The Boolean equations for the BCD-to-2421 decoder are:

$$Y_4 = D_4 + D_3D_2 + D_3D_1$$
$$Y_3 = D_4 + D_3D_2 + D_3\overline{D}_1$$
$$Y_2 = D_4 + \overline{D}_3D_2 + D_3\overline{D}_2D_1$$
$$Y_1 = D_1$$

8.5 Repeat Problem 8.4 for a 2421-to-BCD code converter.

8.4 PAL Devices with Registered Outputs

8.6 What is a registered output?

8.7 State the number of registered outputs for each of the following PAL devices:

a. PAL16R4
b. PAL16R6
c. PAL16R8

8.5 Universal PAL and Generic Array Logic (GAL)

8.8 Name two features of a PALCE16V8 that make it superior to a PAL16L8.

8.9 State the difference between a global architecture cell and a local architecture cell in a PALCE16V8.

8.10 How many macrocells are there in a GAL22V10? How many product lines do these macrocells have?

8.11 State the four configurations possible with a macrocell in a GAL22V10.

8.12 Is there a global output enable function available for a PALCE16V8? For a GAL22V10?

8.13 Can the registered outputs of a PALCE16V8 be clocked by a product term function from the PAL AND matrix?

8.14 Can the registered outputs of a GAL22V10 be clocked by a product term function from the GAL AND matrix?

8.15 Are the Asynchronous Reset *(AR)* and Synchronous Preset *(SP)* functions in a GAL22V10 global or local? Explain your answer in one sentence.

8.6 MAX7000S CPLD

8.16 State one way in which a Complex PLD, such as an Altera MAX7000S, differs from a low-density PAL or GAL.

8.17 How many macrocells are available in the following CPLDs:

a. EPM7032
b. EPM7064
c. EPM7128S
d. EPM7160S

8.18 Which of the CPLDs listed in Problem 8.17 are in-system programmable? What does it mean when a device is in-system programmable?

8.19 How many logic array blocks (LABs) are there in an Altera MAX7000S CPLD?

8.20 How many user I/O pins are there in an EPM7128SLC84 CPLD? How many pins per LAB does this represent?

8.21 What can be done with the macrocells in an LAB that do not connect to I/O pins?

8.22 State the possible clock configurations of a MAX7000S macrocell.

8.23 State the possible reset configurations of a MAX7000S macrocell.

8.24 State the possible preset configurations of a MAX7000S macrocell.

8.25 How many dedicated product terms are available in a MAX7000S macrocell? How can this number of product terms be supplemented? What is the maximum number of product terms available to a macrocell?

8.26 How many shared logic expanders are available in an LAB?

8.7 FLEX10K CPLD

8.27 Briefly state the difference between CPLDs having sum-of-products architecture and look-up table architecture.

8.28 How many inputs can a look-up table accept in an Altera FLEX10K logic element? How can this be expanded?

8.29 What is the purpose of the carry chain in a FLEX10K CPLD?

8.30 How many logic elements are there in a FLEX10K LAB?

8.31 How many bits of storage are there in an Embedded Array Block in a FLEX10K CPLD?

CHAPTER 9

Counters and Shift Registers

OUTLINE

CHAPTER OBJECTIVES

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

- Determine the modulus of a counter.
- Determine the number of outputs required by a counter for a given modulus.
- Determine the maximum modulus of a counter, given the number of circuit outputs.
- Draw the count sequence table, state diagram, and timing diagram of a counter.
- Determine the recycle point of a counter's sequence.
- Calculate the frequencies of each counter output, given the input clock frequency.
- Draw a circuit for any full sequence synchronous counter.
- Determine the count sequence, state diagram, timing diagram, and modulus of any synchronous counter.
- Complete the state diagram of a synchronous counter to account for unused states.
- Design the circuit of a truncated sequence synchronous counter, using flip-flops and logic gates.
- Use MAX+PLUS II to create a graphic design file for any synchronous counter circuit.
- Use behavioral descriptions in VHDL to design synchronous counters of any modulus.
- Use a parameterized counter from the Library of Parameterized Modules in a VHDL file.
- Use the MAX+PLUS II simulation tool to verify the operation of synchronous counters.
- Implement various counter control functions, such as parallel load, clear, count enable, and count direction, both in Graphic Design Files and in VHDL.
- Design a circuit to decode the output of the counter, both in a MAX+PLUS II Graphic Design File or in VHDL.
- Draw a logic circuit of a serial shift register and determine its contents over time given any input data.

- Draw a timing diagram showing the operation of a serial shift register.
- Draw the logic circuit of a general parallel-load shift register.
- Draw a timing diagram showing the operation of a parallel-load shift register.
- Draw the general logic circuit of a bidirectional shift register and explain the concepts of right-shift and left-shift.
- Use timing diagrams to explain the operation of a bidirectional shift register.
- Describe the operation of a universal shift register.
- Design shift registers, ring counters, and Johnson counters with the MAX+PLUS II Graphic Editor or VHDL.
- Verify the operation of shift registers, ring counters, and Johnson counters using the MAX+PLUS II simulation tool.
- Design a decoder for a Johnson counter.
- Use a ring counter or a Johnson counter as an event sequencer.
- Compare binary, ring, and Johnson counters in terms of the modulus and the required decoding for each circuit.

Counters and shift registers are two important classes of sequential circuits. In the simplest terms, a counter is a circuit that counts pulses. As such, it is used in many circuit applications, such as event counting and sequencing, timing, frequency division, and control. A basic counter can be enhanced to incorporate functions such as synchronous or asynchronous parallel loading, synchronous or asynchronous clear, count enable, directional control, and output decoding. In this chapter, we will design counters using schematic entry, VHDL, and counters from the Library of Parameterized Modules and verify their operation using the MAX+PLUS II simulator.

Shift registers are circuits that store and move data. They can be used in serial data transfer, serial/parallel conversion, arithmetic functions, and delay elements. As with counters, many shift registers have additional functions such as parallel load, clear, and directional control. We can implement these circuits using schematic entry, VHDL, and LPM components. ■

9.1 Basic Concepts of Digital Counters

KEY TERMS

Counter A sequential digital circuit whose output progresses in a predictable repeating pattern, advancing by one state for each clock pulse.

Recycle To make a transition from the last state of the count sequence to the first state.

Count sequence The specific series of output states through which a counter progresses.

State diagram A diagram showing the progression of states of a sequential circuit.

Modulus The number of states through which a counter sequences before repeating.

Modulo-*n* (or mod-*n*) counter A counter with a modulus of *n*.

UP counter A counter with an ascending sequence.

DOWN counter A counter with a descending sequence.

The simplest definition of a **counter** is "a circuit that counts pulses." Knowing only this, let us look at an example of how we might use a counter circuit.

EXAMPLE 9.1

Figure 9.1 shows a 10-bit binary counter that can be used to count the number of people passing by an optical sensor. Every time the sensor detects a person passing by, it produces a pulse. Briefly describe the counter's operation. What is the maximum number of people it can count? What happens if this number is exceeded?

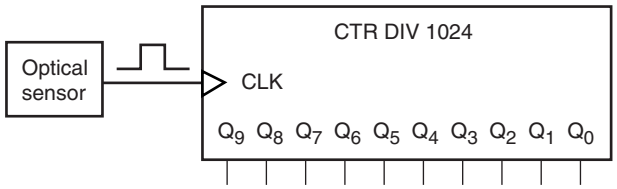


FIGURE 9.1
Example 9.1
10-bit Counter

Solution The counter has a 10-bit output, allowing a binary number from 00 0000 0000 to 11 1111 1111 (0 to 1023) to appear at its output. The sensor causes the counter to advance by one binary number for every pulse applied to the counter's clock *(CLK)* input. If the counter is allowed to register *no people* (i.e., 00 0000 0000), then the circuit can count 1023 people, since there are 1024 unique binary combinations of a 10-bit number, including 0. (This is because $2^{10} = 1024$.) When the 1024^{th} pulse is applied to the clock input, the counter rolls over to 0 (or **recycles**) and starts counting again. (After this point, the counter would not accurately reflect the number of people counted.)

The counter is labeled CTR DIV 1024 to indicate that one full cycle of the counter requires 1024 clock pulses (i.e., the frequency of the MSB output signal (Q_9) is the clock frequency divided by 1024).

A counter is a digital circuit that has a number of binary outputs whose states progress through a fixed sequence. This **count sequence** can be ascending, descending, or nonlinear.

The output sequence of a counter is usually defined by its **modulus,** that is, the number of states through which the counter progresses. An **UP counter** with a modulus of 12 counts through 12 states from 0000 up to 1011 (0 to 11 in decimal), recycles to 0000, and continues. A **DOWN counter** with a modulus of 12 counts from 1011 down to 0000, recycles to 1011, and continues downward. Both types of counter are called **modulo-12,** or just **mod-12** counters, since they both have sequences of 12 states.

State Diagram

The states of a counter can be represented by a **state diagram.** Figure 9.2 compares the state diagram of a mod-12 UP counter to an analog clock face. Each counter state is illustrated in the state diagram by a circle containing its binary value. The progression is shown by a series of directional arrows.

Both the clock face and the state diagram represent a closed system of counting. In each case, when we reach the end of the count sequence, we start over from the beginning of the cycle.

For instance, if it is 10:00 a.m. and we want to meet a friend in four hours, we know we should turn up for the appointment at 2:00 p.m. We arrive at this figure by starting at 10 on the clock face and counting 4 digits forward in a "clockwise" circle. This takes us two digits past 12, the "recycle point" of the clock face.

Similarly, if we want to know the 8th state after 0111 in a mod-12 UP counter, we start at state 0111 and count 8 positions in the direction of the arrows. This brings us to state 0000 (the recycle point) in 5 counts and then on to state 0011 in another 3 counts.

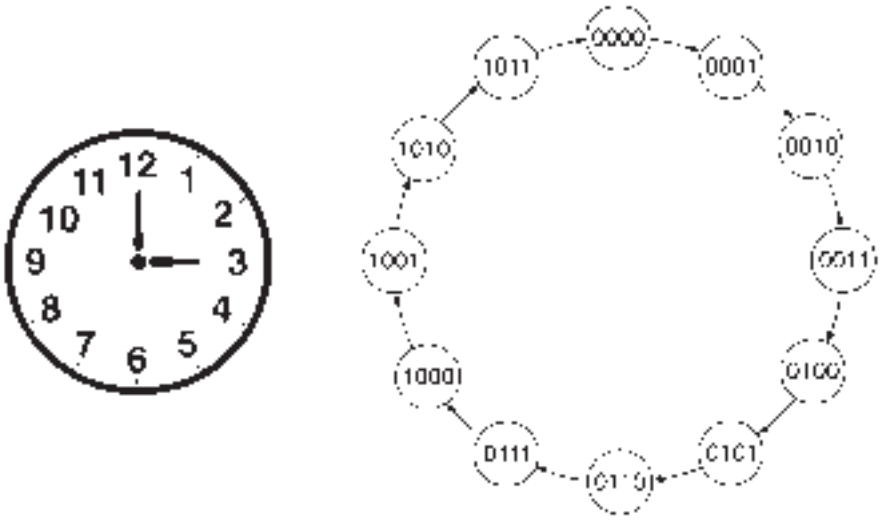


FIGURE 9.2
Mod-12 State Diagram and Analog Clock Face

Number of Bits and Maximum Modulus

KEY TERMS

Maximum modulus (m_{max}) The largest number of counter states that can be represented by n bits ($m_{max} = 2^n$).

Full-sequence counter A counter whose modulus is the same as its maximum modulus ($m = 2^n$ for an n-bit counter).

Binary counter A counter that generates a binary count sequence.

Truncated-sequence counter A counter whose modulus is less than its maximum modulus ($m < 2^n$ for an n-bit counter).

The state diagram of Figure 9.2 represents the states of a mod-12 counter as a series of 4-bit numbers. Counter states are always written with a fixed number of bits, since each bit represents the logic level of a physical location in the counter circuit. A mod-12 counter requires four bits because its highest count value is a 4-bit number: 1011.

The **maximum modulus** of a 4-bit counter is 16 ($= 2^4$). The count sequence of a mod-16 UP counter is from 0000 to 1111 (0 to 15 in decimal), as illustrated in the state diagram of Figure 9.3.

In general, an n-bit counter has a maximum modulus of 2^n and a count sequence from 0 to $2^n - 1$ (i.e., all 0s to all 1s). Since a mod-16 counter has a modulus of 2^n ($= m_{max}$), we say that it is a **full-sequence counter.** We can also call this a **binary counter** if it generates the sequence in binary order. A counter, such as a mod-12 counter, whose modulus is less than 2^n, is called a **truncated sequence counter.**

Count-Sequence Table and Timing Diagram

KEY TERMS

Count-sequence table A list of counter states in the order of the count sequence.

Two ways to represent a count sequence other than a state diagram are by a **count sequence table** and by a timing diagram. The count sequence table is simply a list of counter states in the same order as the count sequence. Tables 9.1 and 9.2 show the count sequence tables of a mod-16 UP counter and a mod-12 UP counter, respectively.

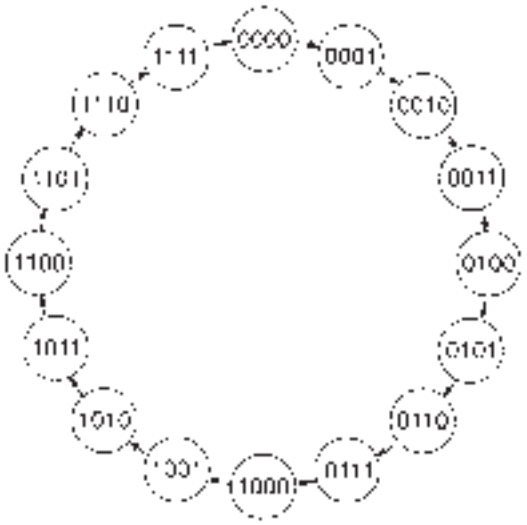


FIGURE 9.3
State Diagram of a Mod-16 Counter

Table 9.1 Mod-16 Count Sequence Table

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

Table 9.2 Mod-12 Count-Sequence Table

$Q_3Q_2Q_1Q_0$
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011

We can derive timing diagrams from each of these tables. We know that each counter advances by one state with each applied clock pulse. The mod-16 count sequence shows us that the Q_0 waveform changes state with each clock pulse. Q_1 changes with every two clock pulses, Q_2 with every four, and Q_3 with every eight. Figure 9.4 shows this pattern for the mod-16 UP counter, assuming the counter is a positive edge-triggered device.

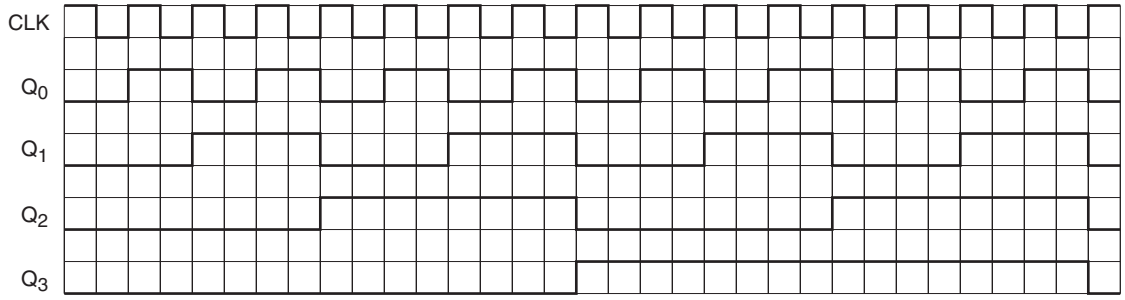


FIGURE 9.4
Mod-16 Timing Diagram

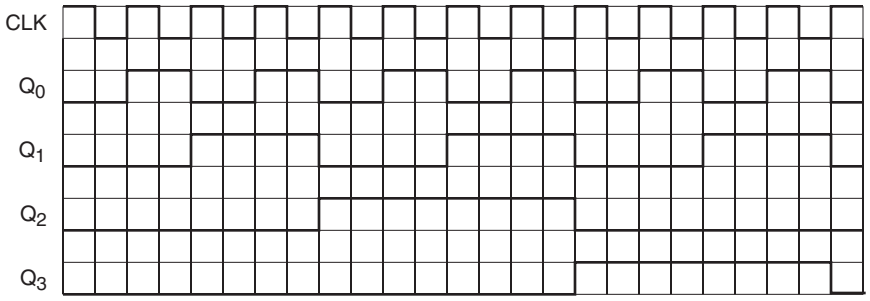


FIGURE 9.5
Mod-12 Timing Diagram

A divide-by-two ratio relates the frequencies of adjacent outputs of a binary counter. For example, if the clock frequency is $f_c = 16$ MHz, the frequencies of the output waveforms are: 8 MHz ($f_0 = f_c/2$); 4 MHz ($f_1 = f_c/4$); 2 MHz ($f_2 = f_c/8$); 1 MHz ($f_3 = f_c/16$).

We can construct a similar timing diagram, illustrated in Figure 9.5, for a mod-12 UP counter. The changes of state can be monitored by noting where Q_0 (the least significant bit) changes. This occurs on each positive edge of the *CLK* waveform. The sequence progresses by 1 with each *CLK* pulse until the outputs all go to 0 on the first *CLK* pulse after state $Q_3Q_2Q_1Q_0 = 1011$.

The output waveform frequencies of a truncated sequence counter do not necessarily have a simple relationship to one another as do binary counters. For the mod-12 counter the relationships between clock frequency, f_c, and output frequencies are: $f_0 = f_c/2$; $f_1 = f_c/4$; $f_2 = f_c/12$; $f_3 = f_c/12$. Note that both Q_2 and Q_3 have the same frequencies (f_2 and f_3), but are out of phase with one another.

EXAMPLE 9.2

Draw the state diagram, count sequence table, and timing diagram for a mod-12 DOWN counter.

Solution Figure 9.6 shows the state diagram for the mod-12 DOWN counter. The states are identical to those of a mod-12 UP counter, but progress in the opposite direction. Table 9.3 shows the count sequence table of this circuit.

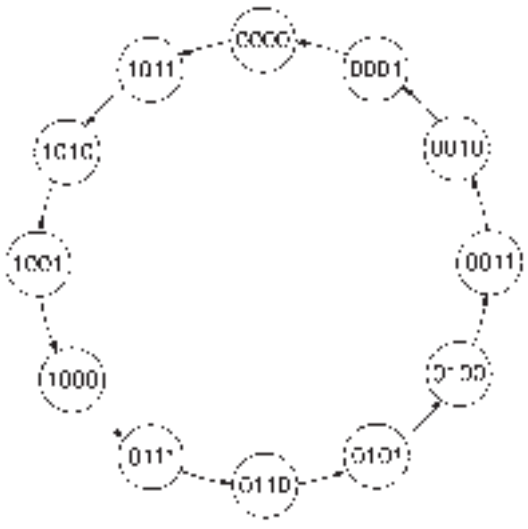


FIGURE 9.6
Example 9.2
State Diagram of a Mod-12 DOWN Counter

Table 9.3 Count-Sequence Table for a Mod-12 DOWN Counter

$Q_3Q_2Q_1Q_0$
1011
1010
1001
1000
0111
0110
0101
0100
0011
0010
0001
0000

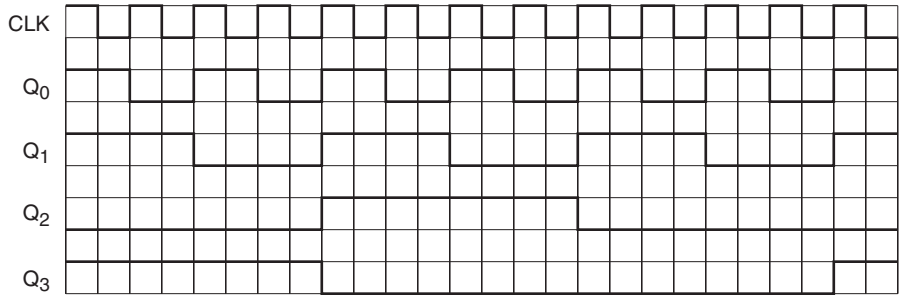


FIGURE 9.7
Example 9.2
Timing Diagram of a Mod-12 DOWN Counter

The timing diagram of this counter is illustrated in Figure 9.7. The output starts in state $Q_3Q_2Q_1Q_0 = 1011$ and counts DOWN until it reaches 0000. On the next pulse, it recycles to 1011 and starts over.

SECTION 9.1 REVIEW PROBLEM

9.1 How many outputs does a mod-24 counter require? Is this a full-sequence or a truncated sequence counter? Explain your answer.

9.2 Synchronous Counters

KEY TERMS

Synchronous counter A counter whose flip-flops are all clocked by the same source and thus change in synchronization with each other.

Present state The current state of flip-flop outputs in a synchronous sequential circuit.

Next state The desired future state of flip-flop outputs in a synchronous sequential circuit after the next clock pulse is applied.

Memory section A set of flip-flops in a synchronous circuit that hold its present state.

Control section The combinational logic portion of a synchronous circuit that determines the next state of the circuit.

Status lines Signals that communicate the present state of a synchronous circuit from its memory section to its control section.

Command lines Signals that connect the control section of a synchronous circuit to its memory section and direct the circuit from its present to its next state.

In Chapter 7, we briefly examined the circuits of a 3-bit and a 4-bit **synchronous counter** (Figures 7.53 and 7.87, respectively). A synchronous counter is a circuit consisting of flip-flops and control logic, whose outputs progress through a regular predictable sequence, driven by a clock signal. The counter is synchronous because all flip-flops are clocked at the same time.

Figure 9.8 shows the block diagram of a synchronous counter, which consists of a **memory section** to keep track of the **present state** of the counter and a **control section** to direct the counter to its **next state.** The memory section is a sequential circuit (flip-flops) and the control section is combinational (gates). They communicate through a set of **status lines** that go from the *Q* outputs of the flip-flops to the control gate inputs and **command lines** that connect the control gate outputs to the synchronous inputs *(J, K, D,* or *T)* of the flip-flops. Outputs can be tied directly to the status lines or can be decoded to give a sequence other than that of the flip-flop output states. The circuit might have inputs to implement one or more control functions, such as changing the count direction, clearing the counter, or presetting the counter to a specific value.

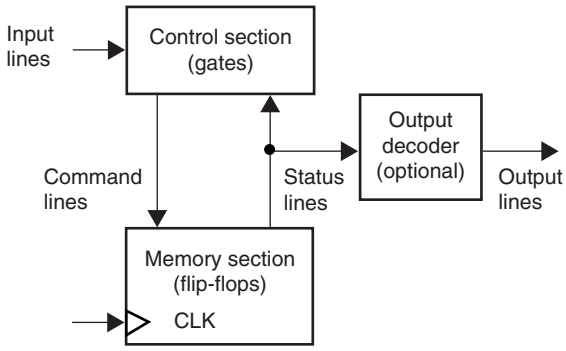


FIGURE 9.8
Synchronous Counter Block Diagram

Analysis of Synchronous Counters

A 3-bit synchronous binary counter based on JK flip-flops is shown in Figure 9.9. Let us analyze its count sequence in detail so that we can see how the *J* and *K* inputs are affected by the *Q* outputs and how transitions between states are made. Later we will look at the function of truncated sequence counter circuits and counters that are made from flip-flops other than JK.

The synchronous input equations are given by:

$$J_2 = K_2 = Q_1 \cdot Q_0$$
$$J_1 = K_1 = Q_0$$
$$J_0 = K_0 = 1$$

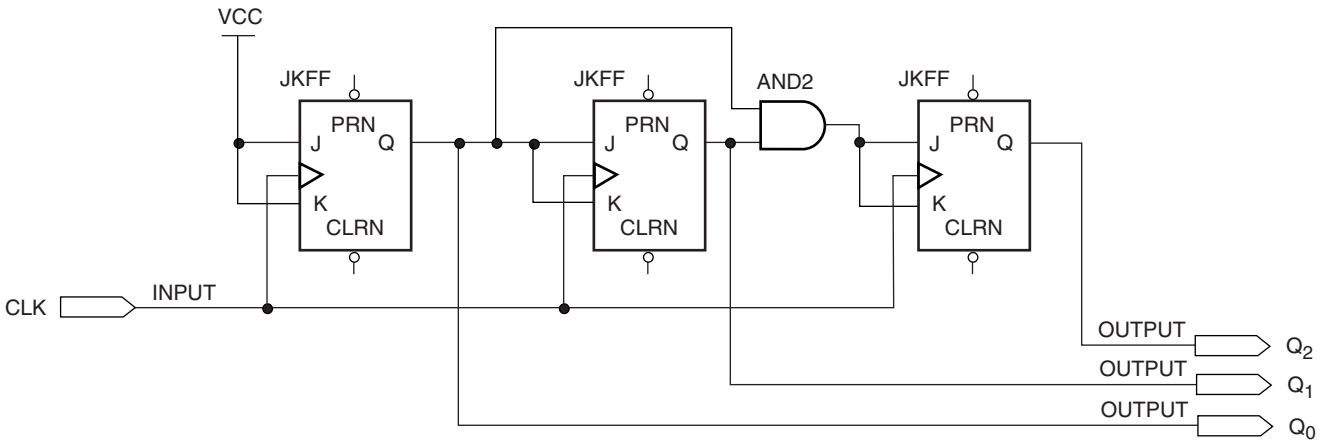


FIGURE 9.9
3-bit Synchronous Binary Counter

For reference, the JK flip-flop function table is shown in Table 9.4:

Table 9.4 Function Table of a JK Flip-Flop

J	*K*	Q_{t+1}	Function
0	0	Q_t	No change
0	1	0	Reset
1	0	1	Set
1	1	$\overline{Q}_t$	Toggle

Q_t indicates the state of Q before a clock pulse is applied. Q_{t+1} indicates the state of Q after the clock pulse.

Assume the counter output is initially $Q_2Q_1Q_1 = 000$. Before any clock pulses are applied, the J and K inputs are at the following states:

$$J_2 = K_2 = Q_1 \cdot Q_0 = 0 \cdot 0 = 0 \quad \text{(No change)}$$
$$J_1 = K_1 = Q_0 = 0 \quad \text{(No change)}$$
$$J_0 = K_0 = 1 \text{ (Constant)} \quad \text{(Toggle)}$$

The transitions of the outputs after the clock pulse are:

$$Q_2\text{: } 0 \rightarrow 0 \quad \text{(No change)}$$
$$Q_1\text{: } 0 \rightarrow 0 \quad \text{(No change)}$$
$$Q_0\text{: } 0 \rightarrow 1 \quad \text{(Toggle)}$$

The output goes from $Q_2Q_1Q_1 = 000$ to $Q_2Q_1Q_1 = 001$ (see Figure 9.10). The transition is defined by the values of J and K *before* the clock pulse, since the propagation delays of the flip-flops prevent the new output conditions from changing the J and K values until after the transition.

The new conditions of the J and K inputs are:

$$J_2 = K_2 = Q_1 \cdot Q_0 = 0 \cdot 1 = 0 \quad \text{(No change)}$$
$$J_1 = K_1 = Q_0 = 1 \quad \text{(Toggle)}$$
$$J_0 = K_0 = 1 \text{ (Constant)} \quad \text{(Toggle)}$$

The transitions of the outputs generated by the second clock pulse are:

Q_2: $0 \rightarrow 0$ (No change)
Q_1: $0 \rightarrow 1$ (Toggle)
Q_0: $1 \rightarrow 0$ (Toggle)

The new output is $Q_2Q_1Q_0 = 010$, since both Q_0 and Q_1 change and Q_2 stays the same. The J and K conditions are now:

$J_2 = K_2 = Q_1 \cdot Q_0 = 1 \cdot 0 = 0$ (No change)
$J_1 = K_1 = Q_0 = 0$ (No change)
$J_0 = K_0 = 1$ (Constant) (Toggle)

The output transitions are:

Q_2: $0 \rightarrow 0$ (No change)
Q_1: $1 \rightarrow 1$ (No change)
Q_0: $0 \rightarrow 1$ (Toggle)

The output is now $Q_2Q_1Q_0 = 011$, which results in the JK conditions:

$J_2 = K_2 = Q_1 \cdot Q_0 = 1 \cdot 1 = 1$ (Toggle)
$J_1 = K_1 = Q_0 = 1$ (Toggle)
$J_0 = K_0 = 1$ (Constant) (Toggle)

The above conditions result in output transitions:

Q_2: $0 \rightarrow 1$ (Toggle)
Q_1: $1 \rightarrow 0$ (Toggle)
Q_0: $1 \rightarrow 0$ (Toggle)

All the outputs toggle and the new output state is $Q_2Q_1Q_0 = 100$. The J and K values repeat the above pattern in the second half of the counter cycle (states 100 to 111). Go through the exercise of calculating the J, K, and Q values for the rest of the cycle. Compare the result with the timing diagram in Figure 9.10.

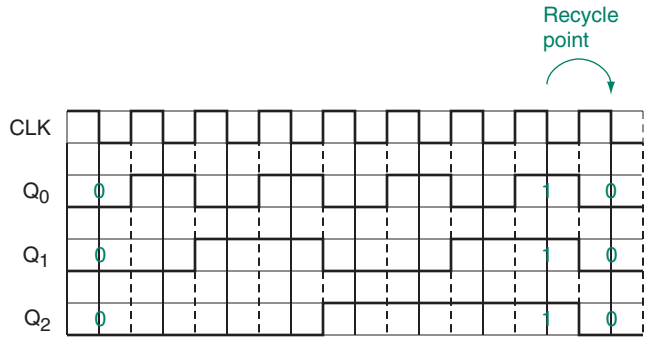


FIGURE 9.10
Timing Diagram for a Synchronous 3-bit Binary Counter

In the counter we have just analyzed, the combinational circuit generates either a toggle ($JK = 11$) or a no change ($JK = 00$) state at each point through the count sequence. We could use any combination of JK modes (no change, reset, set, or toggle) to make the transitions from one state to the next. For instance, instead of using only the no change and toggle modes, the $000 \rightarrow 001$ transition could also be done by making Q_0 set ($J_0 = 1$,

$K_0 = 0$) and Q_1 and Q_2 reset ($J_1 = 0$, $K_1 = 1$ and $J_2 = 0$, $K_2 = 1$). To do so we would need a different set of combinational logic in the circuit.

The simplest synchronous counter design uses only the no change ($JK = 00$) or toggle ($JK = 11$) modes, since the J and K inputs of each flip-flop can be connected together. The no change and toggle modes allow us to make any transition (i.e., not just in a linear sequence), even though for truncated sequence and nonbinary counters this is not usually the most efficient design.

There is a simple progression of algebraic expressions for the J and K inputs of a synchronous binary (full sequence) counter, which uses only the no change and toggle states:

$$J_0 = K_0 = 1$$
$$J_1 = K_1 = Q_0$$
$$J_2 = K_2 = Q_1 \cdot Q_0$$
$$J_3 = K_3 = Q_2 \cdot Q_1 \cdot Q_0$$
$$J_4 = K_4 = Q_3 \cdot Q_2 \cdot Q_1 \cdot Q_0$$
etc.

The J and K inputs of each stage are the ANDed outputs of all previous stages. This implies that a flip-flop toggles only when the outputs of *all* previous stages are HIGH. For example, Q_2 doesn't change unless *both* Q_1 AND Q_0 are HIGH (and therefore $J_2 = K_2 = 1$) before the clock pulse. In a 3-bit counter, this occurs only at states 011 and 111, after which Q_2 will toggle, along with Q_1 and Q_0, giving transitions to states 100 and 000 respectively. Look at the timing diagram of Figure 9.10 to confirm this.

Determining the Modulus of a Synchronous Counter

We can use a more formal technique to analyze any synchronous counter, as follows.

1. Determine the equations for the synchronous inputs *(JK, D,* or *T)* in terms of the Q outputs for all flip-flops. (For counters other than straight binary full sequence types, the equations will *not* be the same as the algebraic progressions previously listed.)
2. Lay out a table with headings for the Present State of the counter (Q outputs before *CLK* pulse), each Synchronous Input before *CLK* pulse, and Next State of the counter (Q outputs after the clock pulse).
3. Choose a starting point for the count sequence, usually 0, and enter the starting point in the Present State column.
4. Substitute the Q values of the initial present state into the synchronous input equations and enter the results under the appropriate columns.
5. Determine the action of each flip-flop on the next *CLK* pulse (e.g., for a JK flip-flop, the output either will not change ($JK = 00$), or will reset ($JK = 01$), set ($JK = 10$), or toggle ($JK = 11$)).
6. Look at the Q values for every flip-flop. Change them according to the function determined in Step 5 and enter them in the column for the counter's next state.
7. Enter the result from Step 6 on the next line of the column for the counter's present state (i.e., this line's next state is the next line's present state).
8. Repeat the above process until the result in the next state column is the same as the initial state.

EXAMPLE 9.3

Find the count sequence of the synchronous counter shown in Figure 9.11 and, from the count sequence table, draw the timing diagram and state diagram. What is the modulus of the counter?

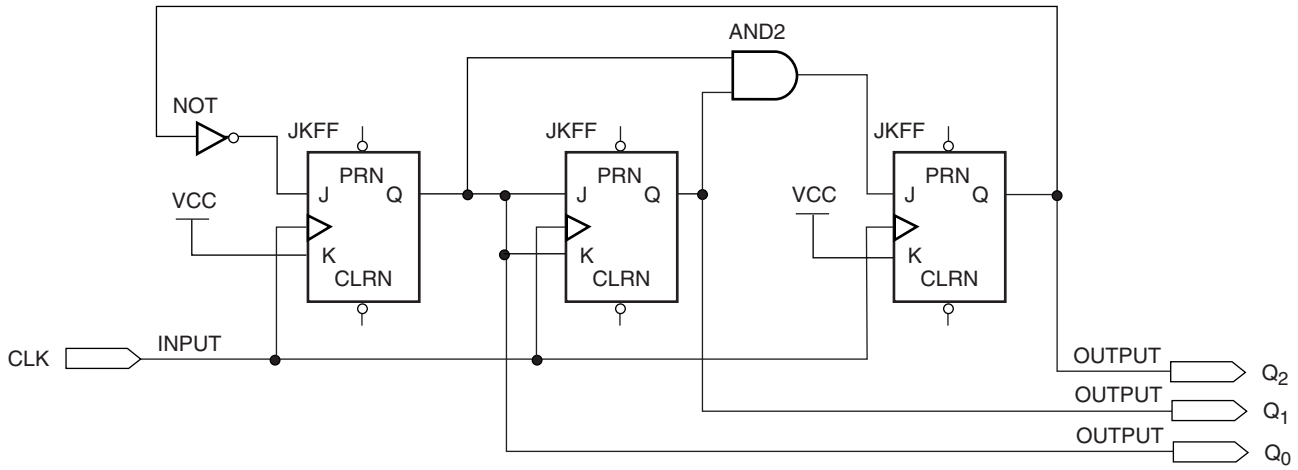


FIGURE 9.11
Synchronous Counter of Unknown Modulus

Solution The J and K equations are:

$$J_2 = Q_1 \cdot Q_0 \qquad J_1 = Q_0 \qquad J_0 = \overline{Q}_2$$
$$K_2 = 1 \qquad K_1 = Q_0 \qquad K_0 = 1$$

The output transitions can be determined from the values of the J and K functions before each clock pulse, as shown in Table 9.5.

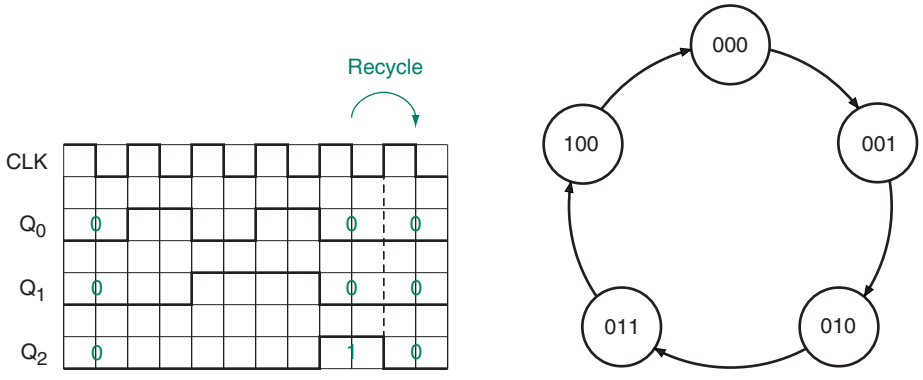
Table 9.5 State Table for Figure 9.11

Present State	Synchronous Inputs						Next State
$Q_2Q_1Q_0$	J_2K_2		J_1K_1		J_0K_0		$Q_2Q_1Q_0$
000	01	(R)	00	(NC)	11	(T)	001
001	01	(R)	11	(T)	11	(T)	010
010	01	(R)	00	(NC)	11	(T)	011
011	11	(T)	11	(T)	11	(T)	100
100	01	(R)	00	(NC)	01	(R)	000

Since there are five unique output states, the counter's modulus is 5.

The timing diagram and state diagram are shown in Figure 9.12. Since this circuit produces one pulse on Q_2 for every 5 clock pulses, we can use it as a divide-by-5 circuit.

FIGURE 9.12
Example 9.3
Timing Diagram and State Diagram of a Mod-5 Counter



a. Timing diagram **b. State diagram**

The analysis in Example 9.3 did not account for the fact that the counter uses only 5 of a possible 8 output states. In any truncated sequence counter, it is good practice to determine the next state for each unused state to ensure that if the counter powers up in one of these unused states, it will eventually enter the main sequence.

EXAMPLE 9.4

Extend the analysis of the counter in Example 9.3 to include its unused states. Redraw the counter's state diagram to show how these unused states enter the main sequence (if they do).

Solution The synchronous input equations are:

$$J_2 = Q_1 \cdot Q_0 \qquad J_1 = Q_0 \qquad J_0 = \overline{Q}_2$$
$$K_2 = 1 \qquad K_1 = Q_0 \qquad K_0 = 1$$

The unused states are $Q_2Q_1Q_0 = 101$, 110, and 111. Table 9.6 shows the transitions made by the unused states. Figure 9.13 shows the completed state diagram.

Table 9.6 State Table for Mod-5 Counter Including Unused States

Present State	Synchronous Inputs						Next State
$Q_2Q_1Q_0$	J_2K_2		J_1K_1		J_0K_0		$Q_2Q_1Q_0$
000	01	(R)	00	(NC)	11	(T)	001
001	01	(R)	11	(T)	11	(T)	010
010	01	(R)	00	(NC)	11	(T)	011
011	11	(T)	11	(T)	11	(T)	100
100	01	(R)	00	(NC)	01	(R)	000
101	01	(R)	11	(T)	01	(R)	010
110	01	(R)	00	(NC)	01	(R)	010
111	11	(T)	11	(T)	01	(R)	000

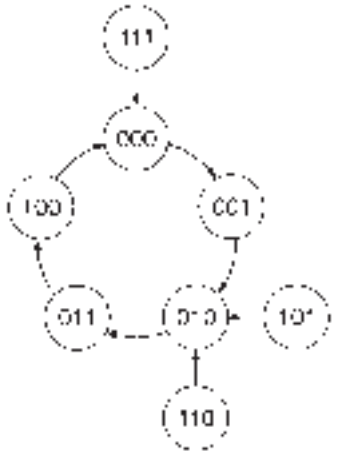


FIGURE 9.13
Example 9.4
Complete State Diagram

SECTION 9.2 REVIEW PROBLEM

9.2 A 4-bit synchronous counter based on JK flip-flops is described by the following set of equations:

$$J_3 = Q_2Q_1Q_0 \qquad J_2 = Q_1Q_0 \qquad J_1 = \overline{Q}_3Q_0 \qquad J_0 = 1$$
$$K_3 = Q_0 \qquad K_2 = Q_1Q_0 \qquad K_1 = Q_0 \qquad K_0 = 1$$

Assume the counter output is at 1000 in the count sequence. What will the output be after one clock pulse? After two clock pulses?

9.3 Design of Synchronous Counters

KEY TERMS

Excitation table A table showing the required input conditions for every possible transition of a flip-flop output.

State machine A synchronous sequential circuit.

A synchronous counter can be designed using established techniques that involve the derivation of Boolean equations for the counter's next state logic. Alternatively, several VHDL structures can be used to define counters; we can use a behavioral description of the counter, or we can use a **state machine** definition in VHDL that specifies each present and next state explicitly.

