CHAPTER '

COUNTERS AND REGISTERS

OUTLINE

Part 1

- 7-1 Asynchronous (Ripple) Counters
- 7-2 Propagation Delay in Ripple Counters
- 7-3 Synchronous (Parallel) Counters
- 7-4 Counters with MOD Numbers $< 2^N$
- 7-5 Synchronous Down and Up/Down Counters
- 7-6 Presettable Counters
- 7-7 IC Synchronous Counters
- 7-8 Decoding a Counter
- 7-9 Analyzing Synchronous Counters
- 7-10 Synchronous Counter Design
- 7-11 Basic Counters Using HDLs
- 7-12 Full-Featured Counters in HDL

- 7-13 Wiring HDL Modules Together
- 7-14 State Machines

Part 2

- 7-15 Integrated Circuit Registers
- 7-16 Parallel In/Parallel Out— The 74ALS174/74HC174
- 7-17 Serial In/Serial Out— The 74ALS166/74HC166
- 7-18 Parallel In/Serial Out— The 74ALS165/74HC165
- 7-19 Serial In/Parallel Out— The 74ALS164/74HC164
- 7-20 Shift-Register Counters
- 7-21 Troubleshooting
- 7-22 HDL Registers
- 7-23 HDL Ring Counters
- 7-24 HDL One-Shots

OBJECTIVES

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

- Understand the operation and characteristics of synchronous and asynchronous counters.
- Construct counters with MOD numbers less than 2^N .
- Construct both up and down counters.
- Connect multistage counters.
- Analyze and evaluate various types of counters.
- Design arbitrary-sequence synchronous counters.
- Understand several types of schemes used to decode different types of counters.
- Describe counter circuits using different levels of abstraction in HDL.
- Compare the major differences between ring and Johnson counters.
- Recognize and understand the operation of various types of IC registers.
- Describe shift registers and shift register counters using HDL.
- Apply existing troubleshooting techniques used for combinational logic systems to troubleshoot sequential logic systems.

INTRODUCTION

In Chapter 5, we saw how flip-flops could be connected to function as counters and registers. At that time we studied only the basic counter and register circuits. Digital systems employ many variations of these basic circuits, mostly in integrated-circuit form. In this chapter, we will look at how FFs and logic gates can be combined to produce different types of counters and registers.

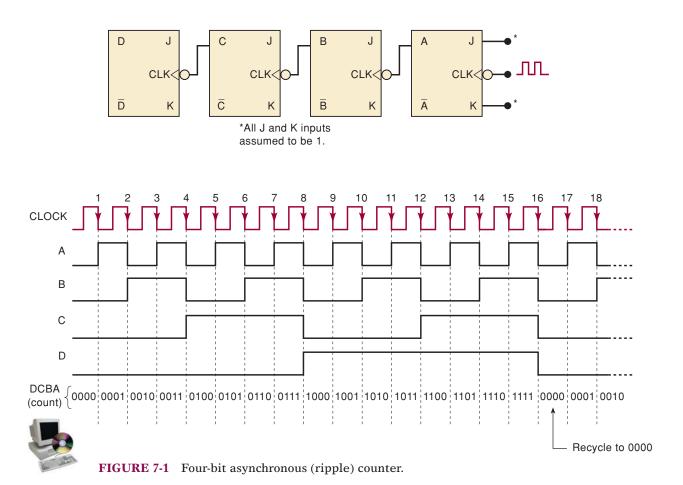
Because there are a great number of topics in this chapter, it has been divided into two parts. In **PART 1**, we will cover the principles of counter operation, the various counter circuit arrangements, and representative IC counters. **PART 2** will present several types of IC registers, shift register counters, and troubleshooting. Each part includes a section containing HDL descriptions of counters and registers.

As you progress through this chapter, you will find that you are constantly drawing on your understanding of the material we have covered in the preceding chapters. It is a good idea to go back and review previously learned material whenever you need to.

PART 1 7-1 ASYNCHRONOUS (RIPPLE) COUNTERS

Figure 7-1 shows a four-bit binary counter circuit such as the one discussed in Chapter 5. Recall the following points concerning its operation:

- 1. The clock pulses are applied only to the *CLK* input of flip-flop *A*. Thus, flip-flop *A* will toggle (change to its opposite state) each time the clock pulses make a negative (HIGH-to-LOW) transition. Note that J = K = 1 for all FFs.
- 2. The normal output of flip-flop *A* acts as the *CLK* input for flip-flop *B*, and so flip-flop *B* will toggle each time the *A* output goes from 1 to 0. Similarly, flip-flop *C* will toggle when *B* goes from 1 to 0, and flip-flop *D* will toggle when *C* goes from 1 to 0.
- 3. FF outputs *D*, *C*, *B*, and *A* represent a four-bit binary number, with *D* as the MSB. Let's assume that all FFs have been cleared to the 0 state (CLEAR inputs are not shown). The waveforms in Figure 7-1 show that a binary counting sequence from 0000 to 1111 is followed as clock pulses are continuously applied.
- 4. After the NGT of the fifteenth clock pulse has occurred, the counter FFs are in the 1111 condition. On the sixteenth NGT, flip-flop *A* goes from 1 to 0, which causes flip-flop *B* to go from 1 to 0, and so on, until the



counter is in the 0000 state. In other words, the counter has gone through one complete cycle (0000 through 1111) and has *recycled* back to 0000. From this point, it will begin a new counting cycle as subsequent clock pulses are applied.

In this counter, each FF output drives the *CLK* input of the next FF. This type of counter arrangement is called an **asynchronous counter** because the FFs do not change states in exact synchronism with the applied clock pulses; only flip-flop A responds to the clock pulses. FF B must wait for FF A to change states before it can toggle; FF C must wait for FF B, and so on. Thus, there is a delay between the responses of successive FFs. This delay is typically 5–20 ns per FF. In some cases, as we shall see, this delay can be troublesome. This type of counter is also often referred to as a **ripple counter** because of the way the FFs respond one after another in a kind of rippling effect. We will use the terms *asynchronous counter* and *ripple counter* interchangeably.

Signal Flow

It is conventional in circuit schematics to draw the circuits (wherever possible) so that the signal flow is from left to right, with inputs on the left and outputs on the right. In this chapter, we will often break with this convention, especially in diagrams showing counters. For example, in Figure 7-1, the *CLK* inputs of each FF are on the right, the outputs are on the left, and the input clock signal is shown coming in from the right. We will use this arrangement because it makes the counter operation easier to understand and follow (because the order of the FFs is the same as the order of the bits in the binary number that the counter represents). In other words, FF A (which is the LSB) is the rightmost FF, and FF D (which is the MSB) is the leftmost FF. If we adhered to the conventional left-to-right signal flow, we would have to put FF A on the left and FF D on the right, which is opposite to their positions in the binary number that the counter represents. In some of the counter diagrams later in the chapter, we will employ the conventional left-to-right signal flow so that you will get used to seeing it.

EXAMPLE 7-1

The counter in Figure 7-1 starts off in the 0000 state, and then clock pulses are applied. Some time later the clock pulses are removed, and the counter FFs read 0011. How many clock pulses have occurred?

Solution

The apparent answer seems to be 3 because 0011 is the binary equivalent of 3. With the information given, however there is no way to tell whether or not the counter has recycled. This means that there could have been 19 clock pulses; the first 16 pulses bring the counter back to 0000, and the last 3 bring it to 0011. There could have been 35 pulses (two complete cycles and then three more), or 51 pulses, and so on.

MOD Number

The counter in Figure 7-1 has 16 distinctly different states (0000 through 1111). Thus, it is a *MOD-16 ripple counter*. Recall that the **MOD number** is generally equal to the number of states that the counter goes through in

each complete cycle before it recycles back to its starting state. The MOD number can be increased simply by adding more FFs to the counter. That is,

$$MOD number = 2^N$$
 (7-1)

where N is the number of FFs connected in the arrangement of Figure 7-1.

EXAMPLE 7-2

A counter is needed that will count the number of items passing on a conveyor belt. A photocell and light source combination is used to generate a single pulse each time an item crosses its path. The counter must be able to count as many as one thousand items. How many FFs are required?

Solution

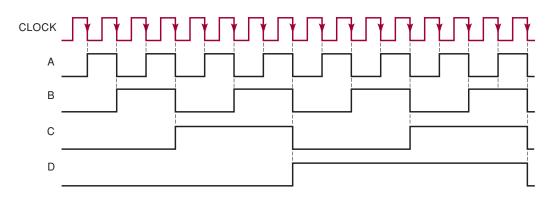
It is a simple matter to determine what value of N is needed so that $2^N \ge 1000$. Since $2^9 = 512$, 9 FFs will not be enough. $2^{10} = 1024$, so 10 FFs would produce a counter that could count as high as $111111111_2 = 1023_{10}$. Therefore, we should use 10 FFs. We could use more than 10, but it would be a waste of FFs because any FF past the tenth one will not be needed.

Frequency Division

In Chapter 5, we saw that in the basic counter each FF provides an output waveform that is exactly *half* the frequency of the waveform at its *CLK* input. To illustrate, suppose that the clock signal in Figure 7-1 is 16 kHz. Figure 7-2 shows the FF output waveforms. The waveform at output *A* is an 8-kHz *square wave*, at output *B* it is 4 kHz, at output *C* it is 2 kHz, and at output *D* it is 1 kHz. Notice that the output of flip-flop *D* has a frequency equal to the original clock frequency divided by 16. In general,

In any counter, the signal at the output of the last FF (i.e., the MSB) will have a frequency equal to the input clock frequency divided by the MOD number of the counter.

For example, in a MOD-16 counter, the output from the last FF will have a frequency of 1/16 of the input clock frequency. Thus, it can also be called a *divide-by-16 counter*. Likewise, a MOD-8 counter has an output frequency of $\frac{1}{8}$ the input frequency; it is a *divide-by-8 counter*.

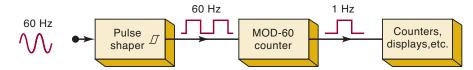




EXAMPLE 7-3

The first step involved in building a digital clock is to take the 60-Hz signal and feed it into a Schmitt-trigger, pulse-shaping circuit* to produce a square wave, as illustrated in Figure 7-3. The 60-Hz square wave is then put into a MOD-60 counter, which is used to divide the 60-Hz frequency by exactly 60 to produce a 1-Hz waveform. This 1-Hz waveform is fed to a series of counters, which then count seconds, minutes, hours, and so on. How many FFs are required for the MOD-60 counter?

FIGURE 7-3 Example 7-3.



Solution

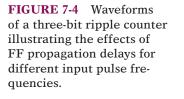
There is no integer power of 2 that will equal 60. The closest is $2^6 = 64$. Thus, a counter using six FFs would act as a MOD-64 counter. Obviously, this will not satisfy the requirement. It seems that there is no solution using a counter of the type shown in Figure 7-1. This is partly true; in Section 7-4, we will see how to modify basic binary counters so that almost *any* MOD number can be obtained and we will not be limited to values of 2^N .

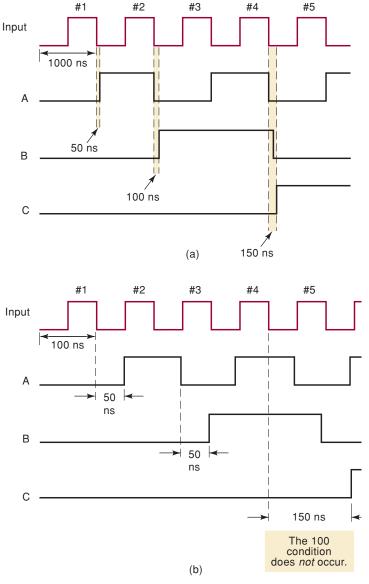
REVIEW QUESTIONS1. *True or false:* In an asynchronous counter, all FFs change states at the same time. 2. Assume that the counter in Figure 7-1 is holding the count 0101. What will be the count after 27 clock pulses? 3. What would be the MOD number of the counter if three more FFs were added?

7-2 PROPAGATION DELAY IN RIPPLE COUNTERS

Ripple counters are the simplest type of binary counters because they require the fewest components to produce a given counting operation. They do, however, have one major drawback, which is caused by their basic principle of operation: each FF is triggered by the transition at the output of the preceding FF. Because of the inherent propagation delay time (t_{pd}) of each FF, this means that the second FF will not respond until a time t_{pd} after the first FF receives an active clock transition; the third FF will not respond until a time equal to $2 \times t_{pd}$ after that clock transition; and so on. In other words, the propagation delays of the FFs accumulate so that the *N*th FF cannot change states until a time equal to $N \times t_{pd}$ after the clock transition occurs. This is illustrated in Figure 7-4, where the waveforms for a three-bit ripple counter are shown.

The first set of waveforms in Figure 7-4(a) shows a situation where an input pulse occurs every 1000 ns (the clock period T = 1000 ns) and it is assumed that each FF has a propagation delay of 50 ns ($t_{pd} = 50$ ns). Notice





that the *A* flip-flop output toggles 50 ns after the NGT of each input pulse. Similarly, *B* toggles 50 ns after *A* goes from 1 to 0, and *C* toggles 50 ns after *B* goes from 1 to 0. As a result, when the fourth input NGT occurs, the *C* output goes HIGH after a delay of 150 ns. In this situation, the counter does operate properly in the sense that the FFs do eventually get to their correct states, representing the binary count. However, the situation worsens if the input pulses are applied at a much higher frequency.

