overline{Q_0}$$
$$J_1 = \overline{Q_2}Q_0$$
$$K_1 = Q_2Q_0$$
$$J_0 = \overline{Q_2}\,\overline{Q_1} + Q_2Q_1 = \overline{Q_2 \oplus \mathrm{Q}_1}$$
$$K_0 = \overline{Q_2}Q_1 + Q_2\overline{Q_1} = Q_2 \oplus Q_1$$

Section 10.3

10.3 The output flip-flop synchronizes the output to the system clock, yielding the following advantages: (1) the output is always a known width of one clock cycle; and (2) the output is not vulnerable to change due to asynchronous changes of input.

Section 10.4

10.4 $T_c = 3.75$ ms; $f_c = 267$ Hz

Section 10.5

10.5 Moore machine. The outputs are derived entirely from the output states of the state machine and are not vulnerable to asynchronous changes of input.

CHAPTER 11

Logic Gate Circuitry

OUTLINE

CHAPTER OBJECTIVES

Upon successful completion of this chapter, you will be able to:

- Name the various logic families most commonly in use today and state several advantages and disadvantages of each.
- Define propagation delay.
- Calculate propagation delay of simple circuits, using data sheets.
- Define fanout and calculate its value, using data sheets.
- Calculate power dissipation of TTL and CMOS circuits.
- Calculate noise margin of a logic gate from data sheets.
- Draw circuits that will interface various CMOS and TTL gates.
- Explain how a bipolar junction transistor can be used as a logic inverter.
- Describe the function of a TTL input transistor in all possible input states: HIGH, LOW, and open-circuit.
- Explain the operation of a totem pole output.
- Illustrate how a totem pole output generates power line noise and describe how to remedy this problem.
- Illustrate why totem pole outputs cannot be tied together.
- Explain the difference between open-collector and totem pole outputs of a TTL gate.
- Illustrate the operation of TTL open-collector inverter, NAND, and NOR gates.
- Write the Boolean expression of a wired-AND circuit.
- Design a circuit that uses an open-collector gate to drive a high-current load.
- Calculate the value of a pull-up resistor at the output of an open-collector gate.
- Explain the operation of a tristate gate and name several of its advantages.
- Design a circuit using a tristate bus driver to direct the flow of data from one device to another.
- Describe the basic structure of a MOSFET and state its bias voltage requirements.
- Draw the circuit of an CMOS inverter and show how it works.