In addition to the classical counter design techniques, we will examine the design of a counter through a behavioral description in VHDL. We will leave the state machine design for the following chapter.

Classical Design Technique

There are several steps involved in the classical design of a synchronous counter.

1. Define the problem. Before you can begin design of a circuit, you have to know what its purpose is and what it should do under all possible conditions.
2. Draw a state diagram showing the progression of states under various input conditions and what outputs the circuit should produce, if any.
3. Make a state table which lists all possible Present States and the Next State for each one. *List the present states 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.
5. The logic levels of the synchronous inputs are Boolean functions of the flip-flop outputs and the control inputs. Simplify the expression for each input and write the simplified Boolean expression.
6. Use the Boolean expressions found in step 5 to draw the required logic circuit.

Flip-flop Excitation Tables

In the synchronous counter circuits we examined earlier in this chapter, we used JK flip-flops that were configured to operate only in toggle or no change mode. We can use any type of flip-flop for a synchronous sequential circuit. If we choose to use JK flip-flops, we can use any of the modes (no change, reset, set, or toggle) to make transitions from one state to another.

A flip-flop excitation table shows all possible transitions of a flip-flop output and the synchronous input levels needed to effect these transitions. Table 9.7 is the excitation table of a JK flip-flop.

If we want a flip-flop to make a transition from 0 to 1, we can use either the toggle function ($JK = 11$) or the set function ($JK = 10$). It doesn't matter what K is, as long as $J = 1$. This is reflected by the variable pair ($JK = 1X$) beside the $0 \rightarrow 1$ entry in Table 9.7. The X is a don't care state, a 0 or 1 depending on which is more convenient for the simplification of the Boolean function of the J or K input affected.

Table 9.8 shows a condensed version of the JK flip-flop excitation table.

Table 9.7 JK Flip-Flop Excitation Table

Transition	Function	*JK*	
$0 \rightarrow 0$	No change or reset	00 01	0X
$0 \rightarrow 1$	Toggle or set	11 10	1X
$1 \rightarrow 0$	Toggle or reset	11 01	X1
$1 \rightarrow 1$	No change or set	00 10	X0

Table 9.8 Condensed Excitation Table for a JK Flip-Flop

Transition	*JK*
$0 \rightarrow 0$	0X
$0 \rightarrow 1$	1X
$1 \rightarrow 0$	X1
$1 \rightarrow 1$	X0

Design of a Synchronous Mod-12 Counter

We will follow the procedure outlined above to design a synchronous mod-12 counter circuit, using JK flip-flops. The aim is to derive the Boolean equations of all *J* and *K* inputs and to draw the counter circuit.

1. *Define the problem.* The circuit must count in binary sequence from 0000 to 1011 and repeat. The output progresses by 1 for each applied clock pulse. Since the outputs are 4-bit numbers, we require 4 flip-flops.
2. *Draw a state diagram.* The state diagram for this problem is shown in Figure 9.14.
3. *Make a state table showing each present state and the corresponding next state.*
4. *Use flip-flop excitation tables to fill in the J and K entries in the state table.* Table 9.9 shows the combined result of steps 3 and 4. Note that all present states are in binary order.

   We assume for now that states 1100 to 1111 never occur. If we assign their corresponding next states to be don't care states, they can be used to simplify the *J* and *K* expressions we derive from the state table.

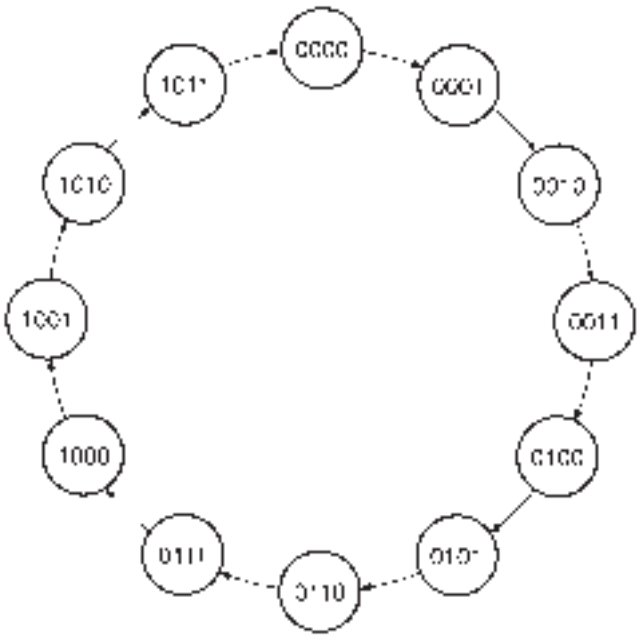


FIGURE 9.14
State Diagram for a Mod-12 Counter

Table 9.9 State Table for a Mod-12 Counter

Present State	Next State	Synchronous Inputs			
$Q_3Q_2Q_1Q_0$	$Q_3Q_2Q_1Q_0$	J_3K_3	J_2K_2	J_1K_1	J_0K_0
0000	0001	0X	0X	0X	1X
0001	0010	0X	0X	1X	X1
0010	0011	0X	0X	X0	1X
0011	0100	0X	1X	X1	X1
0100	0101	0X	X0	0X	1X
0101	0110	0X	X0	1X	X1
0110	0111	0X	X0	X0	1X
0111	1000	1X	X1	X1	X1
1000	1001	X0	0X	0X	1X
1001	1010	X0	0X	1X	X1
1010	1011	X0	0X	X0	1X
1011	0000	X1	0X	X1	X1
1100	XXXX	XX	XX	XX	XX
1101	XXXX	XX	XX	XX	XX
1110	XXXX	XX	XX	XX	XX
1111	XXXX	XX	XX	XX	XX

Let us examine one transition to show how the table is completed. The transition from $Q_3Q_2Q_1Q_0 = 0101$ to $Q_3Q_2Q_1Q_0 = 0110$ consists of the following individual flip-flop transitions.

Q_3: $0 \rightarrow 0$ (No change or reset; $J_3K_3 = 0X$)
Q_2: $1 \rightarrow 1$ (No change or set; $J_2K_2 = X0$)
Q_1: $0 \rightarrow 1$ (Toggle or set; $J_1K_1 = 1X$)
Q_0: $1 \rightarrow 0$ (Toggle or reset; $J_0K_0 = X1$)

The other lines of the table are similarly completed.

5. *Simplify the Boolean expression for each input.* Table 9.9 can be treated as eight truth tables, one for each J or K input. We can simplify each function by Boolean algebra or by using a Karnaugh map.

   Figure 9.15 shows K-map simplification for all 8 synchronous inputs. These maps yield the following simplified Boolean expressions.

$$J_0 = 1$$
$$K_0 = 1$$

$$J_1 = Q_0$$
$$K_1 = Q_0$$

$$J_2 = \overline{Q}_3Q_1Q_0$$
$$K_2 = Q_1Q_0$$

$$J_3 = Q_2Q_1Q_0$$
$$K_3 = Q_1Q_0$$

6. *Draw the required logic circuit.* Figure 9.16 shows the circuit corresponding to the above Boolean expressions.

We have assumed that states 1100 to 1111 will never occur in the operation of the mod-12 counter. This is normally the case, but when the circuit is powered up, there is no guarantee that the flip-flops will be in any particular state.

If a counter powers up in an unused state, the circuit should enter the main sequence after one or more clock pulses. To test whether or not this happens, let us make a state

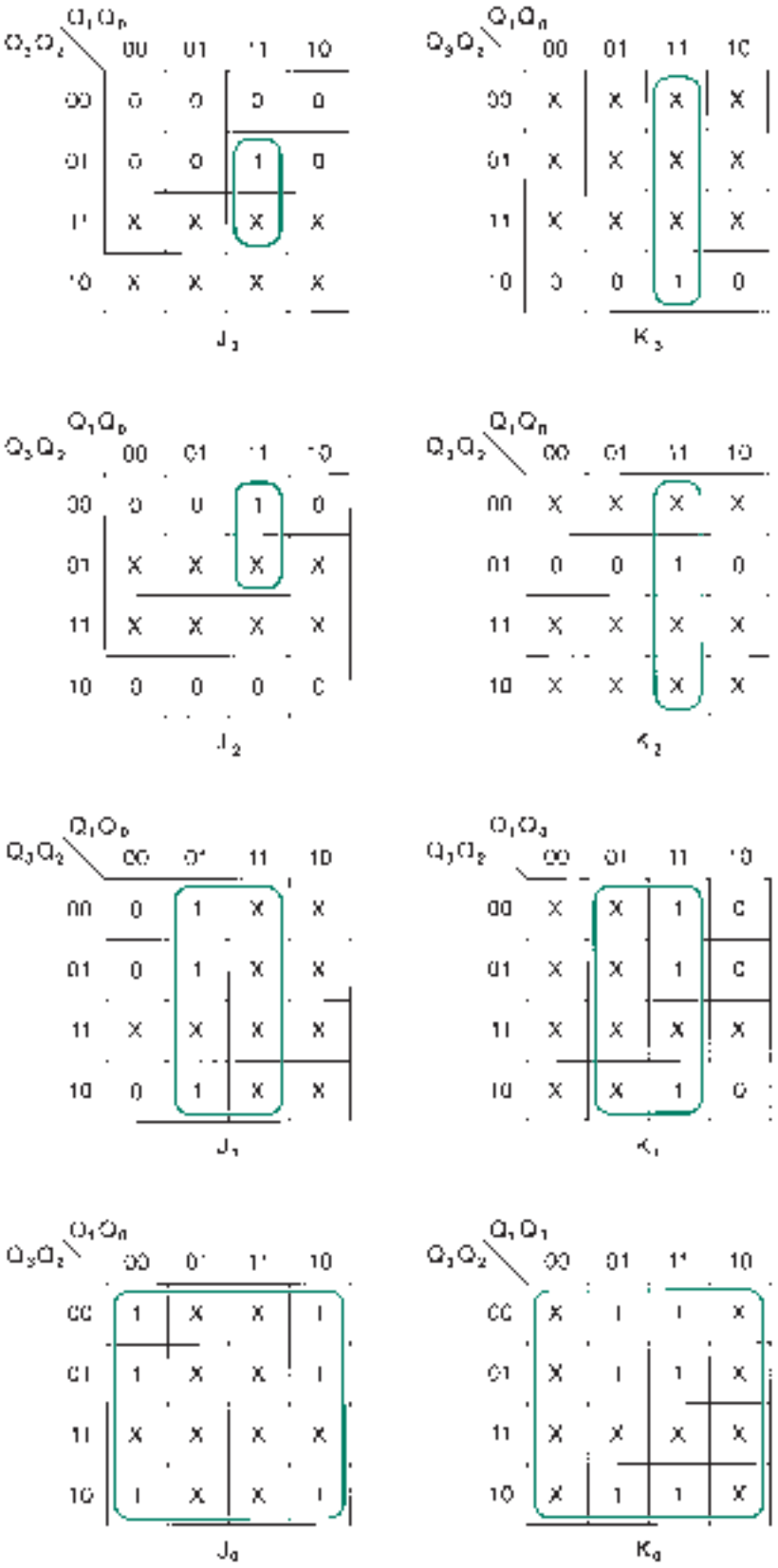


FIGURE 9.15
K-Map Simplification of Table 9.9

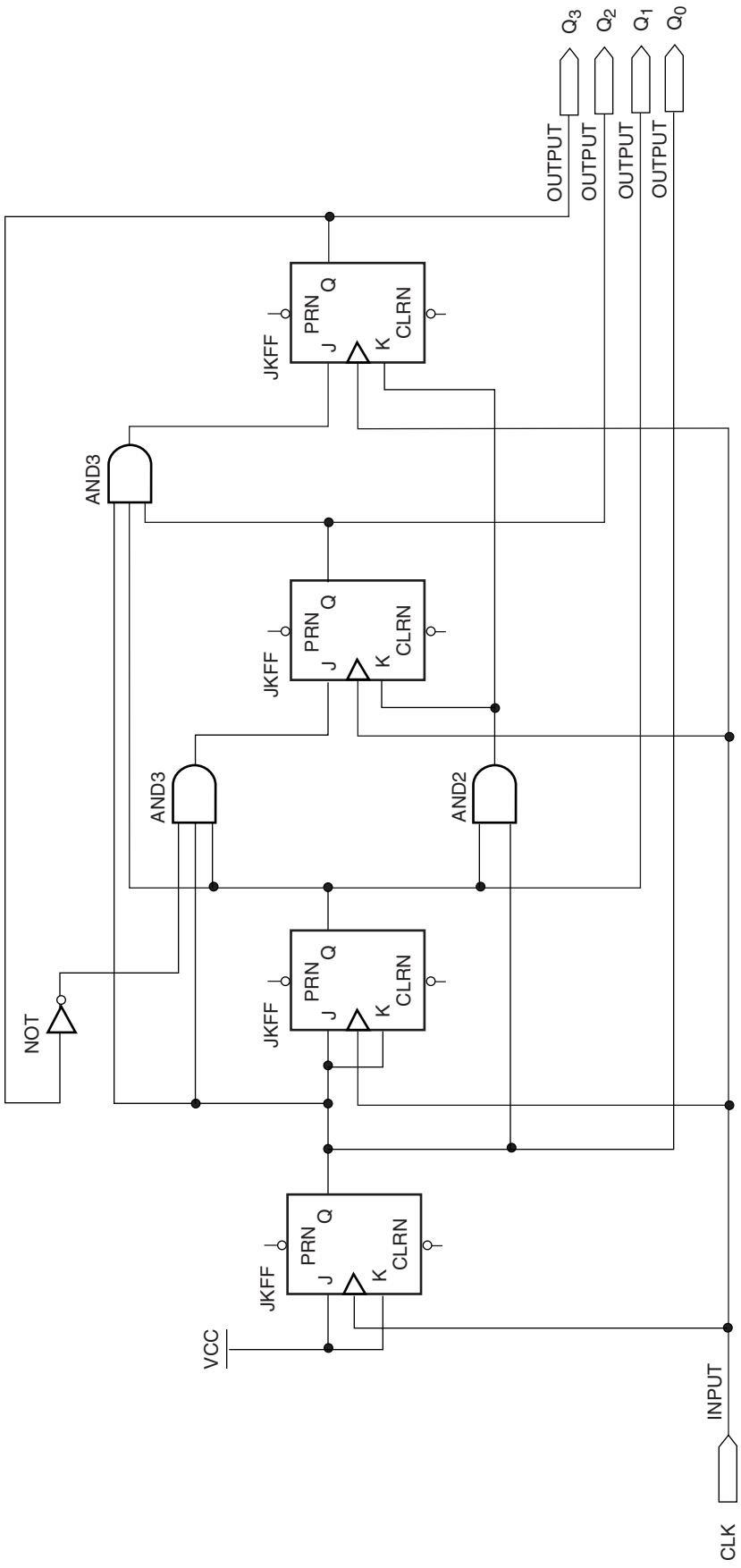


FIGURE 9.16
Synchronous Mod-12 Counter

Table 9.10 Unused States in a Mod-12 Counter

Present State	Synchronous Inputs				Next State
$Q_3Q_2Q_1Q_0$	J_3K_3	J_2K_2	J_1K_1	J_0K_0	$Q_3Q_2Q_1Q_0$
0000	00	00	00	11	1101
1101	00	00	11	11	1110
1110	00	00	00	11	1111
1111	11	01	11	11	0000

table, applying each unused state to the J and K equations as implemented, to see what the Next State is for each case. This analysis is shown in Table 9.10.

Figure 9.17 shows the complete state diagram for the designed mod-12 counter. If the counter powers up in an unused state, it will enter the main sequence in no more than four clock pulses.

If we want an unused state to make a transition directly to 0000 in one clock pulse, we have a couple of options:

1. We could reset the counter asynchronously and otherwise leave the design as is.
2. We could rewrite the state table to specify these transitions, rather than make the unused states don't cares.

Option 1 is the simplest and is considered perfectly acceptable as a design practice. Option 2 would yield a more complicated set of Boolean equations and hence a more complex circuit, but might be worthwhile if a direct synchronous transition to 0000 were required.

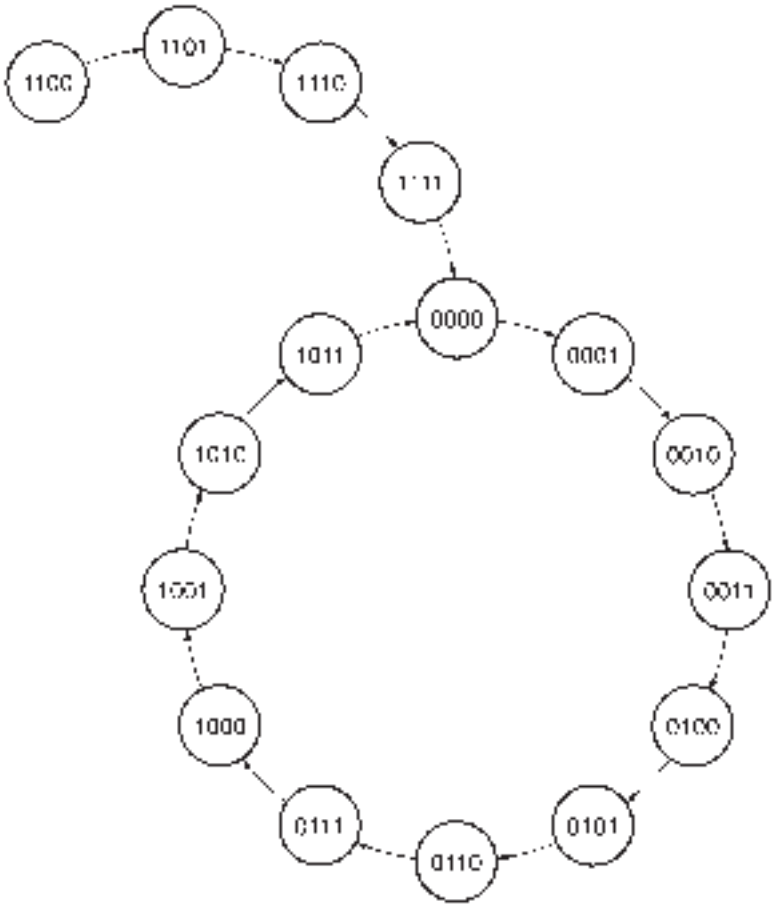


FIGURE 9.17
Complete State Diagram of Mod-12 Counter in Figure 9.16

EXAMPLE 9.5

Derive the synchronous input equations of a 4-bit synchronous binary counter based on D flip-flops. Draw the corresponding counter circuit.

Solution The first step in the counter design is to derive the excitation table of a D flip-flop. Recall that Q follows D when the flip-flop is clocked. Therefore the next state of Q is the same as the input D for any transition. This is illustrated in Table 9.11.

Table 9.11 Excitation Table of a D Flip-Flop

Transition	D
$0 \rightarrow 0$	0
$0 \rightarrow 1$	1
$1 \rightarrow 0$	0
$1 \rightarrow 1$	1

Next, we must construct a state table, shown in Table 9.12, with present and next states for all possible transitions. Note that the binary value of $D_3D_2D_1D_0$ is the same as the next state of the counter.

Table 9.12 State Table for a 4-bit Binary Counter

Present State	Next State	Synchronous Inputs
$Q_3Q_2Q_1Q_0$	$Q_3Q_2Q_1Q_0$	$D_3D_2D_1D_0$
0000	0001	0001
0001	0010	0010
0010	0011	0011
0011	0100	0100
0100	0101	0101
0101	0110	0110
0110	0111	0111
0111	1000	1000
1000	1001	1001
1001	1010	1010
1010	1011	1011
1011	1100	1100
1100	1101	1101
1101	1110	1110
1110	1111	1111
1111	0000	0000

This state table yields four Boolean equations, for D_3 through D_0, in terms of the present state outputs. Figure 9.18 shows four Karnaugh maps used to simplify these functions.

The simplified equations are:

$$D_3 = \overline{Q}_3Q_2Q_1Q_0 + Q_3\overline{Q}_2 + Q_3\overline{Q}_1 + Q_3\overline{Q}_0$$
$$D_2 = \overline{Q}_2Q_1Q_0 + Q_2\overline{Q}_1 + Q_1\overline{Q}_0$$
$$D_1 = \overline{Q}_1Q_0 + Q_1\overline{Q}_0$$
$$D_0 = \overline{Q}_0$$

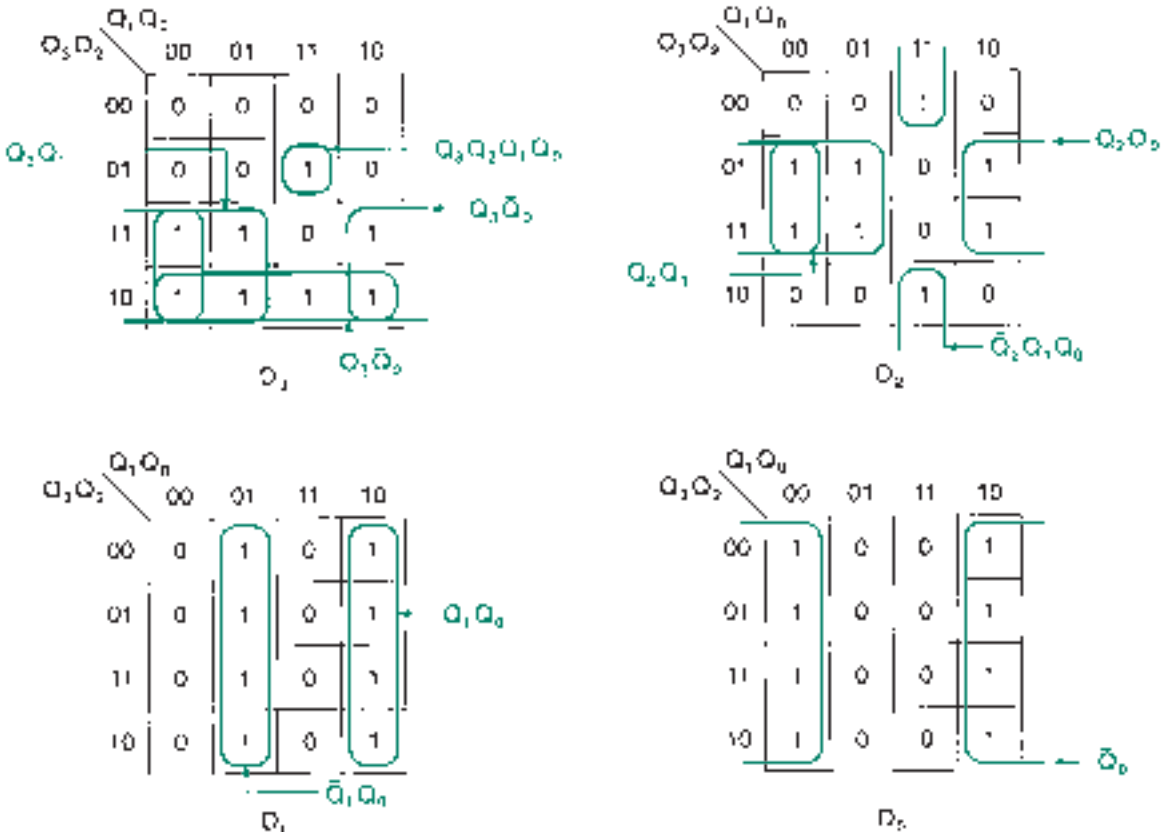


FIGURE 9.18
Example 9.5
K-Maps for a 4-bit Counter Based on D Flip-Flops

These equations represent the maximum SOP simplifications of the input functions. However, we can rewrite them to make them more compact. For example the equation for D_3 can be rewritten, using DeMorgan's theorem ($\bar{x} + \bar{y} + \bar{z} = \overline{xyz}$) and our knowledge of Exclusive OR (XOR) functions ($\bar{x}y + x\bar{y} = x \oplus y$).

$$\begin{aligned} D_3 &= \overline{Q}_3Q_2Q_1Q_0 + Q_3\overline{Q}_2 + Q_3\overline{Q}_1 + Q_3\overline{Q}_0 \\ &= \overline{Q}_3(Q_2Q_1Q_0) + Q_3(\overline{Q}_2 + \overline{Q}_1 + \overline{Q}_0) \\ &= \overline{Q}_3(Q_2Q_1Q_0) + Q_3(\overline{Q_2Q_1Q_0}) \\ &= Q_3 \oplus Q_2Q_1Q_0 \end{aligned}$$

We can write similar equations for the other D inputs as follows:

$$\begin{aligned} D_2 &= Q_2 \oplus Q_1Q_0 \\ D_1 &= Q_1 \oplus Q_0 \\ D_0 &= Q_0 \oplus 1 \end{aligned}$$

These equations follow a predictable pattern of expansion. Each equation for an input D_n is simply Q_n XORed with the logical product (AND) of all previous Qs.

Figure 9.19 shows the circuit for the 4-bit counter, including an asynchronous reset.

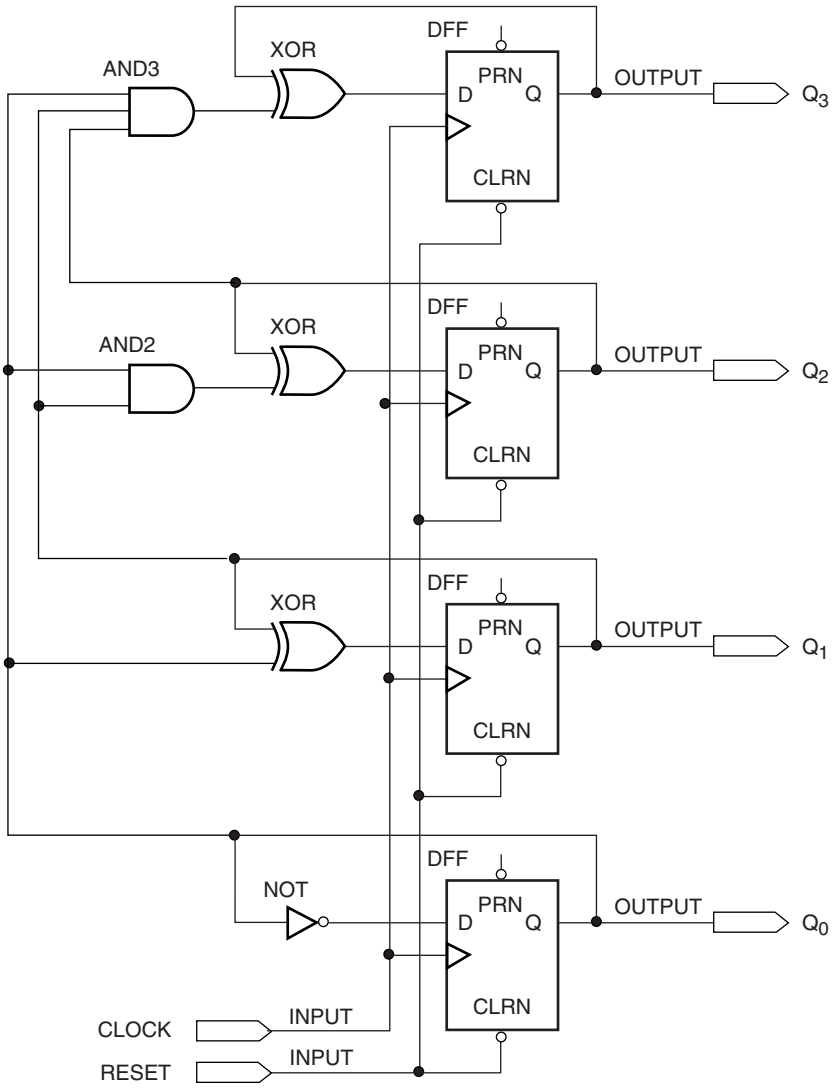


FIGURE 9.19
Example 9.5
4-bit Counter Using D Flip-Flops

In Section 7.6 (Edge-Triggered T Flip-Flops) of Chapter 7, we saw how a D flip-flop could be configured for a switchable toggle function (refer to Figure 7.59). The flip-flops in Figure 9.19 are similarly configured. Each flip-flop output, except Q_0, is fed back to its input through an Exclusive OR gate. The other input to the XOR controls whether this feedback is inverted (for toggle mode) or not (for no change mode). Recall that $x \oplus 0 = x$ and $x \oplus 1 = \bar{x}$.

For example, Q_3 is fed back to D_3 through an XOR gate. The feedback is inverted only if the 3-input AND gate has a HIGH output. Thus, the Q_3 output toggles only if all previous bits are HIGH ($Q_3Q_2Q_1Q_0 = 0111$ or 1111). The flip-flop toggle mode is therefore controlled by the states of the XOR and AND gates in the circuit.

SECTION 9.3 REVIEW PROBLEM

9.3 A 4-bit synchronous counter must make a transition from state $Q_3Q_2Q_1Q_0 = 1011$ to $Q_3Q_2Q_1Q_0 = 1100$. Write the required states of the synchronous inputs for a set of four JK flip-flops used to implement the counter. Write the required states of the synchronous inputs if the counter is made from D flip-flops.

9.4 Programming Binary Counters in VHDL

KEY TERMS

If statement A VHDL construct in which statements within the IF statement are executed only when a specified Boolean condition is satisfied.

Attribute A property associated with a named identifier in VHDL. (For example, the attribute **EVENT,** when associated with the identifier **clk** (written **clk'EVENT**), indicates, when true, that a transition has occurred on the input called **clk.**)

When using VHDL to create a counter, we can take several approaches. We can encode the Boolean equations of the counter directly with concurrent signal assignment statements; we can use VHDL code to describe the behavior of the counter; we can use a CASE statement to implement the state diagram of the counter; or we can use a predefined counter, such as those found in the MAX+PLUS II Library of Parameterized Modules (LPM) and map its ports to the ports of a VHDL design entity.

If we chose to use concurrent signal assignments to encode the Boolean equations of a counter, we could derive the following equations for a 4-bit counter with D flip-flops.

```
d(3)<= q(3)xor(q(2)and q(1)and q(0));,
d(2)<= q(2)xor(q(1)and q(0));
d(1)<= q(1)xor q(1);,
d(0)<= not q(0);,
```

In Chapter 5, we saw that using concurrent signal assignment statements is an inefficient way to code many digital functions. (For one thing, if we use this procedure, we must know what the equations are. Getting to that point requires a lot of work that can be done by the VHDL compiler.) While acknowledging this as a possible option, we will not examine this method any further for the count logic of binary counters.

In this section, we will design a counter using a behavioral description and using an LPM counter. The design of a counter as a state machine will be examined in the next chapter.

Behavioral Description of Counters

The following VHDL code shows the behavioral description of a simple 8-bit counter **(ct_simp.vhd)** with asynchronous clear.


ct_simp.vhd

```
ENTITY ct_simp IS
    PORT(
        clk      : IN    BIT;
        clear    : IN    BIT;
        q        : OUT   INTEGER RANGE 0 TO 255);
END ct_simp;

ARCHITECTURE a OF ct_simp IS
    BEGIN
    PROCESS (clk, clear)
        VARIABLE count : INTEGER RANGE 0 TO 255;
    BEGIN
        If (clear = '0') THEN
            count := 0;
        ELSE
            IF (clk'EVENT AND clk = '1') THEN
                count := count + 1;
            END IF;
```

```
        END IF;
    q <= count;
    END PROCESS;
END a;
```

Recall that the PROCESS statement has the following syntax:

```
PROCESS (sensitivity list)
   [VARIABLE variable name :type [range]; ]
BEGIN
   Process statements
END PROCESS;
```

Square brackets [ ] indicate an optional part of the code.

When there is a change in an item in the sensitivity list, the process statements are executed. For a synchronous counter, the list would often only include **clock,** since any action in a synchronous circuit depends on a clock transition. Since the clear function in this counter is asynchronous, the **clear** input must also be monitored for any changes.

To hold the accumulating output value of the counter, we define a variable called **count,** presumed to have an initial value of 0, but defined for the range of 0 to 255. (This 8-bit value rolls over to 0 when the count exceeds 255.) The variable (*any* variable) is local to the process in which it is defined. We update the value of **count** by an **IF statement,** with the form:

```
IF (condition) THEN
    Statement[s];
[ELSIF (condition) THEN
    statement[s];]
[ELSE
    statement[s];]
END IF;
```

The clause `(IF (clear='0') THEN)` monitors the asynchronous clear function independently of the clock and executes the variable assignment that sets the output to 0 if the Boolean condition `(clear='0')` is true. Otherwise, the clock is monitored for a positive edge by the condition `(clk'EVENT AND clk = '1')`. The clause `clk'EVENT` (pronounced "clock tick event") is a predefined **attribute** of the clock signal and is true if there has just been a change on clock. The combination of this and the condition `clk = '1'` indicates that a positive edge has just occurred. If this is true, the count is incremented.

As a final step, the accumulated count must be assigned to an output port. This is done in the concurrent signal assignment `q <= count` at the end of the process.

Note the difference in types of assignments. A *variable* is assigned by the := operator (e.g., `count := count + 1;`). A *signal* is assigned by the <= operator (eg., `q <= count`).

LPM Counters in VHDL

We can use a component **(lpm_counter)** from the Library of Parameterized Modules (LPM) to instantiate a counter in VHDL. When using an LPM counter, we don't need to describe the behavior of the counter, as this has been done for us in the module itself. All we need to do is map the ports and parameters of the LPM component to the ports of the VHDL design entity. We do this by using a **generic map** to specify the parameters we need and a **port map** to map the ports of the LPM device either to an external port or an internal signal. The VHDL code below shows the VHDL implementation **(lpm_simp.vhd)** of the same 8-bit counter as in the previous behavioral example.


lpm_simp.vhd

```
-- lpm_simp.vhd
-- Eight-bit binary counter based on a component
```

```
--   from the Library of Parameterized Modules (LPM)
-- Counter has an active-LOW asynchronous clear.

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

ENTITY lpm_simp IS
   PORT(
        clk, clear : IN  STD_LOGIC;
        q          : OUT STD_LOGIC_VECTOR (7 downto 0));
END lpm_simp;

ARCHITECTURE count OF lpm_simp IS
   SIGNAL clrn : STD_LOGIC;
BEGIN
       count8: lpm_counter
        GENERIC MAP (LPM_WIDTH   => 8)
        PORT MAP ( clock  => clk,
                   aclr   => clrn,
                   q      => q(7 downto 0));

   clrn <= not clear;
END count;
```

LPM components require us to use two packages: the **std_logic_1164** package in the **ieee** library to define STD_LOGIC types used in the LPM components and the **lpm_components** package in the **lpm** library to define the components themselves. Since LPM components are defined using STD_LOGIC and STD_LOGIC_VECTOR types, we should use these types for our other identifiers as well.

The entity declaration defines the inputs and outputs of our counter and need not correspond to the port names for the LPM counter. That correspondence is defined in the architecture body, where we instantiate the counter module. The counter is defined in a component instantiation statement, which takes the following form:

```
__instance_name: __component_name
   GENERIC MAP (__parameter_name => __parameter_value,
                __parameter_name => __parameter_value)
   PORT MAP (__component_port => __connect_port,
             __component_port => __connect_port);
```

The component name is the name of the LPM component. Parameter names are those defined in the LPM component, such as LPM_WIDTH. Parameter values are those values assigned in the instance of the component. Component ports are the LPM port names. Connect ports are the names of identifiers declared in the entity or as signals or variables.

If we want to invert the active level of an LPM input port, we must use a signal assignment statement. (e.g., `clrn <= not clear;`) We need to do this because a VHDL input port cannot be "updated" (modified); only an output can be assigned a new value as a result of a Boolean expression. Thus, we create a signal called **clrn** that maps to the **aclr** (asynchronous clear) port of the LPM counter. This is connected to the **clear** input of the counter circuit via an inverter. Figure 9.20 shows the graphic equivalent of this mapping.

SECTION 9.4 REVIEW PROBLEM

9.4 Write a VHDL code segment that increments a variable called **count** upon detection of a negative edge of an input called **clock.**

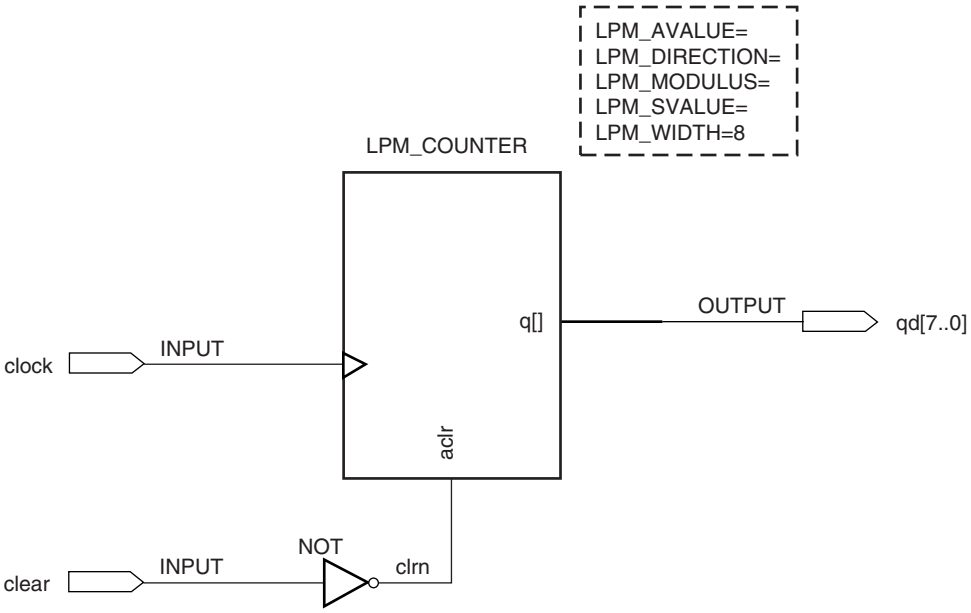


FIGURE 9.20
Graphic Equivalent of an LPM Counter with Active-Low Clear

9.5 Control Options for Synchronous Counters

KEY TERMS

Parallel load A function that allows simultaneous loading of binary values into all flip-flops of a synchronous circuit. Parallel loading can be synchronous or asynchronous.

Presettable counter A counter with a parallel load function.

Clear Reset (synchronous or asynchronous).

Count enable A control function that allows a counter to progress through its count sequence when active and disables the counter when inactive.

Bidirectional counter A counter that can count up or down, depending on the state of a control input.

Terminal count The last state in a count sequence before the sequence repeats (e.g., 1111 is the terminal count of a 4-bit binary UP counter; 0000 is the terminal count of a 4-bit binary DOWN counter).

Ripple carry out or ripple clock out (RCO) An output that produces one pulse with the same period as the clock upon terminal count.

Synchronous counters can be designed with a number of features other than just straight counting. Some of the most common features include:

- Synchronous or asynchronous **parallel load,** which allows the count to be set to any value whenever a *LOAD* input is asserted
- Synchronous or asynchronous **clear** (reset), which sets all of the counter outputs to zero
- **Count enable,** which allows the count sequence to progress when asserted and inhibits the count when deasserted
- **Bidirectional** control, which determines whether the counter counts up or down
- **Output decoding,** which activates one or more outputs when detecting particular states on the counter outputs
- **Ripple carry out or ripple clock out (RCO),** a special case of output decoding that produces a pulse upon detecting the **terminal count,** or last state, of a count sequence.

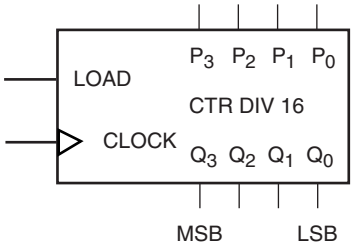
We will examine the implementation of these functions, first as Graphic Design Files in MAX+PLUS II, and then, in the next section, in VHDL, both as behavioral descriptions and as functions of LPM counters.

Parallel Loading

Figure 9.21 shows the symbol of a 4-bit **presettable counter** (i.e., a counter with a parallel load function). The parallel inputs, P_3 to P_0, have direct access to the flip-flops of the counter. When the *LOAD* input is asserted, the values at the *P* inputs are loaded directly into the counter and appear at the *Q* outputs.

> **NOTE**
>
> Parallel loading requires at least two sets of inputs: the load *data* (P_3 to P_0) and the load *command (LOAD)*. If the load function is synchronous, as described below, it also requires a clock input.