The waveforms in Figure 7-4(b) show what happens if the input pulses occur once every 100 ns. Again, each FF output responds 50 ns after the 1-to-0 transition at its *CLK* input (note the change in the relative time scale). Of particular interest is the situation after the falling edge of the *fourth* input pulse, where the *C* output does not go HIGH until 150 ns later, which is the same time that the *A* output goes HIGH in response to the *fifth* input pulse. In other words, the condition C = 1, B = A = 0 (count of 100) never appears because the input frequency is too high. This could cause a serious problem if this condition were supposed to be used to control some other operation in a digital system. Problems such as this can be avoided if the period between

input pulses is made longer than the total propagation delay of the counter. That is, for proper counter operation we need

$$T_{\rm clock} \ge N \times t_{\rm pd}$$
 (7-2)

where N = the number of FFs. Stated in terms of input-clock frequency, the maximum frequency that can be used is given by

$$f_{\max} = \frac{1}{N \times t_{\rm pd}} \tag{7-3}$$

For example, suppose that a four-bit ripple counter is constructed using the 74LS112 J-K flip-flop. Table 5-2 shows that the 74LS112 has $t_{PLH} = 16$ ns and $t_{PHL} = 24$ ns as the propagation delays from *CLK* to *Q*. To calculate f_{max} , we will assume the "worst case"; that is, we will use $t_{pd} = t_{PHL} = 24$ ns, so that

$$f_{\rm max} = \frac{1}{4 \times 24 \, \rm ns} = 10.4 \, \rm MHz$$

Clearly, as the number of FFs in the counter increases, the total propagation delay increases and f_{max} decreases. For example, a ripple counter that uses six 74LS112 FFs will have

$$f_{\rm max} = \frac{1}{6 \times 24 \, \rm ns} = 6.9 \, \rm MHz$$

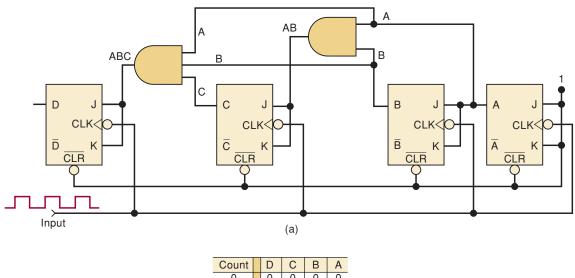
Thus, asynchronous counters are not useful at very high frequencies, especially for counters with large numbers of bits. Another problem caused by propagation delays in asynchronous counters occurs when we try to electronically detect (*decode*) the counter's output states. If you look closely at Figure 7-4(a), for a short period of time (50 ns in our example) right after state 011, you see that state 010 occurs before 100. This is obviously not the correct binary counting sequence, and while the human eye is much too slow to see this temporary state, our digital circuits will be fast enough to detect it. These erroneous count patterns can generate what are called **glitches** in the signals that are produced by digital systems using asynchronous counters. In spite of their simplicity, these problems limit the usefulness of asynchronous counters in digital applications.

REVIEW QUESTIONS

- 1. Explain why a ripple counter's maximum frequency limitation decreases as more FFs are added to the counter.
- 2. A certain J-K flip-flop has $t_{pd} = 12$ ns. What is the largest MOD counter that can be constructed from these FFs and still operate up to 10 MHz?

7-3 SYNCHRONOUS (PARALLEL) COUNTERS

The problems encountered with ripple counters are caused by the accumulated FF propagation delays; stated another way, the FFs do not all change states simultaneously in synchronism with the input pulses. These limitations can be overcome with the use of **synchronous** or **parallel counters** in which all of the FFs are triggered simultaneously (in parallel) by the clock input pulses.



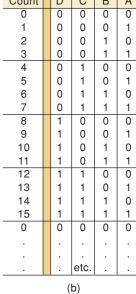




FIGURE 7-5 Synchronous MOD-16 counter. Each FF is clocked by the NGT of the clock input signal so that all FF transitions occur at the same time.

Because the input pulses are applied to all the FFs, some means must be used to control when an FF is to toggle and when it is to remain unaffected by a clock pulse. This is accomplished by using the *J* and *K* inputs and is illustrated in Figure 7-5 for a four-bit, MOD-16 synchronous counter.

If we compare the circuit arrangement for this synchronous counter with its asynchronous counterpart in Figure 7-1, we can see the following notable differences:

- The *CLK* inputs of all of the FFs are connected together so that the input clock signal is applied to each FF simultaneously.
- Only flip-flop A, the LSB, has its J and K inputs permanently at the HIGH level. The J, K inputs of the other FFs are driven by some combination of FF outputs.
- The synchronous counter requires more circuitry than does the asynchronous counter.

Circuit Operation

For this circuit to count properly, on a given NGT of the clock, only those FFs that are supposed to toggle on that NGT should have J = K = 1 when that NGT occurs. Let's look at the counting sequence in Figure 7-5(b) to see what this means for each FF.

The counting sequence shows that the A flip-flop must change states at each NGT. For this reason, its J and K inputs are permanently HIGH so that it will toggle on each NGT of the clock input.

The counting sequence shows that flip-flop *B* must change states on each NGT that occurs while A = 1. For example, when the count is 0001, the next NGT must toggle *B* to the 1 state; when the count is 0011, the next NGT must toggle *B* to the 0 state; and so on. This operation is accomplished by connecting output *A* to the *J* and *K* inputs of flip-flop *B* so that J = K = 1 only when A = 1.

The counting sequence shows that flip-flop *C* must change states on each NGT that occurs while A = B = 1. For example, when the count is 0011, the next NGT must toggle *C* to the 1 state; when the count is 0111, the next NGT must toggle *C* to the 0 state; and so on. By connecting the logic signal *AB* to FF *C*'s *J* and *K* inputs, this FF will toggle only when A = B = 1.

In a like manner, we can see that flip-flop D must toggle on each NGT that occurs while A = B = C = 1. When the count is 0111, the next NGT must toggle D to the 1 state; when the count is 1111, the next NGT must toggle D to the 0 state. By connecting the logic signal *ABC* to FF *D*'s *J* and *K* inputs, this FF will toggle only when A = B = C = 1.

The basic principle for constructing a synchronous counter can therefore be stated as follows:

Each FF should have its *J* and *K* inputs connected so that they are HIGH only when the outputs of all lower-order FFs are in the HIGH state.

Advantage of Synchronous Counters over Asynchronous

In a parallel counter, all of the FFs will change states simultaneously; that is, they are all synchronized to the NGTs of the input clock pulses. Thus, unlike the asynchronous counters, the propagation delays of the FFs do not add together to produce the overall delay. Instead, the total response time of a synchronous counter like the one in Figure 7-5 is the time it takes *one* FF to toggle plus the time for the new logic levels to propagate through a *single* AND gate to reach the *J*, *K* inputs. That is, for a synchronous counter,

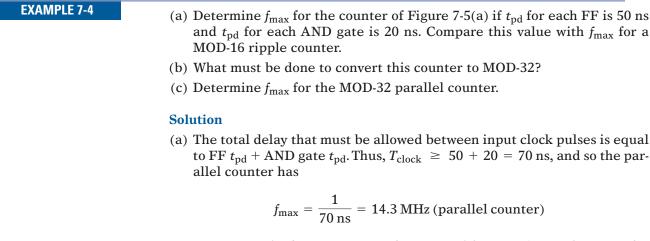
total delay = FF
$$t_{pd}$$
 + AND gate t_{pd}

This total delay is the same no matter how many FFs are in the counter, and it will generally be much lower than with an asynchronous counter with the same number of FFs. Thus, a synchronous counter can operate at a much higher input frequency. Of course, the circuitry of the synchronous counter is more complex than that of the asynchronous counter.

Actual ICs

There are many synchronous IC counters in both the TTL and the CMOS logic families. Some of the most commonly used devices are:

- **74ALS160/162**, 74HC160/162: synchronous decade counters
- 74ALS161/163, 74HC161/163: synchronous MOD-16 counters



A MOD-16 ripple counter uses four FFs with $t_{pd} = 50$ ns. Thus, f_{max} for the ripple counter is

$$f_{\rm max} = \frac{1}{4 \times 50 \, \rm ns} = 5 \, \rm MHz \, (ripple \, \rm counter)$$

- (b) A fifth FF must be added because $2^5 = 32$. The *CLK* input of this FF is also tied to the input pulses. Its *J* and *K* inputs are fed by the output of a four-input AND gate whose inputs are *A*, *B*, *C*, and *D*.
- (c) f_{max} is still determined as in (a) regardless of the number of FFs in the parallel counter. Thus, f_{max} is still 14.3 MHz.

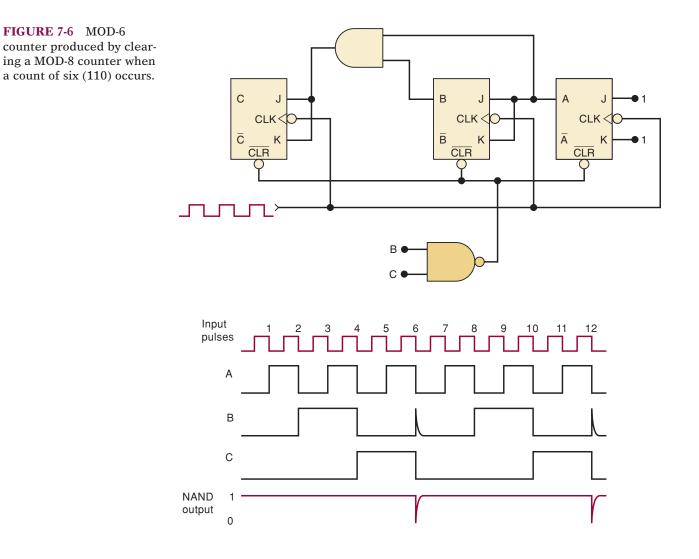
REVIEW QUESTIONS

- 1. What is the advantage of a synchronous counter over an asynchronous counter? What is the disadvantage?
- 2. How many logic devices are required for a MOD-64 parallel counter?
- 3. What logic signal drives the *J*, *K* inputs of the MSB flip-flop for the counter of question 2?

7-4 COUNTERS WITH MOD NUMBERS $< 2^{N}$

The basic synchronous counter of Figure 7-5 is limited to MOD numbers that are equal to 2^N , where *N* is the number of FFs. This value is actually the maximum MOD number that can be obtained using *N* flip-flops. The basic counter can be modified to produce MOD numbers less than 2^N by allowing the counter to *skip states* that are normally part of the counting sequence. One of the most common methods for doing this is illustrated in Figure 7-6, where a three-bit counter is shown. Disregarding the NAND gate for a moment, we can see that the counter is a MOD-8 binary counter that will count in sequence from 000 to 111. However, the presence of the NAND gate will alter this sequence as follows:

1. The NAND output is connected to the asynchronous CLEAR inputs of each FF. As long as the NAND output is HIGH, it will have no effect on the counter. When it goes LOW, however, it will clear all of the FFs so that the counter immediately goes to the 000 state.



- 2. The inputs to the NAND gate are the outputs of the *B* and *C* flip-flops, and so the NAND output will go LOW whenever B = C = 1. This condition will occur when the counter goes from the 101 state to the 110 state on the NGT of input pulse 6. The LOW at the NAND output will immediately (generally within a few nanoseconds) clear the counter to the 000 state. Once the FFs have been cleared, the NAND output goes back HIGH because the B = C = 1 condition no longer exists.
- 3. The counting sequence is, therefore,

CBA	
000 ←	
001	
010	
011	
100	
101	
$110 \rightarrow$	(temporary state needed to clear counter)

Although the counter does go to the 110 state, it remains there for only a few nanoseconds before it recycles to 000. Thus, we can essentially say that this counter counts from 000 (zero) to 101 (five) and then recycles to 000. It essentially skips 110 and 111 so that it goes through only six different states; thus, it is a MOD-6 counter.

Notice that the waveform at the *B* output contains a *spike* or *glitch* caused by the momentary occurrence of the 110 state before clearing. This glitch is very narrow and so would not produce any visible indication on indicator LEDs or numerical displays. It could, however, cause a problem if the *B* output were being used to drive other circuitry outside the counter. It should also be noted that the *C* output has a frequency equal to one-sixth of the input frequency; in other words, this MOD-6 counter has divided the input frequency by *six*. The waveform at *C* is *not* a symmetrical square wave (50 percent duty cycle) because it is HIGH for only two clock cycles, whereas it is LOW for four cycles.

State Transition Diagram

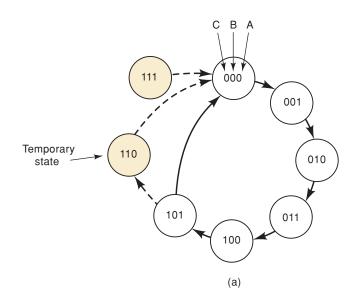
Figure 7-7(a) is the state transition diagram for the MOD-6 counter of Figure 7-6, showing how FFs *C*, *B*, and *A* change states as pulses are applied to the *CLK* input of flip-flop *A*. Recall that each circle represents one of the possible counter states and that the arrows indicate how one state changes to another in response to an input clock pulse.

If we assume a starting count of 000, the diagram shows that the states of the counter change normally up until the count of 101. When the next clock pulse occurs, the counter temporarily goes to the 110 count before going to the stable 000 count. The dotted lines indicate the temporary nature of the 110 state. As stated earlier, the duration of this temporary state is so short that for most purposes we can consider that the counter goes directly from 101 to 000 (solid arrow).

Note that there is no arrow into the 111 state because the counter can never advance to that state. However, the 111 state can occur on power-up when the FFs come up in random states. If that happens, the 111 condition will produce a LOW at the NAND gate output and immediately clear the counter to 000. Thus, the 111 state is also a temporary condition that ends up at 000.