FIGURE 9.21
4-bit Counter with Parallel Load



4b_al_sl.scf

Parallel loading can be synchronous or asynchronous. The MAX+PLUS II simulation in Figure 9.22 shows the difference. Two waveforms, QS[3..0] and QA[3..0], represent the outputs of two 4-bit counters with synchronous and asynchronous load, respectively. Both counters have the same clock, load, and *P* inputs. The count is already in progress at the beginning of the simulation window and shows both counters advancing with each clock pulse: 4, 5, 6.

When *LOAD* goes HIGH at 500 ns, the value of P[3..0] (= AH) is loaded into the asynchronously loading counter (QA[3..0]) immediately after a short propagation delay (12.5 ns). The counter with synchronous load (QS[3..0]) is not loaded until the next positive clock edge, shown at 560 ns.

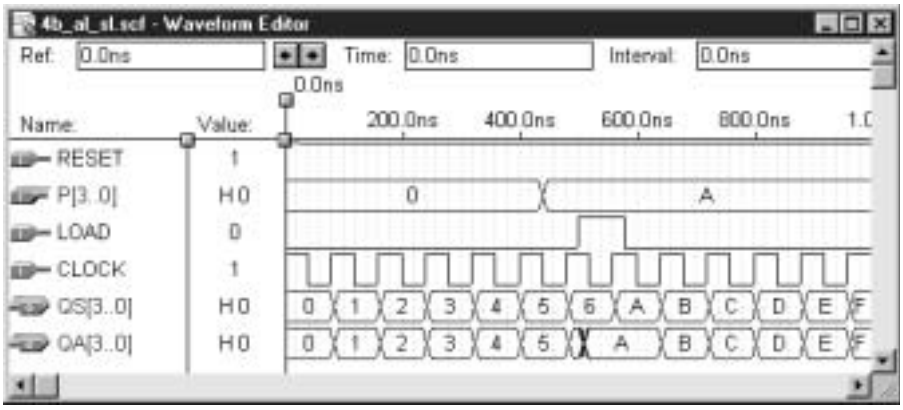


FIGURE 9.22
Synchronous vs. Asynchronous Load

Synchronous Load

The logic diagram of Figure 9.23 shows the concept of synchronous parallel load. Depending on the status of the *LOAD* input, the flip-flop will either count according to its

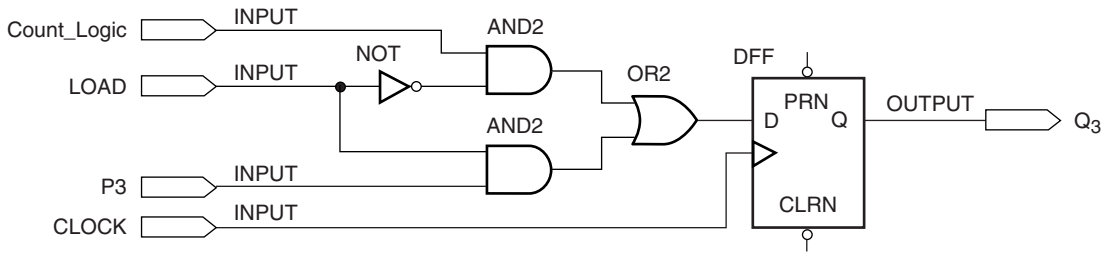


FIGURE 9.23
Count/Load Selection

count logic (the next-state combinational circuit) or load an external value. The flip-flop shown is the most significant bit of a 4-bit binary counter, such as shown in Figure 9.19, but with the count logic represented only by an input pin. (For the fourth bit of a counter, the Boolean equation of the count logic is given by $D_3 = Q_3 \oplus Q_2Q_1Q_0$. It is left out in order to more clearly show the operation of the count/load function select circuit.)

The *LOAD* input selects whether the flip-flop synchronous input will be fed by the count logic or by the parallel input P_3. When $LOAD = 0$, the upper AND gate steers the count logic to the flip-flop, and the count progresses with each clock pulse. When $LOAD = 1$, the lower AND gate loads the logic level at P_3 directly into the flip-flop on the next clock pulse.

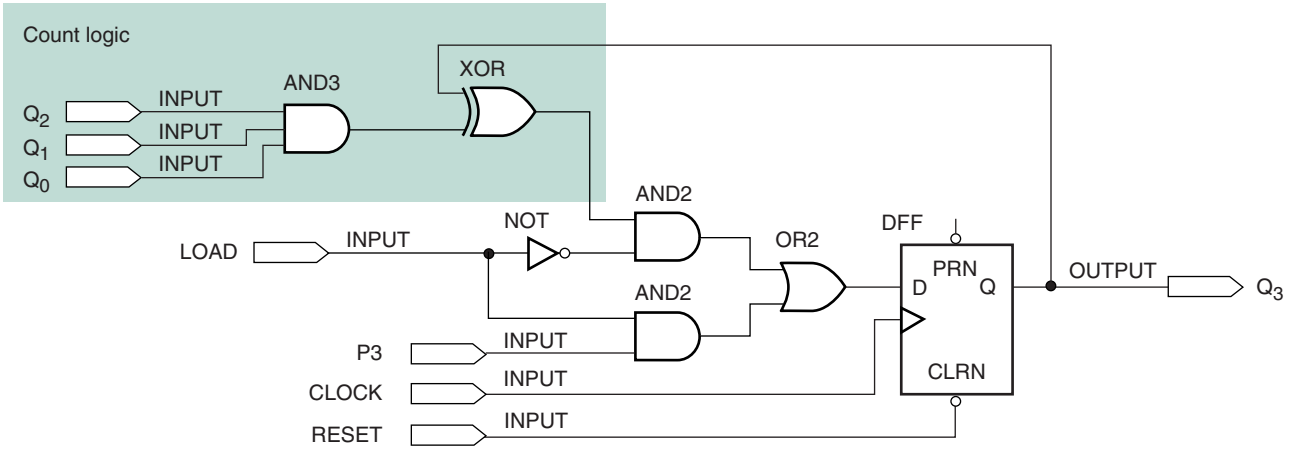


FIGURE 9.24
Counter Element with Synchronous Load and Asynchronous Clear

sl_count.gdf
4bit_sl.gdf
4bit_sl.scf

Figure 9.24 shows the same circuit, but includes the count logic. If we leave out the 3-input AND gate, as in Figure 9.25, we have a circuit that can be used as a general element (called **sl_count**) in a synchronous presettable counter. Figure 9.26 shows the logic diagram of a 4-bit synchronously presettable counter consisting of four instances of the counter element of Figure 9.25 and appropriate AND gates for a synchronous counter. This diagram implements a synchronous counter like that of Figure 9.19, but also incorporates a synchronous load function.

Figure 9.27 shows a simulation of the counter in Figure 9.26. The first 19 clock pulses drive the counter through its normal 4-bit cycle from 0H to FH, then up to 2H. At this point, we set the *LOAD* input HIGH and the value at the *P* inputs (9H) is loaded into the counter on the rising edge of the next clock pulse. An asynchronous *RESET* pulse at 880 ns drives the counter outputs to 0H, after which the count resumes.

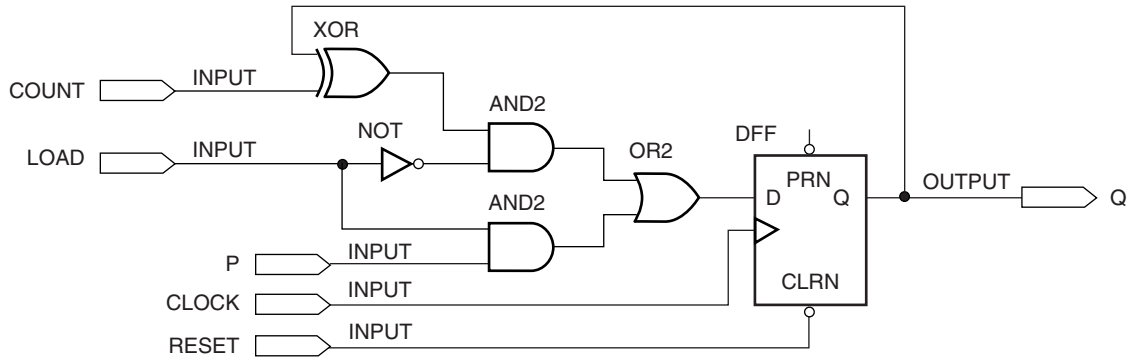


FIGURE 9.25
Counter Element with Synchronous Load and Asychronous Reset (sl_count)

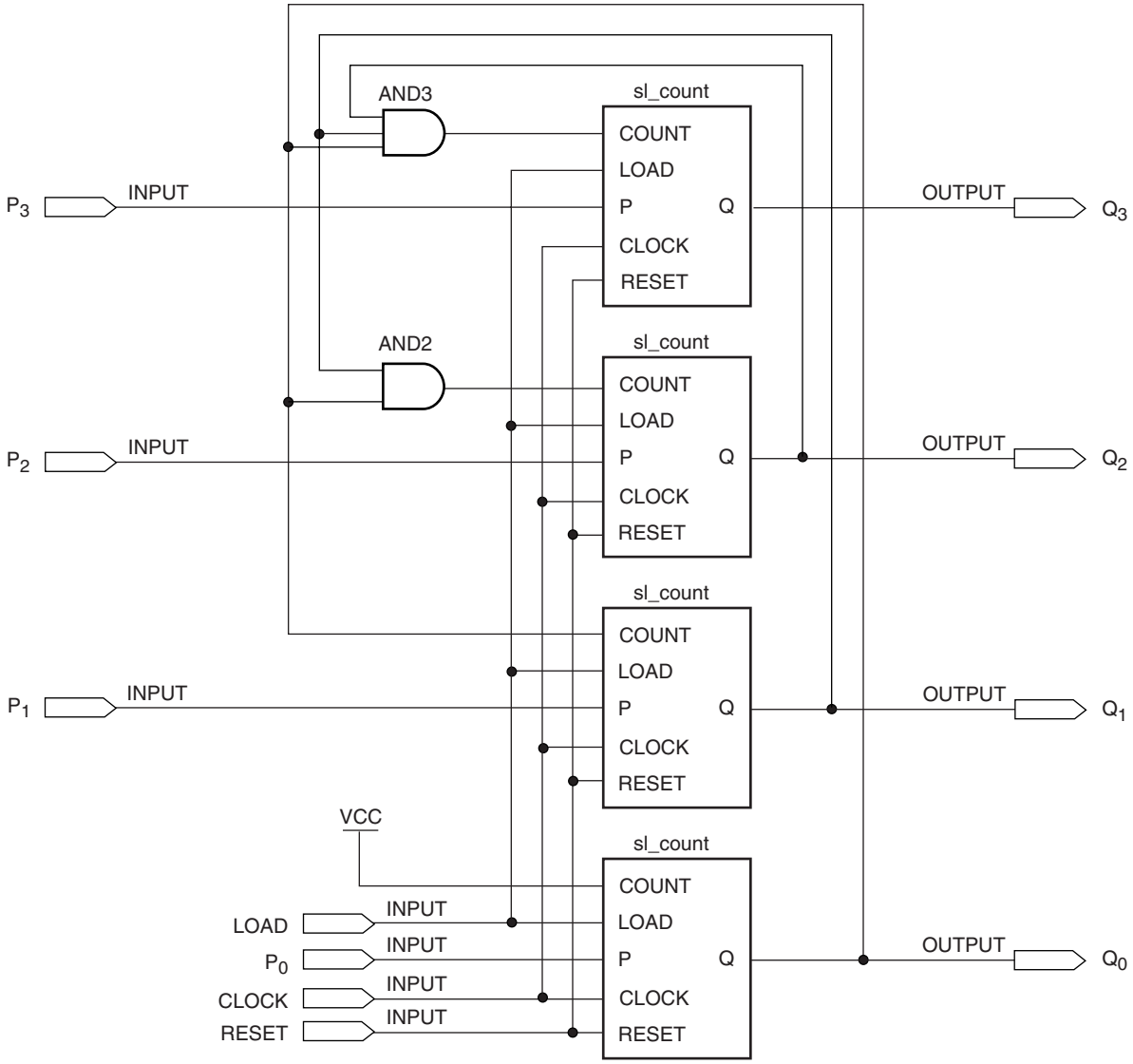


FIGURE 9.26
4-bit Counter with Synchronous Load and Asynchronous Reset

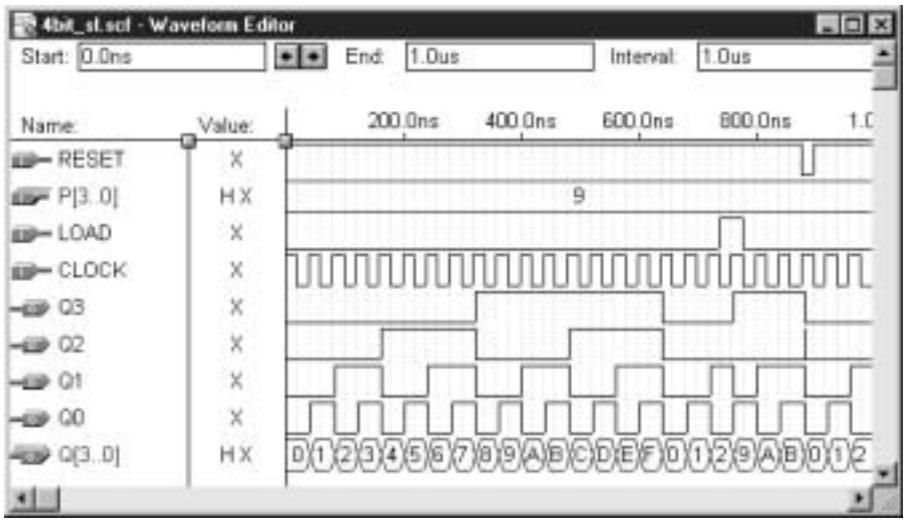


FIGURE 9.27
Simulation of 4-bit Counter with Synchronous Load and Asynchronous Reset

Asynchronous Load

The asynchronous load function of a counter makes use of the asynchronous preset and clear inputs of the counter's flip-flops. Figure 9.28 shows the circuit implementation of the asynchronous load function, without any count logic.

When *ALOAD* (Asynchronous LOAD) is HIGH, both NAND gates in Figure 9.28 are enabled. If the *P* input is HIGH, the output of the upper NAND gate goes LOW, activating the flip-flop's asynchronous *PRESET* input, thus setting $Q = 1$. The lower NAND gate has a HIGH output, thus deactivating the flip-flop's *CLEAR* input.

If *P* is LOW the situation is reversed. The upper NAND output is HIGH and the lower NAND has a LOW output, activating the flip-flop's *CLEAR* input, resetting *Q*. Thus, *Q* will be the same value as *P* when the *ALOAD* input is asserted. When *ALOAD* is not asserted (= 0), both NAND outputs are HIGH and thus do not activate either the preset or clear function of the flip-flop.

Figure 9.29 shows the asynchronous load circuit with an asynchronous clear (reset) function added. The flip-flop can be cleared by a logic LOW either from the *P* input (via the lower NAND gate) or the *CLEAR* input pin. The clear function disables the upper NAND gate when it is LOW, preventing the flip-flop from being cleared and preset simultaneously. This extra connection also ensures that the clear function has priority over the load function.

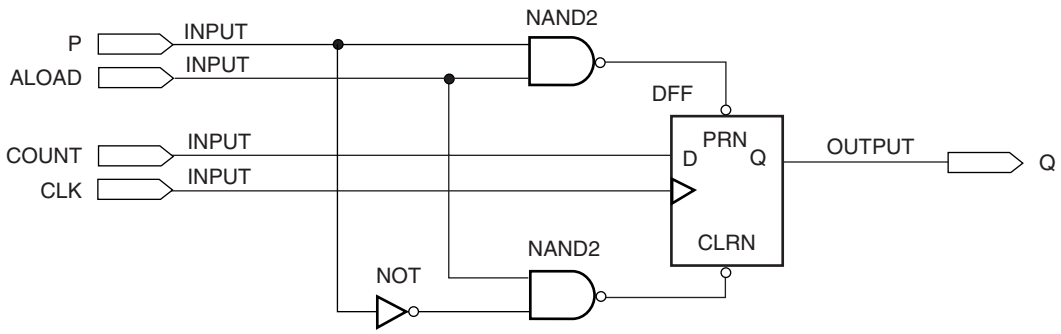


FIGURE 9.28
Asynchronous LOAD Element

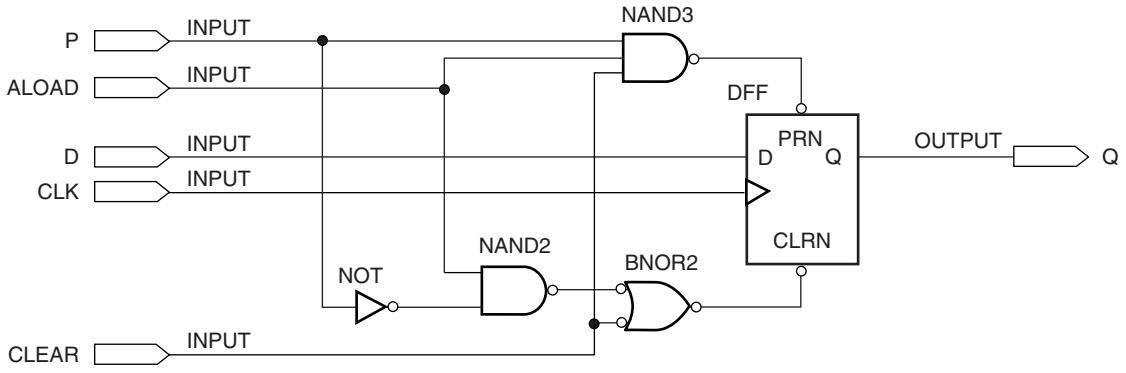


FIGURE 9.29
Asynchronous LOAD Element with Asynchronous Clear

EXAMPLE 9.6

al_count.gdf

Use MAX+PLUS II to redraw the circuit in Figure 9.29 to create a general element called **al_count** that can be used in a synchronous counter with asynchronous load and clear. (Refer to Figure 9.25 for a similar element with *synchronous* load.)

Solution Figure 9.30 shows the modified circuit, which includes an XOR gate for part of the count logic. The remainder of the count logic must be supplied externally to this element for each bit of the counter.

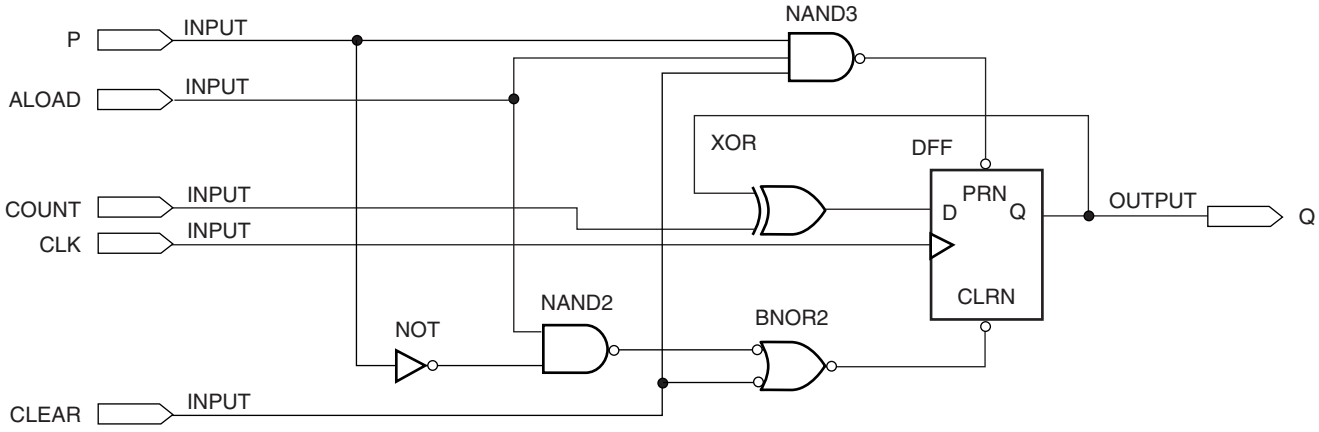


FIGURE 9.30
Example 9.6
Counter Element with Asynchronous Load and Clear (al_count)

EXAMPLE 9.7

4bit_al.gdf
4bit_al.scf

Draw a circuit with four instances of **al_count** (from Example 9.6) to make a 4-bit synchronous counter with asynchronous load and reset. Create a simulation that tests the function of the counter.

Solution Figure 9.31 shows the circuit. (Compare this circuit to the counter with synchronous load in Figure 9.26. This difference between the two is in the load function, not the count logic.)

The Boolean function applied to the *COUNT* input of each instance of **al_count** consists of the logical product of all previous output bits. ($COUNT_3 = Q_2Q_1Q_0$, $COUNT_2 = Q_1Q_0$, $COUNT_1 = Q_0$, $COUNT_0 = 1$.) When combined with the XOR at the *COUNT* input

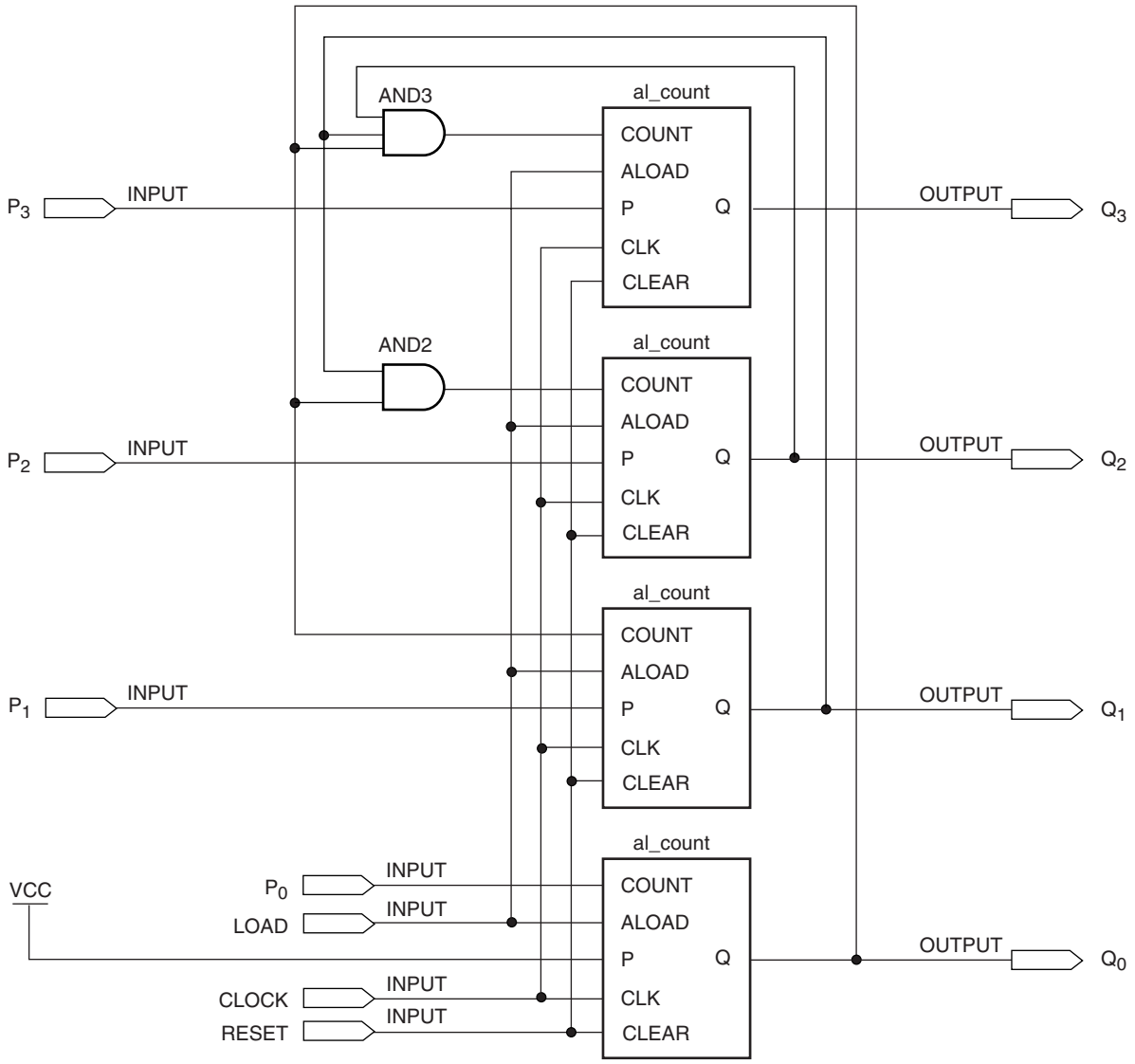
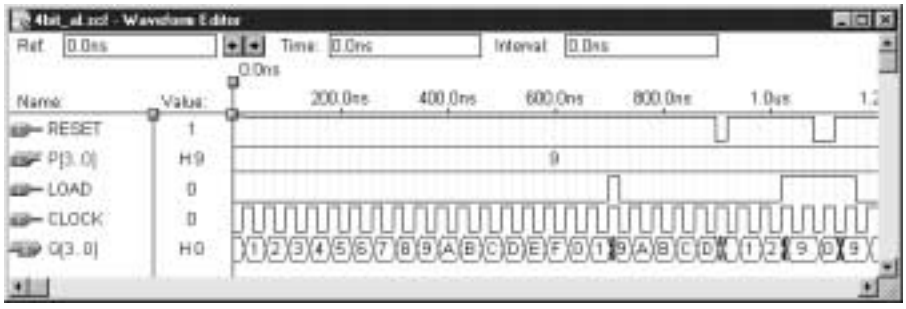


FIGURE 9.31
Example 9.7
4-bit Counter with Asynchronous Load and Reset

of each element, this yields the Boolean equations for a binary counter based on D flip-flops, as derived in Example 9.5. The circuitry inside each instance of **al_count** also generates the asynchronous load and clear functions.

Figure 9.32 shows a MAX+PLUS II simulation of the counter. The counter cycles through its full range and continues. A pulse at 700 ns loads the counter with the value 9H ($= 1001_2$), after which the count continues from that point.

FIGURE 9.32
Example 9.7
Simulation of a 4-bit Counter with Asynchronous Load and Reset



The reset pulse at 900 ns clears the counter. The *LOAD* pulse starting at 1.02 μs shows how the load function has precedence over the count function. When *LOAD* is asserted, 9H is loaded and the count does not increase until *LOAD* is deasserted. The *RESET* pulse at 1.08 μs overrides both load and count functions. When *RESET* is deasserted, 9H is asynchronously reloaded.

Count Enable


4bit_sle.gdf
4bit_sle.scf

The counter elements in Figures 9.25 **(sl_count)** and 9.30 **(al_count)** are just D flip-flops configured for switchable toggle operation with additional circuitry for load and clear functions. Normally, when these elements are used in synchronous counters, the count progresses when the input to the element's XOR gate goes HIGH. In other words, the count progresses when the counter element is switched from a no change to a toggle mode.

In order to arrest the count sequence, we must disable the count logic of the counter circuit. Figure 9.33 shows a simple modification to the 4-bit counter circuit of Figure 9.26 that can achieve this function. Each AND gate has an extra input which is used to enable or inhibit the count logic function to each flip-flop.

Figure 9.34 shows a simulation of the counter. Note that the count progresses normally when *COUNT_ENA* is HIGH and stops when *COUNT_ENA* is LOW, even though the clock pulses remain constant throughout the simulation.

Also note that the count enable has no effect on the synchronous load and asynchronous reset functions. In the latter part of the simulation, the count stops at AH ($Q_3Q_2Q_1Q_0 = 1010_2$), when *COUNT_ENA* goes LOW. At 760 ns, the synchronous load function loads the value of 9H into the counter. The counter stays at this value, even after LOAD is no longer active, since the count is still disabled. At 880 ns, an asynchronous reset pulse clears the counter. The count resumes on the first clock pulse after *COUNT_ENA* goes HIGH again.

Bidirectional Counters

Figure 9.35 shows the logic diagram of a 4-bit synchronous DOWN counter. Its count sequence starts at 1111 and counts backwards to 0000, then repeats. The Boolean equations for this circuit will not be derived at this time, but will be left for an exercise in an end-of-chapter problem.

We can intuitively analyze the operation of the counter if we understand that the upper three flip-flops will each toggle when their associated XOR gates have a HIGH input from the rest of the count logic.


element.gdf

Q_0 is set to toggle on each clock pulse. Q_1 toggles whenever Q_0 is LOW (every second clock pulse, at states 1110, 1100, 1010, 1000, 0110, 0100, 0010, and 0000). Q_2 toggles when Q_1 AND Q_0 are LOW (1100, 1000, 0100, and 0000). Q_3 toggles when Q_2 AND Q_1 AND Q_0 are LOW (1000 and 0000). The result of this analysis can be represented by a timing diagram, such as the simulation shown in Figure 9.36. As we expect, the counter will count down from 1111 (FH) to 0000 (0H) and repeat.

We can create a bidirectional counter by including a circuit to select count logic for an UP or DOWN sequence. Figure 9.37 shows a basic synchronous counter element that can be used to create a synchronous counter. The element is simply a D flip-flop configured for switchable toggle mode.

Four of these elements can be combined with selectable count logic to make a 4-bit bidirectional counter, as shown in Figure 9.38. Each counter element has a pair of AND-shaped gates and an OR gate to steer the count logic to the XOR in the element. When $DIR = 1$, the upper gate in each pair is enabled and the lower gates disabled,

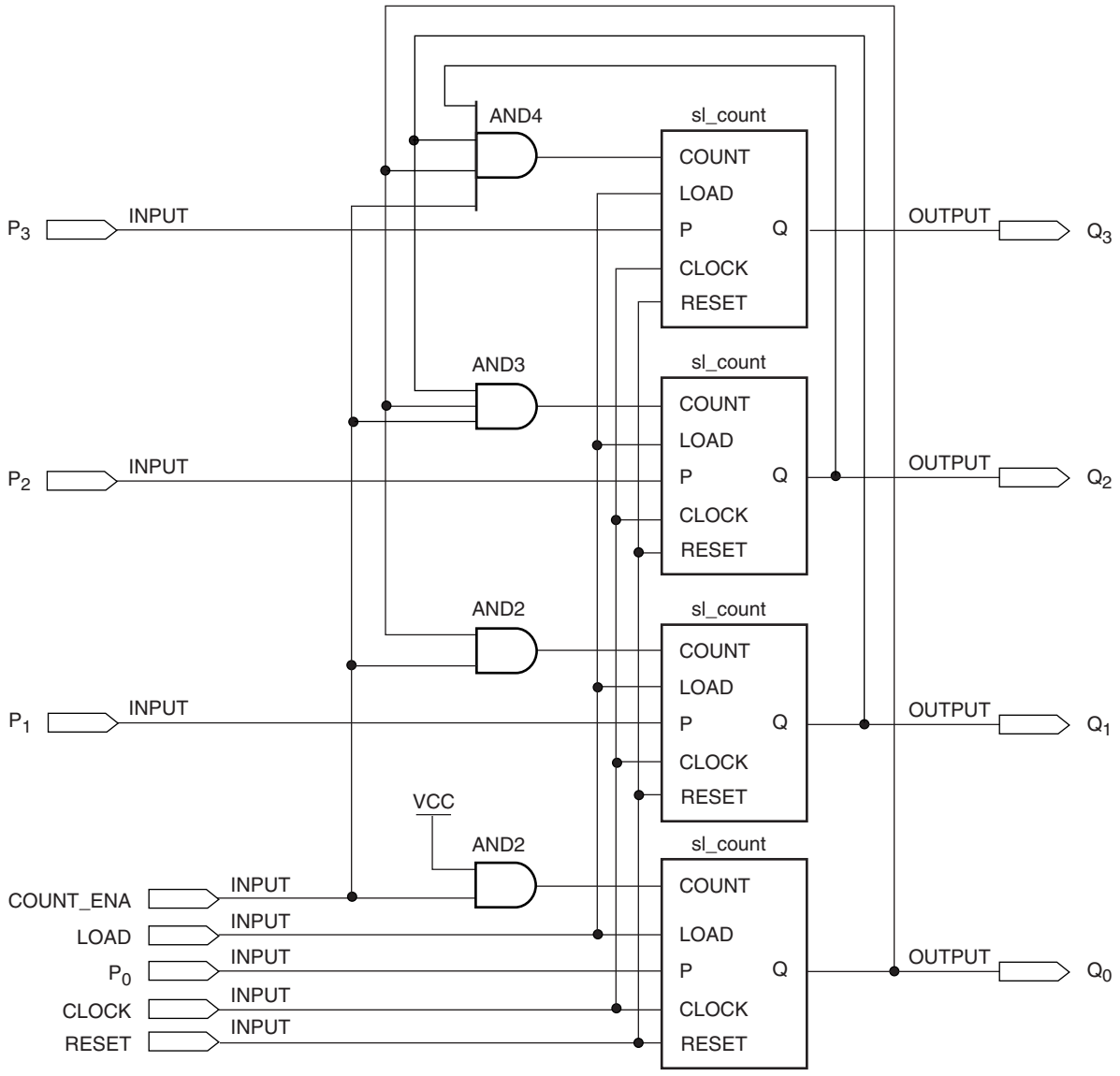
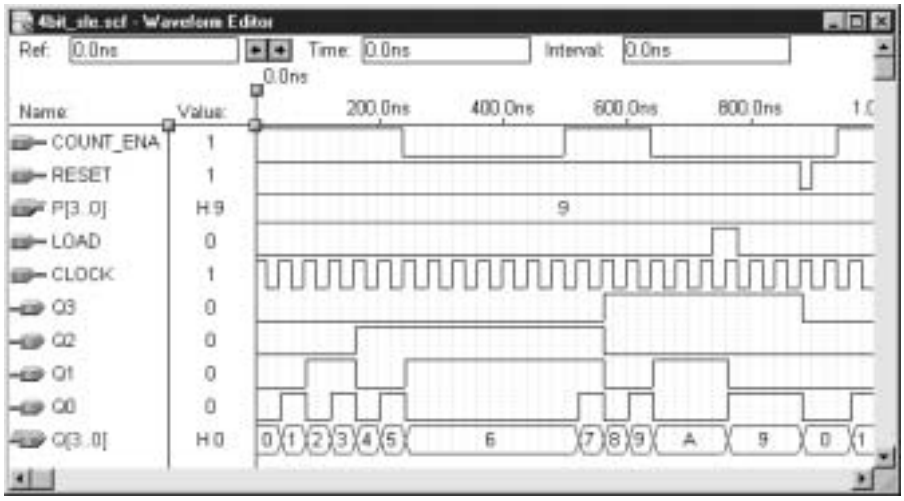


FIGURE 9.33
4-bit Counter with Synchronous Load, Asynchronous Reset, and Count Enable

FIGURE 9.34
Simulation of 4-bit Counter with Synchronous Load, Asynchronous Reset, and Count Enable



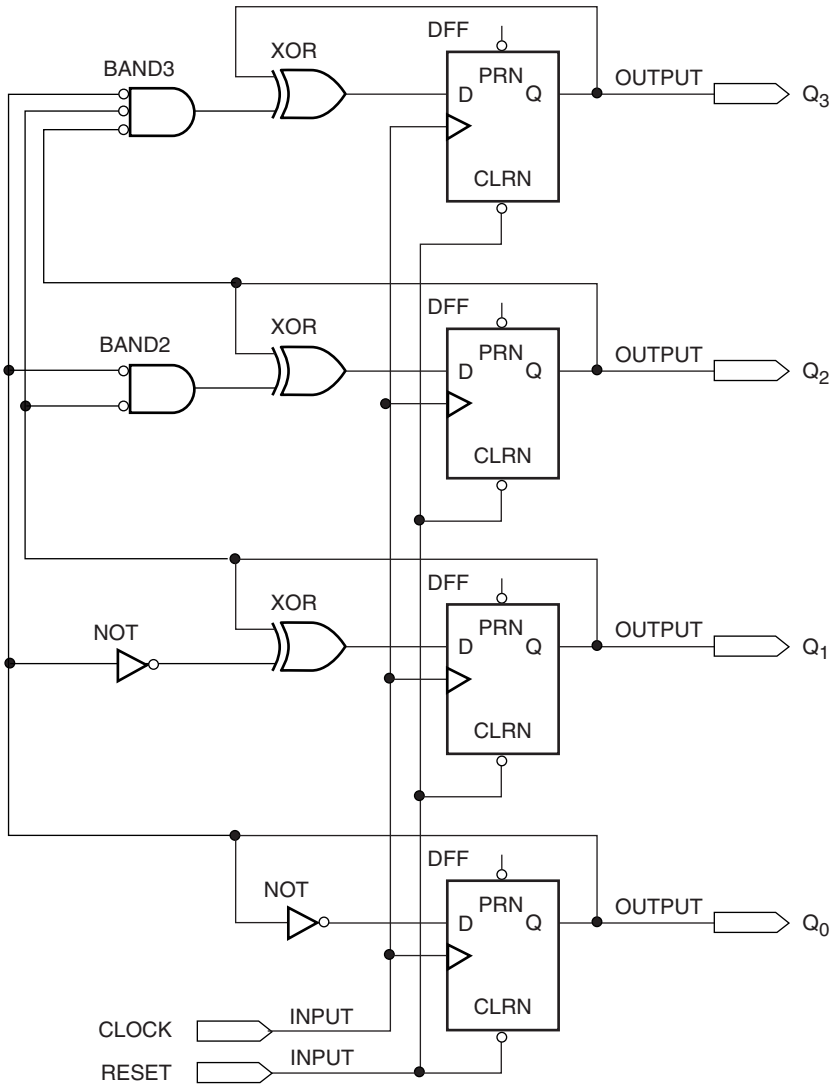


FIGURE 9.35
4-bit Synchronous DOWN Counter

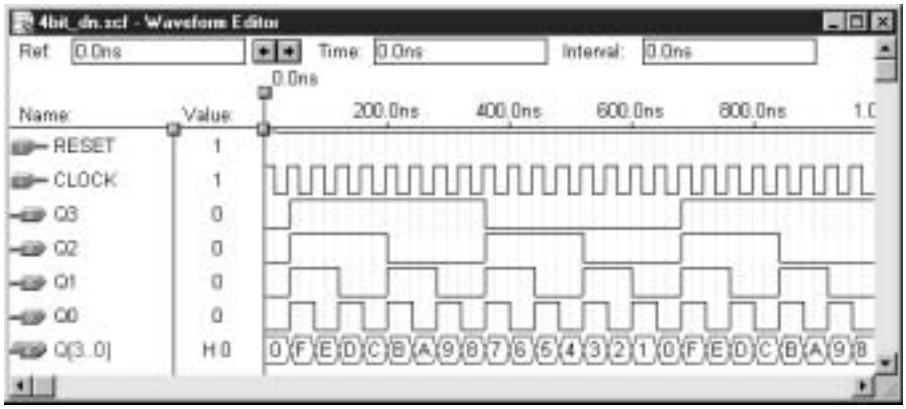


FIGURE 9.36
4-bit DOWN Counter Simulation

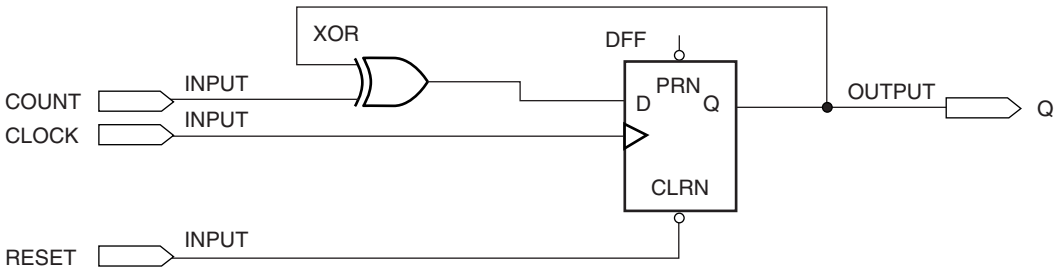


FIGURE 9.37
Synchronous Counter Element (T Flip-Flop)

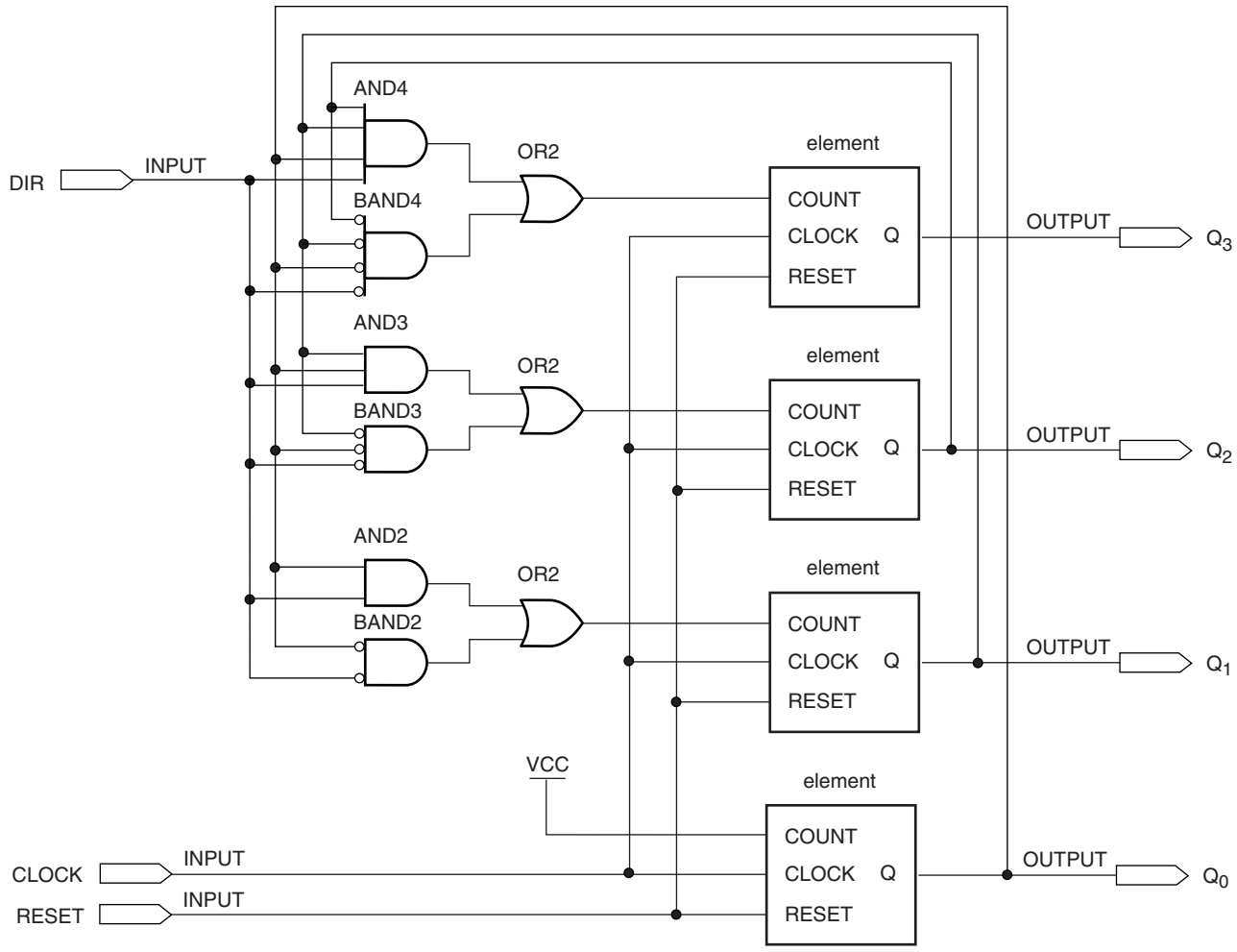


FIGURE 9.38
4-bit Bidirectional Counter

steering the UP count logic to the counter element. When *DIR* = 0, the lower gate in each pair is enabled, steering the DOWN count logic to the counter element. The directional function can also be combined with the load and count enable functions, as was shown for unidirectional UP counters.

Figure 9.39 shows a simulation of the bidirectional counter of Figure 9.38. The waveforms show the UP count when *DIR* is HIGH and the DOWN count when *DIR* is LOW.

4bit_dir.gdf
4bit_dir.scf

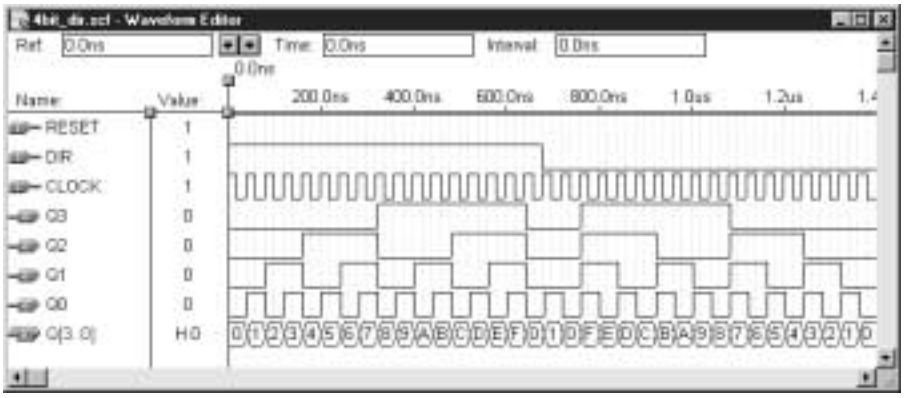


FIGURE 9.39
Simulation of 4-bit Bidirectional Counter

Decoding the Output of a Counter

Figure 9.40 shows a graphic design file of a 4-bit bidirectional counter with an output decoder. The counter is the one shown in Figure 9.38, represented as a logic circuit symbol. The decoder component **decode16** is a module written in VHDL, as listed below.

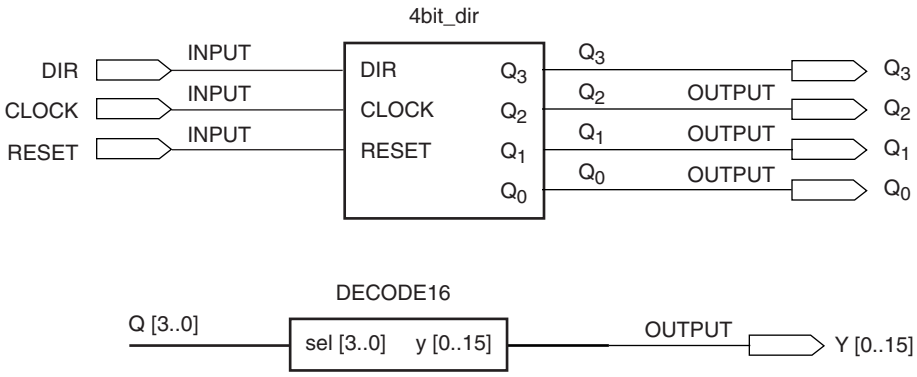


FIGURE 9.40
4-bit Bidirectional Counter with Output Decoder

```
-- decode16.vhd

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY decode16 IS
   PORT(
      sel : IN   INTEGER RANGE 0 to 15;
      y   : OUT  BIT_VECTOR (0 to 15));
END decode16;

ARCHITECTURE a OF decode16 IS
BEGIN
   WITH sel SELECT
      y <=   x"7FFF" WHEN 0,
             x"BFFF" WHEN 1,
             x"DFFF" WHEN 2,
             x"EFFF" WHEN 3,
             x"F7FF" WHEN 4,
             x"FBFF" WHEN 5,
```

CD: decode16.vhd
4bit_dcd.gdf
4bit_dcd.scf

```
                    x"FDFF" WHEN 6,
                    x"FEFF" WHEN 7,
                    x"FF7F" WHEN 8,
                    x"FFBF" WHEN 9,
                    x"FFDF" WHEN 10,
                    x"FFEF" WHEN 11,
                    x"FFF7" WHEN 12,
                    x"FFFB" WHEN 13,
                    x"FFFD" WHEN 14,
                    x"FFFE" WHEN 15,
                    X"FFFF" WHEN others;
END a;
```

The decoder has 16 outputs, one for each state of the counter. For each state, one and only one output will be low. (Refer to the section on binary decoders in Chapter 5 for a more detailed description of *n*-line-to-*m*-line binary decoders.)

Figure 9.41 shows a portion of the simulation waveforms (i.e., only the count value and the decoder outputs) for the circuit in Figure 9.40. As the count progresses up or down, as shown by the waveform for Q[3..0], the decoder outputs respond by going LOW in sequence.

Output decoders for binary counters can also be configured to have active HIGH outputs. In this case, one and only one output would be HIGH for each output state of the counter.

Terminal Count and RCO

A special case of output decoding is a circuit that will detect the **terminal count,** or last state, of a count sequence and activate an output to indicate this state. The terminal count depends on the count sequence. A 4-bit binary UP counter has a terminal count of 1111; a 4-bit binary DOWN counter has a terminal count of 0000. A circuit to detect these conditions must detect the *maximum value* of an UP count and the *minimum value* of a DOWN count.

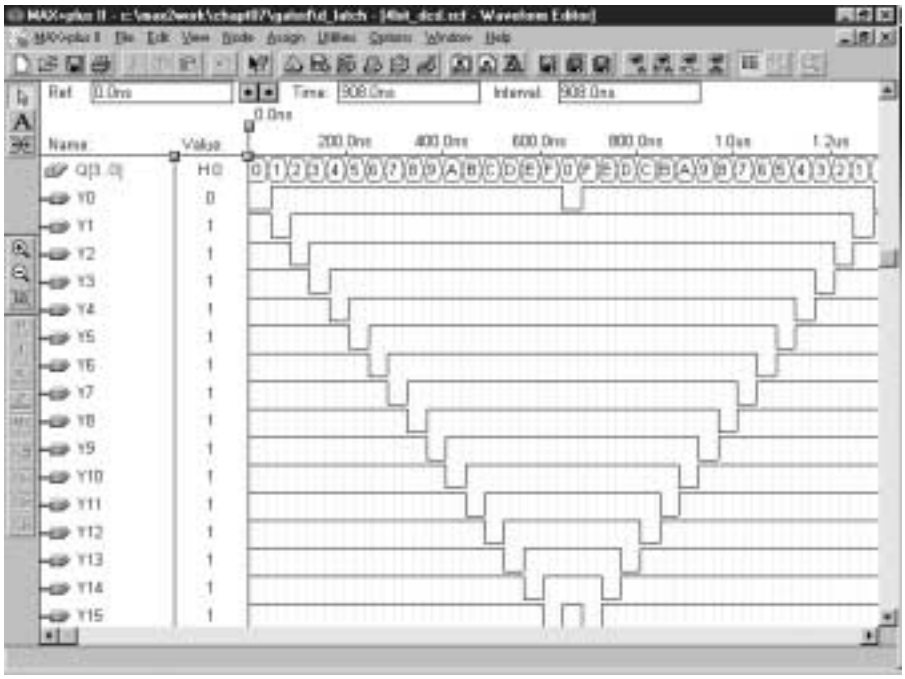


FIGURE 9.41
Simulation of 4-bit Decoder

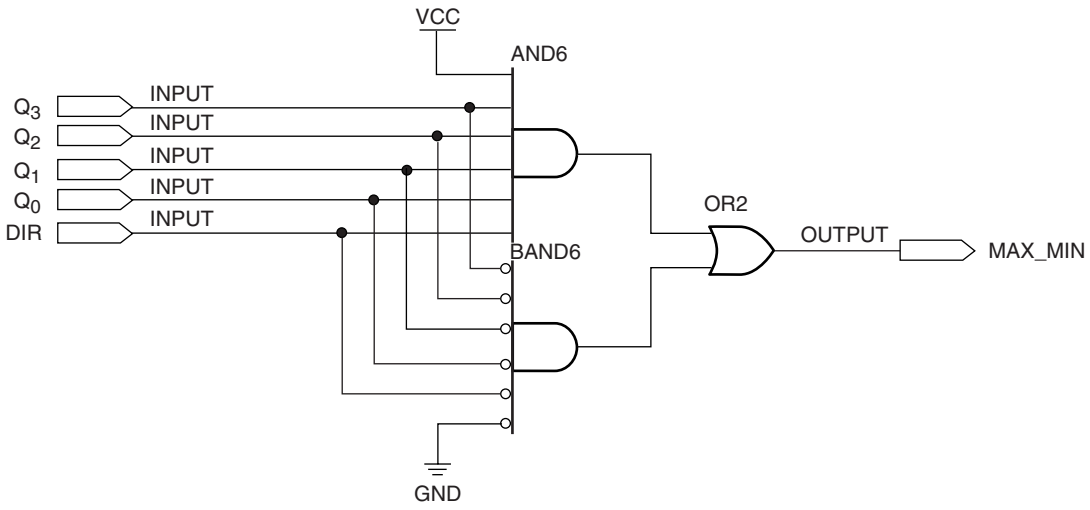


FIGURE 9.42
Terminal Count Decoder for a 4-bit Bidirectional Counter

term_dcd.gdf

4bit_rco.gdf
4bit_rco.scf

The decoder shown in Figure 9.42 fulfills both of these conditions. The directional input *DIR* enables the upper gate when HIGH and the lower gate when LOW. Thus, the upper gate generates a HIGH output when $DIR = 1$ AND $Q_3Q_2Q_1Q_0 = 1111$. The lower gate generates a HIGH when $DIR = 0$ AND $Q_3Q_2Q_1Q_0 = 0000$.

Figure 9.43 shows the terminal count decoder combined with a 4-bit bidirectional counter. The decoder is also used to enable a NAND gate output that generates an *RCO* signal. RCO stands for ripple carry out or ripple clock out. The purpose of *RCO* is to produce exactly one clock pulse upon terminal count and have the positive edge of *RCO* at the end of the counter cycle, for a counter that has a positive edge-triggered clock.

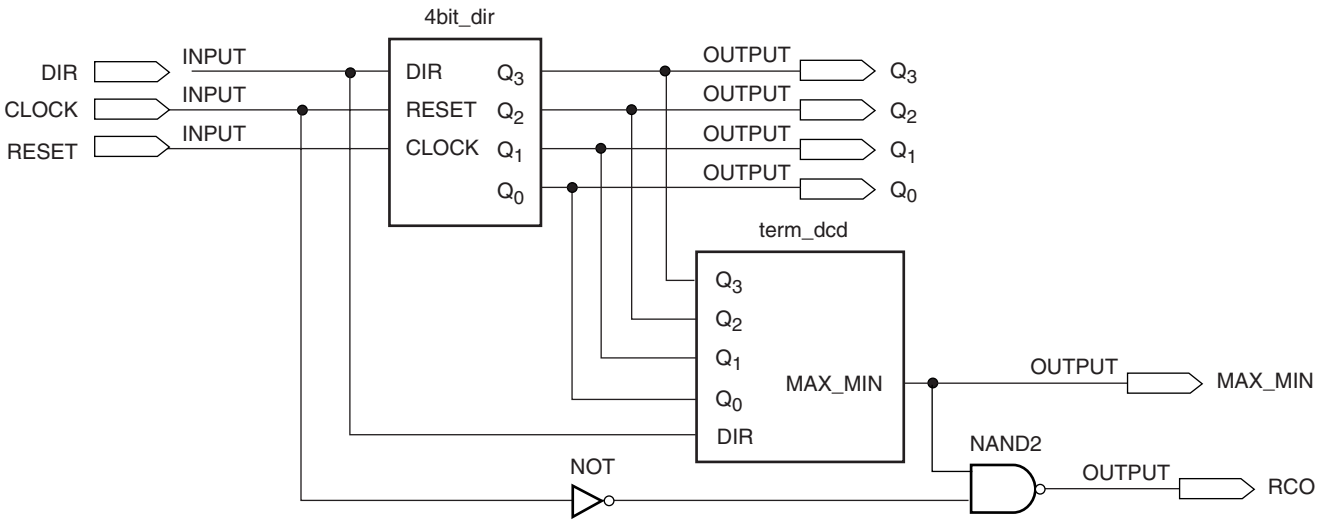


FIGURE 9.43
4-bit Bidirectional Counter with Terminal Count Detection

This function is generally found in counters with a fixed number of bits (i.e., fixed-function counter chips, not PLDs) and is used to asynchronously clock a further counter stage, as in Figure 9.44. This allows us to extend the width of the counter beyond the number of bits available in the fixed-function device. This is not necessary when designing synchronous counters in programmable logic, but is included for the sake of completeness.

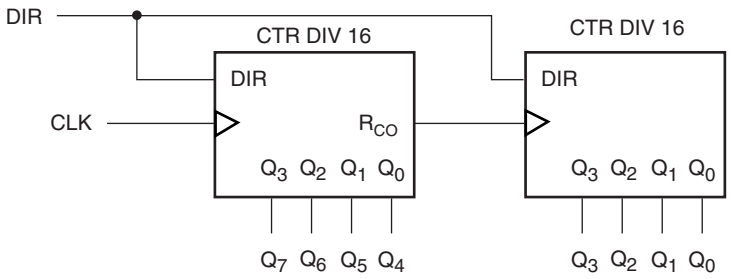


FIGURE 9.44
Counter Expansion Using RCO

The NAND gate in Figure 9.43 is enabled upon terminal count and passes the clock signal through to *RCO*. The NAND output sits HIGH when inhibited. The clock is inverted in the RCO circuit so that when the NAND gate inverts it again, the circuit generates a clock pulse in true form.

Figure 9.45 shows the simulation of the circuit of Figure 9.43. In the first half of the simulation, the counter is counting DOWN. The terminal count decoder output, *MAX_MIN,* goes HIGH when $Q_3Q_2Q_1Q_0 = 0000$. *RCO* generates a pulse at that time. For the second half, the counter is counting UP. *MAX_MIN* is HIGH when $Q_3Q_2Q_1Q_0 = 1111$ and *RCO* generates a pulse at that time.

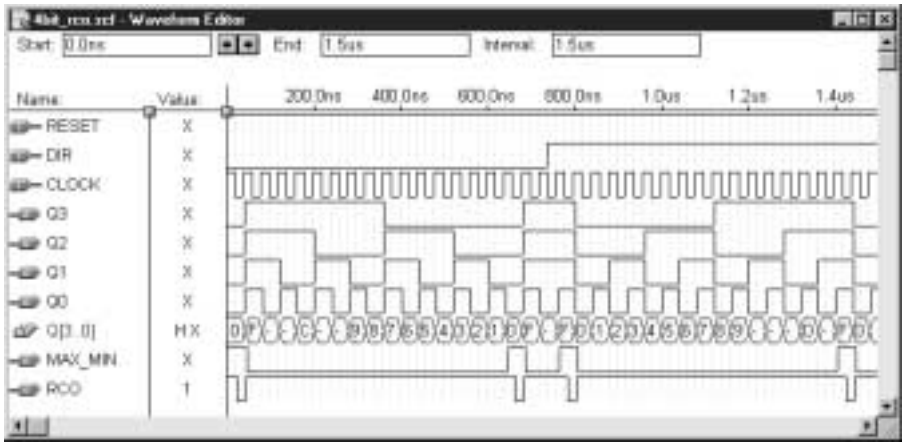


FIGURE 9.45
Simulation of a 4-bit Bidirectional Counter with Terminal Count Detection

Note that the *RCO* pulse appears to be half the width of the *MAX_MIN* pulse. Although the NAND gate that generates *RCO* is enabled for the whole *MAX_MIN* pulse, the clock input is HIGH for the first half-period, which is the same as the *RCO* inhibit level.

The positive edge of *RCO* is at the *end* of the pulse. The idea is to synchronize the positive edge of the clock with the positive edge of *RCO*. However, since the RCO decoder is combinational, a propagation delay of about 7 ns is introduced.

SECTION 9.5 REVIEW PROBLEM

9.5 Figure 9.46 shows two presettable counters, one with asynchronous load and clear, the other with synchronous load and clear. The counter with asynchronous functions has a 4-bit output labeled *QA*. The synchronously loaded counter has a 4-bit output labeled *QS*. The load *and* reset inputs to both counters are *active LOW.*

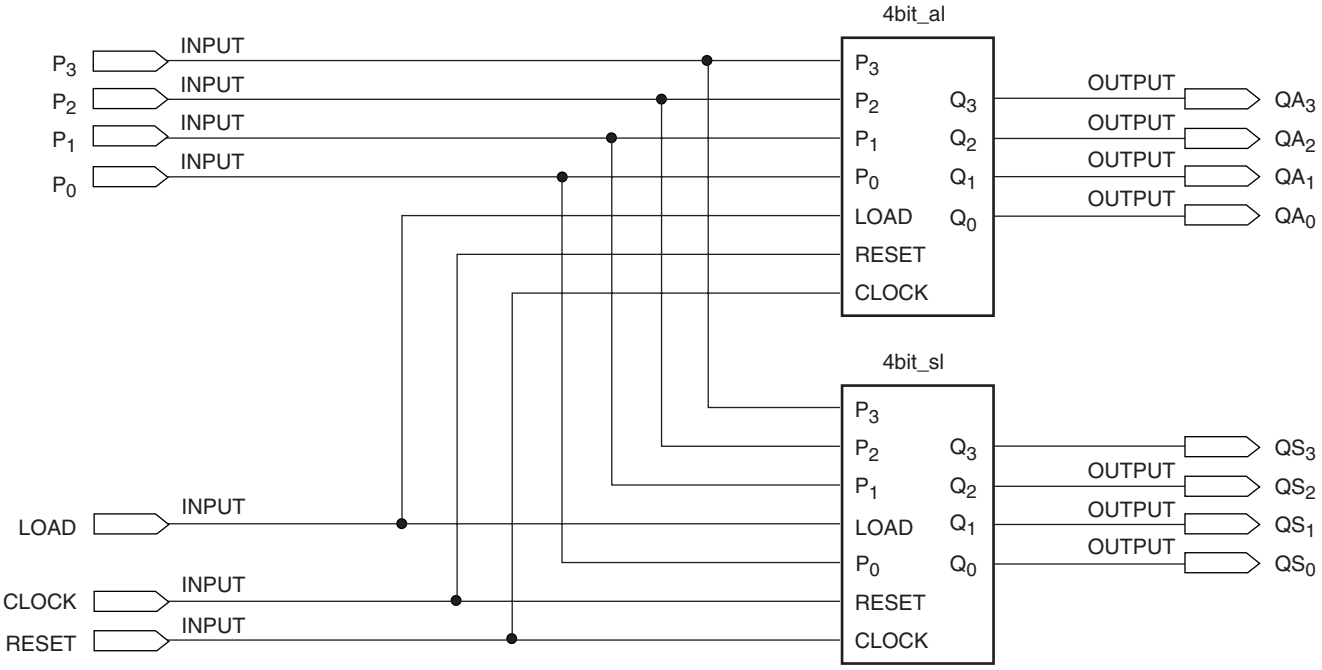


FIGURE 9.46
Section Review Problem 9.5
Two Presettable Counters

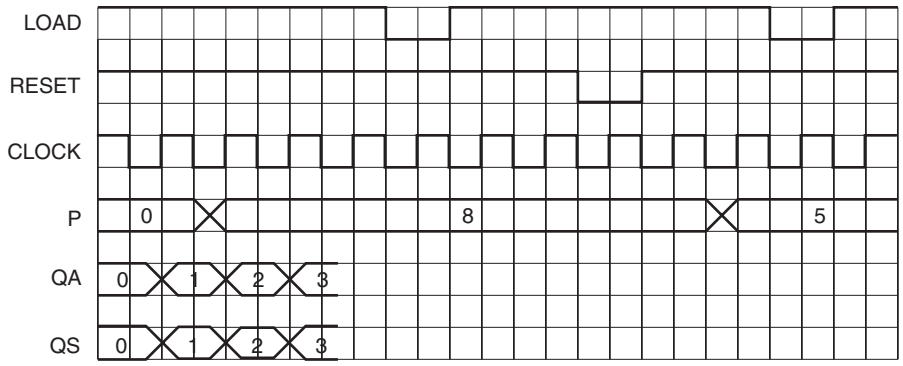


FIGURE 9.47
Timing Diagram for Counters in Figure 9.46

Figure 9.47 shows a partial timing diagram for the counters. Complete the diagram.

9.6 Programming Presettable and Bidirectional Counters in VHDL

The presettable counters and bidirectional counters described in the previous section can be easily implemented in VHDL, either as behavioral descriptions or as LPM components. We will initially examine the behavioral descriptions of two counters, one with asynchronous load and clear and one with synchronous load and clear. We will then examine some options available in the module **lpm_counter.**

Behavioral Description

The following lists the VHDL code for an 8-bit bidirectional counter with count enable, terminal count decoding, and asynchronous load and clear:


pre_ct8a.vhd

```
ENTITY pre_ct8a IS
  PORT (
       clk, count_ena          : IN  BIT;
       clear, load, direction  : IN  BIT;
       p                       : IN  INTEGER RANGE 0 TO 255;
       max_min                 : OUT BIT;
       qd                      : OUT INTEGER RANGE 0 TO 255);
END pre_ct8a;

ARCHITECTURE a OF pre_ct8a IS
 BEGIN
 PROCESS (clk, clear, load)
     VARIABLE cnt : INTEGER RANGE 0 TO 255;
     BEGIN
          IF (clear = '0') THEN                  -- Asynchronous clear
              cnt := 0;
          ELSIF (load = '1' and clear = '1') THEN -- Asynchronous load
              cnt := p;
          ELSE
              IF (clk'EVENT AND clk = '1') THEN
                  IF (count_ena = '1' and direction = '0') THEN
                       cnt := cnt - 1;
                  ELSIF (count_ena = '1' and direction = '1') THEN
                      cnt := cnt + 1;
                  END IF;
              END IF;
          END IF;
          qd <= cnt;

          -- Terminal count decoder
          IF (cnt = 0 and direction = '0') THEN
              max_min <=  '1';
          ELSIF (cnt = 255 and direction = '1') THEN
              max_min <=  '1';
          ELSE
              max_min <=  '0';
          END IF;
     END PROCESS;
END a;
```

The load and clear functions of this counter are asynchronous, so these identifiers are part of the sensitivity list of the PROCESS statement; the statements in the process will execute if there is a change on the clear, load, or clock inputs. Load and clear are checked by IF statements, independently of the clock. Since load and clear are checked first, they have precedence over the clock. Clear has precedence over load since load can only activate if clear is not active.

If clear and load are not asserted, the clock status is checked by a clause in an IF statement: (`IF (clk'EVENT and CLK = '1') THEN`). If this condition is true, then a count variable is incremented or decremented, depending on the states of a count enable input and a directional control input. If the count enable input is not asserted, the count is neither incremented nor decremented.

The count value is assigned to the counter outputs by the signal assignment statement (`qd <= cnt;`) after the clear, load, clock, count enable, and direction inputs have been evaluated. Possible results from the signal assignment are:

- **qd = 0** (clear = 0),
- **qd = p** (load = 1 AND clear = 1),
- **increment qd** (count_ena = 1 AND direction = 1),
- **decrement qd** (count_ena = 1 AND direction = 0), or
- **no change on qd** (count_ena = 0).

The terminal count is decoded by determining the count direction and value of the count variable. If the count is UP and the count value is maximum (255_{10} = FFH) or the count is DOWN and the count value is minimum (0 = 00H), a terminal count decoder output called **max_min** goes HIGH.

The code for the same 8-bit counter, but with synchronous clear and load, is shown next.

```
ENTITY pre_ct8s IS
    PORT (
         clk, count_ena          : IN BIT;
         clear, load, direction  : IN BIT;
         p                       : IN INTEGER RANGE 0 TO 255;
         max_min                 : OUT BIT;
         qd                      : OUT INTEGER RANGE 0 TO 255);
END pre_ct8s;

ARCHITECTURE a OF pre_ct8s IS
    BEGIN
    PROCESS (clk)
        VARIABLE cnt : INTEGER RANGE 0 TO 255;
        BEGIN
            IF (clk'EVENT AND clk = '1') THEN
                IF (clear = '0') THEN -- Synchronous clear
                    cnt := 0;
                ELSIF (load = '1') THEN -- Synchronous load
                    cnt := p;
                ELSIF (count_ena = '1' and direction = '0') THEN
                    cnt := cnt - 1;
                ELSIF (count_ena = '1' and direction = '1') THEN
                    cnt := cnt + 1;
                END IF;
            END IF;
            qd <= cnt;

            -- Terminal count decoder
            IF (cnt = 0 and direction = '0') THEN
                max_min <= '1';
            ELSIF (cnt = 255 and direction = '1') THEN
                max_min <= '1';
            ELSE
                max_min <= '0';
            END IF;
        END PROCESS;
    END a;
```

pre_ct8s.vhd

The PROCESS statement in the synchronous counter has only one identifier in its sensitivity list—that of the clock input. Load and clear status are not evaluated until after the


pre_ct8a.scf

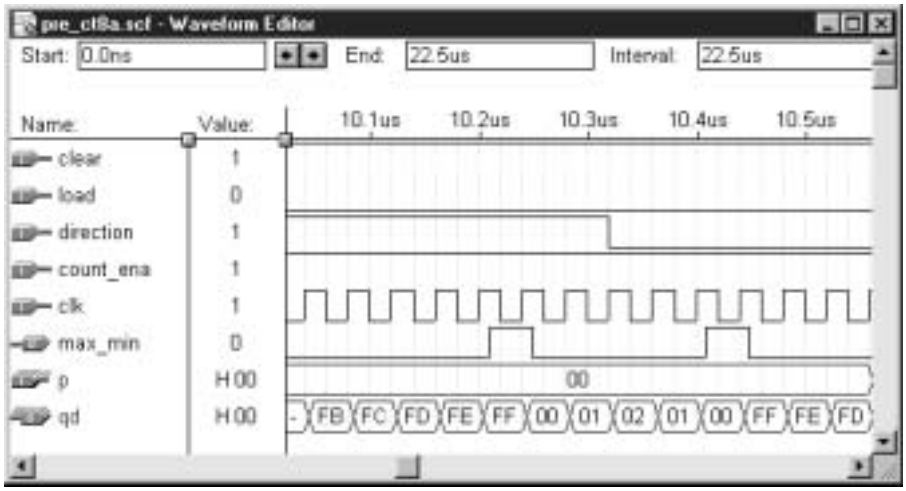


FIGURE 9.48
Simulation Detail of 8-bit VHDL Counter (Bidirectional with Terminal Count Detection)

process checks for a positive clock edge. Otherwise the code is the same as for the asynchronously loading counter.

Figure 9.48 shows a detail of a simulation of the asynchronously loading counter. It shows the point where the count rolls over from FFH to 00H and activates the **max_min** output. The directional output changes shortly after this point and shows the terminal count decoding for a DOWN count, the point where the counter rolls over from 00H to FFH. In the UP count, **max_min** is HIGH when the counter output is FFH, but not 00H. In the DOWN count, **max_min** goes HIGH when the counter output is 00H, but not FFH.

Figure 9.49 shows the operation of the asynchronous load and clear functions. Figure 9.50 show the synchronous load and clear. The inputs are identical for each simulation; each has two pairs of load pulses and a pair of clear pulses. The first pulse of each pair is arranged so that it immediately *follows* a positive clock edge; the second pulse of each pair immediately *precedes* a positive clock edge.

In the counter with asynchronous load and clear, these functions are activated by the first pulse of each pair and again on the second pulse. For the counter with synchronous load and clear, only the second pulse of each pair has an effect, since the load and clear functions must be active *during* or *just prior to* an active clock edge, in order to satisfy

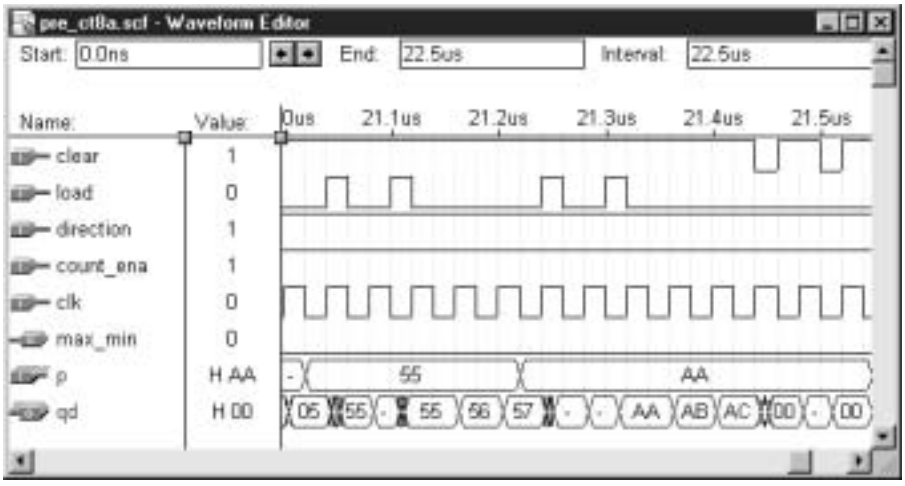


FIGURE 9.49
Simulation Detail of 8-bit VHDL Counter with Asynchronous Load and Clear

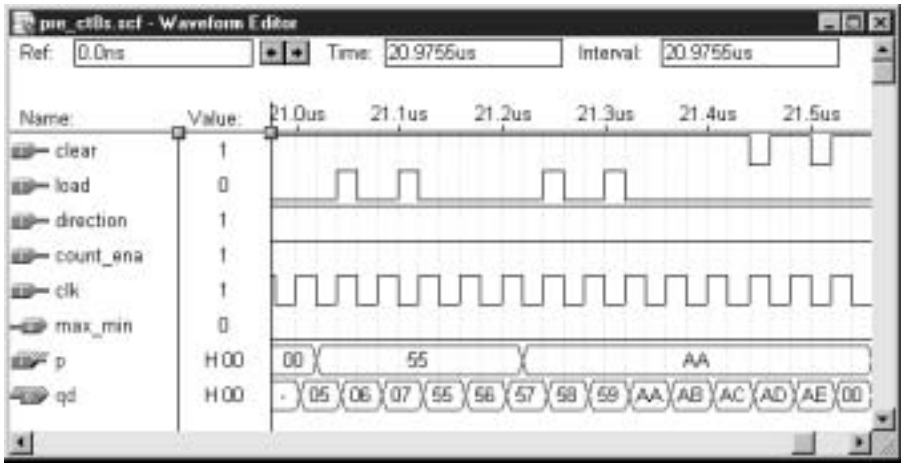


FIGURE 9.50
Simulation Detail of 8-bit VHDL Counter with Synchronous Load and Clear

setup-time requirements of the counter flip-flops. The end of the load and clear pulse can correspond to the positive clock edge, as the flip-flop hold time is zero.

Also note that the counter with synchronous load and clear has no intermediate glitch states on its outputs. (The simulation for the asynchronously loading counter shows glitch states on output *qd* at 21.04 μs, 21.10 μs, 21.22 μs, and 21.44 μs. Refer to the section on Synchronous versus Asynchronous Circuits in Chapter 7 for further discussion of intermediate states in asynchronous circuits.)

Figures 9.51 and 9.52 show further simulation details for our two VHDL counters. Both show the priority of the load, clear, and count enable functions. Both diagrams show that load and clear are independent of count enable and that clear has precedence over load. Again note that the counter with synchronous load and clear is free of intermediate glitch states.

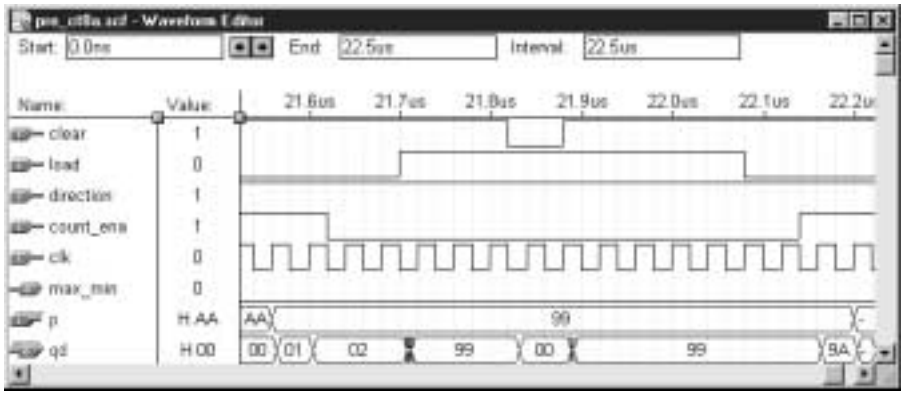


FIGURE 9.51
Simulation Detail of 8-bit VHDL Counter Showing Priority of Count Enable, Asynchronous Load, and Asynchronous Clear

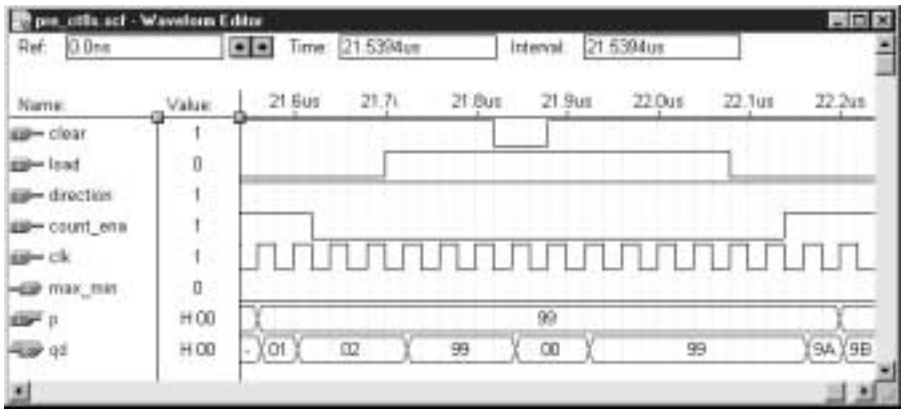


FIGURE 9.52
Simulation Detail of 8-bit VHDL Counter Showing Priority of Count Enable, Synchronous Load, and Synchronous Clear

LPM Counters

Earlier in this chapter, we saw how a parameterized counter from the Library of Parameterized Modules (LPM) could be used as a simple 8-bit counter. The component **lpm_counter** has a number of other functions that can be implemented using specific ports and parameters. These functions are indicated in Table 9.13.

Table 9.13 Available Functions of an LPM counter

Function	Ports	Parameters	Description
Basic count operation	clock, q []	LPM_WIDTH	Output **q[]** increases by one with each positive clock edge. **LPM_WIDTH** is the number of output bits.
Synchronous load	sload, data []	none	When **sload** = 1, output **q[]** goes to the value at input **data[]** on the next positive clock edge. **data[]** has the same width as **q[].**
Synchronous clear	sclr	none	When **sclr** = 1, output **q[]** goes to zero on positive clock edge.
Synchronous set	sset	LPM_SVALUE	When **sset** = 1, output goes to value of **LPM_SVALUE** on positive clock edge. If **LPM_SVALUE** is not specified, **q[]** goes to all 1s.
Asynchronous load	aload, data[]	none	Output goes to value at **data[]** when **aload** = 1.
Asynchronous clear	aclr	none	Output goes to zero when **aclr** = 1.
Asynchronous set	aset	LPM_AVALUE	Output goes to value of **LPM_AVALUE** when **aset** = 1. If **LPM_AVALUE** is not specified, outputs all go HIGH when **aset** = 1.
Directional control	updown	LPM_DIRECTION	Optional direction control. Default direction is UP. Only one of **updown** and **LPM_DIRECTION** can be used. **updown** = 1 for UP count, **updown** = 0 for DOWN count. **LPM_DIRECTION** = "UP", "DOWN", or "DEFAULT"
Count enable	cnt_en	none	When **cnt_en** = 1, count proceeds upon positive clock edges. No effect on other synchronous functions **(sload, sclr, sset).** Defaults to "enabled" when not specified.
Clock enable	clk_en	none	All synchronous functions are enabled when **clk_en** = 1. Defaults to "enabled" when not specified.
Modulus control	none	LPM_MODULUS	Modulus of counter is set to value of **LPM_MODULUS**
Output decoding (GDF or AHDL only; not available in VHDL)	eq[15..0]	none	Sixteen active-HIGH decoded outputs, one for each internal counter value from 0 to 15.

The only ports that are required by an LPM counter are **clock,** and one of **q[]** (counter outputs) or **eq[]** (decoder outputs). The only required parameter is **LPM_WIDTH,** which specifies the number of counter output bits. All other ports and parameters are optional, although certain ones must be used together. (For instance, ports **sload** and **data[]** are optional, but both must be used for the synchronous load function.) If unused, a port or parameter will be held at a default logic level.

To use any of the functions of an LPM component in a VHDL file, we use a component instantiation statement and specify the required parameters in a generic map and the ports in a port map.