Displaying Counter States

Sometimes during normal operation, and very often during testing, it is necessary to have a visible display of how a counter is changing states in response to the input pulses. We will take a detailed look at several ways of doing this later in the text. For now, Figure 7-7(b) shows one of the simplest methods using individual indicator LEDs for each FF output. Each FF output is connected to an INVERTER whose output provides the current path for the LED. For example, when output A is HIGH, the INVERTER output goes LOW and the LED turns ON. An LED that is turned on indicates A = 1. When output A is LOW, the INVERTER output is HIGH and the LED turns OFF. When the LED is turned off, it indicates A = 0.



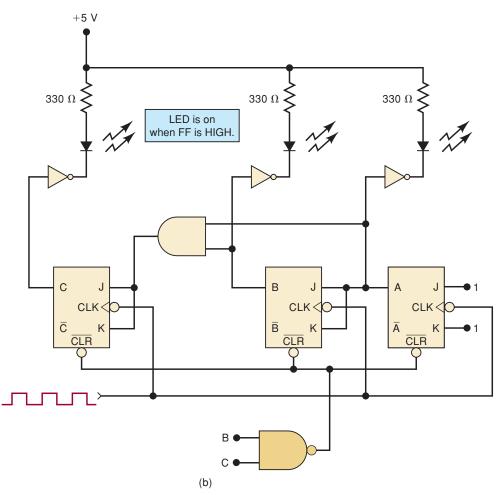


FIGURE 7-7 (a) State transition diagram for the MOD-6 counter of Figure 7-6. (b) LEDs are often used to display the states of a counter.

EXAMPLE 7-5

- (a) What will be the status of the LEDs when the counter is holding the count of five?
- (b) What will the LEDs display as the counter is clocked by a 1-kHz input?
- (c) Will the 110 state be visible on the LEDs?

Solution

- (a) Because $5_{10} = 101_2$, the 2^0 and 2^2 LEDs will be ON, and the 2^1 LED will be OFF.
- (b) At 1 kHz, the LEDs will be switching ON and OFF so rapidly that they will appear to the human eye to be ON all the time at about half the normal brightness.
- (c) No; the 110 state will persist for only a few nanoseconds as the counter recycles to 000.

Changing the MOD Number

The counter of Figures 7-6 and 7-7 is a MOD-6 counter because of the choice of inputs to the NAND gate. Any desired MOD number can be obtained by changing these inputs. For example, using a three-input NAND gate with inputs *A*, *B*, and *C*, the counter would function normally until the 111 condition was reached, at which point it would immediately reset to the 000 state. Ignoring the very temporary excursion into the 111 state, the counter would go from 000 through 110 and then recycle back to 000, resulting in a MOD-7 counter (seven states).

EXAMPLE 7-6

Determine the MOD number of the counter in Figure 7-8(a). Also determine the frequency at the D output.

Solution

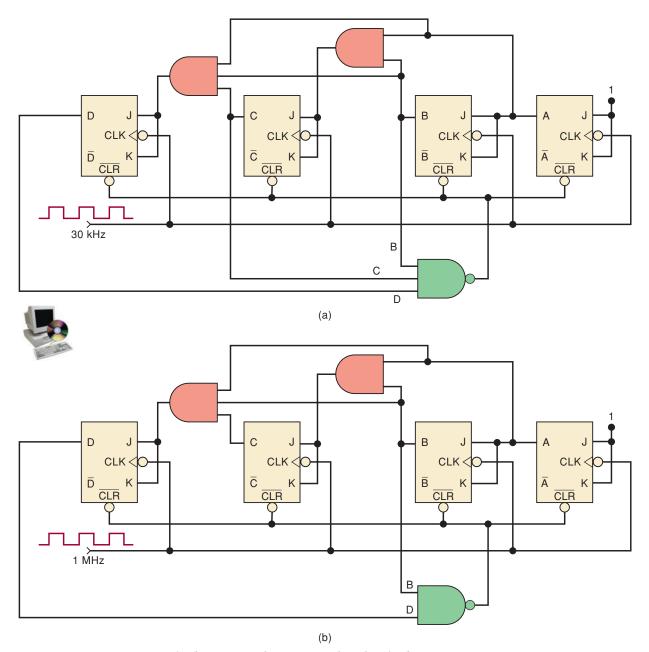
This is a four-bit counter, which would normally count from 0000 through 1111. The NAND inputs are D, C, and B, which means that the counter will immediately recycle to 0000 when the 1110 (decimal 14) count is reached. Thus, the counter actually has 14 stable states 0000 through 1101 and is therefore a *MOD-14* counter. Because the input frequency is 30 kHz, the frequency at output D will be

$$\frac{30 \text{ kHz}}{14} = 2.14 \text{ kHz}$$

General Procedure

To construct a counter that starts counting from all 0s and has a MOD number of *X*:

- 1. Find the smallest number of FFs such that $2^N \ge X$, and connect them as a counter. If $2^N = X$, do not do steps 2 and 3.
- 2. Connect a NAND gate to the asynchronous CLEAR inputs of all the FFs.
- 3. Determine which FFs will be in the HIGH state at a count = X; then connect the normal outputs of these FFs to the NAND gate inputs.





EXAMPLE 7-7

Construct a MOD-10 counter that will count from 0000 (zero) through 1001 (decimal 9).

Solution

 $2^3 = 8$ and $2^4 = 16$; thus, four FFs are required. Because the counter is to have stable operation up to the count of 1001, it must be reset to zero when the count of 1010 is reached. Therefore, FF outputs *D* and *B* must be connected as the NAND gate inputs. Figure 7-8(b) shows the arrangement.

Decade Counters/BCD Counters

The MOD-10 counter of Example 7-7 is also referred to as a **decade counter**. In fact, a decade counter is any counter that has 10 distinct states, no matter what

the sequence. A decade counter such as the one in Figure 7-8(b), which counts in sequence from 0000 (zero) through 1001 (decimal 9), is also commonly called a **BCD counter** because it uses only the 10 BCD code groups 0000, 0001, ..., 1000, and 1001. To reiterate, any MOD-10 counter is a decade counter; and any decade counter that counts in binary from 0000 to 1001 is a BCD counter.

Decade counters, especially the BCD type, find widespread use in applications where pulses or events are to be counted and the results displayed on some type of decimal numerical readout. We shall examine this later in more detail. A decade counter is also often used for dividing a pulse frequency *exactly* by 10. The input pulses are applied to the paralleled clock inputs, and the output pulses are taken from the output of flip-flop *D*, which has onetenth the frequency of the input signal.

EXAMPLE 7-8

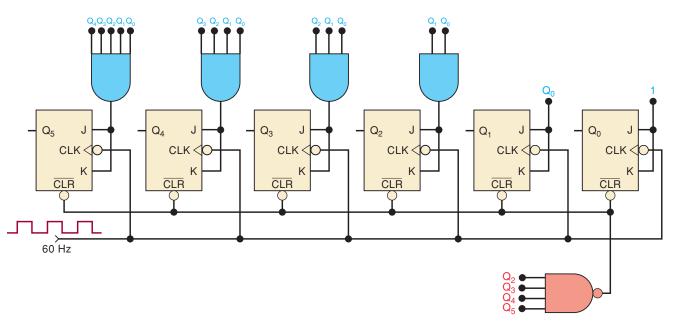
In Example 7-3, a MOD-60 counter was needed to divide the 60-Hz line frequency down to 1 Hz. Construct an appropriate MOD-60 counter.

Solution

 $2^5 = 32$ and $2^6 = 64$, and so we need six FFs, as shown in Figure 7-9. The counter is to be cleared when it reaches the count of 60 (111100). Thus, the outputs of flip-flops Q_5 , Q_4 , Q_3 , and Q_2 must be connected to the NAND gate. The output of flip-flop Q_5 will have a frequency of 1 Hz.

REVIEW QUESTIONS

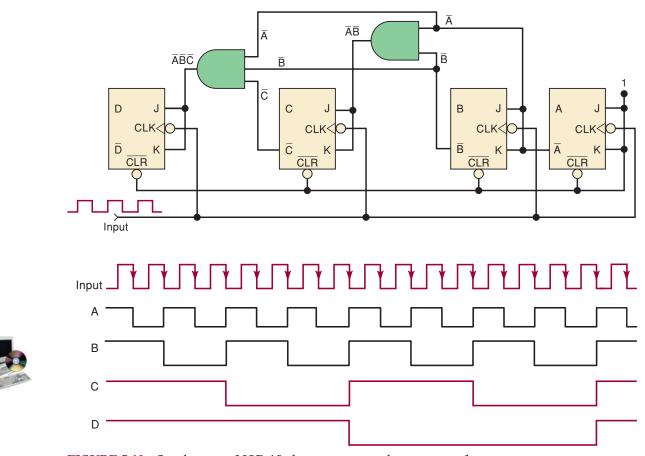
- 1. What FF outputs should be connected to the clearing NAND gate to form a MOD-13 counter?
- 2. True or false: All BCD counters are decade counters.
- 3. What is the output frequency of a decade counter that is clocked from a 50-kHz signal?

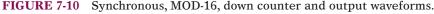


7-5 SYNCHRONOUS DOWN AND UP/DOWN COUNTERS

In Section 7-3, we saw that using the output of lower-order FFs to control the toggling of each FF creates a synchronous **up counter**. A synchronous **down counter** is constructed in a similar manner except that we use the inverted FF outputs to control the higher-order *J*, *K* inputs. Comparing the synchronous, MOD-16, down counter in Figure 7-10 with the up counter in Figure 7-5 shows that we need only to substitute the corresponding inverted FF output in place of the *A*, *B*, and *C* outputs. For a down count sequence, the LSB FF (A) still needs to toggle with each NGT of the clock input signal. Flip-flop B must change states on the next NGT of the clock when A = 0 ($\overline{A} = 1$). Flip-flop C changes states when A = B = 0 ($\overline{A} \cdot \overline{B} = 1$), and flip-flop *D* changes states when A = B = C = 0($\overline{A} \cdot \overline{B} \cdot \overline{C} = 1$). This circuit configuration will produce the count sequence: 15, 14, 13, 12, ..., 3, 2, 1, 0, 15, 14, and so on, as shown in the timing diagram.

Figure 7-11(a) shows how to form a parallel **up/down counter**. The control input Up/Down controls whether the normal FF outputs or the inverted FF outputs are fed to the *J* and *K* inputs of the successive FFs. When Up/Down is held HIGH, AND gates 1 and 2 are enabled while AND gates 3 and 4 are disabled (note the inverter). This allows the *A* and *B* outputs through gates 1 and 2 to the *J* and *K* inputs of FFs *B* and *C*. When Up/Down is held LOW, AND gates 1 and 2 are disabled while AND gates 3 and 4 are enabled. This allows the inverted *A* and *B* outputs through gates 5 and 4 are enabled. This allows the inverted *A* and *B* outputs through gates of FFs *B* and *C*. The waveforms in Figure 7-11(b) illustrate the operation. Notice that for the first five clock pulses, Up/Down = 1 and the counter counts up; for the last five pulses, Up/Down = 0, and the counter counts down.





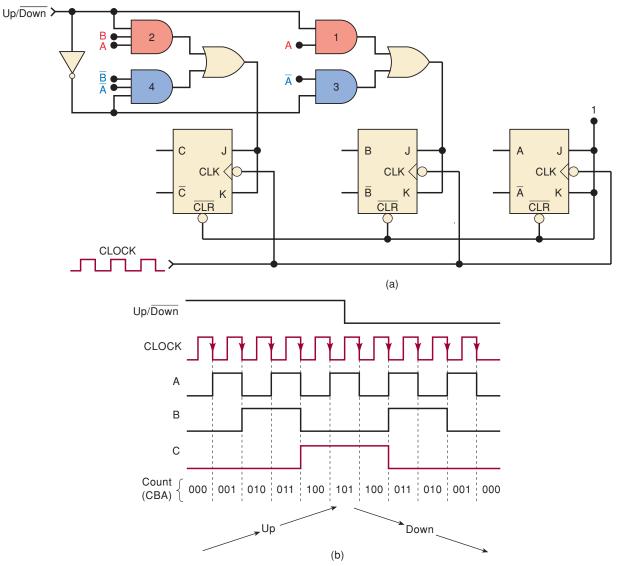




FIGURE 7-11 (a) MOD-8 synchronous up/down counter. (b) The counter counts up when the control input Up/ $\overline{Down} = 1$; it counts down when the control input Up/ $\overline{Down} = 0$.

The nomenclature used for the control signal (Up/Down) was chosen to make it clear how it affects the counter. The count-up operation is active-HIGH; the count-down operation is active-LOW.

EXAMPLE 7-9

What problems might be caused if the Up/Down signal changes levels on the NGT of the clock?

Solution

The FFs might operate unpredictably because some of them would have their J and K inputs changing at about the same time that a NGT occurs at their *CLK* input. However, the effects of the change in the control signal must propagate through two gates before reaching the J, K inputs, so it is more likely that the FFs will respond predictably to the levels that are at J, K prior to the NGT of *CLK*.