NOTE

The VHDL component declaration, shown below, indicates that all parameters except LPM_WIDTH are defined as having type STRING, which requires the parameter value to be written in double quotes, even if numeric. (e.g, `LPM_MODULUS => "12"`). Since LPM_WIDTH is defined as type POSITIVE (i.e., any integer > 0) it must be written without quotes (e.g., `LPM_WIDTH => 8`). Default values of all ports and parameters are also included in the component declaration (e.g., `clk_en: IN STD_LOGIC := '1'`; the default value of the clock enable input is '1'). The LPM component declaration can also be found in the MAX+PLUS II Help menu **(Help; Megafunctions/LPM; lpm_counter).**

VHDL Component Declaration for lpm_counter:

```
COMPONENT lpm_counter
   GENERIC (LPM_WIDTH: POSITIVE;
      LPM_MODULUS: STRING := "UNUSED";
      LPM_AVALUE: STRING:= "UNUSED";
      LPM_SVALUE: STRING := "UNUSED";
      LPM_DIRECTION: STRING := "UNUSED";
      LPM_TYPE: STRING := "L_COUNTER";
      LPM_PVALUE: STRING := "UNUSED";
      LPM_HINT : STRING := "UNUSED");
   PORT (data: IN STD_LOGIC_VECTOR (LPM_WIDTH-1 DOWNTO 0) := (OTHERS => '0');
      clock: IN STD_LOGIC;
      cin: IN STD_LOGIC := '0';
      clk_en: IN STD_LOGIC := '1';
      cnt_en: IN STD_LOGIC := '1';
      updown: IN STD_LOGIC := '1'
      sload: IN STD_LOGIC := '0';
      sset: IN STD_LOGIC := '0';
      sclr: IN STD_LOGIC := '0';
      aload: IN STD_LOGIC := '0';
      aset: IN STD_LOGIC := '0';
      aclr: IN STD_LOGIC := '0';
      cout: OUT STD_LOGIC;
      q: OUT STD_LOGIC_VECTOR (LPM_WIDTH-1 DOWNTO 0) );
END COMPONENT;
```

EXAMPLE 9.8

Write a VHDL file for an 8-bit LPM counter with ports for the following functions: asynchronous load, asynchronous clear, directional control, and count enable.

Solution The required VHDL file is shown below. Note that no behavioral descriptions are required for the functions, only a mapping from the defined port names to the entity inputs and outputs.

```
-- pre_lpm8
-- 8-bit presettable counter with asynchronous clear and load,
-- count enable, and a directional control port

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


pre_lpm8.vhd

```
ENTITY pre_lpm8 IS
   PORT (
        clk, count_ena    : IN     STD_LOGIC;
        clear, load, direction    : IN    STD_LOGIC;
        p                 : IN     STD_LOGIC_VECTOR(7 downto 0);
        qd                : OUT    STD_LOGIC_VECTOR(7 downto 0));
END pre_lpm8;

ARCHITECTURE a OF pre_lpm8 IS
BEGIN
   counter1: lpm_counter
       GENERIC MAP (LPM_WIDTH => 8)
       PORT MAP ( clock  => clk,
                  updown => direction,
                  cnt_en => count_ena,
                  data   => p,
                  aload  => load,
                  aclr   => clear,
                  q      => qd);
END a;
```

EXAMPLE 9.9

Write a VHDL file that uses an LPM counter to generate a DOWN counter with a modulus of 500. Create a MAX+PLUS II simulation file to verify the counter's operation.

Solution A mod-500 counter requires nine bits since $2^8 < 500 < 2^9$. Since the counter always counts DOWN, we can use the parameter LPM_DIRECTION to specify the DOWN counter rather than using an unnecessary port. The required VHDL code is given below.

Note that the value of LPM_WIDTH is written without quotes, since it is defined as type POSITIVE in the component declaration. LPM_MODULUS and LPM_DIRECTION are written in double quotes, since the component declaration defines them as type STRING.


mod5c_lpm.vhd
mod5c_lpm.scf

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

ENTITY mod5c_lpm IS
   PORT (
        clk : IN      STD_LOGIC;
        q   : OUT     STD_LOGIC_VECTOR (8 downto 0) );
END mod5c_lpm;

ARCHITECTURE a OF mod5c_lpm IS
BEGIN
   counter1: lpm_counter
       GENERIC MAP(LPM_WIDTH     => 9,
                   LPM_DIRECTION => "DOWN",
                   LPM_MODULUS   => "500")
       PORT MAP ( clock => clk,
                  q     => q);
END a;
```

Figure 9.53 shows a partial simulation of the counter, indicating the point at which the output rolls over from 0 to 499 (decimal).

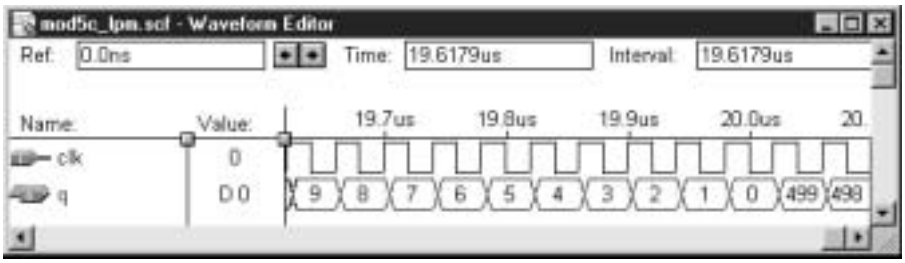


FIGURE 9.53
Example 9.9
Partial Simulation of a Mod-500 LPM DOWN Counter

If we are designing a counter for the Altera UP-1 circuit board, we can simulate the on-board oscillator by choosing a clock period of 40 ns, which corresponds to a clock frequency of 25 MHz. The default simulation period is from 0 to 1 μs, which only gives 1 μs ÷ 40 ns/clock period = 25 clock periods. This is not enough time to show the entire count cycle. The minimum value for the end of the simulation time is:

$$40 \text{ ns/clock period} \times 500 \text{ clock periods} = 20000 \text{ ns} = 20 \text{ μs}.$$

If we wish to see a few clock cycles past the recycle point, we can set the simulation end time to 20.1 μs. (In the MAX+PLUS II Simulator window, select **File menu; End Time.** Enter the value **20.1us** (no spaces) into the **Time** window and click **OK**.)

To view the count waveform, **q**, in decimal rather than hexadecimal, select the waveform by clicking on it. Either right-click to get a pop-up menu or select **Enter Group** from the simulator **Node** menu, as in Figure 9.54. This will bring up the **Enter Group** dialog box shown in Figure 9.55. Select **DEC** (for decimal) and click **OK**.

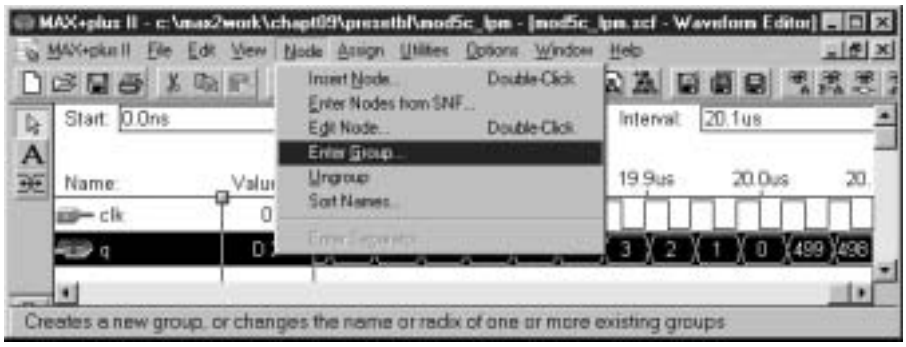


FIGURE 9.54
Selecting a Group in a MAX+PLUS II Simulation

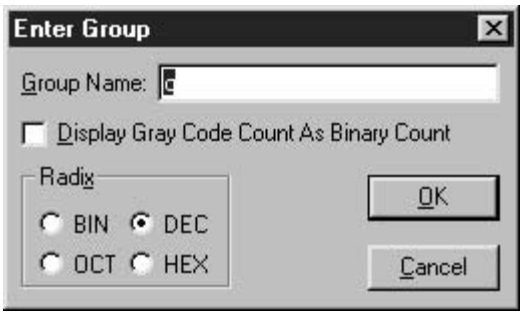


FIGURE 9.55
Changing the Name or Radix of a Group

EXAMPLE 9.10

Write a VHDL file that instantiates a 12-bit LPM counter with asynchronous clear and synchronous set functions. Design the counter to set to 2047 (decimal). Create a simulation to verify the counter operation.

Solution The required VHDL file is:

```
-- sset_lpm.vhd
-- 12-bit LPM counter with sset and aclr

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

ENTITY sset_lpm IS
    PORT(
            clk         : IN  STD_LOGIC;
            clear, set  : IN  STD_LOGIC;
            q           : OUT STD_LOGIC_VECTOR (11 downto 0) );
END sset_lpm;

ARCHITECTURE a OF sset_lpm IS
BEGIN
   counter1: lpm_counter
      GENERIC MAP    (LPM_WIDTH => 12,
                      LPM_SVALUE => "2047")
      PORT MAP ( clock   => clk,
                 sset    => set,
                 aclr    => clear,
                 q       => q);
END a;
```

sset_lpm.vhd
sset_lpm.scf

Figure 9.56 shows the simulation file of the counter. The full count sequence would take over 160 μs, so we will assume the count portion of the design works properly. Only the set and clear functions are fully simulated. The count waveform is shown in decimal.

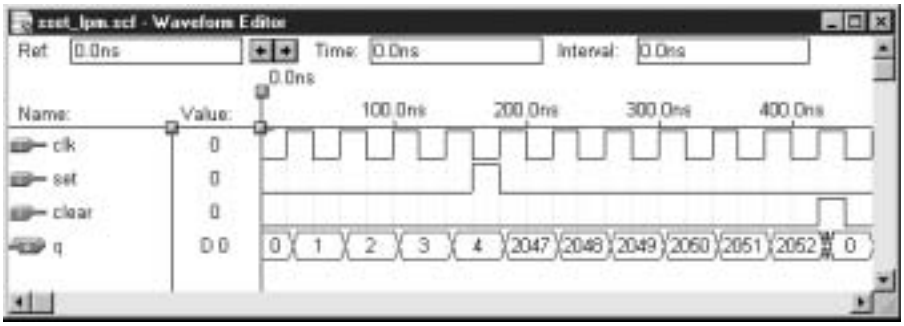


FIGURE 9.56
Example 9.10
Simulation of a 12-bit Counter with Synchronous Set to 2047 and Asynchronous Clear

SECTION 9.6 REVIEW PROBLEM

9.6 The first part of a VHDL process statement includes a sensitivity list: `PROCESS (sensitivity list)`. How should this be written for a counter with asynchronous clear and for a counter with synchronous clear?

9.7 Shift Registers

KEY TERMS

Shift register A synchronous sequential circuit that will store and move *n*-bit data, either serially or in parallel, in *n* flip-flops.

SRG*n* Abbreviation for an *n*-bit shift register (e.g., SRG4 indicates a 4-bit shift register).

Serial shifting Movement of data from one end of a shift register to the other at a rate of one bit per clock pulse.

Parallel transfer Movement of data into all flip-flops of a shift register at the same time.

Rotation Serial shifting of data with the output(s) of the last flip-flop connected to the synchronous input(s) of the first flip-flop. The result is continuous circulation of the same data.

Right shift A movement of data from the left to the right in a shift register. (Right is defined in MAX+PLUS II as toward the LSB.)

Left shift A movement of data from the right to the left in a shift register. (Left is defined in MAX+PLUS II as toward the MSB.)

Bidirectional shift register A shift register that can serially shift bits left or right according to the state of a direction control input.

Parallel-load shift register A shift register that can be preset to any value by directly loading a binary number into its internal flip-flops.

Universal shift register A shift register that can operate with any combination of serial and parallel inputs and outputs (i.e., serial in/serial out, serial in/parallel out, parallel in/serial out, parallel in/parallel out). A universal shift register is often bidirectional, as well.

A **shift register** is a synchronous sequential circuit used to store or move data. It consists of several flip-flops, connected so that data are transferred into and out of the flip-flops in a standard pattern.

Figure 9.57 represents three types of data movement in three 4-bit shift registers. The circuits each contain four flip-flops, configured to move data in one of the ways shown.

Figure 9.57a shows the operation of **serial shifting.** The stored data are taken in one at a time from the input and moved one position toward the output with each applied clock pulse.

Parallel transfer is illustrated in Figure 9.57b. As with the synchronous parallel load function of a presettable counter, data move simultaneously into all flip-flops when a clock pulse is applied. The data are available in parallel at the register outputs.

Rotation, depicted in Figure 9.57c, is similar to serial shifting in that data are shifted one place to the right with each clock pulse. In this operation, however, data are continuously circulated in the shift register by moving the rightmost bit back to the leftmost flip-flop with each clock pulse.

Serial Shift Registers

srg4_sr.gdf
srg4_sr.scf

Figure 9.58 shows the most basic shift register circuit: the serial shift register, so called because data are shifted through the circuit in a linear or serial fashion. The circuit shown consists of four D flip-flops connected in cascade and clocked synchronously.

For a D flip-flop, *Q* follows *D*. The value of a bit stored in any flip-flop *after* a clock pulse is the same as the bit in the flip-flop to its left *before* the pulse. The result is that when a clock pulse is applied to the circuit, the contents of the flip-flops move one position to the

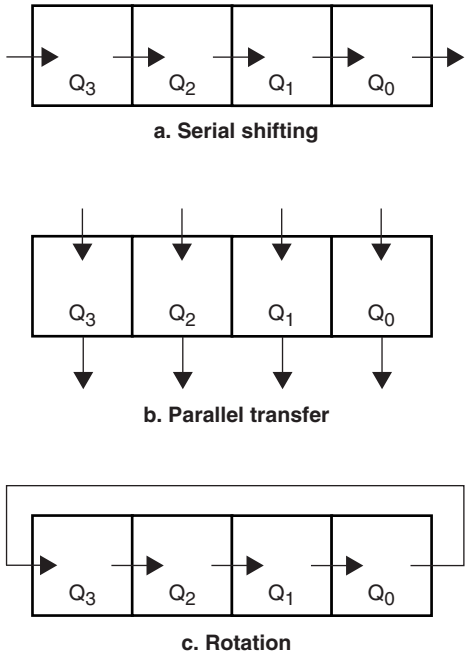


FIGURE 9.57
Data Movement in a 4-bit Shift Register

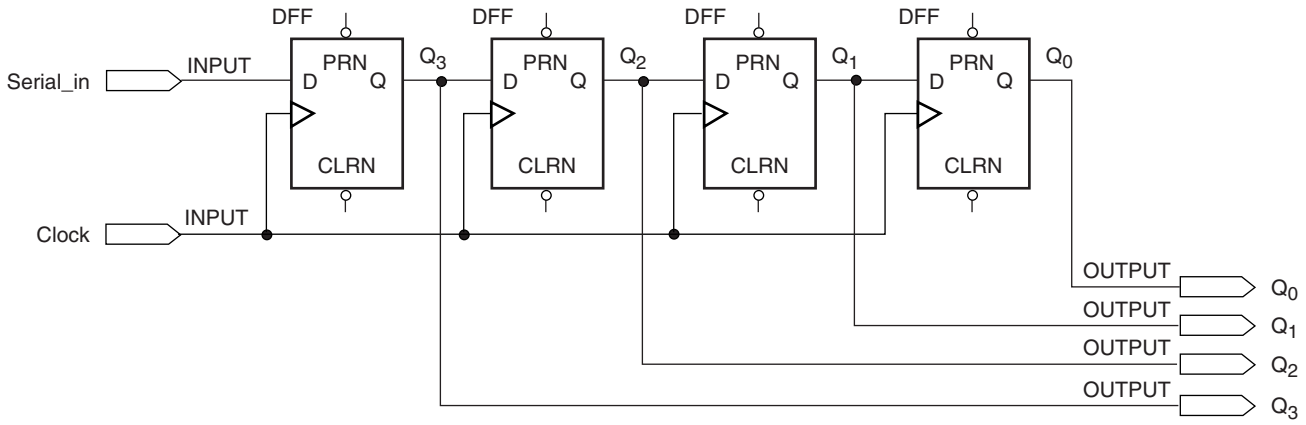


FIGURE 9.58
4-bit Serial Shift Register Configured to Shift Right

right and the bit at the circuit input is shifted into Q_3. The bit stored in Q_0 is overwritten by the former value of Q_1 and is lost. Since the data move from left to right, we say that the shift register implements a **right shift** function. (Data movement in the other direction, requiring a different circuit connection, is called **left shift.**)

Let us track the progress of data through the circuit in two cases. All flip-flops are initially cleared in each case.

Case 1: A 1 is clocked into the shift register, followed by a string of 0s, as shown in Figure 9.59. The flip-flop containing the 1 is shaded.

Before the first clock pulse, all flip-flops are filled with 0s. Data In goes to a 1 and on the first clock pulse, the 1 is clocked into the first flip-flop. After that, the input goes to 0. The 1 moves one position right with each clock pulse, the register filling up with 0s behind it, fed by the 0 at Data In. After four clock pulses, the 1 reaches the Data Out flip-flop. On the fifth pulse, the 0 coming behind overwrites the 1 at Q_0, leaving the register filled with 0s.

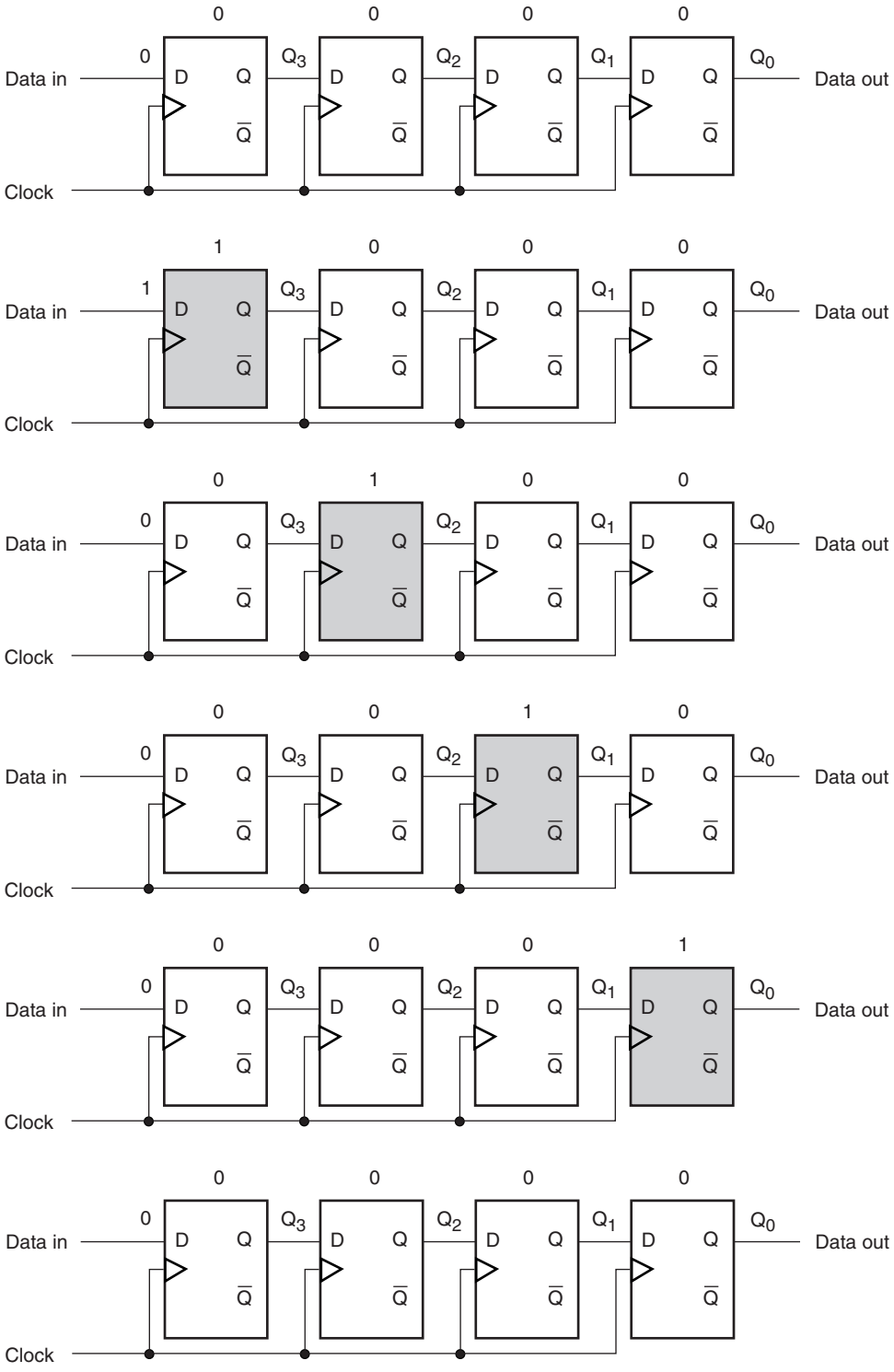


FIGURE 9.59
Shifting a "1" Through a Shift Register (Shift Right)

Case 2: Figure 9.60 shows a shift register, initially cleared, being filled with 1s.

As before, the initial 1 is clocked into the shift register and reaches the Data Out line on the fourth clock pulse. This time, the register fills up with 1s, not 0s, because the Data input remains HIGH.

Figure 9.61 shows a MAX+PLUS II simulation of the 4-bit serial shift register in Figure 9.58 through 9.60. The first half of the simulation shows the circuit operation for Case

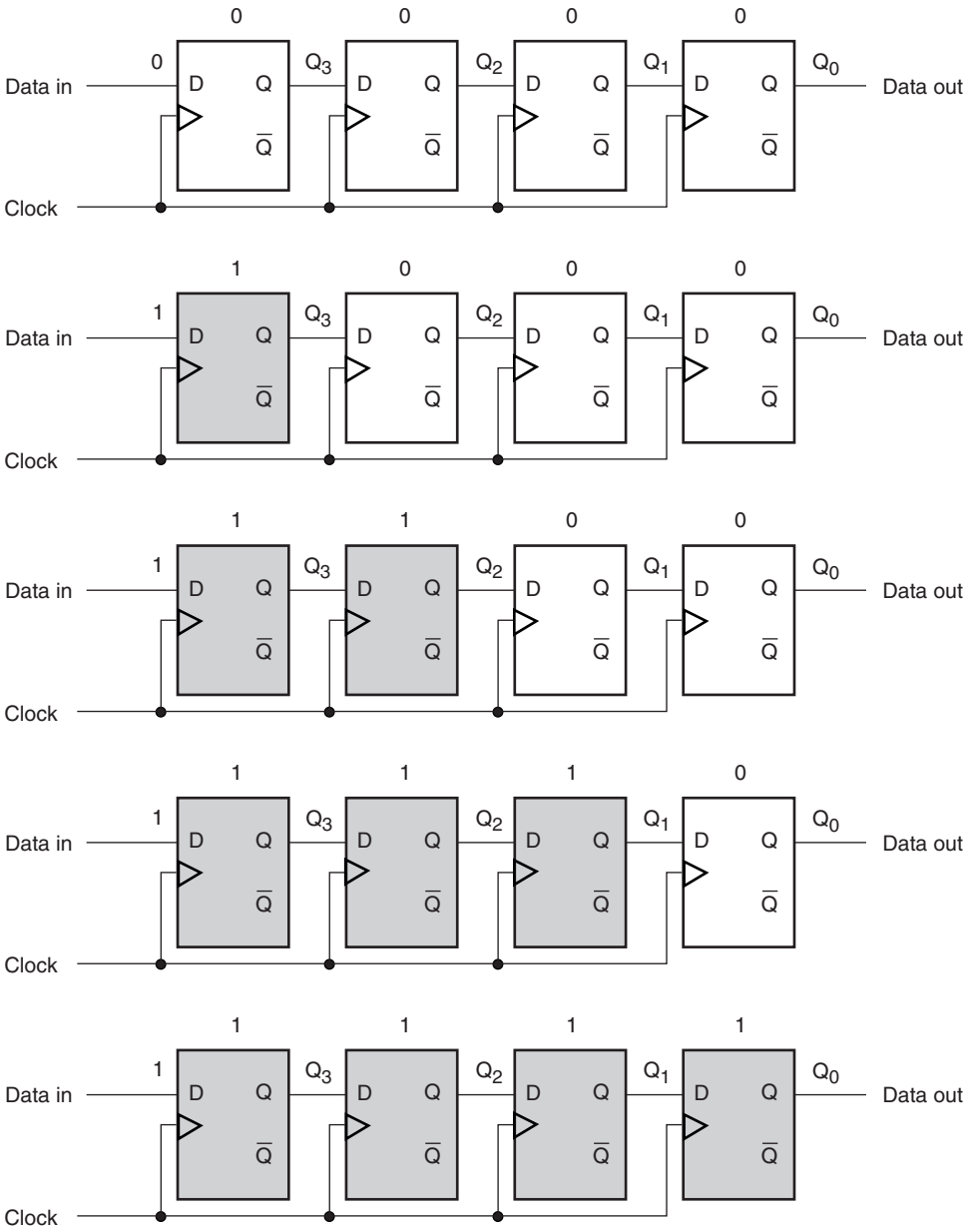


FIGURE 9.60
Filling a Shift Register with "1"s (Shift Right)

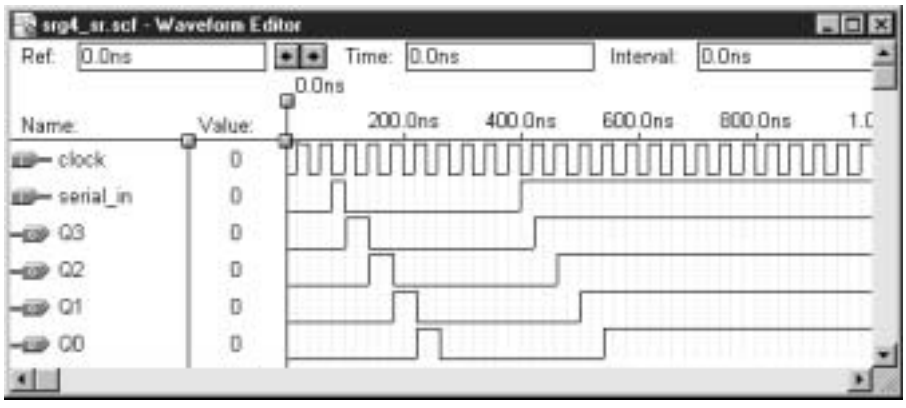


FIGURE 9.61
Simulation of a 4-bit Shift Register (Shift Right)

1, above. The 1 enters the register at Q_3 on the first clock pulse after **serial_in** (Data In) goes HIGH. The 1 moves one position for each clock pulse, which is seen in the simulation as a pulse moving through the Q outputs.

Case 2 is shown in the second half of the simulation. Again, a 1 enters the register at Q_3. The 1 continues to be applied to **serial_in,** so all Q outputs stay HIGH after receiving the 1 from the previous flip-flop.

NOTE

Conventions differ about whether the rightmost or leftmost bit in a shift register should be considered the most significant bit. The Altera Library of Parameterized Modules uses the convention of the leftmost bit being the MSB, so this is the convention we will follow. The convention has no physical meaning; the concept of right or left shift only makes sense on a logic diagram. The actual flip-flops may be laid out in any configuration at all in the physical circuit and still implement the right or left shift functions as defined on the logic diagram. (That is to say, wires, circuit board traces, and internal programmable logic connections can run wherever you want; left and right are defined on the logic diagram.)

EXAMPLE 9.11

Use the MAX+PLUS II Graphic Editor to create the logic diagram of a 4-bit serial shift register that shifts left, rather than right.

srg4_sl.gdf
srg4_sl.scf

Solution Figure 9.62 shows the required logic diagram. The flip-flops are laid out the same way as in Figure 9.58, with the MSB (Q_3) on the left. The D input of each flip-flop is connected to the Q output of the flip-flop to its right, resulting in a looped-back connection. A bit at D_0 is clocked into the rightmost flip-flop. Data in the other flip-flops are moved one place to the left. The bit in Q_2 overwrites Q_3. The previous value of Q_3 is lost.

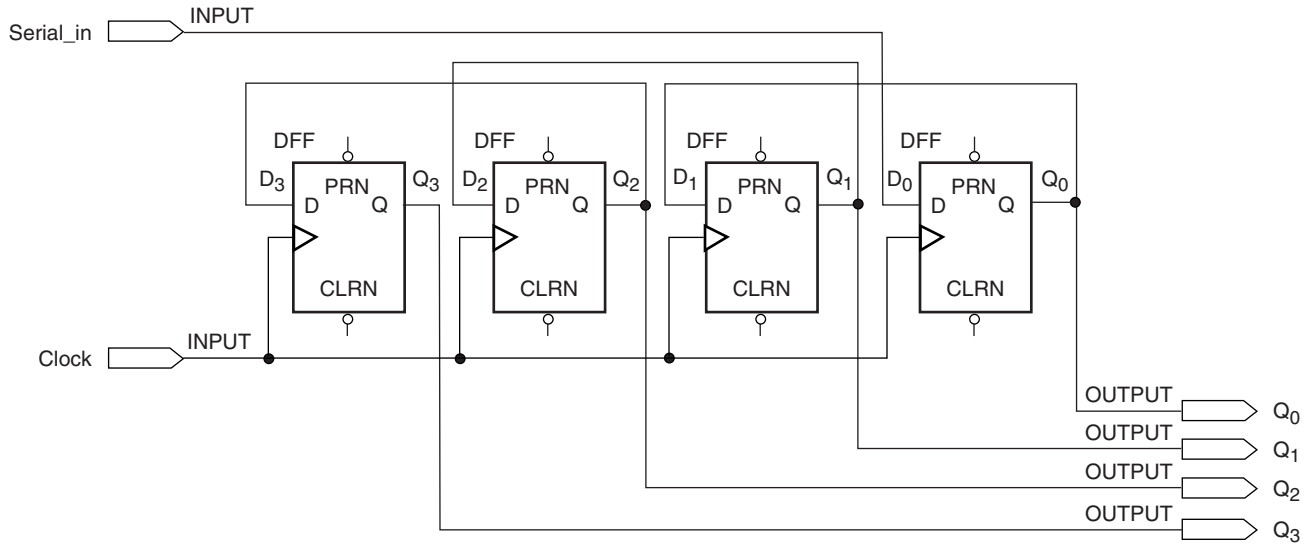


FIGURE 9.62
4-bit Serial Shift Register Configured to Shift Left

EXAMPLE 9.12

Draw a diagram showing the movement of a single 1 through the register in Figure 9.62. Also draw a diagram showing how the register can be filled up with 1s.

Solution Figures 9.63 and 9.64 show the required data movements.

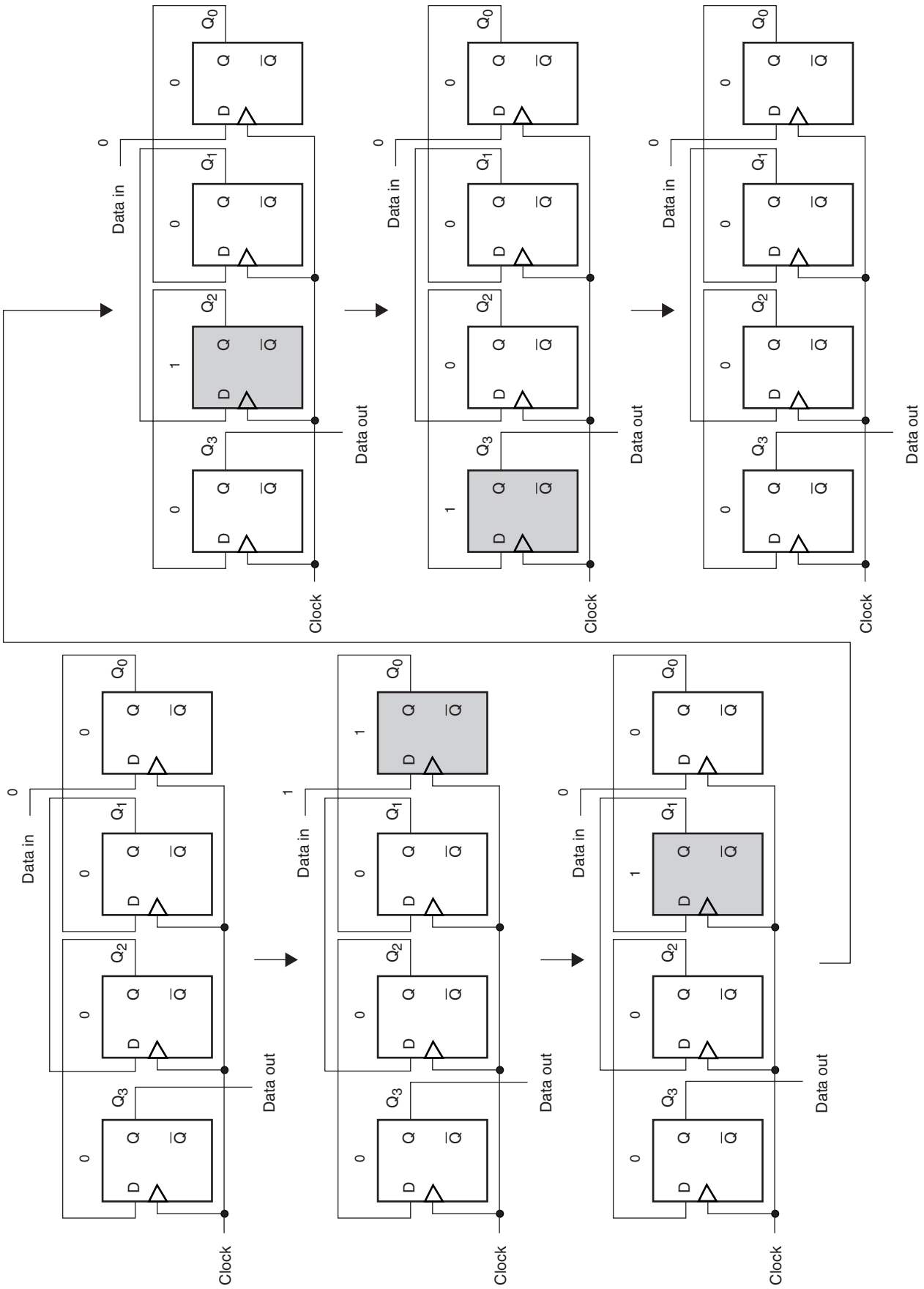


FIGURE 9.63
Shifting a "1" Through a Shift Register (Shift Left)

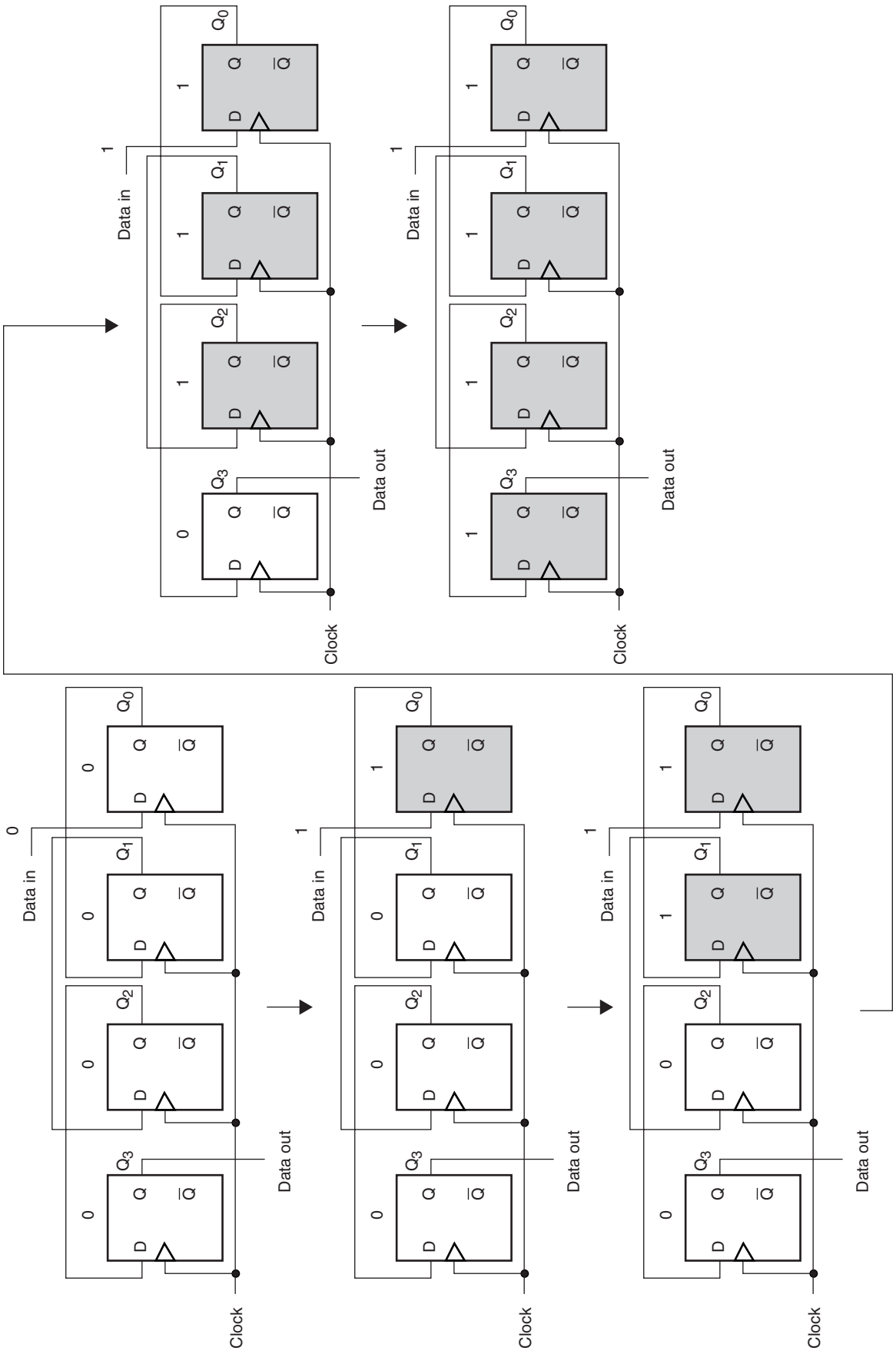


FIGURE 9.64
Filling a Shift Register with "1"s (Shift Left)

EXAMPLE 9.13

Use the MAX+PLUS II simulator to verify the operation of the shift-left serial shift register in Figure 9.62.

Solution Figure 9.65 shows the simulation of the shift operations shown in Example 9.12. Compare this simulation to the one in Figure 9.61 to see how the opposite shift direction appears on a timing diagram.

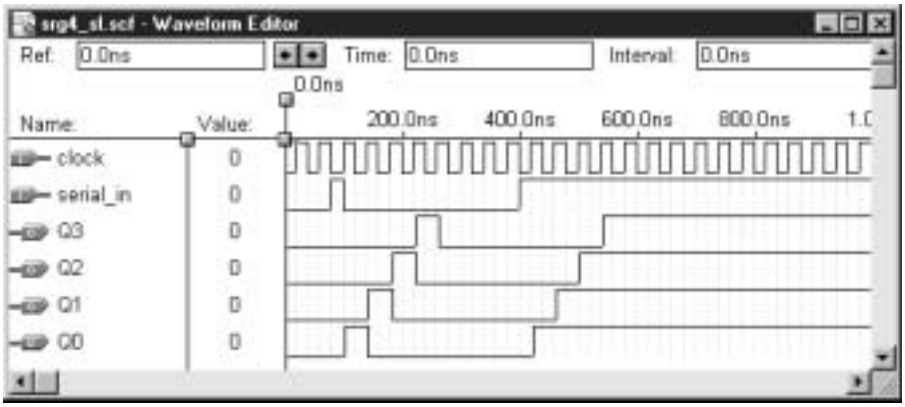


FIGURE 9.65
Simulation of a 4-bit Shift Register (Shift Left)

Bidirectional Shift Registers

srg4_bi.gdf
srg4_bi.scf

Figure 9.66 shows the logic diagram of a **bidirectional shift register.** This circuit combines the properties of the right shift and left shift circuits, seen earlier in Figures 9.58 and 9.62. This circuit can serially move data right or left, depending on the state of a control input, called *DIRECTION*.

The shift direction is controlled by enabling or inhibiting four pairs of AND-OR circuit paths that direct the bits at the flip-flop outputs to other flip-flop inputs. When *DIRECTION* = 0, the right-hand AND gate in each pair is enabled and the flip-flop outputs are directed to the *D* inputs of the flip-flops one position left. Thus the enabled pathway is from *Left_Shift_In* to Q_0, then to Q_1, Q_2, and Q_3.

When *DIRECTION* = 1, the left-hand AND gate of each pair is enabled, directing the data from *Right_Shift_In* to Q_3, then to Q_2, Q_1, and Q_0. Thus, *DIRECTION* = 0 selects left shift and *DIRECTION* = 1 selects right-shift.

Figure 9.67 shows a MAX+PLUS II simulation of the bidirectional shift register in Figure 9.66. The simulation shows the left shift function from 0 to 500 ns and right shift after 500 ns. Both *Right_Shift_In* and *Left_Shift_In* are applied in both parts of the simulation, but the circuit responds only to one for each function.

For the left shift function, a 1 is applied to Q_0 at 140 ns and shifted left. The *Right_Shift_In* pulse is ignored. Similarly, for the right shift function, a 1 is applied to Q_3 at 540 ns and shifted right. *Left_Shift_In* is ignored.

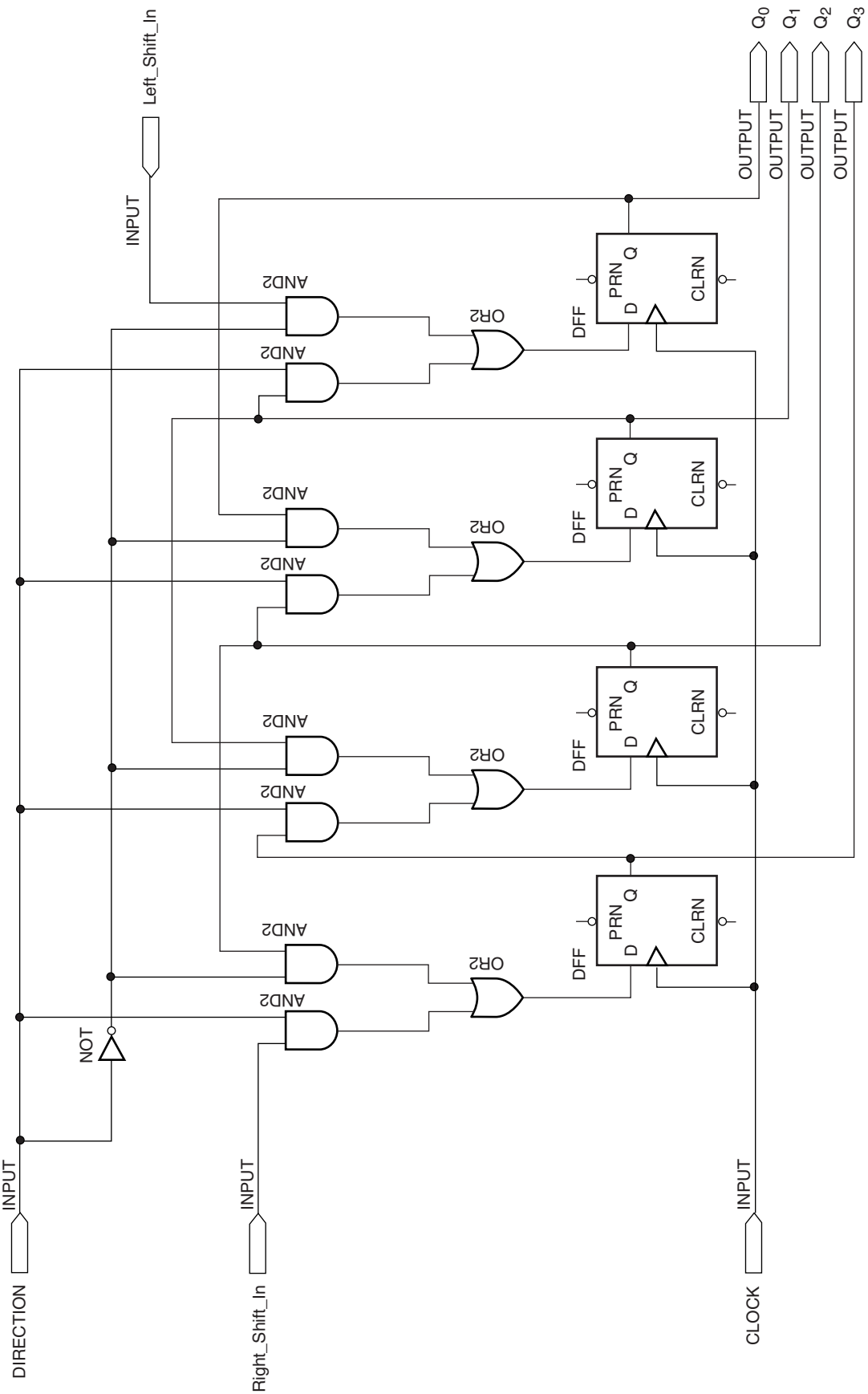


FIGURE 9.66
Bidirectional Shift Register

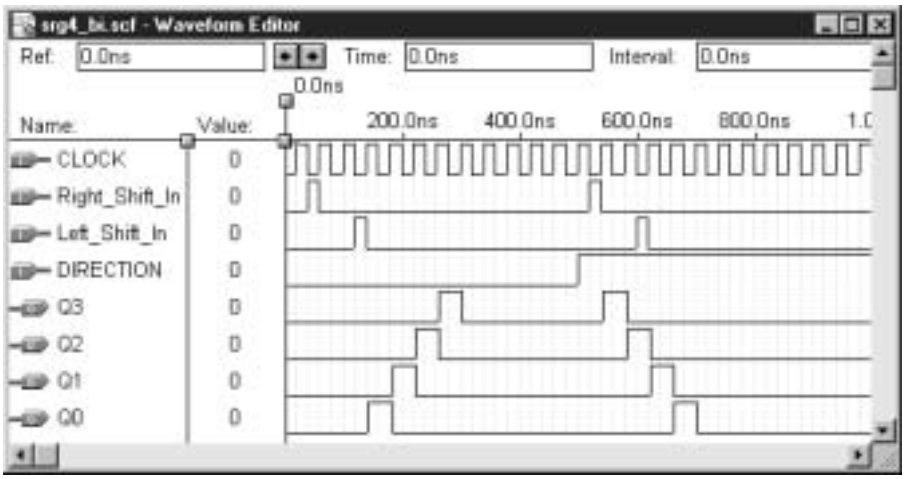


FIGURE 9.67
Simulation of a 4-bit Bidirectional Shift Register

srg4_par.gdf
srg4_par.scf

Shift Register with Parallel Load

Earlier in this chapter, we saw how a counter could be set to any value by synchronously loading a set of external inputs directly into the counter flip-flops. We can implement the same function in a shift register, as shown in Figure 9.68.

The circuit is similar to that of the bidirectional shift register in Figure 9.66. The synchronous input of each flip-flop is fed by an AND-OR circuit that directs one of two signals to the flip-flop: the output of the previous flip-flop (shift function) or a parallel input (load function). The circuit is configured such that the shift function is enabled when $LOAD = 0$ and the load function is enabled when $LOAD = 1$.

Figure 9.69 shows a simulation of the parallel-load shift register circuit of Figure 9.68. In the first part of the simulation, the shift function is selected. This is tested by sending a 1 through the circuit in a right-shift pattern. Next, at 400 ns, *LOAD* goes HIGH, and the parallel input value AH ($= 1010_2$) is synchronously loaded into the circuit. The *LOAD* input goes LOW, thus causing the circuit to revert to the shift function. The data in the register are right-shifted out, followed by 0s. At 640 ns, the value FH ($= 1111_2$) is loaded into the circuit, then right-shifted out.

srg4_uni.gdf
srg4_uni.scf

Figure 9.70 shows the logic circuit of a **universal shift register.** This circuit can implement any combination of serial and parallel inputs and outputs. It can also serially shift data left or right or hold data, depending on the states of S_1 and S_0, which form a 2-bit function select input.

Each AND-OR circuit acts as a multiplexer to direct one of several possible data sources to the synchronous inputs of each flip-flop. For instance, if we trace the paths through the corresponding AND-OR circuit, we find that the possible sources of data at D_2, the synchronous input of the second flip-flop, are Q_3 ($S_1S_0 = 01$), P_2 ($S_1S_0 = 11$), Q_1 ($S_1S_0 = 10$), and Q_2($S_1S_0 = 00$). These are the inputs required for the right-shift, parallel load, left-shift, and hold functions, respectively. All functions are synchronous, including the parallel load and hold functions.

The hold function is a synchronous no change function, implemented by feeding back the *Q* output of a flip-flop to its synchronous *(D)* input. It is necessary to have this function, so that the flip-flops will not synchronously clear when none of the other functions is selected.

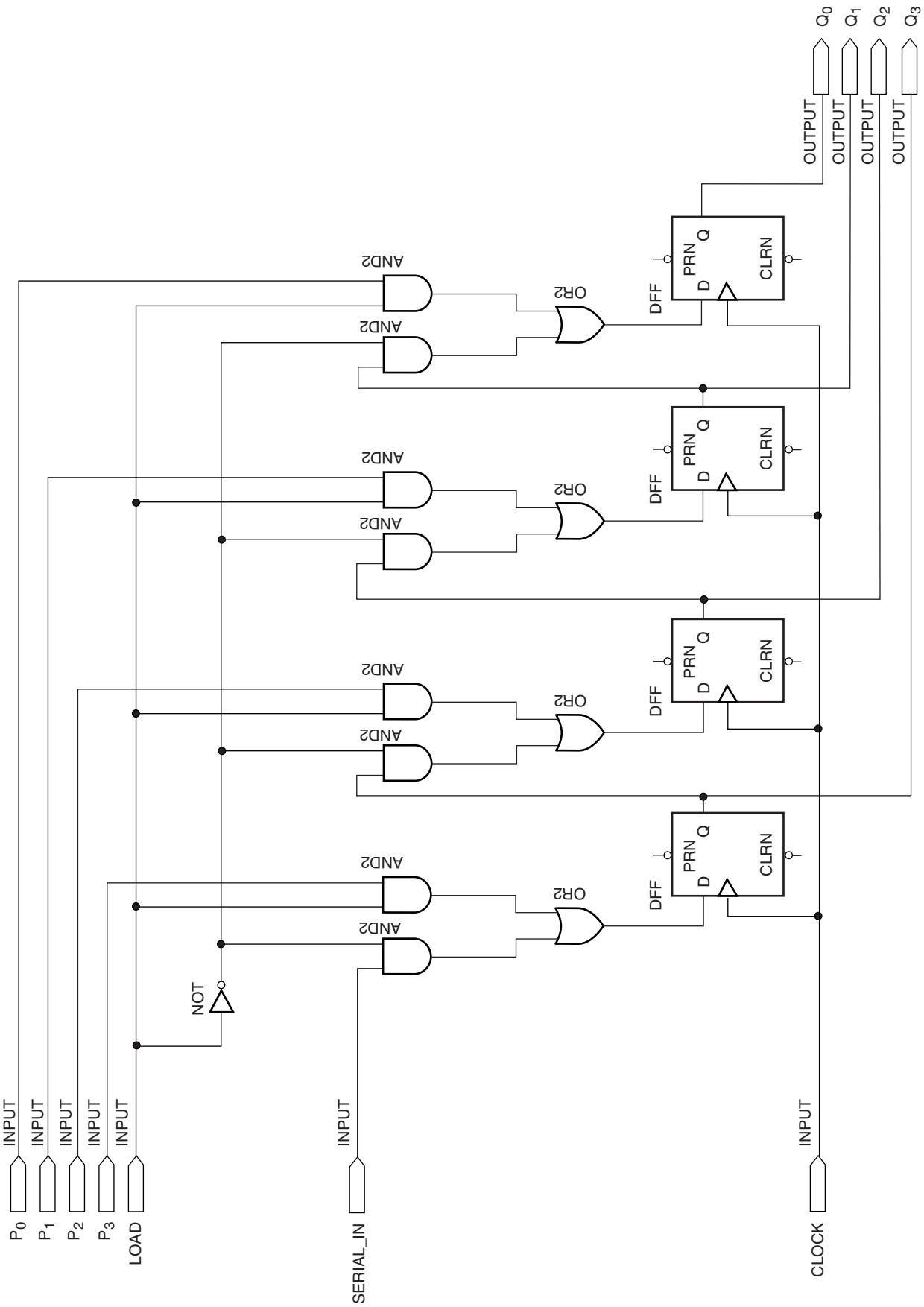


FIGURE 9.68
Serial Shift Register with Parallel Load

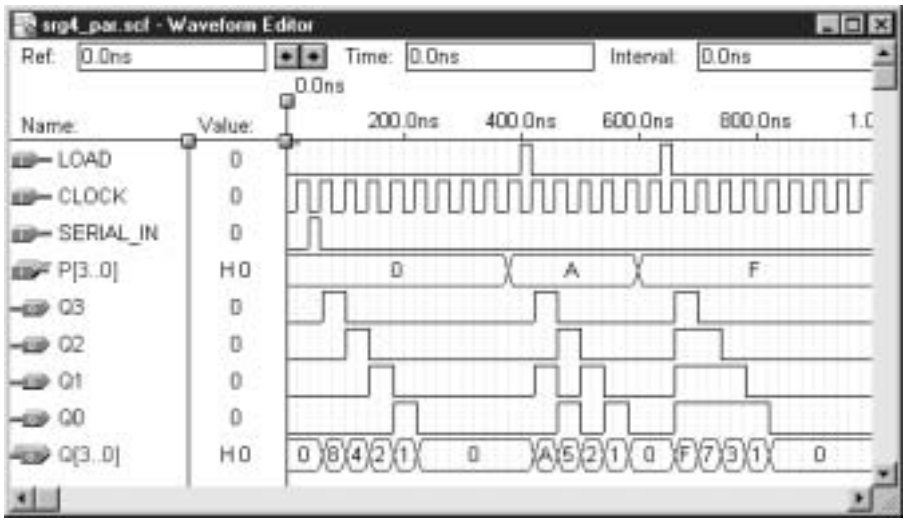


FIGURE 9.69
Simulation of a 4-bit Serial Shift Register with Parallel Load

Table 9.14 summarizes the various possible inputs to each flip-flop as a function of S_1 and S_0.

Table 9.14 Flip-Flop Inputs as a Function of S_1S_0 in a Universal Shift Register

S_1	S_0	Function	D_3	D_2	D_1	D_0
0	0	Hold	Q_3	Q_2	Q_1	Q_0
0	1	Shift Right	*RSI**	Q_3	Q_2	Q_1
1	0	Shift Left	Q_2	Q_1	Q_0	*LSI***
1	1	Load	P_3	P_2	P_1	P_0

**RSI* = Right-shift input
***LSI* = Left-shift input

EXAMPLE 9.14

Create a simulation file to verify the operation of the universal shift register of Figure 9.70.

Solution Figure 9.71 shows a possible solution. The following functions are tested: hold, right shift (*LSI* ignored), hold, left shift (*RSI* ignored), load FH, asynchronous clear, load FH, shift right for two clocks, shift left for three clocks.

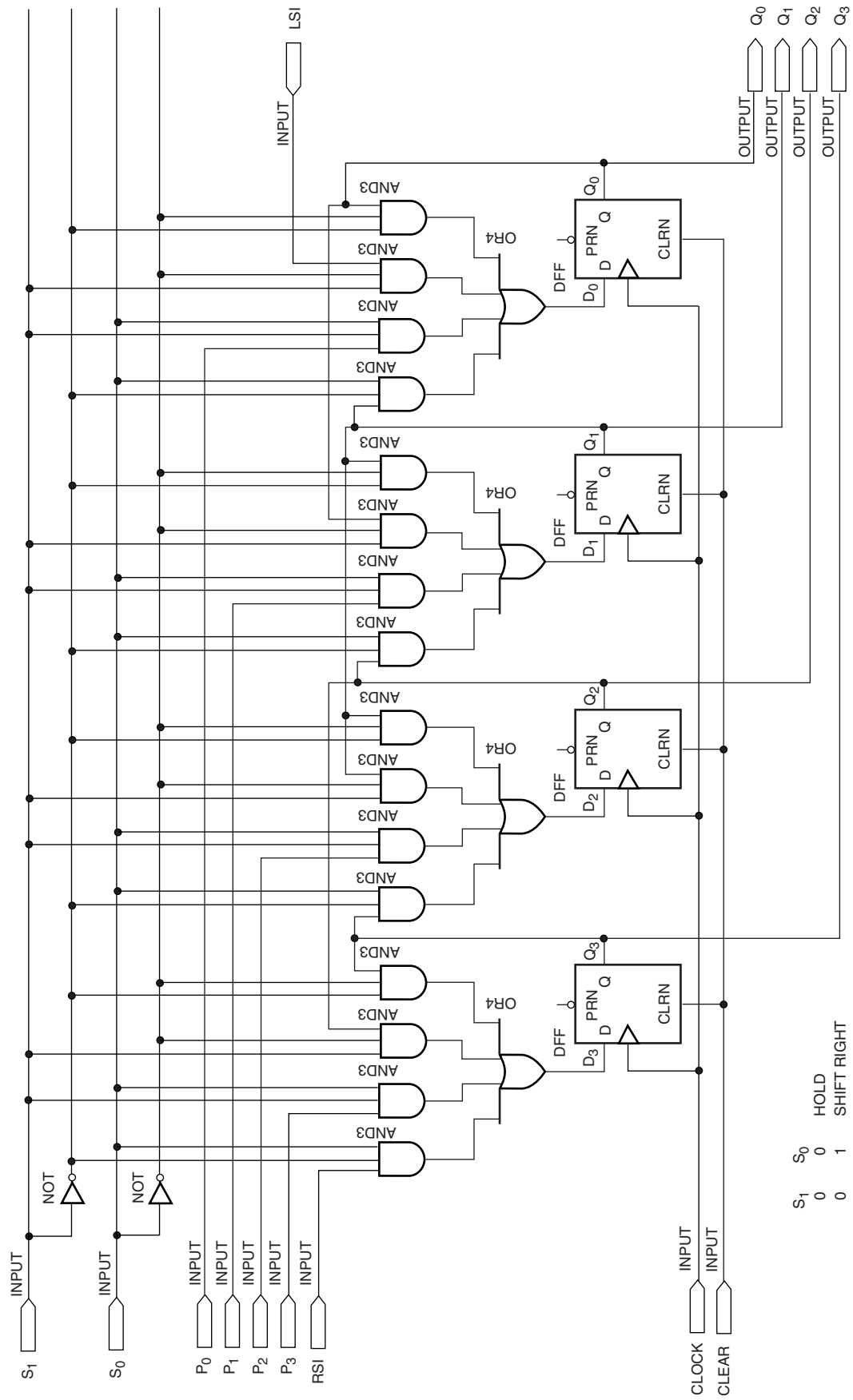


FIGURE 9.70
4-bit Universal Shift Register

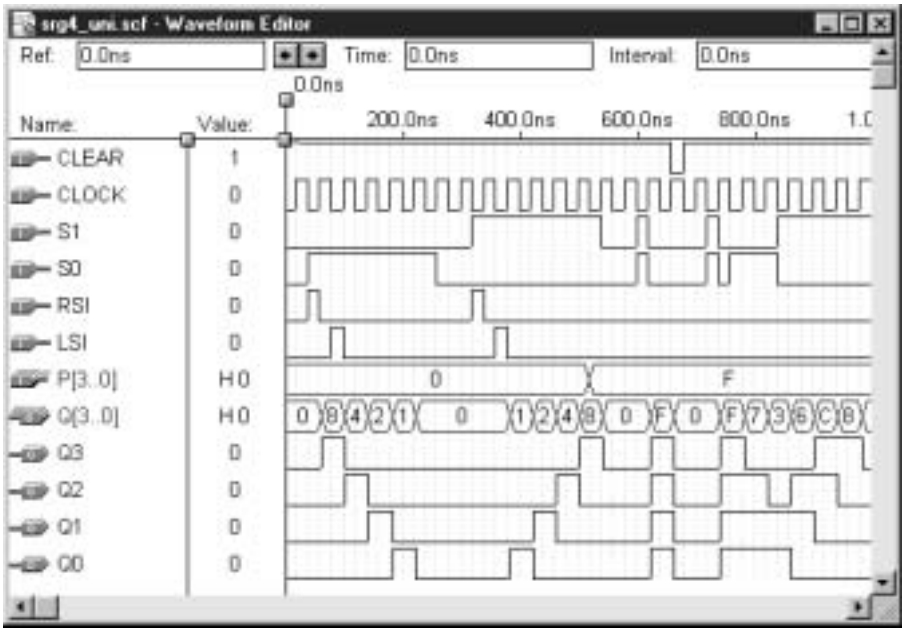


FIGURE 9.71
Example 9.14
Simulation of a 4-bit Universal Shift Register

SECTION 9.7 REVIEW PROBLEM

9.7 Can the D flip-flops in Figure 9.58 be replaced by JK flip-flops? If so, what modifications to the existing circuit are required?

9.8 Programming Shift Registers in VHDL

KEY TERMS

Structural design A VHDL design technique that connects predesigned components using internal signals.

Dataflow design A VHDL design technique that uses Boolean equations to define relationships between inputs and outputs.

Behavioral design A VHDL design technique that uses descriptions of required behavior to describe the design.

As with other circuit applications, we can take several approaches to programming shift registers in VHDL. Three basic design techniques are **structural, dataflow,** and **behavioral** descriptions. We will use each of these techniques to design a 4-bit shift register, such as the one shown in Figure 9.58.

Structural Design

Structural design is like taking components out of a bin and connecting them together to make a circuit. We can use the **DFF** component from the MAX+PLUS II primitives library and instantiate enough components to make a shift register, with connections made

by internal signals. The code to make a 4-bit shift register using the structural design technique is shown here in the file **srg4strc.vhd**.


srg4strc.vhd
srgstrc.scf

```
-- srg4strc.vhd
-- Structural description of a 4-bit serial shift register
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
LIBRARY altera;
USE altera.maxplus2.ALL;

ENTITY srg4strc IS
    PORT(
       serial_in, clk : IN      STD_LOGIC;
       qo             : BUFFER  STD_LOGIC_VECTOR(3 downto 0) );
END srg4strc;

ARCHITECTURE right_shift of srg4strc IS
    COMPONENT DFF
       PORT (d : IN STD_LOGIC;
           clk : IN STD_LOGIC;
           q   : OUT STD_LOGIC);
    END COMPONENT;
BEGIN
    flip_flop_3: dff
       PORT MAP (serial_in, clk, qo(3) );

    dffs:
    FOR i IN 2 downto 0 GENERATE
    flip_flops_2_to_0: dff
       PORT MAP (qo(i + 1), clk, qo(i) );
    END GENERATE;
END right_shift;
```

The design entity **srg4strc.vhd** instantiates four D flip-flops from the **altera.maxplus2** package and connects them by assigning common inputs and outputs to related components. A different way of writing the component instantiations would be as follows.

```
flip_flop_3: dff
    PORT MAP (serial_in, clk, qo(3) );
flip_flop_2: dff
    PORT MAP(qo(3), clk, qo(2) );
flip_flop_1: dff
    PORT MAP(qo(2), clk, qo(1) );
flip_flop_0: dff
    PORT MAP(qo(1), clk, qo(0) );
```

Since the component ports are in the sequence (D, clk, Q), the component instantiations shown above imply that the *D* input of a flip-flop is fed by the *Q* of the previous flip-flop.

The port identifier **qo** is defined as mode BUFFER, not as OUT, because it is sometimes used as an input and sometimes as an output. A port of mode OUT can only be used as an output. A port of mode BUFFER has a feedback connection so that the output can be reused in the programmed AND matrix of the CPLD macrocell. Figure 9.72 illustrates the difference between these modes.

Rather than defining connections in the component instantiations, we would also be able to use an internal signal to connect the flip-flops. This method allows us to use a port of mode OUT, rather than BUFFER. The file **srg4str2.vhd** shows this alternative way.

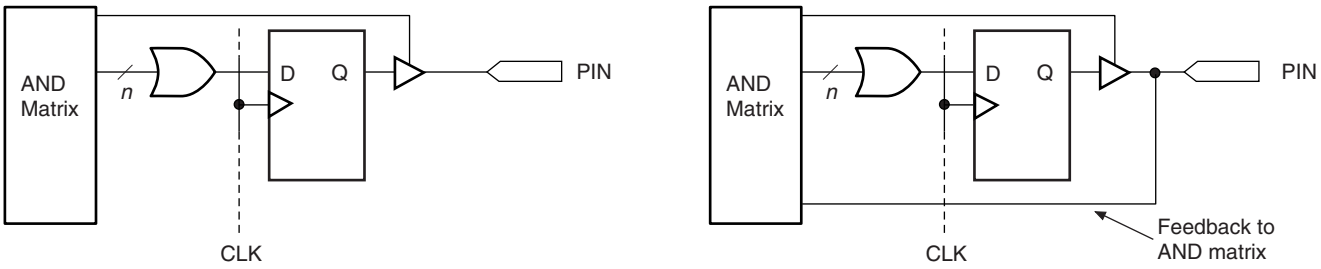


FIGURE 9.72
OUT vs. BUFFER

srg4str2.vhd
srg4str2.scf

```
--srg4str2.vhd
-- Structural description of a 4-bit serial shift register
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
LIBRARY altera;
USE altera.maxplus2.ALL;

ENTITY srg4str2 IS
    PORT (
        serial_in, clk : IN  STD_LOGIC;
        qo             : OUT STD_LOGIC_VECTOR(3 downto 0) );
END srg4str2;

ARCHITECTURE right_shift of srg4str2 IS
    COMPONENT DFF
        PORT (d   : IN STD_LOGIC;
              clk : IN STD_LOGIC;
              q   : OUT STD_LOGIC);
    END COMPONENT;
    SIGNAL connect : STD_LOGIC_VECTOR(3 downto 0);
BEGIN
    flip_flop_3: dff
       PORT MAP (serial_in, clk, connect(3) );
    dffs:
    FOR i IN 2 downto 0 GENERATE
    flip_flops_2_to_0: dff
        PORT MAP (connect(i + 1), clk, connect(i) );
    END GENERATE;

    qo <= connect;
END right_shift;
```

In this case, the internal signal **connect** is used to tie the flip-flops together. The circuit output derives from a signal assignment statement at the end of the file. Since the internal signal **connect** is used to fulfil the flip-flop input/output functions, **qo** can be defined solely as an output.

Dataflow Design

Dataflow design describes a design entity in terms of the Boolean relationships between different parts of the circuit. The Boolean relationships in a 4-bit shift register are defined by the expressions for the flip-flop synchronous inputs:

$$D_3 = \text{serial\_in}$$
$$D_2 = Q_3$$
$$D_1 = Q_2$$
$$D_0 = Q_1$$

The design entity **srg4dflw.vhd** illustrates the use of the dataflow design method for a 4-bit serial shift register.

srg4dflw.vhd
srg4dflw.scf

```
-- srg4dflw.vhd
-- Dataflow description of a 4-bit serial shift register
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY srg4dflw IS
    PORT (
        serial_in, clk : IN      STD_LOGIC;
        q              : BUFFER  STD_LOGIC_VECTOR(3 downto 0) );
END srg4dflw;

ARCHITECTURE right_shift of srg4dflw IS
    SIGNAL d : STD_LOGIC_VECTOR(3 downto 0);
BEGIN

    PROCESS (clk)
    BEGIN
        -- Define a 4-bit D flip-flop
        IF clk'EVENT and clk = '1' THEN
            q <= d;
        END IF;
    END PROCESS;

    d <= serial_in & q(3 downto 1);
END right_shift;
```

Before the flip-flops can be connected, they must be defined in a PROCESS statement. The statements inside the process are sequential, as they must be to define a flip-flop, but the process itself is a concurrent statement. Signals are applied concurrently (simultaneously) to the construct implied by the process (the flip-flops) and all other concurrent constructs in the design entity (the connections between **q** and **d** and the serial input).

A signal assignment statement implements the Boolean equations for the shift register. It is written as a single statement for efficiency, but could also be written as four separate assignment statements, as follows:

```
d(3) <= serial_in;
d(2) <= q(3);
d(1) <= q(2);
d(0) <= q(1);
```

We must define **q** as mode BUFFER, since we are using it as both input and output.

Behavioral Design

We can create a VHDL design entity from the description of its desired behavior. In the case of a shift register, we know that after a clock pulse all data move over one position and the first flip-flop in the chain accepts a bit from a serial input, as indicated in Table 9.15. We can use this behavioral description to implement a serial shift register, as shown in the VHDL file **srg4behv.vhd**.

Table 9.15 Next States of Flip-Flops in a Serial Shift Register

Q_3	Q_2	Q_1	Q_0
serial_in	Q_3	Q_2	Q_1

srg4behv.vhd
srg4behv.scf

```
-- srg4behv.vhd
-- Behavioral description of a 4-bit serial shift register
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY srg4behv IS
    PORT (
        serial_in, clk : IN      STD_LOGIC;
        q              : BUFFER  STD_LOGIC_VECTOR(3 downto 0) );
END srg4behv;

ARCHITECTURE right_shift of srg4behv IS
BEGIN
    PROCESS (clk)
    BEGIN
       IF (clk'EVENT and clk = '1') THEN
          q <= serial_in & q(3 downto 1);
       END IF;
    END PROCESS;
END right_shift;
```

In the behavioral design, we are not concerned with the flip-flop inputs or other internal connections; the behavioral description is sufficient for the VHDL compiler to synthesize the required hardware. Compare this to the dataflow description, where we created a set of flip-flops, then assigned Boolean functions to the *D* inputs. In this case, the behavioral design method combines these two steps into one.

EXAMPLE 9.15

Write the code for a VHDL design entity that implements a 4-bit bidirectional shift register with asynchronous clear. Create a simulation that verifies the design function.

Solution The VHDL code for the bidirectional shift register, **srg4bidi.vhd**, follows. A CASE statement monitors the directional control of the shift register. We require the **others** clause of the CASE statement since the identifier **direction** is of type STD_LOGIC; the cases '0' and '1' do not cover all possible values of STD_LOGIC. Since we want no action to be taken in the default case, we use the keyword **NULL**.

www.electronictech.com

srg4bidi.vhd
srg4bidi.scf

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

ENTITY srg4bidi IS
    PORT (
        clk, clear : IN STD_LOGIC;
        rsi, lsi   : IN STD_LOGIC;
        direction  : IN STD_LOGIC;
        q          : BUFFER STD_LOGIC_VECTOR(3 downto 0) );
END srg4bidi;

ARCHITECTURE bidirectional_shift of srg4bidi IS
BEGIN
    PROCESS (clk, clear)
    BEGIN
       IF clear = '0' THEN
              q <= "0000"; -- asynchronous clear
       ELSIF (clk'EVENT and clk = '1') THEN
          CASE direction IS
             WHEN '0' =>
                q <= q(2 downto 0) & lsi; -- left shift
```

```
            WHEN '1' =>
               q <= rsi & q(3 downto 1); -- right shift
            WHEN others =>
               NULL;
         END CASE;
      END IF;
   END PROCESS;
END bidirectional_shift;
```

Figure 9.73 shows the simulation of the shift register, with the left shift function in the first half of the simulation and the right shift function in the second half.

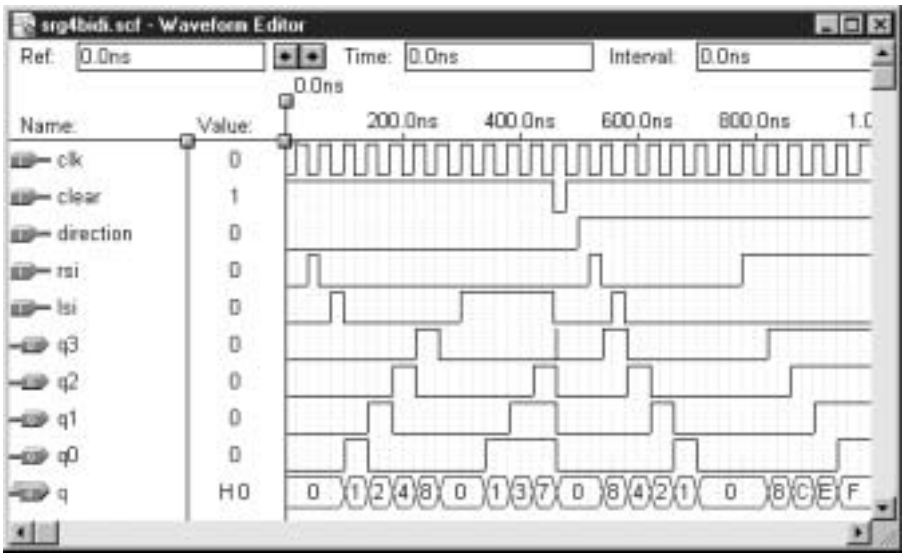


FIGURE 9.73
Example 9.15
4-bit Bidirectional Shift Register

Shift Registers of Generic Width

KEY TERM

GENERIC A clause in the entity declaration of a VHDL component that lists the parameters that can be specified when the component is instantiated.

All multibit VHDL components we have examined until now have been of a specified width (e.g., 2-to-4 decoder, 8-bit MUX, 8-bit adder, 4-bit counter). VHDL allows us to create components having a generic, or unspecified, width or other parameter which is specified when the component is instantiated. In the entity declaration of such a component, we indicate an unspecified parameter (such as width) in a **GENERIC** clause. The unspecified parameter must be given a default value in the GENERIC clause, indicated by := *value*.

When we instantiate the component, we specify the parameter value in a generic map, as we have done with components from the Library of Parameterized Modules. The design entity **srt_bhv.vhd** below behaviorally defines an *n*-bit right-shift register, with a default width of four bits given by the statement (`GENERIC (width : POSITIVE := 4);`).

The entity **srt8_bhv.vhd** instantiates the *n*-bit register as an 8-bit circuit by specifying the bit width in a generic map. If no value is specified, the component is presumed to have a default width of four, as defined in the component's entity declaration.

```
-- srt_bhv.vhd
```

srt_bhv.vhd
srt8_bhv.vhd
srt8_bhv.scf

```
-- Behavioral description of an n-bit shift register
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY srt_bhv IS
  GENERIC (width : POSITIVE := 4);
  PORT (
     serial_in, clk : IN       STD_LOGIC;
     q              : BUFFER   STD_LOGIC_VECTOR(width-1 downto 0) );
END srt_bhv;

ARCHITECTURE right_shift of srt_bhv IS
BEGIN
  PROCESS (clk)
  BEGIN
     IF (clk'EVENT and clk = '1') THEN
         q(width-1 downto 0) <= serial_in & q(width-1 downto 1);
  END IF;
END PROCESS;
END right_shift;

-- srt8_bhv.vhd
-- 8-bit shift register that instantiates srt_bhv
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY srt8_bhv IS
 PORT(
     data_in, clock : IN         STD_LOGIC;
     qo             : BUFFER     STD_LOGIC_VECTOR(7 downto 0) );
END srt8_bhv;

ARCHITECTURE right_shift of srt8_bhv IS
COMPONENT srt_bhv
  GENERIC (width : POSITIVE);
  PORT (
     serial_in, clk   : IN     STD_LOGIC;
     q                : OUT    STD_LOGIC_VECTOR(7 downto 0) );
END COMPONENT;
BEGIN
  Shift_right_8: srt_bhv
     GENERIC MAP (width=> 8)
     PORT MAP (serial_in => data_in,
               clk       => clock,
               q         => qo);
END right_shift;
```

EXAMPLE 9.16

Write the code for a VHDL design entity that defines a universal shift register with a generic width. (The default width is eight bits.) Instantiate this entity as a component in a file for a 16-bit universal shift register.

Solution

```
-- srg_univ.vhd
-- Universal shift register with generic width
-- Default width = 8 bits
```

srg_univ.vhd
srg16uni.vhd

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

ENTITY srg_univ IS
    GENERIC (width : POSITIVE := 8);
    PORT (
       clk, clear     : IN STD_LOGIC;
       rsi, lsi       : IN STD_LOGIC;
       function_select   : IN STD_LOGIC_VECTOR(1 downto 0);
       p              : IN STD_LOGIC_VECTOR(width-1 downto 0);
       q              : BUFFER STD_LOGIC_VECTOR(width-1 downto 0) );
END srg_univ;

ARCHITECTURE universal_shift of srg_univ IS
BEGIN
   PROCESS (clk, clear)
   BEGIN
      IF clear = '0' THEN
         -- Conversion function to convert integer 0 to vector
         -- of any width. Requires ieee.std_logic_arith package.
         q <= CONV_STD_LOGIC_VECTOR(0, width);
      ELSIF (clk'EVENT and clk = '1') THEN
         CASE function_select IS
            WHEN "00" =>
               q <= q; -- Hold
            WHEN "01" =>
               q <= rsi & q(width-1 downto 1); -- Shift right
            WHEN "10" =>
               q <= q(width-2 downto 0) & lsi; -- Shift left
            WHEN "11" =>
               q <= p; -- Load
            WHEN OTHERS =>
               NULL;
          END CASE;
      END IF;
   END PROCESS;
END universal_shift;

-- srg16uni.vhd
-- 16-bit universal shift register (instantiates srg_univ)
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY srg16uni IS
    PORT (
        clock, clr  : IN      STD_LOGIC;
        rsi, lsi    : IN      STD_LOGIC;
        s           : IN      STD_LOGIC_VECTOR(1 downto 0);
        parallel_in : IN      STD_LOGIC_VECTOR(15 downto 0);
        qo          : BUFFER  STD_LOGIC_VECTOR(15 downto 0) );
END srg16uni;

ARCHITECTURE universal_shift of srg16uni IS
COMPONENT srg_univ
   GENERIC (width : POSITIVE);
   PORT (
      clk, clear      : IN STD_LOGIC;
```

```
      rsi, lsi        : IN STD_LOGIC;
      function_select    : IN STD_LOGIC_VECTOR(1 downto 0);
      p               : IN STD_LOGIC_VECTOR(width-1 downto 0);
      q               : BUFFER STD_LOGIC_VECTOR(width-1 downto 0));
END COMPONENT;
BEGIN
    Shift_universal_16: srg_univ
        GENERIC MAP (width=> 16)
        PORT MAP (clk             => clock,
                  clear           => clr,
                  rsi             => rsi,
                  lsi             => lsi,
                  function_select => s,
                  p               => parallel_in,
                  q               => qo);
END universal_shift;
```

When we are designing the clear function in **srg_univ.vhd**, we must account for the fact that we must set all bits of a vector of unknown width to '0'. To get around this problem, we use a conversion function that changes an INTEGER value of 0 to a STD_LOGIC_VECTOR of **width** bits and assigns the value to the output. The required conversion function, `CONV_STD_LOGIC_VECTOR(value, number_of_bits)`, is found in the **std_logic_arith** package in the **ieee** library. We could also use the construct

```
q <= (others => '0');
```

which states that the default case is to set all bits of **q** to 0 when **clear** is 0. Since there is no other case specified, all bits of **q** are cleared.

LPM Shift Registers

The Library of Parameterized Modules contains a shift register component, **lpm_shiftreg,** that we can instantiate in a VHDL design entity. The various functions of **lpm_shiftreg** are listed in Table 9.16.