REVIEW QUESTIONS	REVIEW QUESTIONS	- 1	τ
------------------	------------------	-----	---

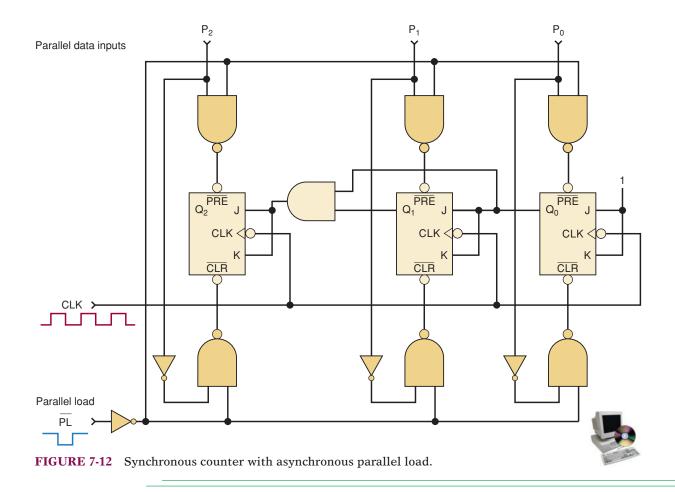
- 1. What is the difference between the counting sequence of an up counter and a down counter?
- 2. What circuit changes will convert a synchronous, binary up counter into a binary down counter?

7-6 PRESETTABLE COUNTERS

Many synchronous (parallel) counters that are available as ICs are designed to be **presettable**; in other words, they can be preset to any desired starting count either asynchronously (independent of the clock signal) or synchronously (on the active transition of the clock signal). This presetting operation is also referred to as **parallel loading** the counter.

Figure 7-12 shows the logic circuit for a three-bit presettable parallel up counter. The *J*, *K*, and *CLK* inputs are wired for operation as a parallel up counter. The asynchronous PRESET and CLEAR inputs are wired to perform asynchronous presetting. The counter is loaded with any desired count at any time by doing the following:

1. Apply the desired count to the parallel data inputs, P_2 , P_1 , and P_0 .



2. Apply a LOW pulse to the PARALLEL LOAD input, \overline{PL} .

This procedure will perform an asynchronous transfer of the P_2 , P_1 , and P_0 levels into flip-flops Q_2 , Q_1 , and Q_0 , respectively (Section 5-17). This *jam transfer* occurs independently of the *J*, *K*, and *CLK* inputs. The effect of the *CLK* input will be disabled as long as \overline{PL} is in its active-LOW state because each FF will have one of its asynchronous inputs activated while $\overline{PL} = 0$. Once \overline{PL} returns HIGH, the FFs can respond to their *CLK* inputs and can resume the counting-up operation starting from the count that was loaded into the counter.

For example, let's say that $P_2 = 1$, $P_1 = 0$, and $P_0 = 1$. While \overline{PL} is HIGH, these parallel data inputs have no effect. If clock pulses are present, the counter will perform the normal count-up operation. Now let's say that \overline{PL} is pulsed LOW when the counter is at the 010 count (i.e., $Q_2 = 0$, $Q_1 = 1$, and $Q_0 = 0$). This LOW at \overline{PL} will produce LOWs at the *CLR* input of Q_1 and at the *PRE* inputs of Q_2 and Q_0 so that the counter will go to the 101 count *regardless of what is occurring at the CLK input*. The count will hold at 101 until \overline{PL} is deactivated (returned HIGH); at that time the counter will resume counting up at each clock pulse from the count of 101.

This asynchronous presetting is used by several IC counters, such as the TTL 74ALS190, 74ALS191, 74ALS192, and 74ALS193 and the CMOS equivalents, 74HC190, 74HC191, 74HC192, and 74HC193.

Synchronous Presetting

Many IC parallel counters use *synchronous presetting* whereby the counter is preset on the active transition of the same clock signal that is used for counting. The logic level on the parallel load control input determines if the counter is preset with the applied input data at the next active clock transition.

Examples of IC counters that use synchronous presetting include the TTL 74ALS160, 74ALS161, 74ALS162, and 74ALS163 and their CMOS equivalents, 74HC160, 74HC161, 74HC162, and 74HC163.

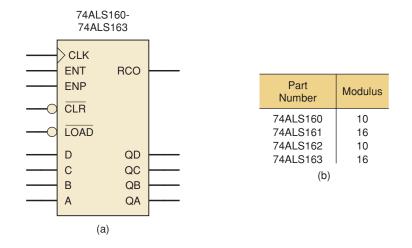
REVIEW QUESTIONS

- 1. What is meant when we say that a counter is presettable?
- 2. Describe the difference between asynchronous and synchronous presetting.

7-7 IC SYNCHRONOUS COUNTERS

The 74ALS160-163/74HC160-163 Series

Figure 7-13 shows the logic symbol, modulus, and function table for the 74ALS160 through 74ALS163 series of IC counters (and the equivalent CMOS counterparts, 74HC160 through 74HC163). These recycling, four-bit counters have outputs labeled QD, QC, QB, QA, where QA is the LSB and QD is the MSB. They are clocked by a PGT applied to CLK. Each of the four different part numbers has a different combination of two feature variations. As seen in Figure 7-13(b), two of the counters are MOD-10 counters (74ALS160 and 74ALS162), while the other two are MOD-16 binary counters (74ALS161 and 74ALS163). The other variation for these parts is in the operation of the clear function [as highlighted in Figure 7-13(c)]. The 74ALS160 and 74ALS161 each has an asynchronous clear input. This means that as soon as CLR goes LOW (CLR is active-LOW for all four parts), the counter's output will be reset to 0000. On the other hand, the 74ALS162 and 74ALS163 IC counters are synchronously cleared. For these counters to be synchronously cleared, the CLR input must be LOW and a PGT must be applied to the clock input. The clear input has priority over all other functions FIGURE 7-13 74ALS160-74ALS163 series synchronous counters: (a) logic symbol; (b) modules; (c) function table.



7441	C160	74 41	C162	Function	Toble
74AL	.5100-	74AL	5103	FUNCTION	rapie

CLR	LOAD	ENP	ENT	CLK	Function	Part Numbers
L	Х	Х	Х	Х	Asynch. Clear	74ALS160 & 74ALS161
L	Х	Х	Х	↑	Synchr. Clear	74ALS162 & 74ALS163
Н	L	Х	х	↑	Synchr. Load	All
Н	Н	Н	н	↑	Count up	All
Н	Н	L	х	Х	No change	All
Н	Н	Х	L	Х	No change	All
					(C)	

for this series of IC counters. Clear will override all other control inputs, as indicated by the Xs in the Figure 7-13(c) function table.

The second priority function available in this series of IC counters is the parallel loading of data into the counter's flip-flops. To preset a data value, make the clear input inactive (HIGH), apply the desired four-bit value to the data input pins D, C, B, A (A is LSB and D is MSB), apply a LOW to the $\overline{\text{LOAD}}$ input control, and then clock the chip with a PGT. The load function is therefore synchronous and has priority over counting, so it does not matter what logic levels are applied to ENT or ENP. To count from the preset state it will be necessary to disable the load (with a HIGH) and enable the count function. If the load function is inactive, it does not matter what is applied to the data input pins.

To enable counting, the lowest-priority function, both CLR and LOAD control inputs must be inactive. Additionally, there are two active-HIGH count enable controls, ENT and ENP. ENT and ENP are essentially ANDed together to control the count function. If either or both of the **count enable** controls is inactive (LOW), the counter will hold the current state. Therefore, to increment the count with each PGT on CLK, all four of the control inputs must be HIGH. When counting, the decade counters (74ALS160 and 74ALS162) will automatically recycle to 0000 after state 1001 (9) and the binary counters (74ALS161 and 74ALS163) will automatically recycle after 1111 (15).

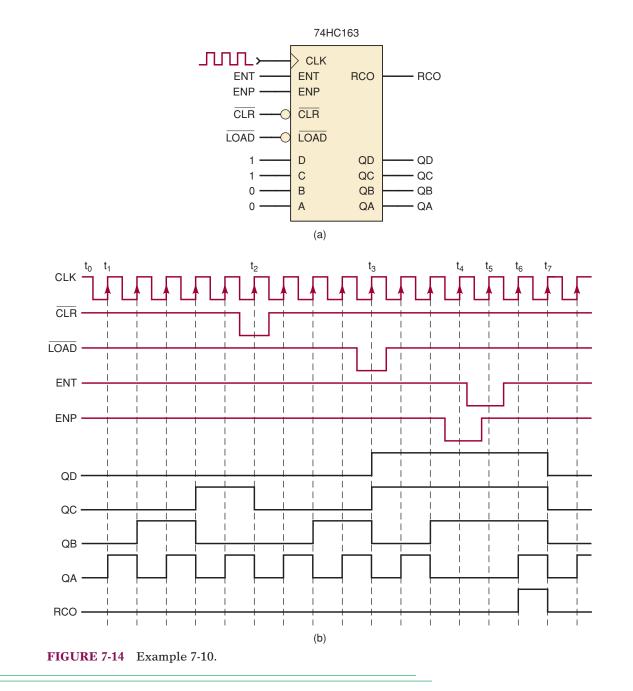
This series of IC counter chips has one more output pin, RCO. The function of this active-HIGH output is to detect (*decode*) the last or terminal state of the counter. The terminal state for a decade counter is 1001 (9), while the terminal state for a MOD-16 counter is 1111 (15). ENT, the primary count enable input, also controls the operation of RCO. ENT must be HIGH for the counter to indicate with the RCO output that it has reached its terminal state. You will see that this feature is very useful in connecting two or more counter chips together in a multistage arrangement to create larger counters.

EXAMPLE 7-10

Refer to Figure 7-14, where a 74HC163 has the input signals given in the timing diagram applied. The parallel data inputs are permanently connected as 1100. Assume the counter is initially in the 0000 state, and determine the counter output waveforms.

Solution

Initially (at t_0), the counter's FFs are all LOW. Since this is not the terminal state for the counter, output RCO will be LOW also. The first PGT on the CLK input occurs at t_1 and, since all control inputs are HIGH, the counter will increment to 0001. The counter continues to count up with each PGT until t_2 . The CLR input is LOW for t_2 . This will synchronously reset the counter to 0000 at t_2 . After t_2 , the CLR input goes inactive (HIGH) so the counter will



start counting up again from 0000 with each subsequent PGT. The $\overline{\text{LOAD}}$ input is LOW for t_3 . This will synchronously load the applied data value 1100 (12) into the counter at t_3 . After t_3 , the $\overline{\text{LOAD}}$ input goes inactive (HIGH), so the counter will continue counting up from 1100 with each subsequent PGT until t_4 . The counter output does not change at t_4 or t_5 , since either ENP or ENT (the count enable inputs) is LOW. This holds the count at 1110 (14). At t_6 , the counter is enabled again and counts up to 1111 (15), its terminal state. As a result, the RCO output now goes HIGH. At t_7 , another PGT on CLK will make the counter recycle to 0000 and RCO returns to a LOW output.

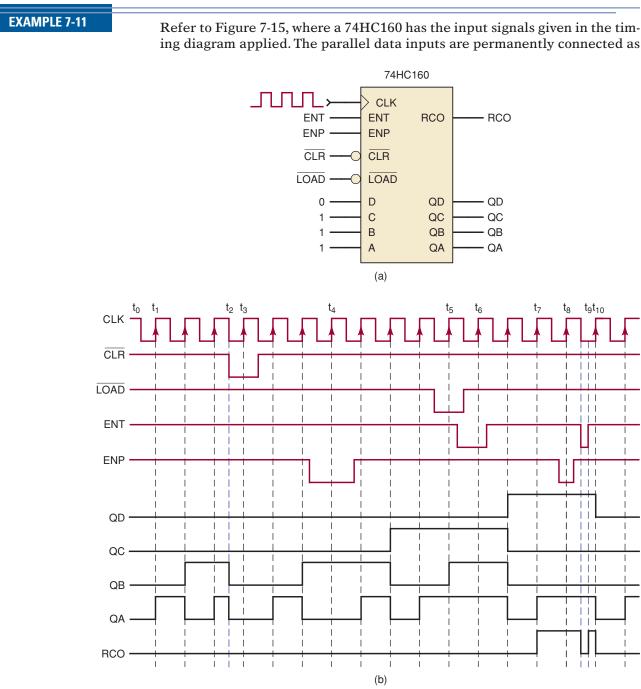


FIGURE 7-15 Example 7-11.

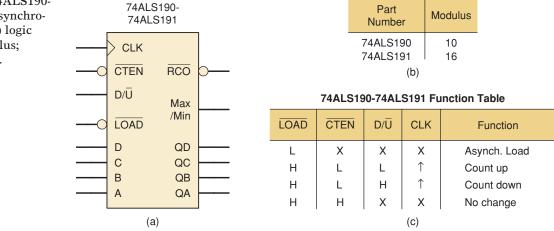
0111. Assume the counter is initially in the 0000 state, and determine the counter output waveforms.

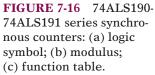
Solution

Initially (at t_0) the counter's FFs are all LOW. Since this is not the terminal state for the BCD counter, output RCO will be LOW also. The first PGT on the CLK input occurs at t_1 and, since all control inputs are HIGH, the counter will increment to 0001. The counter continues to count up with each PGT until t_2 . The asynchronous $\overline{\text{CLR}}$ input goes LOW at t_2 and will immediately reset the counter to 0000 at that point. At t_3 , the CLR input is still active (LOW), so the PGT of the CLK input will be ignored and the counter will stay at 0000. Later the CLR input goes inactive again and the counter will count up to 0001 and then to 0010. At t_4 , the count enable ENP is LOW, so the count holds at 0010. For subsequent PGTs of the CLK input, the counter is enabled and counts up until t_5 . The LOAD input is LOW for t_5 . This will synchronously load the applied data value 0111 (7) into the counter at t_5 . At t_6 , the count enable ENT is LOW, so the count holds at 0111. For the two subsequent PGTs after t_6 , the counter will continue counting up since it is re-enabled. At t_7 , the BCD counter reaches its terminal state 1001 (9) and the RCO output now goes HIGH. At t_8 , ENP is LOW and the counter stops counting (remaining at 1001). At t_9 , while ENT is LOW, the RCO output will be disabled so that it returns to a LOW even though the counter is still at its terminal state (1001). Recall that only ENT controls the RCO output. When ENT returns HIGH during the counter's terminal state, RCO goes HIGH again. At t_{10} the counter is enabled, and it recycles to 0000 and then counts to 0001 on the last PGT.

The 74ALS190-191/74HC190-191 Series

Figure 7-16 shows the logic symbol, modulus, and function table for the 74ALS190 and 74ALS191 series of IC counters (and the equivalent CMOS counterparts, 74HC190 and 74HC191). These recycling, four-bit counters have outputs labeled QD, QC, QB, QA, where QA is the LSB and QD is the MSB. They are clocked by a PGT applied to CLK. The only difference between the two part numbers is the counter's modulus. The 74ALS190 is a MOD-10 counter and the 74ALS191 is a MOD-16 binary counter. Both chips are up/down counters and have an asynchronous, active-LOW load input. This





means that as soon as \overline{LOAD} goes LOW, the counter will be preset to the parallel data on the D, C, B, A (A is LSB and D is MSB) input pins. If the load function is inactive, it does not matter what is applied to the data input pins. The load input has priority over the counting function.

To count, the $\overline{\text{LOAD}}$ control input must be inactive (HIGH) and the count enable control $\overline{\text{CTEN}}$ must be LOW. The count direction is controlled by the D/ $\overline{\text{U}}$ control input. If D/ $\overline{\text{U}}$ is LOW, the count is incremented with each PGT on CLK, while a HIGH on D/ $\overline{\text{U}}$ will decrement the count. Both counters automatically recycle in either count direction. The decade counter recycles to 0000 after state 1001 (9) when counting up or to 1001 after state 0000 when counting down. The binary counter will recycle to 0000 after 1111 (15) when counting up or to 1111 after state 0000 when counting down.

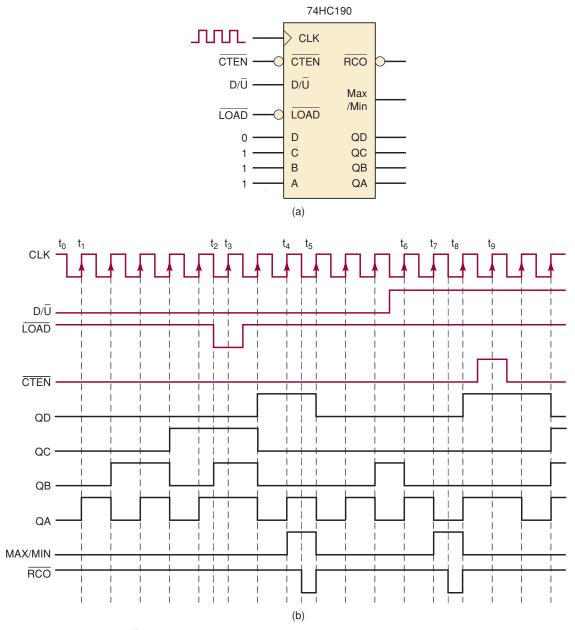
These counter chips have two more output pins, MAX/MIN and RCO. MAX/MIN is an active-HIGH output that detects (decodes) the terminal state of the counter. Since they are up/down counters, the terminal state depends on the direction of the count. The terminal state (MIN) for either counter when counting down is 0000 (0). However when counting up, the terminal state (MAX) for a decade counter is 1001 (9), while the terminal state for a MOD-16 counter is 1111 (15). Note that MAX/MIN detects only one state in the count sequence—it just depends on whether it is counting up or down. The active-LOW RCO output also detects the appropriate terminal state for the counter, but it is a bit more complicated. First, it is only enabled when CTEN is LOW. Additionally, RCO will only be LOW while the CLK input is also LOW. So essentially RCO will mimic the CLK waveform only during the terminal state while the counter is enabled.

EXAMPLE 7-12

Refer to Figure 7-17, where a 74HC190 has the input signals given in the timing diagram applied. The parallel data inputs are permanently connected as 0111. Assume the counter is initially in the 0000 state, and determine the counter output waveforms.

Solution

Initially (at t_0), the counter's FFs are all LOW. Since the counter is enabled (CTEN = 0) and the count direction control D/U = 0, the BCD counter will start counting up on the first PGT applied to CLK at t_1 and continues to count up with each PGT until t_2 , where the count has reached 0101. The asynchronous $\overline{\text{LOAD}}$ input goes LOW at t_2 and will immediately load 0111 into the counter at that point. At t_3 , the LOAD input is still active (LOW), so the PGT of the CLK input will be ignored and the counter will stay at 0111. Later the LOAD input goes HIGH again and the counter will count up to 1000 at the next PGT. At t₄, the counter increments to 1001, which is the terminal state for a BCD up counter and the MAX/MIN output goes HIGH. During t_5 , the counter is at its terminal state and the CLK input is LOW, so RCO goes LOW. For subsequent PGTs of the CLK input, the counter recycles to 0000 and continues to count up until t_6 . Just prior to t_6 , the D/U control changes to a HIGH. This will make the counter count down at t₆ and again at t_7 , where it will be at state 0000, which now is the terminal state since we are counting down, and MAX/MIN will output a HIGH. During t₈, when the CLK input goes LOW, the RCO output again will be LOW. At t_9 , the counter is disabled with CTEN = 1 and the counter holds at 1001. For the subsequent CLK pulses, the counter continues to count down.





EXAMPLE 7-13

Compare the operation of two counters, one with synchronous load and the other with asynchronous load. Refer to Figure 7-18(a), in which a 74ALS163 and a 74ALS191 have been wired in a similar fashion to count up in binary. Both chips are driven by the same clock signal and have their QD and QC outputs NANDed together to control the respective LOAD input control. Assume that both counters are initially in the 0000 state.

- (a) Determine the output waveform for each counter.
- (b) What is the recycling count sequence and modulus for each counter?
- (c) Why do they have different count sequences?

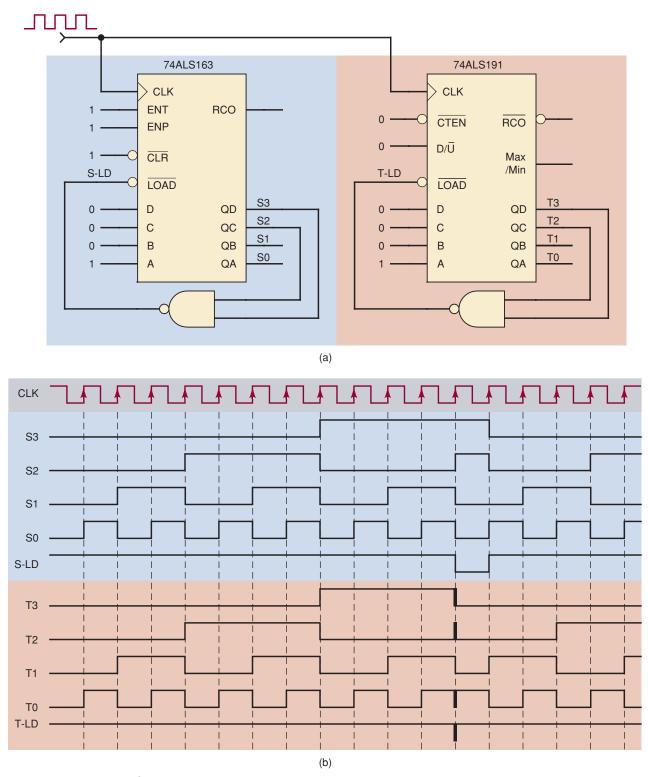


FIGURE 7-18 Example 7-13.

Solution

(a) Starting at state 0000, each counter will count up until it reaches state 1100 (12) as shown in Figure 7-18(b). The output of each NAND gate will apply a LOW to the respective LOAD input at that time. The 74ALS163 has a synchronous LOAD and will wait until the next PGT on CLK to load the data input 0001 into the counter. The 74ALS191 has an asynchronous LOAD and will immediately load the data input 0001 into the counter. This will make the 1100 state a temporary or transient state for the 74ALS191. The transient state will produce some spikes or glitches for some of the counter's outputs because of their rapid switching back and forth.

- (b) The 74ALS163 circuit has a recycling count sequence of 0001 through 1100 and is a MOD-12 counter. The 74ALS191 circuit has a recycling count sequence of 0001 through 1011 and is a MOD-11 counter. Transient states are not included in determining the modulus for a counter.
- (c) The counter circuits have different count sequences because one has a synchronous load and the other has an asynchronous load.

Multistage Arrangement

Many standard IC counters have been designed to make it easy to connect multiple chips together to create circuits with a higher counting range. All of the counter chips presented in this section can be simply connected in a **multistage** or **cascading** arrangement. In Figure 7-19, two 74ALS163s are connected in a two-stage counter arrangement that produces a recycling, binary sequence from 0 to 255 for a maximum modulus of 256. Applying a LOW to the $\overline{\text{CLR}}$ input will synchronously clear both counter stages, and applying a LOW to $\overline{\text{LD}}$ will synchronously preset the eight-bit counter to the binary value on inputs D7, D6, D5, D4, D3, D2, D1, D0 (D0 = LSB). The block on the left (stage 1) is the low-order stage and provides the least-significant counter outputs Q3, Q2, Q1, Q0 (with Q0 = LSB). Stage 2 on the right provides the most-significant counter outputs Q7, Q6, Q5, Q4 (with Q7 = MSB).

EN, the enable for the eight-bit counter, is connected to the ENT input on stage 1. Note that we must use the ENT input and not ENP, since only ENT controls the RCO output. Using ENT and RCO makes cascading very easy. Both counter blocks are clocked together synchronously, but the block on the right (stage 2) is disabled until the least-significant output nibble has reached its terminal state, which will be indicated by the TC1 output. When Q3, Q2, Q1, Q0 reaches 1111 and if EN is HIGH, then TC1 will output a HIGH. This will allow both counter stages to count up one with the next PGT on the clock. Stage 1

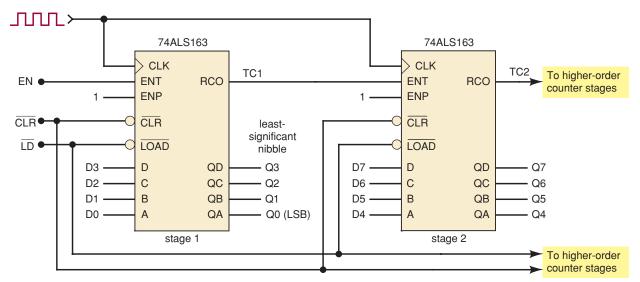


FIGURE 7-19 Two 74ALS163s connected in a two-stage arrangement to extend the maximum counting range.

will recycle back to 0000 and stage 2 will increment from its previous output state. TC1 will return to a LOW, since stage 1 is no longer at its terminal state. With subsequent clock pulses, stage 1 will continue to count up if EN=1 until it again reaches 1111 and the process repeats. When the eight-bit counter reaches 11111111, it will recycle back to 00000000 on the next clock pulse.

Additional 74ALS163 counter chips can be cascaded in the same fashion. TC2 would be connected to the ENT control on the next chip, and so on. TC2 will be HIGH when Q7, Q6, Q5, Q4 is equal to 1111 and TC1 is HIGH, which in turn means that Q3, Q2, Q1, Q0 is also equal to 1111 and EN is HIGH. This cascading technique works for all chips (TTL or CMOS families) in this series, even for the BCD counters. The 74ALS190-191 (or 74HC190-191) series also can be cascaded similarly using the active-LOW CTEN and RCO pins. A multistage counter using 74ALS190-191 chips connected in this fashion can count up or down.

REVIEW QUESTIONS	1. Describe the function of the inputs $\overline{\text{LOAD}}$ and D, C, B, A.
	2. Describe the function of the $\overline{\text{CLR}}$ input.
	3. <i>True or false:</i> The 74HC161 cannot be preset while $\overline{\text{CLR}}$ is active.
	4. What logic levels must be present on the control inputs in order for the 74ALS162 to count pulses that appear on the CLK?
	5. What logic levels must be present on the control inputs in order for the 74HC190 to count down with pulses that appear on the CLK?
	6. What would be the maximum counting range for a four-stage counter made up of 74HC163 ICs? What is the maximum counting range for 74ALS190 ICs?

7-8 DECODING A COUNTER

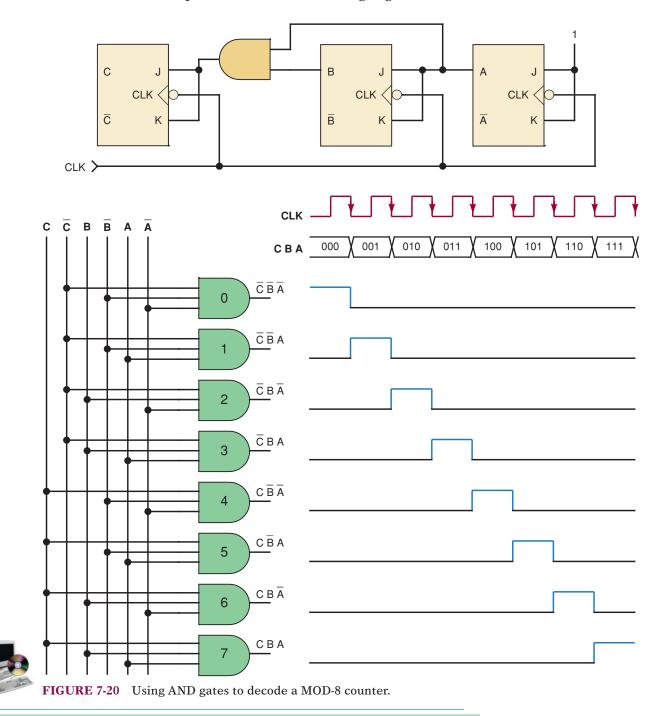
Digital counters are often used in applications where the count represented by the states of the FFs must somehow be determined or displayed. One of the simplest means for displaying the contents of a counter involves just connecting the output of each FF to a small indicator LED [see Figure 7-7(b)]. In this way the states of the FFs are visibly represented by the LEDs (on = 1, off = 0), and the count can be mentally determined by **decoding** the binary states of the LEDs. For instance, suppose that this method is used for a BCD counter and the states of the LEDs are off-on-on-off, respectively. This would represent 0110, which we would mentally decode as decimal 6. Other combinations of LED states would represent the other possible counts.