The following VHDL code instantiates **lpm_shiftreg** as an 8-bit shift register with serial input and serial output. In this case, the LPM component is declared explicitly, with the component declaration statement listing only the ports and parameters used by the design entity. The component instantiation statement lists the port names from the design entity in the same order as the corresponding component port names. By default the register direction is LEFT (i.e., toward the MSB).

```
-- srg8_lpm.vhd
-- 8-bit serial shift register (shift left by default)
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
LIBRARY lpm;
USE lpm.lpm_components.ALL;
```

Table 9.16 Available Functions for ***lpm_shiftreg***

Function	Ports	Parameters	Description
Basic serial operation	clock, shiftin, shiftout, q[]	LPM_WIDTH	Data moves serially from **shiftin** to **shiftout.** Parallel outputs appear at **q[].**
Load	sload, data[]	none	When **sload** = 1, **q[]** goes to the value at input **data[]** on the next positive clock edge. **Data[]** has the same width as **LPM_WIDTH**.
Synchronous clear	sclr	none	When **sclr** = 1, **q[]** goes to zero on positive clock edge
Synchronous set	sset	LPM_SVALUE	When **sset** = 1, output goes to value of **LPM_SVALUE** on positive clock edge. If **LPM_SVALUE** is not specified **q[]** goes to all 1s.
Asynchronous clear	aclr	none	Output goes to zero when **aclr** = 1.
Asynchronous set	aset	LPM_AVALUE	Output goes to value of **LPM_AVALUE** when **aset** = 1. If **LPM_AVALUE** is not specified, outputs all go HIGH when **aset** = 1.
Directional control	none	LPM_DIRECTION	Optional direction control. Default direction is LEFT. **LPM_DIRECTION** = "LEFT" or "RIGHT". If **shiftin** and **shiftout** are used, the serial shift always goes through the entire shift register, in the direction given by **LPM_DIRECTION.**
Clock enable	enable	none	All synchronous functions are enabled when **enable** = 1. Defaults to "enabled" when not specified.

srg8_lpm.vhd
srg8_lpm.scf

```
ENTITY srg8_lpm IS
   PORT (
      clk        : IN  STD_LOGIC;
      serial_in  : IN  STD_LOGIC;
      serial_out : OUT STD_LOGIC);
END srg8_lpm;

ARCHITECTURE lpm_shift of srg8_lpm IS
   COMPONENT lpm_shiftreg
   GENERIC(LPM_WIDTH: POSITIVE);
      PORT (
         clock, shiftin : IN  STD_LOGIC;
         shiftout       : OUT STD_LOGIC);
END COMPONENT;
BEGIN
   Shift_8: lpm_shiftreg
      GENERIC MAP (LPM_WIDTH=> 8)
      PORT MAP (clk, serial_in, serial_out);
END lpm_shift;
```

Figure 9.74 shows a simulation of the shift register, with the data shifting from right to left (LSB to MSB). Since there is no parallel output **(q[])** instantiated in our design, we would not normally be able to monitor the progress of bits from flip-flop to flip-flop; we would only see **shiftin**, and then, eight clock cycles later, **shiftout**. However, we are able to monitor the flip-flop states as buried nodes (**Shift_8|dffs[7..0].Q**). These buried nodes are the last eight lines in the simulation.

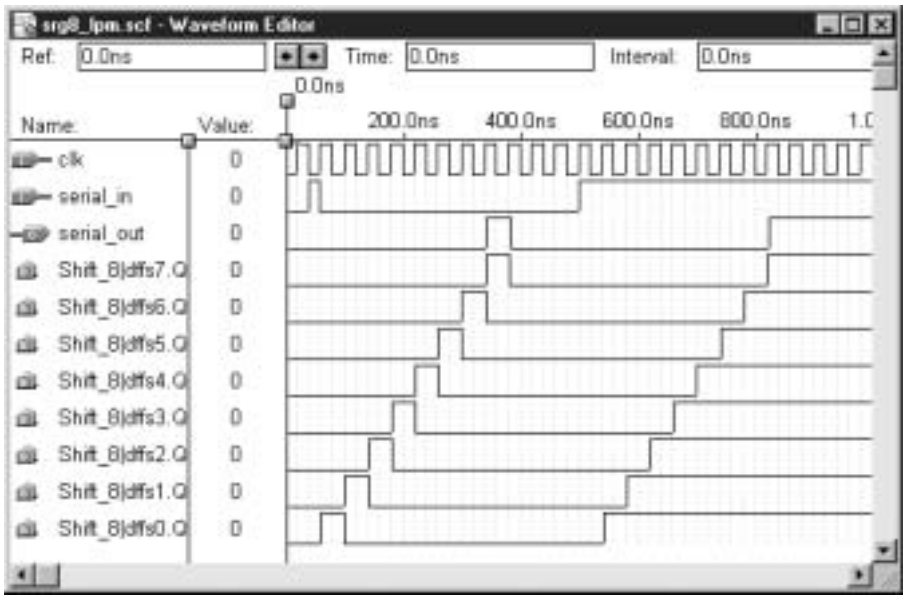


FIGURE 9.74
Simulation of an 8-bit LPM Shift Register (Shift Left)

EXAMPLE 9.17

Modify the VHDL code just shown to make the serial shift register shift right, rather than left. Create a simulation to verify the circuit function. How do the positions of **shiftin** and **shiftout** ports relate to the internal flip-flops for the right-shift and left-shift implementations?

Solution The modified VHDL code is shown next as design entity **srg8lpm2.** The only difference is the addition of the parameter LPM_DIRECTION to both the component declaration and component instantiation statements.

```
-- srg8_lpm2.vhd
-- 8-bit serial shift register (shift right)
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
LIBRARY lpm;
USE lpm.lpm_components.ALL;

ENTITY srg8lpm2 IS
    PORT (
        clk        : IN  STD_LOGIC;
        serial_in  : IN  STD_LOGIC;
        serial_out : OUT STD_LOGIC);
END srg8_lpm2;
```

srg8_lpm2.vhd
srg8_lpm2.scf

```
ARCHITECTURE lpm_shift of srg8_lpm2 IS
COMPONENT lpm_shiftreg
    GENERIC(LPM_WIDTH: POSITIVE; LPM_DIRECTION: STRING);
    PORT (
         clock, shiftin : IN  STD_LOGIC;
         shiftout       : OUT STD_LOGIC);
END COMPONENT;
BEGIN
    shift_8: lpm)_shiftreg
       GENERIC MAP (LPM_WIDTH=> 8, LPM_DIRECTION => "RIGHT")
       PORT MAP (clk, serial_in, serial_out);
END lpm_shift;
```

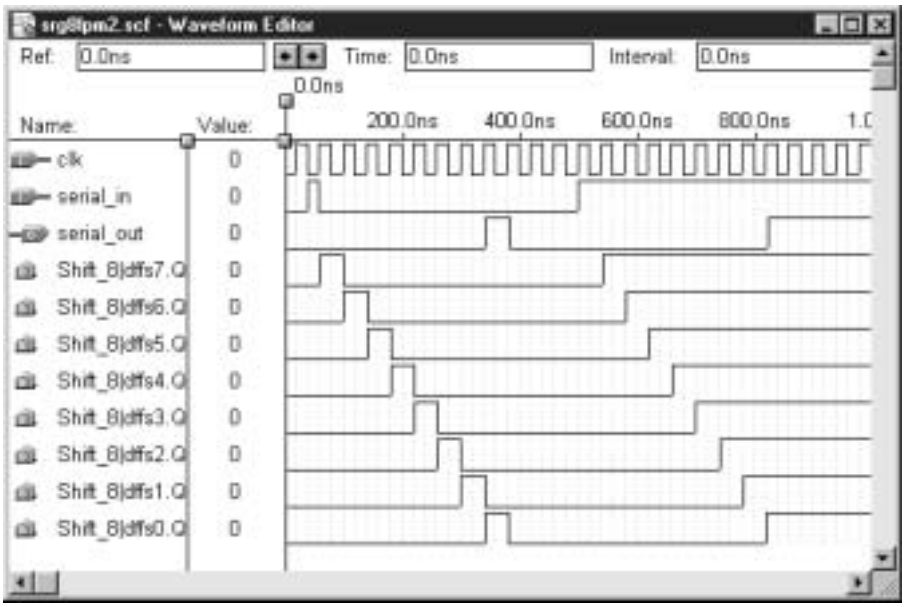


FIGURE 9.75
Example 9.17
Simulation of an 8-bit LPM Shift Register (Shift Right)

The simulation for the right-shift register is shown in Figure 9.75. The inputs are identical to those of Figure 9.74, but the internal shift direction is opposite. The LPM component configures the serial shift input and output such that they allow data to go through the entire register, regardless of shift direction. For left-shift, **serial_in (shiftin)** is applied to **D0**, and is shifted toward **Q7**. For right-shift, the same **serial_in** is applied to **D7** and shifted toward **Q0**. Thus, there is no right shift input or left shift input in this component, and also no bidirectional shift that can be controlled by an input port. Shift direction can only be set by the value of a parameter and is therefore fixed when a component is instantiated.

EXAMPLE 9.18

Write the VHDL code for an 8-bit LPM shift register with both parallel and serial outputs, parallel load and asynchronous clear. Create a simulation to verify the design operation.

Solution The VHDL code for the parallel load shift register follows as **srg8lpm3**.

srg8lpm3.vhd
srg8lpm3.scf

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

ENTITY srg8lpm3 IS
    PORT (
        clk, ld, clr   : IN   STD_LOGIC;
        p              : IN   STD_LOGIC_VECTOR(7 downto 0);
        q_out          : OUT  STD_LOGIC_VECTOR(7 downto 0);
        serial_out     : OUT  STD_LOGIC);
END srg8lpm3;

ARCHITECTURE lpm_shift of srg8lpm3 IS
COMPONENT lpm_shiftreg
    GENERIC(LPM_WIDTH: POSITIVE);
    PORT (
         clock, load     : IN   STD_LOGIC;
         aclr            : IN   STD_LOGIC;
         data            : IN   STD_LOGIC_VECTOR (7 downto 0);
         q               : OUT  STD_LOGIC_VECTOR (7 downto 0);
         shiftout        : OUT  STD_LOGIC);
END COMPONENT;
BEGIN
```

```
    Shift_8: lpm_shiftreg
        GENERIC MAP (LPM_WIDTH=> 8)
        PORT MAP (clk, ld, clr, p, q_out, serial_out);
END lpm_shift;
```

The simulation for **srg8lpm3** is shown in Figure 9.76. The load input is initially HIGH, causing the shift register to load 55H (= 01010101) on the first clock pulse. Since we have not instantiated the serial input **shiftin**, the serial input reverts to a default value of `'1'`, causing the register to be filled with 1s. If we did not want this to be the case, we would have to instantiate the **shiftin** port and set it to `'0'`.

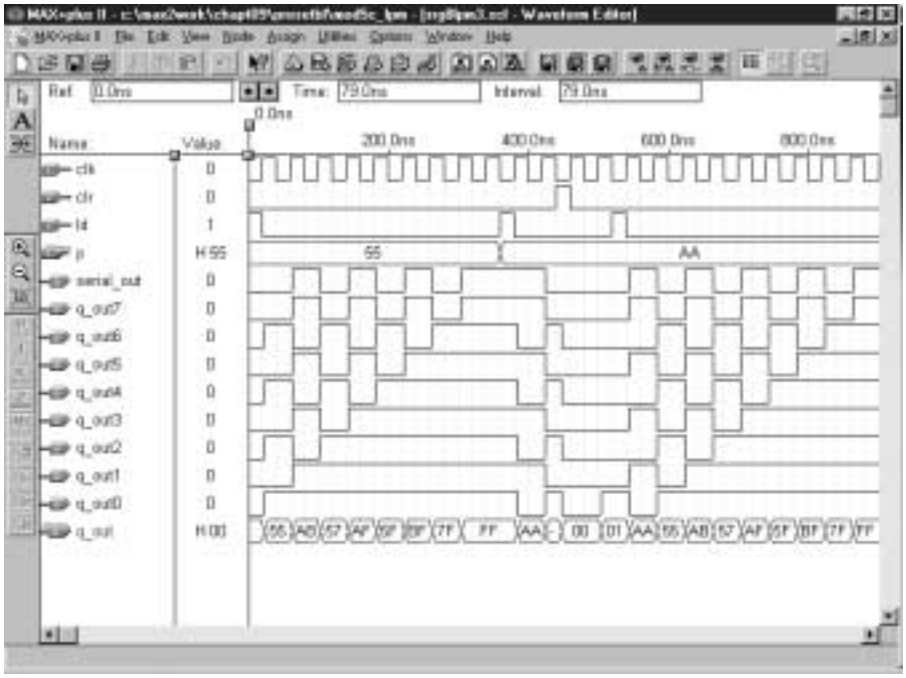


FIGURE 9.76
Example 9.18
Simulation of an 8-bit LPM Shift Register with Parallel Load

SECTION 9.8 REVIEW PROBLEM

9.8 When a shift register is encoded in VHDL, why are its outputs defined as BUFFER, not OUT?

9.9 Shift Register Counters

KEY TERMS

Ring counter A serial shift register with feedback from the output of the last flip-flop to the input of the first.

Johnson counter A serial shift register with complemented feedback from the output of the last flip-flop to the input of the first. Also called a twisted ring counter

By introducing feedback into a serial shift register, we can create a class of synchronous counters based on continuous circulation, or rotation, of data.

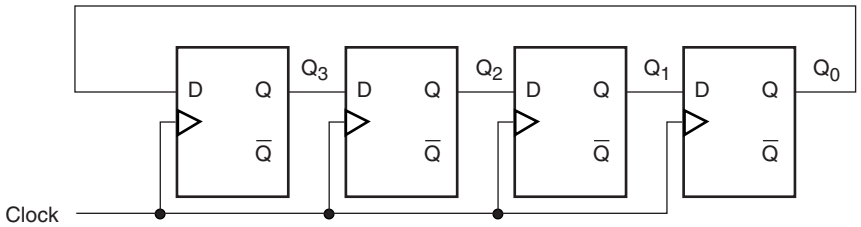
If we feed back the output of a serial shift register to its input without inversion, we create a circuit called a **ring counter**. If we introduce inversion into the feedback loop, we have a circuit called a **Johnson counter**. These circuits can be decoded more easily than binary counters of similar size and are particularly useful for event sequencing.

Ring Counters

Figure 9.77 shows a 4-bit ring counter made from D flip-flops. This circuit could also be constructed from SR or JK flip-flops, as can any serial shift register.

A ring counter circulates the same data in a continuous loop. This assumes that the

FIGURE 9.77
4-bit Ring Counter



data have somehow been placed into the circuit upon initialization, usually by synchronous or asynchronous preset and clear inputs, which are not shown.

Figure 9.78 shows the circulation of a logic 1 through a 4-bit ring counter. If we assume that the circuit is initialized to the state $Q_3Q_2Q_1Q_0 = 1000$, it is easy to see that the 1 is shifted one place right with each clock pulse. The feedback connection from Q_0 to D_3 ensures that the input of flip-flop 3 will be filled by the contents of Q_0, thus recirculating the initial data. The final transition in the sequence shows the 1 recirculated to Q_3.

A ring counter is not restricted to circulating a logic 1. We can program the counter to circulate any data pattern we happen to find convenient.

Figure 9.79 shows a ring counter circulating a 0 by starting with an initial state of $Q_3Q_2Q_1Q_0 = 0111$. The circuit is the same as before; only the initial state has changed. Figure 9.80 shows the timing diagrams for the circuit in Figures 9.78 and 9.79.

Ring Counter Modulus and Decoding

The maximum modulus of a ring counter is the maximum number of unique states in its count sequence. In Figures 9.78 and 9.79, the ring counters each had a maximum modulus of 4. We say that 4 is the *maximum* modulus of the ring counters shown, since we can change the modulus of a ring counter by loading different data at initialization.

For example, if we load a 4-bit ring counter with the data $Q_3Q_2Q_1Q_0 = 1000$, the following unique states are possible: 1000, 0100, 0010, and 0001. If we load the same circuit with the data $Q_3Q_2Q_1Q_0 = 1010$, there are only two unique states: 1010 and 0101. Depending on which data are loaded, the modulus is 4 or 2.

Most input data in this circuit will yield a modulus of 4. Try a few combinations.

NOTE

The maximum modulus of a ring counter is the same as the number of bits in its output.

A ring counter requires more flip-flops than a binary counter to produce the same number of unique states. Specifically, for n flip-flops, a binary counter has 2^n unique states and a ring counter has n.

This is offset by the fact that a ring counter requires no decoding. A binary counter used to sequence eight events requires three flip-flops andeight 3-input decoding gates. To perform the same task, a ring counter requires eight flip-flops and no decoding gates.

As the number of output states of an event sequencer increases, the complexity of the decoder for the binary counter also increases. A circuit requiring 16 output states can be implemented with a 4-bit binary counter and sixteen 4-input decoding gates. If you need 18 output states, you must have a 5-bit counter ($2^4 \leq 18 \leq 2^5$) and eighteen 5-input decoding gates.

The only required modification to the ring counter is one more flip-flop for each addi-

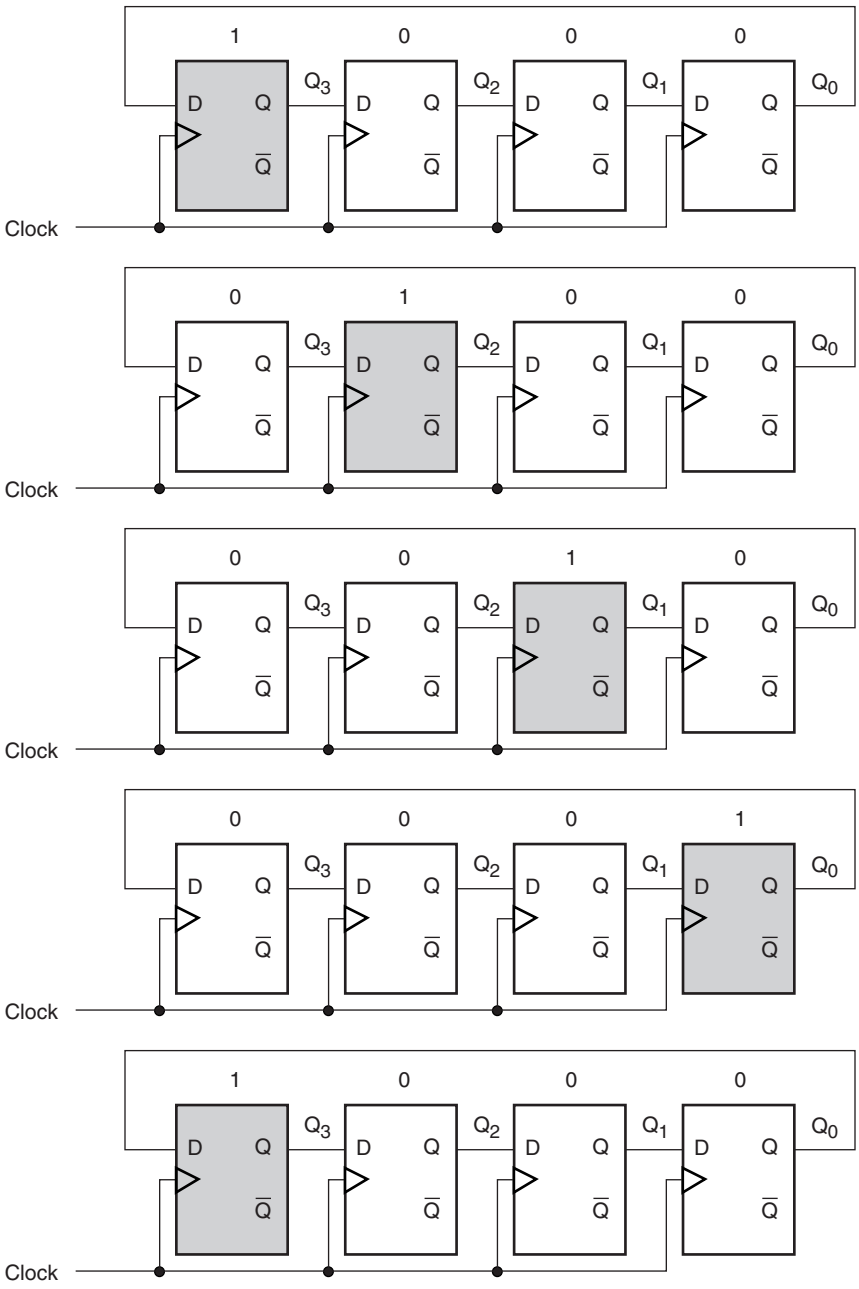


FIGURE 9.78
Circulating a 1 in a Ring Counter

tional state. A 16-state ring counter needs 16 flip-flops and an 18-state ring counter must have 18 flip-flops. No decoding is required for either circuit.

Johnson Counters

Figure 9.81 shows a 4-bit Johnson counter constructed from D flip-flops. It is the same as a ring counter except for the inversion in the feedback loop where $\overline{Q}_0$ is connected to D_3. The circuit output is taken from flip-flop outputs Q_3 through Q_0. Since the feedback introduces a "twist" into the recirculating data, a Johnson counter is also called a "twisted ring

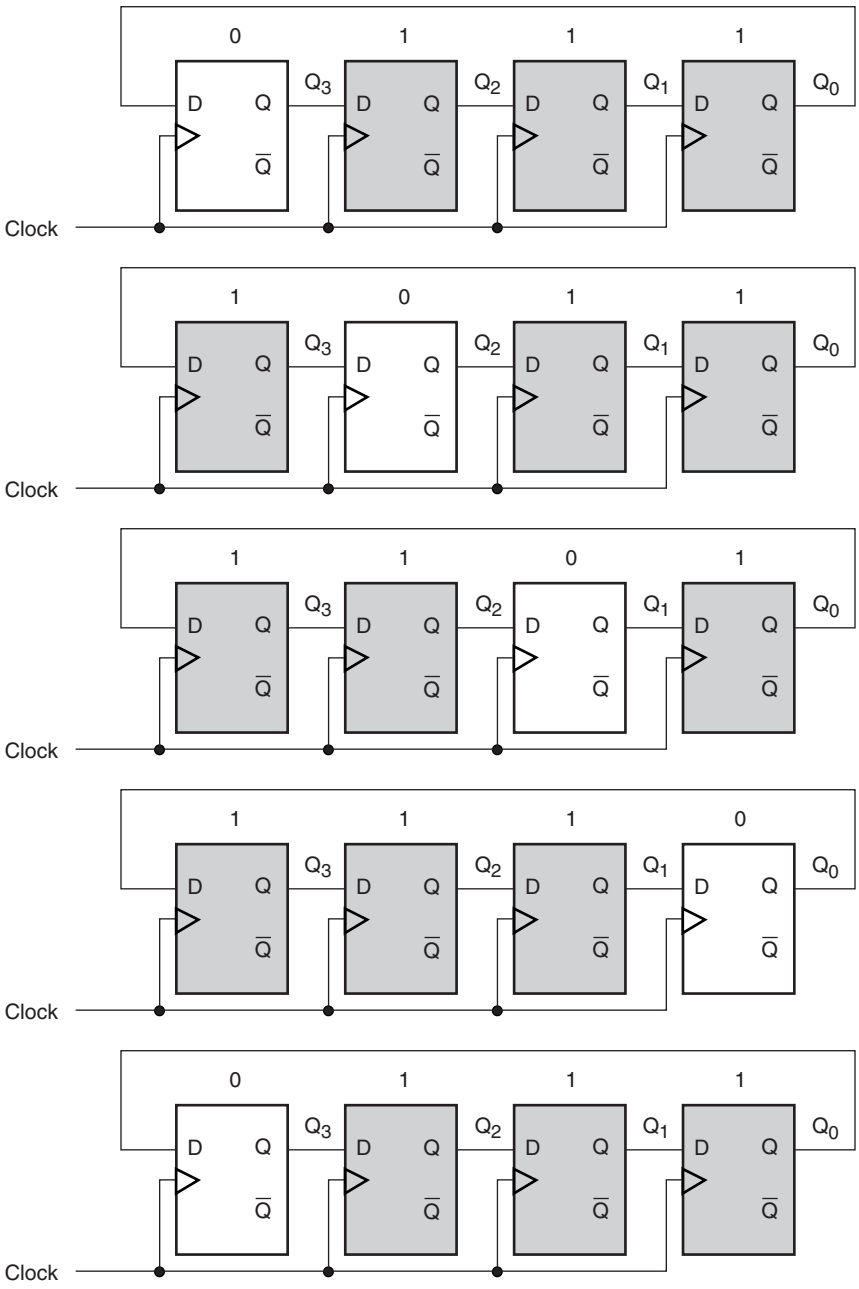


FIGURE 9.79
Circulating a 0 in a Ring Counter

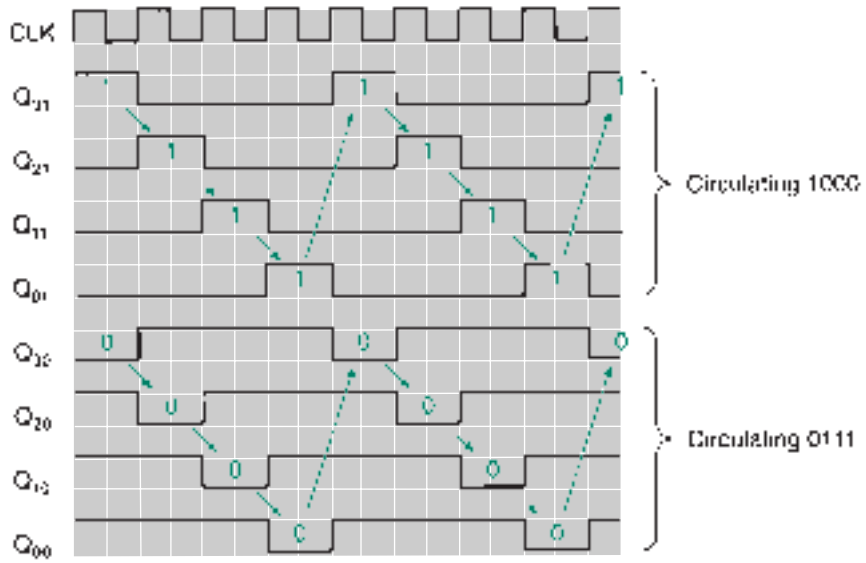


FIGURE 9.80
Timing Diagrams for Figures 9.78 and 9.79

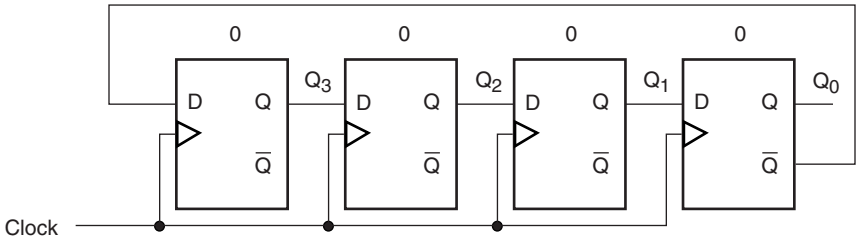


FIGURE 9.81
4-bit Johnson Counter

Table 9.17 Count Sequence of a 4-bit Johnson Counter

Q_3	Q_2	Q_1	Q_0
0	0	0	0
1	0	0	0
1	1	0	0
1	1	1	0
1	1	1	1
0	1	1	1
0	0	1	1
0	0	0	1

counter."

Figure 9.82 shows the progress of data through a Johnson counter that starts cleared ($Q_3Q_2Q_1Q_0 = 0000$). The shaded flip-flops represents 1s and the unshaded flip-flops are 0s.

Every 0 at Q_0 is fed back to D_3 as a 1 and every 1 is fed back as a 0. The count sequence for this circuit is given in Table 9.17. There are 8 unique states in the count sequence table.

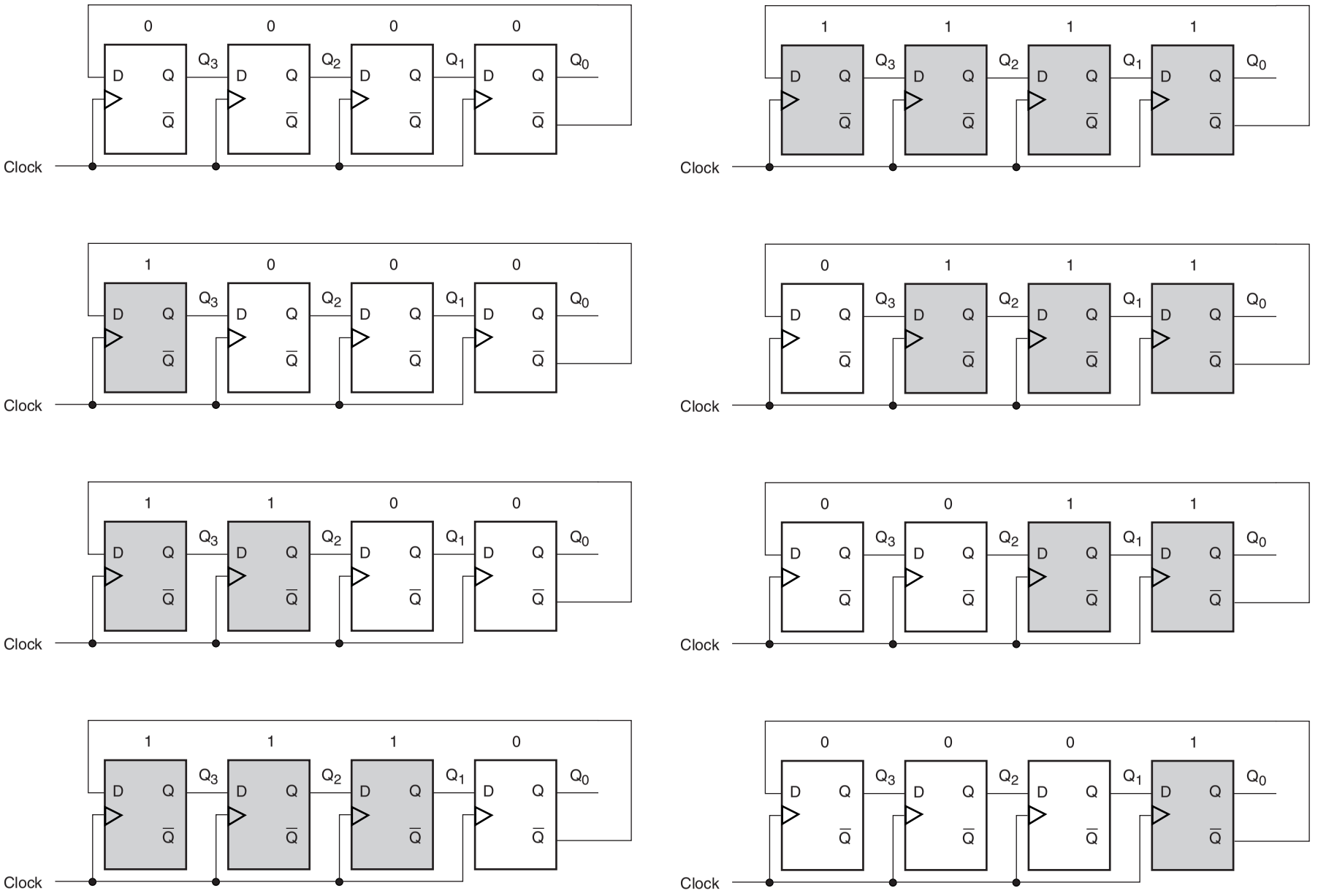


FIGURE 9.82
Data Circulation in a 4-bit Johnson Counter

EXAMPLE 9.19

Write the VHDL code for a Johnson counter of generic width and instantiate it as an 8-bit 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).

Solution The VHDL design entities for the generic-width component and the 8-bit Johnson counter follow.

jnsn_ct.vhd
jnsn_ct8.vhd
jnsn_ct8.scf

```
-- jnsn_ct.vhd
-- Johnson counter of generic width
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY jnsn_ct IS
  GENERIC (width : POSITIVE := 4);
  PORT (
       clk, clr : IN       STD_LOGIC;
       q        : BUFFER   STD_LOGIC_VECTOR(width-1 downto 0) );
END   jnsn_ct;

ARCHITECTURE johnson_counter of jnsn_ct IS
BEGIN
  PROCESS (clk)
  BEGIN
       IF (clk'EVENT and clk = '1') THEN
          IF clr = '0' THEN
               q <= (others => '0'); -- n-bit clear function (n = width)
       ELSE
               q(width-1 downto 0) <= (not q(0) ) & q(width-1 downto 1);
       END IF;
     END IF;
  END PROCESS;
END johnson_counter;

-- jnsn_ct8.vhd
-- 8-bit Johnson counter using component jnsn_ct
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY jnsn_ct8 IS
 PORT(
     clock, clear : IN      STD_LOGIC;
     qo           : BUFFER  STD_LOGIC_VECTOR(7 downto 0));
END jnsn_ct8;

ARCHITECTURE johnson_counter of jnsn_ct8 IS
COMPONENT jnsn_ct GENERIC (width : POSITIVE);
 PORT(
     clk, clr : IN         STD_LOGIC;
     q        : BUFFER     STD_LOGIC_VECTOR(7 downto 0) );
END  COMPONENT;
BEGIN
  johnson: jnsn_ct
     GENERIC MAP (width=> 8)
     PORT MAP (clk => clock
               clr  => clear,
                q      => qo);
END johnson_counter;
```

Table 9.18 Count Sequence of an 8-bit Johnson Counter

$Q_7Q_6Q_5Q_4Q_3Q_2Q_1Q_0$
00000000
10000000
11000000
11100000
11110000
11111000
11111100
11111110
11111111
01111111
00111111
00011111
00001111
00000111
00000011
00000001

Note that in the component file **(jnsn_ct.vhd),** the counter is cleared synchronously by the statement (`q <= (others => '0');`). Recall that the clause `(others => '0')` can be used to set all bits of a signal aggregate to the value `'0'`. This is a simple way to clear a vector of unknown width without using a conversion function.

Table 9.18 shows the count sequence for the 8-bit Johnson counter.

The simulation of the Johnson counter, including one full cycle and a clear, is shown in Figure 9.83.

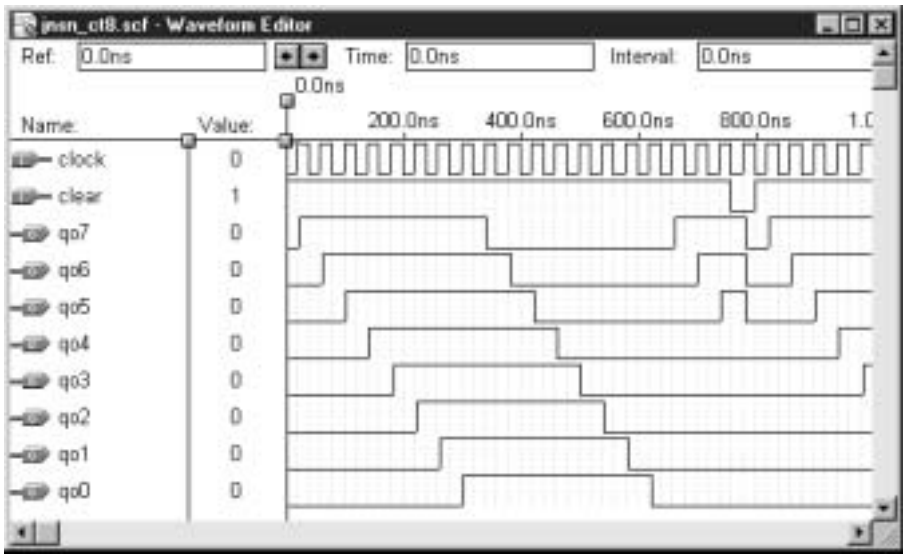


FIGURE 9.83
Example 9.19
Simulation of an 8-bit Johnson Counter

Johnson Counter Modulus and Decoding

NOTE

The maximum modulus of a Johnson counter is $2n$ for a circuit with n flip-flops.

The Johnson counter represents a compromise between binary and ring counters, whose maximum moduli are, respectively, 2^n and n for an n-bit counter.

If it is used for event sequencing, a Johnson counter must be decoded, unlike a ring counter. Its output states are such that each state can be decoded uniquely by a 2-input AND or NAND gate, depending on whether you need active-HIGH or active-LOW indication. This yields a simpler decoder than is required for a binary counter.

Table 9.19 shows the decoding of a 4-bit Johnson counter.

Table 9.19 Decoding a 4-bit Johnson Counter

Q_3	Q_2	Q_1	Q_0	Decoder Outputs	Comment
0	0	0	0	$\overline{Q}_3\overline{Q}_0$	MSB = LSB = 0
1	0	0	0	$Q_3\overline{Q}_2$	"1/0"
1	1	0	0	$Q_2\overline{Q}_1$	Pairs
1	1	1	0	$Q_1\overline{Q}_0$	
1	1	1	1	Q_3Q_0	MSB = LSB = 1
0	1	1	1	$\overline{Q}_3Q_2$	"0/1"
0	0	1	1	$\overline{Q}_2Q_1$	Pairs
0	0	0	1	$\overline{Q}_1Q_0$	

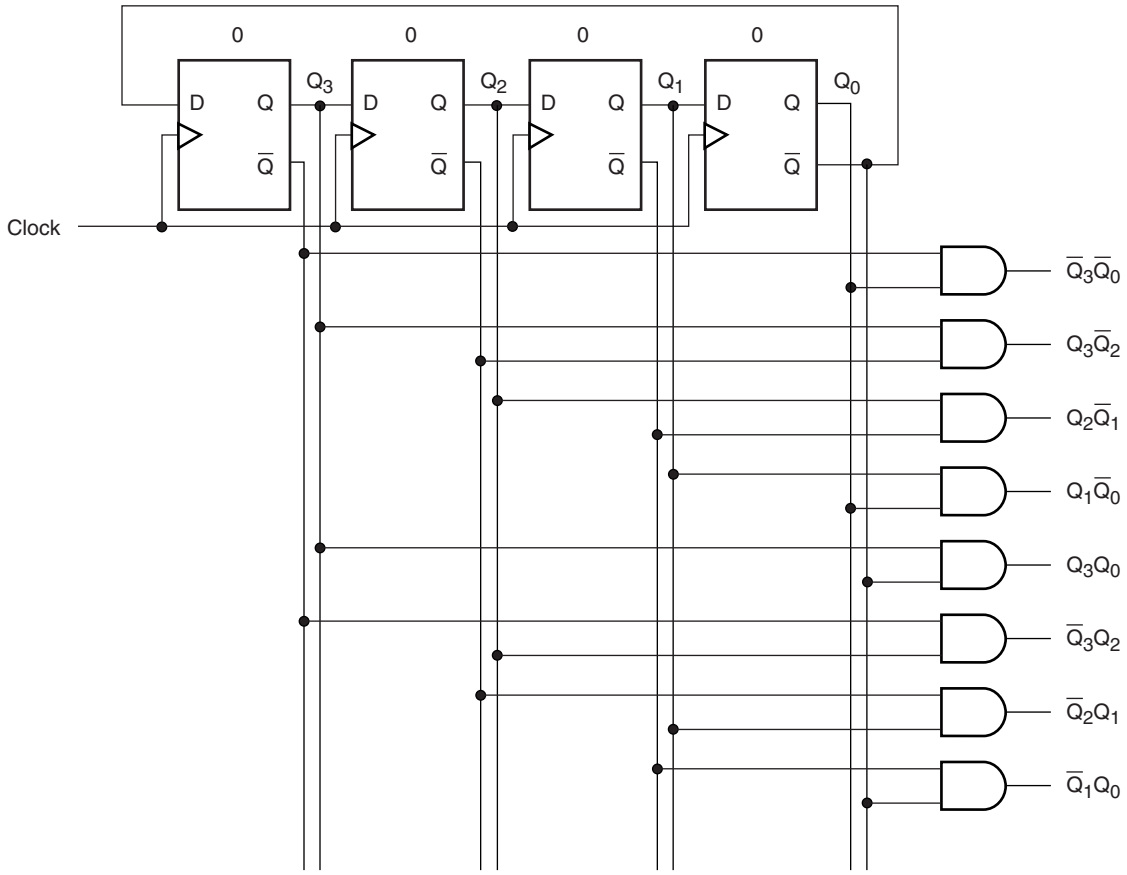


FIGURE 9.84
4-bit Johnson Counter with Output Decoding

Decoding a sequential circuit depends on the decoder responding uniquely to every possible state of the circuit outputs. If we want to use only 2-input gates in our decoder, it must recognize two variables for every state that are *both* active *only* in that state.

A Johnson counter decoder exploits what might be called the "1/0 interface" of the count sequence table. Careful examination of Tables 9.17 and 9.18 reveals that for every state, except where the outputs are all 1s or all 0s, there is a side-by-side 10 or 01 pair which exists only in that state.

Each of these pairs can be decoded to give unique indication of a particular state. For example, the pair $Q_3\overline{Q}_2$ uniquely indicates the second state since $Q_3 = 1$ AND $Q_2 = 0$ *only* in the second line of the count sequence table. (This is true for any size of Johnson counter; compare the second lines of Tables 9.17 and 9.18. In the second line of both tables, the MSB is 1 and the 2nd MSB is 0.)

For the states where the outputs are all 1s or all 0s, the most significant AND least significant bits can be decoded uniquely, these being the only states where MSB = LSB.

Figure 9.84 shows the decoder circuit for a 4-bit Johnson counter.

The output decoder of a Johnson counter does not increase in complexity as the modulus of the counter increases. The decoder will always consist of $2n$ 2-input AND or NAND gates for an n-bit counter. (For example, for an 8-bit Johnson counter, the decoder will consist of sixteen 2-input AND or NAND gates.)

EXAMPLE 9.20

Draw the timing diagram of the Johnson counter decoder of Figure 9.84, assuming the counter is initially cleared.

Solution Figure 9.85 shows the timing diagram of the Johnson counter and its decoder outputs.

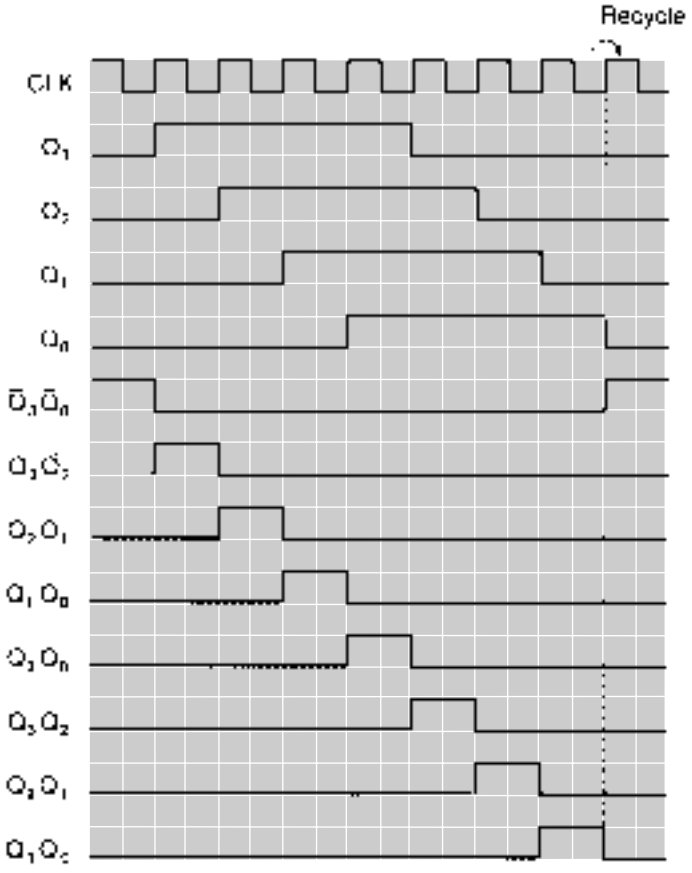


FIGURE 9.85
Example 9.20
Johnson Counter Decoder Outputs

SECTION 9.9 REVIEW PROBLEM

9.9 How many flip-flops are required to produce 24 unique states in each of the following types of counters: binary counter, ring counter, Johnson counter? How many and what type of decoding gates are required to produce an active-LOW decoder for each type of counter?

SUMMARY

1. A counter is a circuit that progresses in a defined sequence at the rate of one state per clock pulse.
2. The modulus of a counter is the number of states through which the counter output progresses before repeating.
3. A counter with an ascending sequence of states is called an UP counter. A counter with a descending sequence of states is called a DOWN counter.
4. In general, the maximum modulus of a counter is given by 2^n for an n-bit counter.
5. A counter whose modulus is 2^n is called a full-sequence counter. The count progresses from 0 to $2^n - 1$, which corresponds to a binary output of all 0s to all 1s.
6. A counter whose output is less than 2^n is called a truncated sequence counter.
7. The adjacent outputs of a full-sequence binary counter have a frequency ratio of 2:1. The less significant of the two bits has the higher frequency.
8. The outputs of a truncated sequence counter do not necessarily have a simple frequency relationship.
9. A synchronous counter consists of a series of flip-flops, all clocked from the same source, that stores the present state of the counter and a combinational circuit that monitors the counter's present state and determines its next state.
10. A synchronous counter can be analyzed by a formal procedure that includes the following steps:
    a. Write the Boolean equations for the synchronous inputs of the counter flip-flops in terms of the present state of the flip-flip outputs.
    b. Evaluate each Boolean equation for an initial state to find the states of the synchronous inputs.
    c. Use flip-flop function tables to determine each flip-flop next state.
    d. Set the next state to the new present state.
    e. Continue until the sequence repeats.
11. The analysis procedure above should be applied to any unused states of the counter to ensure that they will enter the count sequence properly.
12. A synchronous counter can be designed using a formal method that relies on the excitation tables of the flip-flops used in the counter. An excitation table indicates the required logic levels on the flip-flop inputs to effect a particular transition.
13. The synchronous counter design procedure is based on the following steps:
    a. Draw the state diagram of the counter and use it to list the relationship between the counter's present and next states. The table should list the counter's present states in binary order.
    b. For the initial design, unused states can be set to a known destination, such as 0, or treated as don't care states.
    c. Use the flip-flop excitation table to determine the synchronous input levels for each present-to-next state transition.
    d. Use Boolean algebra or Karnaugh maps to find the simplest equations for the flip-flop inputs *(JK, D,* or *T)* in terms of *Q*.
    e. Unused states should be analyzed by substituting their values into the Boolean equations of the counter. This will verify whether or not an unused state will enter the count sequence properly.
14. If a counter must reset to 0 from an unused state, the flip-flops can be reset asynchronously to their initial states or the counter can be designed with the unused states always having 0 as their next state.
15. A counter can be designed in VHDL by using a behavioral description or a structural design that uses a component from the Library of Parameterized Modules (LPM).
16. A behavioral counter design requires a PROCESS statement that lists the clock signal and any asynchronous inputs in its sensitivity list. An IF statement inside the PROCESS can monitor the active clock edge by using the predefined EVENT attribute (e.g., clk′EVENT) and increment a count variable.
17. A variable is local to a PROCESS and is assigned with the := operator. A signal is global to the VHDL design entity and is assigned with the <= operator. (Recall that a signal is like an internal connecting wire and a variable is a piece of working memory.)
18. A structural counter design can use an LPM component (lpm_counter) and instantiate the component in a component instantiation statement. The statement's generic map specifies the component parameters, and its port map indicates the correspondence between the component port names and the user port, signal, or variable names.
19. Some of the most common control features available in synchronous counters include:
    a. Synchronous or asynchronous parallel load, which allows the count to be set to any value whenever a LOAD input is asserted
    b. Synchronous or asynchronous clear (reset), which sets all of the counter outputs to zero
    c. Count enable, which allows the count sequence to progress when asserted and inhibits the count when deasserted
    d. Bidirectional control, which determines whether the counter counts up or down

e. Output decoding, which activates one or more outputs when detecting particular states on the counter outputs
f. Ripple carry out or ripple clock out (RCO), a special case of output decoding that produces a pulse upon detecting the terminal count, or last state, of a count sequence

20. The parallel load function of a counter requires load *data* (the parallel input values) and a load *command* input, such as *LOAD,* that transfer the parallel data when asserted. If the load function is synchronous, a clock pulse is also required.
21. Synchronous load transfers data to the counter outputs on an active clock edge. Asynchronous load operates as soon as the load input activates, without waiting for the clock.
22. Synchronous load is implemented by a function select circuit that selects either the count logic or the direct parallel input to be applied to the synchronous input(s) of a flip-flop.
23. Asynchronous load is implemented by enabling or inhibiting a pair of NAND gates, one of which asserts a flip-flop clear input and the other of which asserts a preset input for the same flip-flop.
24. The count enable function enables or disables the count logic of a counter without affecting other functions, such as clock or clear. This can be done by ANDing the count logic with the count enable input signal.
25. A flip-flop in an UP counter toggles when all previous bits are HIGH. A flip-flop in a DOWN counter toggles when all previous bits are LOW. A circuit that selects one of these two conditions (a pair of AND-shaped gates, combined in an OR gate; essentially a 2-to-1 multiplexer) can implement a bidirectional count.
26. An output decoder asserts one output for each counter state. A special case is a terminal count decoder that detects the last state of a count sequence.
27. RCO (ripple clock out) generates one clock pulse upon terminal count, with its positive edge at the end of the count cycle.
28. Asynchronous inputs to a behaviorally defined counter in VHDL must be included in the sensitivity list of the process defining the counter. Asynchronous inputs must be checked inside the process before the clock is checked for an active edge.
29. Synchronous inputs to a behaviorally defined counter should not be included in the sensitivity list of the process defining the counter. Synchronous inputs must be checked inside the IF statement that checks the clock edge.
30. A shift register is a circuit for storing and moving data. Three basic movements in a shift register are: serial (from one flip-flop to another), parallel (into all flip-flops at once), and rotation (serial shift with a connection from the last flip-flop output to the first flip-flop input).
31. Serial shifting can be left (toward the MSB) or right (away from the MSB). This is the convention used by MAX+PLUS II. Some data sheets indicate the opposite relationship between right/left and LSB/MSB.
32. A function select circuit can implement several shift register variations: bidirectional serial shift, parallel load with serial shift, and universal shift (parallel/serial in/out and bidirectional in one device). The circuit directs data to the *D* inputs of each flip-flop from one of several sources, such as from the flip-flop immediately to the left or right or from an external parallel input.
33. A shift register can be created in VHDL by the structural, dataflow, or behavioral method.
34. A structural design instantiates components, such as D flip-flops, and connects them with internal signals.
35. A dataflow design uses internal Boolean relationships between inputs and outputs. It is similar to a structural model, except that it must contain a process to create the flip-flops.
36. A behavioral design method uses a description of the shift register function to generate the required hardware.
37. A VHDL component can be created with parameters (such as width) that are specified when the component is instantiated. The parameters are listed in a GENERIC clause in the component's entity declaration. Each parameter must be given a default value. The parameters are specified in a generic map in the design entity that instantiates the component.
38. A ring counter is a serial shift register with the serial output fed back to the serial input so that the internal data is continuously circulated. The initial value is generally set by asynchronous preset and clear functions.
39. The maximum modulus of a ring counter is n for a circuit with n flip-flops, as compared to 2^n for a binary counter. A ring counter output is self-decoding, whereas a binary counter requires $m \leq 2^n$ AND or NAND gates with n inputs each.
40. A Johnson counter is a ring counter where the feedback is complemented. A Johnson counter has $2n$ states for an n-bit counter which can be uniquely decoded by $2n$ 2-input AND or NAND gates.

GLOSSARY

Attribute A property associated with a named identifier in VHDL. (e.g., the attribute EVENT, when associated with the identifier clk (written clk′EVENT), indicates whether a transition has occurred on the input called clk.)

Behavioral design A VHDL design technique that uses descriptions of required behavior to describe the design.

Bidirectional counter A counter that can count up or down, depending on the state of a control input.

Bidirectional shift register A shift register that can serially shift bits left or right according to the state of a direction control input.

Binary counter A counter that generates a binary count sequence.

Clear Reset (synchronous or asynchronous)

Command lines Signals that connect the control section of a synchronous circuit to its memory section and direct the circuit from its present to its next state.

Conditional signal assignment statement A signal assignment statement that is executed only when a Boolean condition is satisfied.

Control section The combinational logic portion of a synchronous circuit that determines the next state of the circuit.

Count enable A control function that allows a counter to progress through its count sequence when active and disables the counter when inactive.

Count sequence The specific series of output states through which a counter progresses.

Counter A sequential digital circuit whose output progresses in a predictable repeating pattern, advancing by one state for each clock pulse.

Count-sequence table A list of counter states in the order of the count sequence.

Dataflow design A VHDL design technique that uses Boolean equations to define relationships between inputs and outputs.

DOWN counter A counter with a descending sequence.

Excitation table A table showing the required input conditions for every possible transition of a flip-flop output.

Full-sequence counter A counter whose modulus is the same as its maximum modulus ($m = 2^n$ for an n-bit counter).

GENERIC A clause in the entity declaration of a VHDL component that lists the parameters that can be specified when the component is instantiated.

Johnson counter A serial shift register with complemented feedback from the output of the last flip-flop to the input of the first. Also called a twisted ring counter

Left shift A movement of data from the right to the left in a shift register. (Left is defined in MAX+PLUS II as toward the MSB.)

Maximum modulus (m_{max}) The largest number of counter states that can be represented by n bits ($m_{max} = 2^n$)

Memory section A set of flip-flops in a synchronous circuit that hold its present state.

Modulo-*n* (or mod-*n*) counter A counter with a modulus of n.

Modulus The number of states through which a counter sequences before repeating.

Next state The desired future state of flip-flop outputs in a synchronous sequential circuit after the next clock pulse is applied.

Parallel load A function that allows simultaneous loading of binary values into all flip-flops of a synchronous circuit. Parallel loading can be synchronous or asynchronous.

Parallel-load shift register A shift register that can be preset to any value by directly loading a binary number into its internal flip-flops.

Parallel transfer Movement of data into all flip-flops of a shift register at the same time.

Present state The current state of flip-flop outputs in a synchronous sequential circuit.

Presettable counter A counter with a parallel load function.

Recycle To make a transition from the last state of the count sequence to the first state.

Right shift A movement of data from the left to the right in a shift register. (Right is defined in MAX+PLUS II as toward the LSB.)

Ring counter A serial shift register with feedback from the output of the last flip-flop to the input of the first.

Ripple carry out or ripple clock out (RCO) An output that produces one pulse with the same period as the clock upon terminal count.

Rotation Serial shifting of data with the output(s) of the last flip-flop connected to the synchronous input(s) of the first flip-flop. The result is continuous circulation of the same data.

Serial shifting Movement of data from one end of a shift register to the other at a rate of one bit per clock pulse.

Shift register A synchronous sequential circuit that will store and move n-bit data, either serially or in parallel, in n flip-flops.

SRG*n* Symbol for an n-bit shift register (e.g., SRG4 indicates a 4-bit shift register).

State diagram A diagram showing the progression of states of a sequential circuit.

State machine A synchronous sequential circuit.

Status lines Signals that communicate the present state of a synchronous circuit from its memory section to its control section.

Structural design A VHDL design technique that connects predesigned components using internal signals.

Synchronous counter A counter whose flip-flops are all clocked by the same source and thus change in synchronization with each other.

Terminal count The last state in a count sequence before the sequence repeats (e.g., 1111 is the terminal count of a 4-bit binary UP counter; 0000 is the terminal count of a 4-bit binary DOWN counter).

Truncated-sequence counter A counter whose modulus is less than its maximum modulus ($m < 2^n$ for an n-bit counter)

Universal shift register A shift register that can operate with any combination of serial and parallel inputs and outputs (i.e., serial in/serial out, serial in/parallel out, parallel in/serial out, parallel in/parallel out). A universal shift register is often bidirectional, as well.

UP counter A counter with an ascending sequence.

PROBLEMS

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

9.1 Basic Concepts of Digital Counters

9.1 A parking lot at a football stadium is monitored before a game to determine whether or not there is available space for more cars. When a car enters the lot, the driver takes a ticket from a dispenser which also produces a pulse for each ticket taken.

The parking lot has space for 4095 cars. Draw a block diagram which shows how you can use a digital counter to light a LOT FULL sign after 4095 cars have entered. (Assume no cars leave the lot until after the game, so you don't need to keep track of cars leaving the lot.) How many bits should the counter have?

9.2 Figure 9.86 shows a mod-16 which controls the operation of two digital sequential circuits, labeled Circuit 1 and Circuit 2. Circuit 1 is positive edge-triggered and clocked by counter output Q_1. Circuit 2 is negative edge-triggered and clocked by Q_3. (Q_3 is the MSB output of

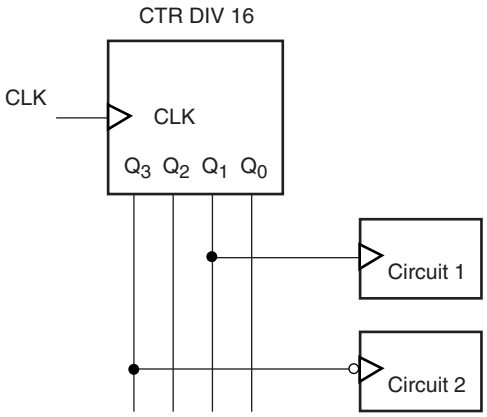


FIGURE 9.86
Problem 9.2
Mod-16 Counter Driving Two Sequential Circuits

the counter.)

a. Draw the timing diagram for one complete cycle of the circuit operation. Draw arrows on the active edges of the waveforms that activate Circuit 1 and Circuit 2.

b. State how many times Circuit 1 is clocked for each time that Circuit 2 is clocked.

9.3 Draw the timing diagram for one complete cycle of a mod-8 counter, including waveforms for *CLK*, Q_0, Q_1, and Q_2, where Q_0 is the LSB.

9.4 How many bits are required to make a counter with a modulus of 64? Why? What is the maximum count of such a counter?

9.5 **a.** Draw the state diagram of a mod-10 UP counter.

b. Use the state diagram drawn in part **a** to answer the following questions:

i. The counter is at state 0111. What is the count after 7 clock pulses are applied?

ii. After 5 clock pulses, the counter output is at 0001. What was the counter state prior to the clock pulses?

iii. The counter output is at 1000 after 15 clock pulses. What was the original output state?

9.6 What is the maximum modulus of a 6-bit counter? A 7-bit? 8-bit?

9.7 Draw the count sequence table and timing diagram of a mod-10 UP counter.

9.8 Draw the state diagram, count sequence table, and timing diagram of a mod-10 DOWN counter.

9.9 A mod-16 counter is clocked by a waveform having a frequency of 48 kHz. What is the frequency of each of the waveforms at Q_0, Q_1, Q_2, and Q_3?

9.10 A mod-10 counter is clocked by a waveform having a frequency of 48 kHz. What is the frequency of the Q_3 output waveform? The Q_0 waveform? Why is it difficult to determine the frequencies of Q_1 and Q_2?

9.2 Synchronous Counters

9.11 Draw the circuit for a synchronous mod-16 UP counter made from negative edge-triggered JK flip-flops.

9.12 Write the Boolean equations required to extend the counter drawn in Problem 9.11 to a mod-64 counter.

9.13 Write the *J* and *K* equations for the MSB of a synchronous mod-256 (8-bit) UP counter.

9.14 Analyze the operation of the synchronous counter in Figure 9.87 by drawing a state table showing all transitions, including unused states. Use this state table to draw a state diagram and a timing diagram. What is the counter's modulus?

9.15 **a.** Write the equations for the *J* and *K* inputs of each flip-

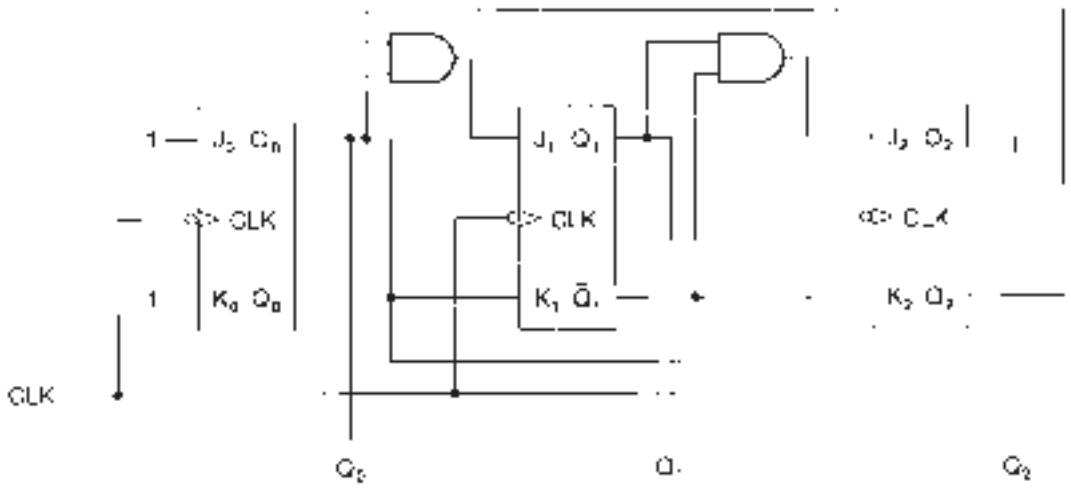


FIGURE 9.87
Problem 9.14
Synchronous Counter

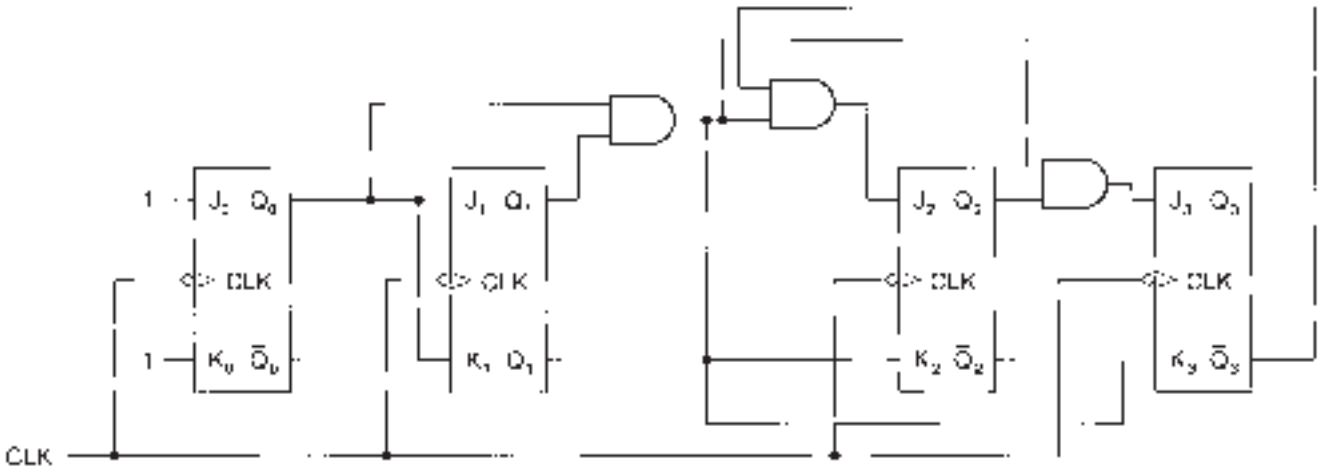


FIGURE 9.88
Problem 9.15
Synchronous Counter

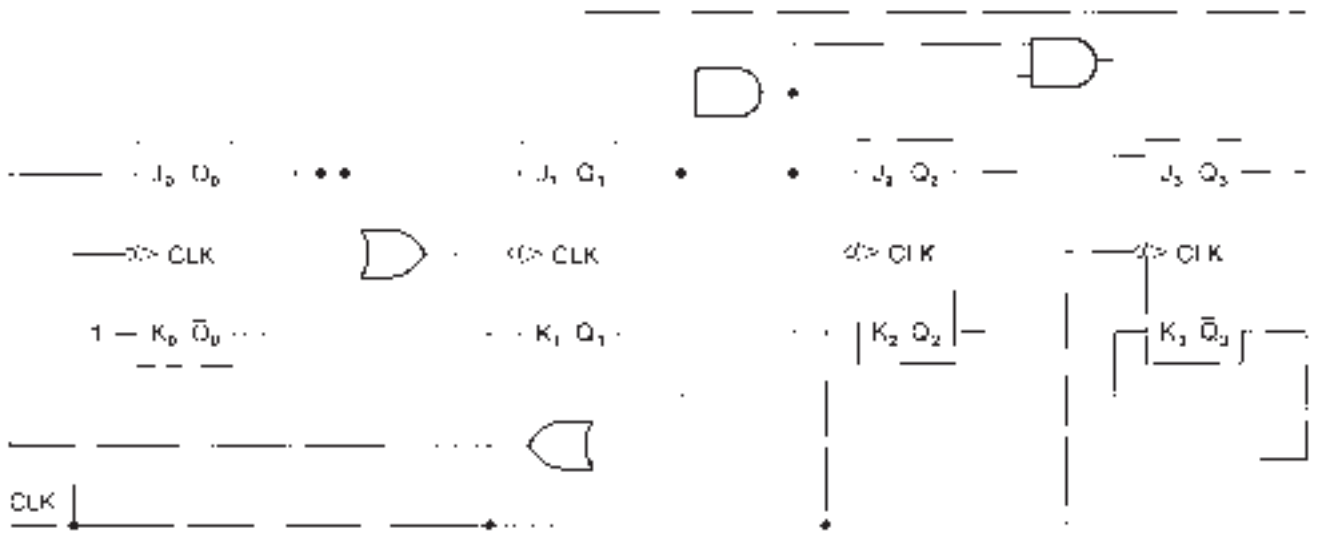


FIGURE 9.89
Problem 9.16
Counter

flop of the synchronous counter represented in Figure 9.88.

b. Assume that $Q_3Q_2Q_1Q_0 = 1010$ at some point in the count sequence. Use the equations from part **a** to predict the circuit outputs after each of three clock pulses.

9.16 Analyze the operation of the counter shown in Figure 9.89. Predict the count sequence by determining the *J* and *K* inputs and resulting transitions for each counter output state. Draw the state diagram and the timing diagram. Assume that all flip-flop outputs are initially 0.

9.3 Design of Synchronous Counters

9.17 Draw the timing diagram and state diagram of a synchronous mod-10 counter with a positive edge-triggered clock.

9.18 Design a synchronous mod-10 counter, using positive edge-triggered JK flip-flops. Check that unused states properly enter the main sequence. Draw a state diagram showing the unused states.

9.19 Design a synchronous mod-10 counter, using positive edge-triggered D flip-flops. Check that unused states properly enter the main sequence. Draw a state diagram showing the unused states.

9.20 Design a synchronous 3-bit binary counter using T flip-flops.

9.21 Table 9.20 shows the count sequence for a **biquinary sequence** counter. The sequence has ten states, but does not progress in binary order. The advantage of the sequence is that its most significant bit has a divide-by-10 ratio, relative to a clock input, and a 50% duty cycle. Design the

Table 9.20 Biquinary Sequence

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

synchronous counter circuit for this sequence, using D flip-flops. *Hint:* When making the state table, list all *present states* in binary order. The next states *will not* be in binary order.

9.4 Programming Binary Counters in VHDL

9.22 Write the VHDL code for a behavioral description of a 6-bit binary counter with asynchronous clear.

9.23 Create a simulation file in MAX+PLUS II to verify the operation of the counter in Problem 9.22. (Use a 40 ns clock, which approximates the clock period of the oscillator on the Altera UP-1 board.) ***Note:*** *To make a useful simulation, you must include the recycle point, which may be beyond the default end time of the simulation (1 μs). To change the end time, select* ***End Time*** *from the MAX+PLUS II* ***File*** *menu in the Simulator menu. To change the clock period, select* ***Grid Size*** *from the MAX+PLUS II* ***Options*** *menu in the Simulator window. The default clock period is two grid spaces.*

9.24 Write a VHDL file that instantiates a counter from the Library of Parameterized Modules to make a 12-bit binary counter. Create a MAX+PLUS II simulation to verify the operation of the counter. (Refer to the note after Problem 9.23.)

9.5 Control Options for Synchronous Counters

9.25 Briefly explain the difference between asynchronous and synchronous parallel load in a synchronous counter. Draw a partial timing diagram that illustrates both functions for a 4-bit counter.

9.26 Refer to the 4-bit counter of Figure 9.26 (p. 391). The graphic design files for the counter are found on the CD accompanying this text as **4bit_sl.gdf** and **sl_count.gdf** in the folder ***drive:*\Student_Files\Chapter09**. Copy these files to a new folder and use the MAX+PLUS II graphic editor to expand the counter of Figure 9.26 to a 5-bit counter with synchronous load and asynchronous reset. Save and compile the file to make sure that there are no design errors.

9.27 Create a MAX+PLUS II simulation to verify the functions of the counter in Problem 9.26. The simulation must include the recycle point of the counter and show that the load is really synchronous and that the reset is really asynchronous.

9.28 Refer to the 4-bit counter of Figure 9.33 (p. 396). The graphic design files for the counter are found on the accompanying CD as **4bit_sle.gdf** and **sl_count.gdf** in the folder ***drive:*\Student Files\Chapter09.** Copy these files to a new folder and modify the synchronous count element **sl_count.gdf** so that it implements an active-HIGH synchronous load and an active-LOW synchronous clear function, as well as the binary count function. Create a default symbol for the new element and substitute it in **4bit_sle.gdf** for the existing counter elements **sl_count.** The load function should have priority over count enable, and clear (reset) should have priority over both. Save and compile the new file. *Hints:* (1) The clear function makes $Q = 0$ after a clock pulse. (2) Q follows D.

9.29 Create a MAX+PLUS II simulation to verify the functions of the counter in Problem 9.28. The simulation must include the recycle point of the counter and show that the load and clear really are synchronous and that load has priority over count enable and clear has priority over both.

9.30 Derive the Boolean equations for the synchronous DOWN-counter in Figure 9.35.

9.31 Write the Boolean equations for the count logic of the 4-bit bidirectional counter in Figure 9.38. Briefly explain how the logic works.

9.32 Draw a MAX+PLUS II Graphic Design File for a bidirectional counter, using T flip-flops. Create a simulation of the counter to verify its function

9.33 Use MAX+PLUS II to create a synchronous bidirectional counter with synchronous load, asynchronous reset, and count enable. The count enable should not affect the operation of the load and reset functions. The functions should have the following priority: (1) clear; (2) load; and (3) count. Create a MAX+PLUS II simulation to verify the operation of your design.

9.6 Programming Presettable and Bidirectional Counters in VHDL

9.34 Write the VHDL code for a counter that uses a behavioral description of the following functions: 12-bit binary UP count; active-LOW asynchronous clear, active-LOW synchronous load, active-LOW count enable, terminal count decoder. The clear function should have the highest priority, followed by load, then count enable. Create a simulation in MAX+PLUS II that verifies the functions of this counter.

9.35 Write the VHDL code for a behavioral description of a bidirectional counter with a modulus of 24. The counter should also have an active-LOW synchronous clear function that has priority over the count. Create a MAX+PLUS II simulation file to verify the counter operation.

9.36 Write the VHDL code for a 4-bit counter with two decoding outputs called **eq8** and **eq12.** Out **eq8** goes HIGH when the count equals 8 and **eq12** goes HIGH when the count equals 12 (decimal). The counter should also have an active-LOW asynchronous clear function that has pri-

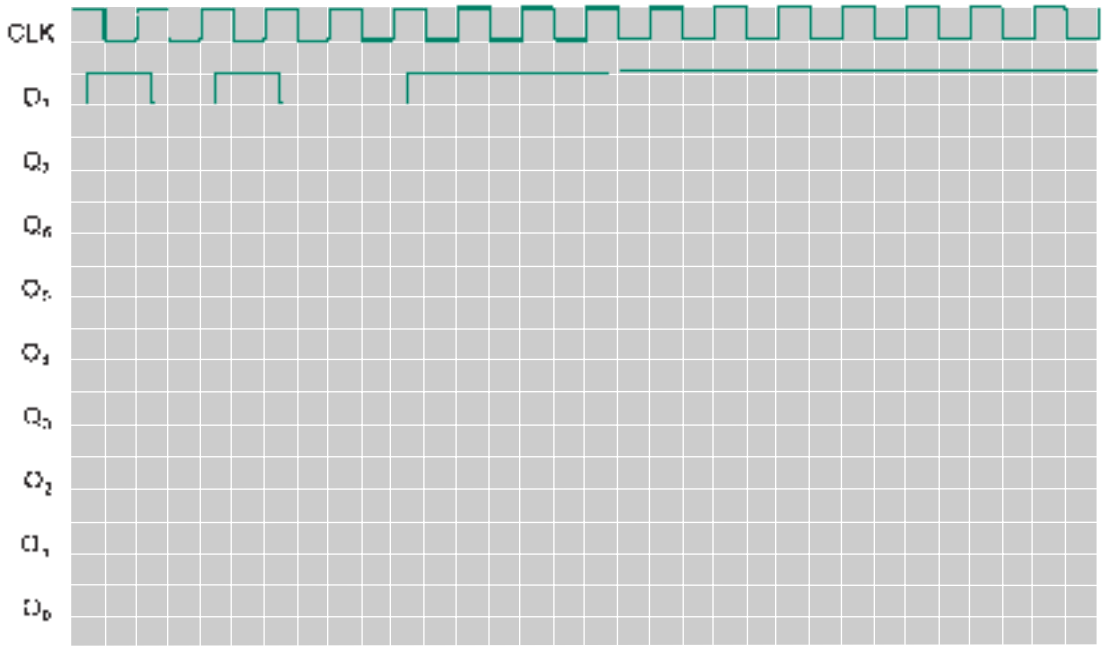


FIGURE 9.90
Problem 9.44
Timing Diagram

ority over the count. Create a MAX+PLUS II simulation file to verify the counter operation.

9.37 Modify the VHDL code in Example 9.10 (p. 412) so that the counter synchronously sets to all 1s (= 4095), rather than to 2047. Do not use SVALUE = 4095. Create a simulation in MAX+PLUS II that verifies the operation of the counter. State the main difference between the code for Example 9.10 and the solution to this problem.

9.38 Use a counter from the Library of Parameterized Modules to implement the counter described in Problem 9.35. Create a MAX+PLUS II simulation file to verify the operation of the counter.

9.39 Write a VHDL file that instantiates an 8-bit LPM count with synchronous load and clear, count enable, and directional control. Also include a terminal count decoder. (The LPM counter has no port for the terminal count function, so it must be done separately.) Create a MAX+PLUS II simulation to verify the operation of the counter.

9.7 Shift Registers

9.40 Use the MAX+PLUS II Graphic Editor to draw the circuit of a serial shift register constructed from JK flip-flops. Create a simulation to verify the operation of the shift register.

9.41 Use the MAX+PLUS II Graphic Editor to create the logic diagram of the 4-bit serial shift register based on JK flip-flops that shifts left, rather than right. Create a simulation to verify the operation of the shift register.

9.42 The following bits are applied in sequence to the input of a 6-bit serial right-shift register: 0111111 (0 is applied first). Draw the timing diagram.

9.43 After the data in Problem 9.42 are applied to the 6-bit shift register, the serial input goes to 0 for the next 8 clock pulses and then returns to 1. Write the internal states, Q_5 through Q_0, of the shift register flip-flops after the first 2 clock pulses. Write the states after 6, 8, and 10 clock pulses.

9.44 Complete the timing diagram of Figure 9.90, which is for a serial shift register (right-shift). Assume the shift register is initially cleared. What happens to the state of the circuit if D_7 stays HIGH beyond the end of the diagram and the *CLK* input continues to pulse?

9.45 An 8-bit right-shift serial-in-serial-out shift register is initially cleared and has the following data clocked into its serial input: 1011001110. Draw a timing diagram of the circuit showing the *CLK, Serial Input,* and *Serial Output.* (Assume the individual flip-flop outputs are not accessible.)

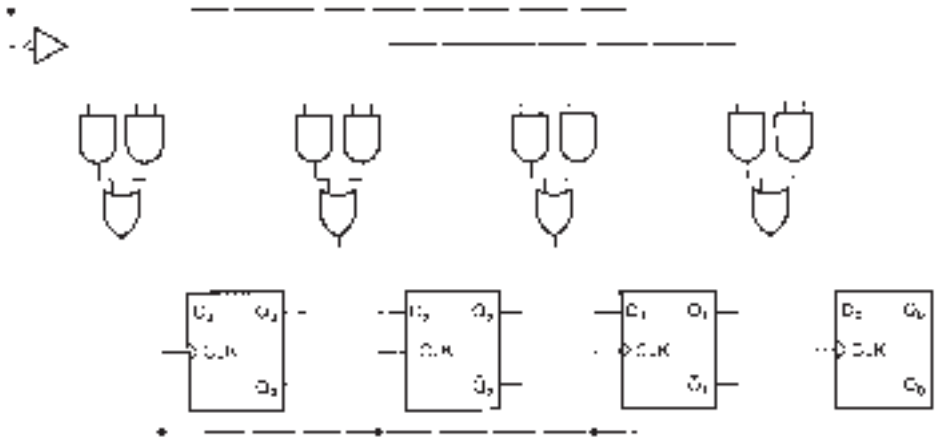


FIGURE 9.91
Problem 9.46
Logic Circuit

9.46 Complete the logic circuit shown in Figure 9.91 to make a bidirectional shift register.

9.47 Complete the logic circuit shown in Figure 9.92 to make a parallel-in-serial-out shift register.

9.8 Programming Shift Registers in VHDL

9.48 Write the VHDL code for an 8-bit serial shift register using a structural design procedure. Use JK flip-flops. (MAX+PLUS II primitive: JKFF.) Create a MAX+PLUS II simulation file to verify the operation of your design.

9.49 Repeat Problem 9.48 using a dataflow design procedure.

9.50 Modify the VHDL code for the behaviorally designed shift register **srg4behv.vhd** so that the shift register moves the data left, not right. *Hint:* The statement `q (3 downto 0) <= serial_in & q(3 downto 1);` is equivalent to the following two statements:

```
q(3) <= serial_in;
q(2 downto 0) <= q(3 downto 1);
```

Create a simulation file to verify the operation of this device.

9.51 Modify the VHDL code for the left-shift register Problem 9.50 to make a shift register of generic width. Use this component in another VHDL file to make a 32-bit shift register that shifts left. Create a simulation file to verify the operation of this design.

9.52 Write the code for a VHDL design entity that implements a 4-bit universal shift register with asynchronous clear. Create a simulation that verifies the design function.

9.53 Use MAX+PLUS II to create simulations for the generic-width and the 16-bit universal shift registers in Example 9.16 (p. 432). What is the difference in width between the default value of the generic shift register and the instantiated component in the 16-bit file? Given this difference, why can the generic-width shift register be correctly used as a component in the 16-bit design entity?

9.54 Use an LPM shift register in a VHDL file to instantiate a 48-bit shift register with the following functions: serial input, parallel output, synchronous clear.

9.55 Use an LPM shift register in a VHDL file to instantiate a 10-bit shift register with the following functions: serial input and output whose internal value can be synchro-

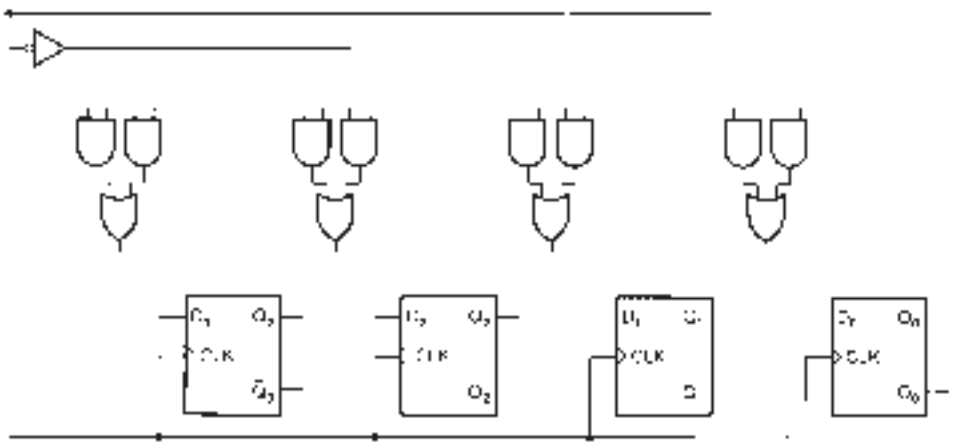


FIGURE 9.92
Problem 9.47
Logic Circuit

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

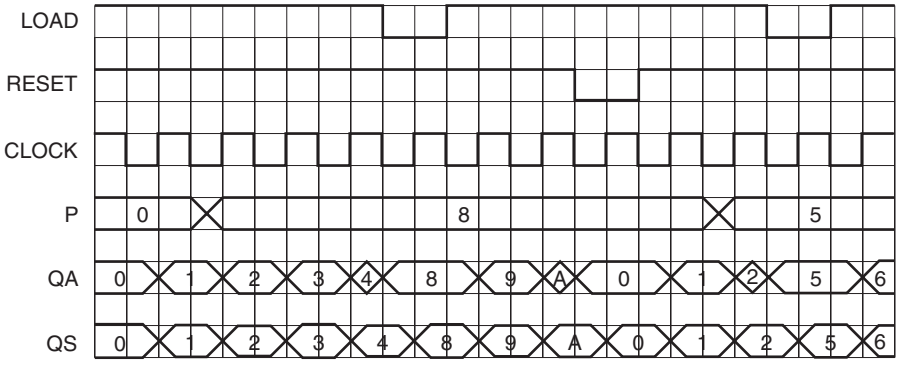


FIGURE 9.93
Answer to Section Review Problem 9.5