The indicator LED method becomes inconvenient as the size (number of bits) of the counter increases because it is much harder to decode the displayed results mentally. For this reason, it is preferable to develop a means for *electronically* decoding the contents of a counter and displaying the results in a form that is immediately recognizable and requires no mental operations.

An even more important reason for electronic decoding of a counter occurs because of the many applications in which counters are used to control the timing or sequencing of operations *automatically* without human intervention. For example, a certain system operation might have to be initiated when a counter reaches the 101100 state (count of 44_{10}). A logic circuit can be used to decode for or detect when this particular count is present and then initiate the operation. Many operations may have to be controlled in this manner in a digital system. Clearly, human intervention in this process would be undesirable except in extremely slow systems.

Active-HIGH Decoding

A MOD-*X* counter has *X* different states; each state is a particular pattern of 0s and 1s stored in the counter FFs. A decoding network is a logic circuit that generates *X* different outputs, each of which detects (decodes) the presence of one particular state of the counter. The decoder outputs can be designed to produce either a HIGH or a LOW level when the detection occurs. An active-HIGH decoder produces HIGH outputs to indicate detection. Figure 7-20 shows the complete active-HIGH decoding logic for a MOD-8 counter. The decoder



consists of eight three-input AND gates. Each AND gate produces a HIGH output for one particular state of the counter.

For example, AND gate 0 has at its inputs the FF outputs \overline{C} , \overline{B} , and \overline{A} . Thus, its output will be LOW at all times *except* when A = B = C = 0, that is, on the count of 000 (zero). Similarly, AND gate 5 has as its inputs the FF outputs C, \overline{B} , and A, so that its output will go HIGH only when C = 1, B = 0, and A = 1, that is, on the count of 101 (decimal 5). The rest of the AND gates perform in the same manner for the other possible counts. At any one time, only one AND gate output is HIGH: the one that is decoding for the particular count present in the counter. The waveforms in Figure 7-20 show this clearly.

The eight AND outputs can be used to control eight separate indicator LEDs, which represent the decimal numbers 0 through 7. Only one LED will be on at a given time, indicating the proper count.

The AND gate decoder can be extended to counters with any number of states. The following example illustrates.

EXAMPLE 7-14

How many AND gates are required to decode completely all of the states of a MOD-32 binary counter? What are the inputs to the gate that decodes for the count of 21?

Solution

A MOD-32 counter has 32 possible states. One AND gate is needed to decode for each state; therefore, the decoder requires 32 AND gates. Because $32 = 2^5$, the counter contains five FFs. Thus, each gate will have five inputs, one from each FF. Decoding for the count of 21 (that is, 10101₂) requires AND gate inputs of *E*, \overline{D} , *C*, \overline{B} , and *A*, where *E* is the MSB flip-flop.

Active-LOW Decoding

If NAND gates are used in place of AND gates, the decoder outputs produce a normally HIGH signal, which goes LOW only when the number being decoded occurs. Both types of decoders are used, depending on the type of circuits being driven by the decoder outputs.

EXAMPLE 7-15

Figure 7-21 shows a common situation in which a counter is used to generate a control waveform, which could be used to control devices such as a motor, solenoid valve, or heater. The MOD-16 counter cycles and recycles through its counting sequence. Each time it goes to the count of 8 (1000), the upper NAND gate will produce a LOW output, which sets flip-flop X to the 1 state. Flip-flop X stays HIGH until the counter reaches the count of 14 (1110), at which time the lower NAND gate decodes it and produces a LOW output to clear X to the 0 state. Thus, the X output is HIGH between the counts of 8 and 14 for each cycle of the counter.

BCD Counter Decoding

A BCD counter has 10 states that can be decoded using the techniques described previously. BCD decoders provide 10 outputs corresponding to the decimal digits 0 through 9 and represented by the states of the counter

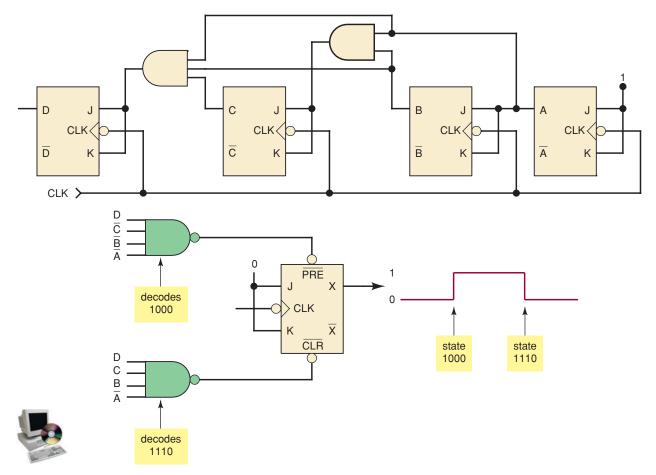
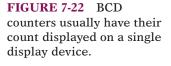
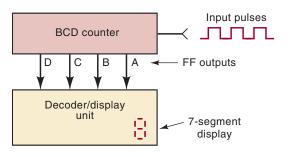


FIGURE 7-21 Example 7-15.

FFs. These 10 outputs can be used to control 10 individual indicator LEDs for a visual display. More often, instead of using 10 separate LEDs, a single display device is used to display the decimal numbers 0 through 9. One class of decimal displays contains seven small segments made of a material (usually LEDs or liquid-crystal displays) that either emits light or reflects ambient light. The BCD decoder outputs control which segments are illuminated in order to produce a pattern representing one of the decimal digits.

We will go into more detail concerning these types of decoders and displays in Chapter 9. However, because BCD counters and their associated decoders and displays are very commonplace, we will use the decoder/display unit (see Figure 7-22) to represent the complete circuitry used to display visually the contents of a BCD counter as a decimal digit.





REVIEW QUESTIONS

- 1. How many gates are needed to decode a six-bit counter fully?
- 2. Describe the decoding gate needed to produce a LOW output when a MOD-64 counter is at the count of 23.

7-9 ANALYZING SYNCHRONOUS COUNTERS

Synchronous counter circuits can be custom-designed to generate any desired count sequence. We can use just the synchronous inputs that are applied to the individual flip-flops to produce the counter's sequence. By not using asynchronous FF controls, such as the clears, to change the counter's sequence, we will never have to deal with transient states and possible glitches in output waveforms. The process of designing completely synchronous counters will be investigated in the next section. First, let's see how to analyze a counter design of this type by predicting the FF control inputs for each state of the counter. A **PRESENT state/NEXT state table** is a very useful tool in this analysis process. The first step is to write the logic expression for each FF control input. Next assume a PRESENT state for the counter and apply that combination of bits to the control logic expressions. The outputs from the control expressions will allow us to predict the commands to each FF and the resulting NEXT state for the counter after clocking. Repeat the analysis process until the entire count sequence is determined.

Figure 7-23 is a synchronous counter that has slightly different J and K inputs than we saw in Section 7-3 for a regular binary up counter. These minor changes to the control circuitry will cause the counter to produce a different count sequence. The control input expressions for this counter are:

$$J_C = A \cdot B$$

$$K_C = C$$

$$J_B = K_B = A$$

$$J_A = K_A = \overline{C}$$

Let us assume that the PRESENT state for the counter is CBA = 000. Applying this combination to the control expressions above will yield $J_C K_C = 0 0, J_B K_B = 0 0$, and $J_A K_A = 1 1$. These control inputs will tell FFs *C* and *B* to hold and FF A to toggle on the next NGT on CLK. Our predicted NEXT state is 001 for CBA. This information has been entered in the first line of the PRE-SENT state/NEXT state table shown in Table 7-1. Next we can use the state 001

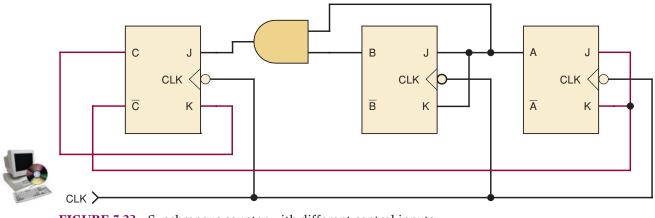


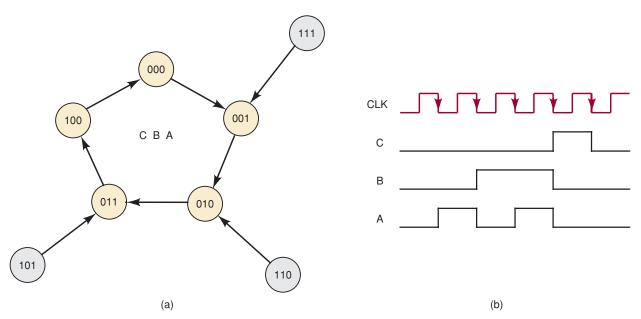
FIGURE 7-23 Synchronous counter with different control inputs.

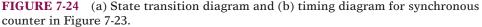
TABLE	7-1
-------	-----

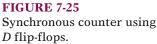
PRES	ESENT State Contr				Control Inputs						
С	В	Α	Jc	K _C	J_B	K _B	J _A K _A	С	В	Α	
0	0	0	0	0	0	0	1 1	0	0	1	
0	0	1	0	0	1	1	1 1	0	1	0	
0	1	0	0	0	0	0	1 1	0	1	1	
0	1	1	1	0	1	1	1 1	1	0	0	
1	0	0	0	1	0	0	0 0	0	0	0	
1	0	1	0	1	1	1	0 0	0	1	1	
1	1	0	0	1	0	0	0 0	0	1	0	
1	1	1	1	1	1	1	0 0	0	0	1	

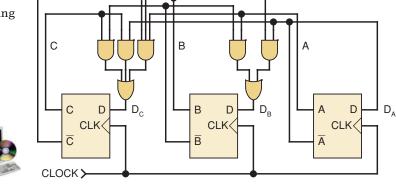
as our PRESENT state. Analyzing the control expressions with this new combination will now yield $J_C K_C = 0$ 0, $J_B K_B = 1$ 1, and $J_A K_A = 1$ 1 giving us a hold command for FF *C* and toggle commands for FFs B and A. This will produce a NEXT state of 010 for CBA, which we have listed on the second line of Table 7-1. Continuing with this process will result in a recycling count sequence of 000, 001, 010, 011, 100, 000. This would be a MOD-5 count sequence. We can also predict the NEXT states for the remaining three possible state combinations in the same way. By doing so, we can determine if the counter design is *selfcorrecting*. A **self-correcting counter** is one in which normally unused states will all somehow return to the normal count sequence. If any of these unused states cannot return to the normal sequence, the counter is said to be not selfcorrecting. Our NEXT-state predictions for all possible states have been entered into Table 7-1. The highlighted information indicates that this counter design happens to be self-correcting. The complete state transition diagram and timing diagram for this counter is shown in Figure 7-24.

We can likewise analyze the operation of counter circuits that use *D* flipflops to store the present state of the counter. The control circuitry for a *D*-type will typically be more complex than for an equivalent JK-type counter that produces the same count sequence, but we will also have half the number of









synchronous inputs to control. Most PLDs utilize *D* flip-flops for their memory elements, so the analysis of this type of counter circuit will give us some insight into how counters are actually programmed inside a PLD.

A synchronous counter designed with D flip-flops is shown in Figure 7-25. The first step is to write the logic expressions for the D inputs:

$$D_{C} = C\overline{B} + C\overline{A} + \overline{C}BA$$
$$D_{B} = \overline{B}A + B\overline{A}$$
$$D_{A} = \overline{A}$$

Then we will determine the PRESENT state/NEXT state table for the counter circuit by assuming a state and applying that set of bit values to the input expressions given above. If we pick CBA = 000 for the initial counter state, we will find that $D_C = 0$, $D_B = 0$, and $D_A = 1$. With a PGT on CLOCK, the flip-flops will "load" in the value 001, which becomes the counter's NEXT state. Using 001 as a PRESENT state will produce inputs of $D_C = 0$, $D_B = 1$, and $D_A = 0$ so that 010 will be the NEXT state, and so on. The completed PRESENT state/NEXT state table, shown in Table 7-2, indicates that this circuit is a recycling MOD-8 binary counter. By applying a little Boolean algebra to the input expressions, we can see that there is actually a fairly simple circuit pattern in creating binary counters from D flip-flops:

$$D_{C} = C\overline{B} + C\overline{A} + \overline{C}BA = C(\overline{B} + \overline{A}) + \overline{C}BA$$

= $C\overline{BA} + \overline{C}(BA) = C \oplus (AB)$
 $D_{B} = \overline{B}A + B\overline{A} = B \oplus A$
 $D_{A} = \overline{A}$

PRESENT State			Control Inputs			NEXT State			
С	В	Α	D _C	D _B	D _A	С	В	Α	
0	0	0	0	0	1	0	0	1	
0	0	1	0	1	0	0	1	0	
0	1	0	0	1	1	0	1	1	
0	1	1	1	0	0	1	0	0	
1	0	0	1	0	1	1	0	1	
1	0	1	1	1	0	1	1	0	
1	1	0	1	1	1	1	1	1	
1	1	1	0	0	0	0	0	0	

TABLE 7-2

It is important to note that the gating resources for most PLDs actually consist of sets of AND-OR circuit arrangements and the SOP logic expression more accurately describes the internal circuit implementation. However, we can see that the expressions have been greatly simplified by using the XOR function. This leads us to predict correctly that to create a MOD-16 binary counter with *D* flip-flops, we would need a fourth FF with:

$$D_D = D \oplus (A B C)$$

REVIEW QUESTIONS

- 1. Why is it desirable to avoid having asynchronous controls on counters?
 - 2. What tool is useful in the analysis of synchronous counters?
 - 3. What determines the count sequence for a counter circuit?
 - 4. What counter characteristic is described by saying that it is self-correcting?

7-10 SYNCHRONOUS COUNTER DESIGN*

Many different counter arrangements are available as ICs—asynchronous, synchronous, and combined asynchronous/synchronous. Most of these count in a normal binary or BCD count sequence, although their counting sequences can be somewhat altered using the clearing or loading methods we demonstrated for the 74ALS160-163 and 74ALS190-191 series of ICs. There are situations, however, where a custom counter is required that follows a sequence that is not a regular binary count pattern, for example, 000, 010, 101, 001, 110, 000, . . .

Several methods exist for designing counters that follow arbitrary sequences. We will present the details for one common method that uses J-K flip-flops in a synchronous counter configuration. The same method can be used in designs with D flip-flops. The technique is one of several design procedures that are part of an area of digital circuit design called **sequential circuit design**, which is normally part of an advanced course.

Basic Idea

In synchronous counters, all of the FFs are clocked at the same time. Before each clock pulse, the J and K input of each FF in the counter must be at the correct level to ensure that the FF goes to the correct state. For example, consider the situation where state 101 for counter CBA is to be followed by state 011. When the next clock pulse occurs, the J and K inputs of the FFs must be at the correct levels that will cause flip-flop C to change from 1 to 0, flip-flop B from 0 to 1, and flip-flop A from 1 to 1 (i.e., no change).

The process of designing a synchronous counter thus becomes one of designing the logic circuits that *decode* the various states of the counter to supply the proper logic levels to each J and K input at the correct time. The inputs to these decoder circuits will come from the outputs of one or more of the FFs. To illustrate, for the synchronous counter of Figure 7-5, the AND gate that feeds the J and K inputs of flip-flop C decodes the states of flip-flops A and B. Likewise, the AND gate that feeds the J and K inputs of flip-flop D decodes the states of A, B, and C.

^{*}This topic may be omitted without affecting the continuity of the remainder of the book.

J-K Excitation Table

Before we begin the process of designing the decoder circuits for each J and K input, we must first review the operation of the J-K flip-flop using a different approach, one called an *excitation table* (Table 7-3). The leftmost column of this table lists each possible FF output transition. The second and third columns list the FF's PRESENT state, symbolized as Q_n , and the NEXT state, symbolized as Q_{n+1} , for each transition. The last two columns list the J and K levels required to produce each transition. Let's examine each case.

 $0 \rightarrow 0$ **TRANSITION** The FF PRESENT state is at 0 and is to remain at 0 when a clock pulse is applied. From our understanding of how a J-K flip-flop works, this can happen when either J = K = 0 (no-change condition) or J = 0 and K = 1 (clear condition). Thus, J must be at 0, but K can be at either level. The table indicates this with a "0" under J and an "x" under K. Recall that "x" means the don't-care condition.

 $0 \rightarrow 1$ **TRANSITION** The PRESENT state is 0 and is to change to a 1, which can happen when either J = 1 and K = 0 (set condition) or J = K = 1 (toggle condition). Thus, *J* must be a 1, but *K* can be at either level for this transition to occur.

 $1 \rightarrow 0$ **TRANSITION** The PRESENT state is 1 and is to change to a 0, which can happen when either J = 0 and K = 1 or J = K = 1. Thus, *K* must be a 1, but *J* can be at either level.

 $1 \rightarrow 1$ **TRANSITION** The PRESENT state is a 1 and is to remain a 1, which can happen when either J = K = 0 or J = 1 and K = 0. Thus, K must be a 0 while J can be at either level.

The use of this **J-K excitation table** (Table 7-3) is a principal part of the synchronous counter design procedure.

Transition at	PRESENT State	NEXT State		
FF Output	Qn	<i>Q</i> _{<i>n</i>+1}	J	К
$0 \rightarrow 0$	0	0	0	x
$0 \rightarrow 1$	0	1	1	Х
$1 \rightarrow 0$	1	0	Х	1
$1 \rightarrow 1$	1	1	х	0

TABLE 7-3J-K flip-flopexcitation table.

Design Procedure

We will now go through a complete synchronous counter design procedure. Although we will do it for a specific counting sequence, the same steps can be followed for any desired sequence.

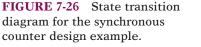
Step 1. Determine the desired number of bits (FFs) and the desired counting sequence.

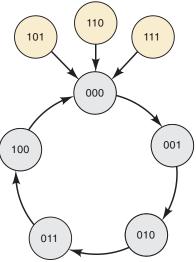
For our example, we will design a three-bit counter that goes through the sequence shown in Table 7-4. Notice that this sequence does not include the 101, 110, and 111 states. We will refer to these states as *undesired states*.

Step 2. Draw the state transition diagram showing *all* possible states, including those that are not part of the desired counting sequence.

IABLE /-4									
С	В	Α							
0	0	0							
0	0	1							
0	1	0							
0	1	1							
1	0	0							
0	0	0							
0	0	1							
	etc.								

TADLE 7 /





For our example, the state transition diagram appears as shown in Figure 7-26. The 000 through 100 states are connected in the expected sequence. We have also included a defined NEXT state for each of the undesired states. This was done in case the counter accidentally gets into one of these states upon power-up or due to noise. The circuit designer can choose to have each of these undesired states go to any state upon the application of the next clock pulse. Alternatively, the designer may choose not to define the counter's action for the undesired states at all. In other words, we may not care about the NEXT state for any undesired state. Using the latter "don't care" design approach will generally produce a simpler design but can be a potential problem in the application where this counter is to be used. For our design example, we will choose to have all undesired states go to the 000 state. This will make our design self-correcting but slightly different from the example MOD-5 counter that was analyzed in Section 7-9.

Step 3. Use the state transition diagram to set up a table that lists *all* PRESENT states and their NEXT states.

For our example, the information is shown in Table 7-5. The left-hand portion of the table lists *every* possible state, even those that are not part of the sequence. We label these as the PRESENT states. The right-hand portion lists the NEXT state for each PRESENT state. These are obtained from the state transition

	PRESENT State				NE	NEXT State				
	С	В	Α	-	С	В	Α			
Line 1	0	0	0		0	0	1			
2	0	0	1		0	1	0			
3	0	1	0		0	1	1			
4	0	1	1		1	0	0			
5	1	0	0		0	0	0			
6	1	0	1		0	0	0			
7	1	1	0		0	0	0			
8	1	1	1		0	0	0			

TA	BL	Е	7-5	
----	----	---	-----	--

diagram in Figure 7-26. For instance, line 1 shows that the PRESENT state of 000 has the NEXT state of 001, and line 5 shows that the PRESENT state of 100 has the NEXT state of 000. Lines 6, 7, and 8 show that the undesired PRESENT states 101, 110, and 111 all have the NEXT state of 000.

Step 4. Add a column to this table for each *J* and *K* input. For each PRESENT state, indicate the levels required at each *J* and *K* input in order to produce the transition to the NEXT state.

Our design example uses three FFs—*C*, *B*, and *A*—and each one has a *J* and a *K* input. Therefore, we must add six new columns as shown in Table 7-6. This completed table is called the **circuit excitation table**. The six new columns are the *J* and *K* inputs of each FF. The entries under each *J* and *K* are obtained from Table 7-3, the J-K flip-flop excitation table that we developed earlier. We will demonstrate this for several of the cases, and you can verify the rest.

Let's look at line 1 in Table 7-6. The PRESENT state of 000 is to go to the NEXT state of 001 on the occurrence of a clock pulse. For this state transition, the *C* flip-flop goes from 0 to 0. From the J-K excitation table, we see that J_C must be at 0 and K_C at "x" for this transition to occur. The *B* flip-flop also goes from 0 to 0, and so $J_B = 0$ and $K_B = x$. The *A* flip-flop goes from 0 to 1. Also from Table 7-3, we see that $J_A = 1$ and $K_A = x$ for this transition.

In line 4 in Table 7-6, the PRESENT state of 011 has a NEXT state of 100. For this state transition, flip-flop *C* goes from 0 to 1, which requires $J_C = 1$ and $K_C = x$. Flip-flops *B* and *A* are both going from 1 to 0. The J-K excitation table indicates that these two FFs need J = x and K = 1 for this to occur.

The required J and K levels for all other lines in Table 7-6 can be determined in the same manner.

Step 5. Design the logic circuits needed to generate the levels required at each *J* and *K* input.

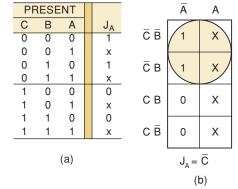
Table 7-6, the circuit excitation table, lists six J, K inputs— J_C , K_C , J_B , K_B , J_A , and K_A . We must consider each of these as an output from its own logic circuit with inputs from flip-flops C, B, and A. Then we must design the circuit for each one. Let's design the circuit for J_A .

To do this, we need to look at the PRESENT states of *C*, *B*, and *A* and the desired levels at J_A for each case. This information has been extracted from Table 7-6 and presented in Figure 7-27(a). This truth table shows the desired

	PRESENT State			e NEXT State								
	С	В	Α	С	В	A	$J_{\rm C}$	K _C	J_{B}	K _B	J_{A}	K _A
Line 1	0	0	0	0	0	1	0	х	0	х	1	х
2	0	0	1	0	1	0	0	х	1	Х	x	1
3	0	1	0	0	1	1	0	x	x	0	1	х
4	0	1	1	1	0	0	1	х	X	1	X	1
5	1	0	0	0	0	0	x	1	0	Х	0	Х
6	1	0	1	0	0	0	x	1	0	х	х	1
7	1	1	0	0	0	0	x	1	x	1	0	х
8	1	1	1	0	0	0	х	1	х	1	х	1

	TAB	LE 7-6
Circuit excit	ation	table.

FIGURE 7-27 (a) Portion of circuit excitation table showing J_A for each PRE-SENT state; (b) K map used to obtain the simplified expression for J_A .



levels at J_A for each PRESENT state. Of course, for some of the cases, J_A is a don't-care. To develop the logic circuit for J_A , we must first determine its expression in terms of *C*, *B*, and *A*. We will do this by transferring the truth-table information to a three-variable Karnaugh map and performing the K-map simplification, as in Figure 7-27(b).

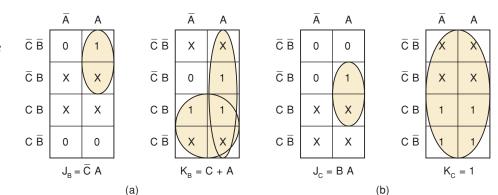
There are only two 1s in this K map, and they can be looped to obtain the term $\overline{A} \ \overline{C}$, but if we use the don't-care conditions at $A \ \overline{B} \ \overline{C}$ and $AB\overline{C}$ as 1s, we can loop a quad to obtain the simpler term \overline{C} . Thus, the final expression is

```
J_A = \overline{C}
```

Now let's consider K_A . We can follow the same steps as we did for J_A . However, a look at the entries under K_A in the circuit excitation table shows only 1s and don't-cares. If we change all the don't-cares to 1s, then K_A is always a 1. Thus, the final expression is

$$K_A = 1$$

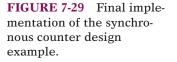
In a similar manner, we can derive the expressions for J_C , K_C , J_B , and K_B . The K maps for these expressions are given in Figure 7-28. You might want to confirm their correctness by checking them against the circuit excitation table.

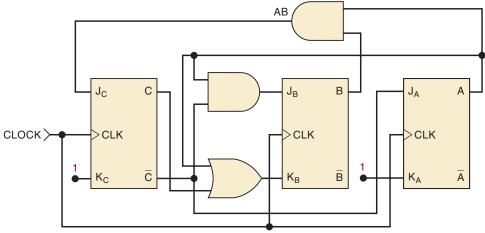


Step 6. Implement the final expressions.

The logic circuits for each J and K input are implemented from the expressions obtained from the K map. The complete synchronous counter design is implemented in Figure 7-29. Note that all FFs are clocked in parallel. You might want to verify that the logic for the J and K inputs agrees with Figures 7-27 and 7-28.

FIGURE 7-28 (a) K maps for the J_B and K_B logic circuits; (b) K maps for the J_C and K_C logic circuits.





Stepper Motor Control

We will now apply this design procedure to a practical situation—driving a *stepper motor*. A stepper motor is a motor that rotates in steps, typically 15° per step, rather than in a continuous motion. Magnetic coils or windings within the motor must be energized and deenergized in a specific sequence in order to produce this stepping action. Digital signals are normally used to control the current in each of the motor's coils. Stepper motors are used extensively in situations where precise position control is needed, such as in positioning of read/write heads on magnetic disks, in controlling print heads in printers, and in robots.

Figure 7-30(a) is a diagram of a typical stepper motor with four coils. For the motor to rotate properly, coils 1 and 2 must always be in opposite states; that is,

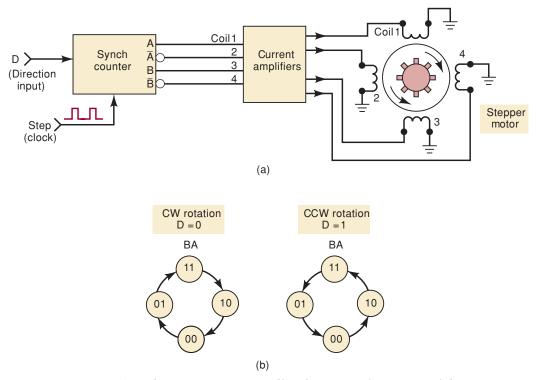


FIGURE 7-30 (a) A synchronous counter supplies the appropriate sequential outputs to drive a stepper motor; (b) state transition diagrams for both states of Direction input, *D*.

when coil 1 is energized, coil 2 is not, and vice versa. Likewise, coil 3 and coil 4 must always be in opposite states. The outputs of a two-bit synchronous counter are used to control the current in the four coils; A and \overline{A} control coils 1 and 2, and B and \overline{B} control coils 3 and 4. The current amplifiers are needed because the FF outputs cannot supply the amount of current that the coils require.

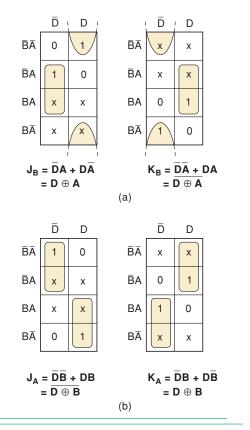
Because this stepper motor can rotate either clockwise (CW) or counterclockwise (CCW), we have a Direction input, D, which is used to control the direction of rotation. The state diagrams in Figure 7-30(b) show the two cases. For CW rotation to occur, we must have D = 0, and the state of the counter, BA, must follow the sequence 11, 10, 00, 01, 11, 10, ..., and so on, as it is clocked by the Step input signal. For CCW rotation, D = 1, and the counter must follow the sequence 11, 01, 00, 10, 11, 01, ..., and so on.

We are now ready to follow the six steps of the synchronous counter design procedure. Steps 1 and 2 have already been done, so we can proceed with steps 3 and 4. Table 7-7 shows each possible PRESENT state of *D*, *B*, and

```
TABLE 7-7
```

PRES	PRESENT State			NEXT State			Control Inputs				
D	В	Α	В	Α		J _B K _B		JA	KA		
0	0	0	0	1		0	x	1	x		
0	0	1	1	1		1	x	x	0		
0	1	0	0	0		x	1	0	x		
0	1	1	1	0		х	0	x	1		
1	0	0	1	0		1	x	0	x		
1	0	1	0	0		0	x	x	1		
1	1	0	1	1		x	0	1	х		
1	1	1	0	1		x	1	x	0		

FIGURE 7-31 (a) K maps for J_B and K_B ; (b) K maps for J_A and K_A .



A and the desired NEXT state, along with the levels at each J and K input needed to achieve the transitions. Note that in all cases, the Direction input, D, does not change in going from the PRESENT to the NEXT state because it is an independent input that is held HIGH or LOW as the counter goes through its sequence.

Step 5 of the design process is presented in Figure 7-31, where the information in Table 7-7 has been transferred to the K maps showing how each *J* and *K* signal is related to the PRESENT states of *D*, *B*, and *A*. Using the appropriate looping, the simplified logic expressions for each *J* and *K* signal are obtained.

The final step is shown in Figure 7-32, where the two-bit synchronous counter is implemented using the J, K expressions obtained from the K maps.

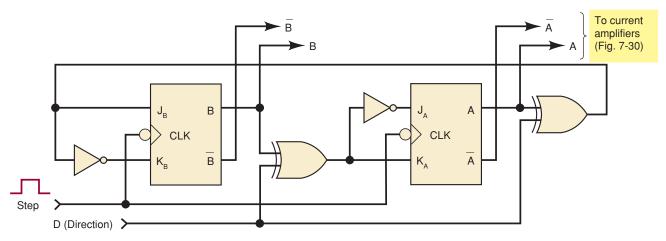


FIGURE 7-32 Synchronous counter implemented from the J, K equations.

Synchronous Counter Design with D FF

We have provided a detailed procedure for designing synchronous counters using J-K flip-flops. Historically, J-K flip-flops have been used to implement counters because the logic circuits needed for the J and K inputs are usually simpler than the logic circuits needed to control an equivalent synchronous counter using D flip-flops. When designing counters that will be implemented in PLDs, where abundant gates are generally available, it makes sense to use D flip-flops instead of J-Ks. Let us now look at synchronous counter design using D FFs.

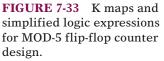
Designing counter circuits using D flip-flops is even easier than using J-K flip-flops. We will demonstrate by designing a D FF circuit that produces the same count sequence as is given in Figure 7-26. The first three steps for synchronous D counter design are identical to the J-K technique. Step 4 for D FF design is trivial since the necessary D inputs are the same as the desired NEXT state as seen in Table 7-8. Step 5 is to generate the logic expressions

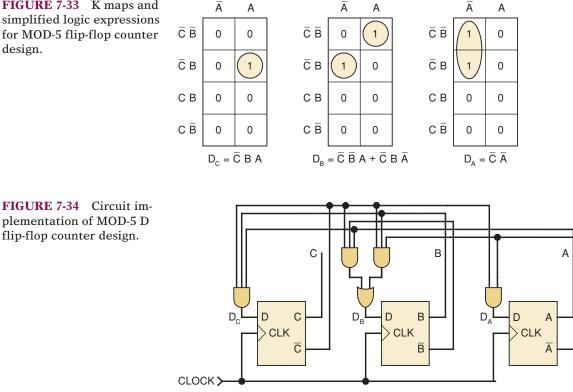
TABLE 7-8

PRESENT State			NE	XT St	ate	Control Inputs			
С	В	Α	С	СВА		D _C	D _B	D _A	
0	0	0	0	0	1	0	0	1	
0	0	1	0	1	0	0	1	0	
0	1	0	0	1	1	0	1	1	
0	1	1	1	0	0	1	0	0	
1	0	0	0	0	0	0	0	0	
1	0	1	0	0	0	0	0	0	
1	1	0	0	0	0	0	0	0	
1	1	1	0	0	0	0	0	0	

from the PRESENT state/NEXT state table for the D inputs. The K maps and simplified expressions are given in Figure 7-33. Finally, for step 6, the counter can be implemented with the circuit shown in Figure 7-34.

A





REVIEW QUESTIONS

- 1. List the six steps in the procedure for designing a synchronous counter.
- 2. What information is contained in a PRESENT state-NEXT state table?
- 3. What information is contained in the circuit excitation table?
- 4. *True or false:* The synchronous counter design procedure can be used for the following sequence: 0010, 0011, 0100, 0111, 1010, 1110, 1111, and repeat.

7-11 BASIC COUNTERS USING HDL

In Chapter 5, we studied flip-flops and the methods used with HDLs to represent flip-flop circuits. The last section in Chapter 5 illustrated how to connect FF components very much like you would wire integrated circuits to one another. By connecting the *Q* output of one FF to the clock input of the next FF, we found that a counter circuit can be created. Using an HDL to describe component connections is referred to as the structural level of abstraction. It is obvious that constructing a complicated circuit using the structural methods would be very tedious and also very difficult to read and interpret. In this section, we will broaden our use of HDL to describe circuits using methods that are considered higher levels of abstraction. This term sounds intimidating, but it only means that there are much more concise and sensible ways to describe what we want a counter to do without worrying about all the details of how to wire flip-flop circuits to do it.

It is still vital that we understand the fundamental principles of flip-flop operation compared with combinational logic gates. As you recall, flip-flops have the following unique characteristics. The output is normally updated according to the condition of the synchronous control inputs when the *active edge* of the clock occurs, which means there is a logic state on the Q output before the clock edge (PRESENT state) and potentially a different state on the Q output after the clock edge (NEXT state). A flip-flop "remembers," or holds its state between clocks, regardless of changes in the synchronous control inputs (e.g., J and K).

Counter circuits using HDL rely on this basic understanding of a circuit going through a sequence of states in response to the event of a clock edge. Ripple counters provide an easy circuit to analyze and understand. They are also much less complicated to build using flip-flops and logic gates than their synchronous counterparts. The problem with ripple counters is the combination of time delay and spurious temporary states that occur when the counter changes state. When we advance to the next level of abstraction and plan to use PLDs to implement our design, we are no longer focusing on wiring issues but rather on describing the circuit's operation concisely. Consequently, the methods we use to describe counter circuits using HDL primarily use synchronous techniques, where all flipflops update simultaneously in response to the same clock event. All the bits in a count sequence go from their PRESENT state to their prescribed NEXT state simultaneously, thereby preventing any intermediate, spurious states.

State Transition Description Methods

The next method of describing circuits that we need to examine uses tables. This method is not concerned with connecting ports of components but rather with assigning values to objects like ports, signals, and variables. In other words, it describes how the output data relates to the input data throughout the circuit. We have already used this method in several of the introductory circuits in Chapters 3 and 4, in the form of truth tables. With sequential counter circuits, the equivalent of the truth table is the PRESENT state/NEXT state table, as we saw in the last section. We can use the HDL essentially to describe the PRESENT state/NEXT state table and thus avoid the tedious details of generating the Boolean equations, as we did in Section 7-10 to design with standard logic devices.

STATE DESCRIPTIONS IN AHDL

As an example of a simple counter circuit, we will implement the MOD-5 counter of Figure 7-26 in AHDL. The inputs and outputs are defined in the SUBDESIGN section of Figure 7-35, as always. In the VARIABLE section on line 7, we have declared (or instantiated) a three-bit array of DFF primitives that are given the instance name *count[]*. This array will be treated basically as a three-bit register in the design and we will essentially define what value should be stored for each NEXT state. Because this is a synchronous counter, we need to tie all the DFF *clk* inputs to the SUBDESIGN's *clock* input. This is accomplished in AHDL by the following statement in the logic section:

count[].clk = clock;

The flip-flop primitives provided in AHDL have standard inputs and outputs that are referred to as "ports." These ports are labeled by a standard port name that is attached to the instance name of the flip-flops. As seen in Table 5-3, the clock port name is *.clk*, a D input is named *.d*, and the FF's output has the name *.q*. To implement the PRESENT state/NEXT state table, a CASE construct is used. For each of the possible values of the register *count[]*, we determine the value that should be placed on the *D* inputs of the flip-flops, which will determine the NEXT state of the counter. The statement on line 21 assigns the value on *count[]* to the output pins. Without this line, the counter would be "buried" in the SUBDESIGN and would not be visible to the outside world.

An alternative design solution is given in Figure 7-36. There are two modifications from Figure 7-35. The first is seen on line 7, where the array name for the D flip-flops is now the same as the output port for the SUBDESIGN.

```
1
     SUBDESIGN fig7_35
 2
     (
 3
           clock
                     : INPUT;
 4
           q[2..0]
                     :OUTPUT;
 5
 6
     VARIABLE
 7
          count[2..0] :DFF;
                               --create a 3-bit register
8
     BEGIN
 9
          count[].clk = clock; --connect all clocks in parallel
10
11
                CASE count[] IS
12
                                     Next
                      Present
13
                      _____
14
                       WHEN 0
                                 => count[].d = 1;
                       WHEN 1
                                 => count[].d = 2;
15
                      WHEN 2
16
                                  =>
                                       count[].d = 3;
17
                      WHEN 3
                                  => count[].d = 4;
18
                                 => count[].d = 0;
                      WHEN 4
19
                      WHEN OTHERS =>
                                     count[].d = 0;
20
                 END CASE;
21
           q[] = count[];
                                 -- assign register to output pins
22
     END;
```

FIGURE 7-35 AHDL MOD-5 counter.

FIGURE 7-36 Another version of the MOD-5 counter described in Figure 7-26.

1

3

4

5

6

7

8

9

10

11

13

14

15

16

17

18

19

20

21

```
SUBDESIGN fig7_36
 2
       (
           clock
                       : INPUT;
           q[2..0]
                      :OUTPUT;
       )
      VARIABLE
          q[2..0]
                       :DFF;
                                -- create a 3-bit register
      BEGIN
          q[].clk = clock;
                               -- connect all clocks in parallel
          TABLE
             q[].q =>
                           q[].d;
12
             0
                           1;
                    =>
             1
                    =>
                           2:
             2
                           3;
                    =>
             3
                    =>
                           4:
             4
                           0;
                    =>
             5
                           0;
                    =>
             6
                    =>
                           0;
             7
                    =>
                           0;
          END TABLE;
       END;
```

This will automatically connect the flip-flop outputs to the SUBDESIGN outputs and eliminate the need to include an assignment statement like line 21 in the first solution. The second modification is the use of an AHDL TABLE instead of the CASE statement used in Figure 7-35. In line 11, the .q port on the *q*[] DFF array represents the PRESENT state side of the table, while the .d port for q[] represents the NEXT state that will be entered into the array's set of D inputs when a PGT is applied to *clock*.

STATE DESCRIPTIONS IN VHDL

As an example of a simple counter circuit, we will implement the MOD-5 counter of Figure 7-26 in VHDL. Our purpose in this example is to demonstrate a counter using a control structure similar to a PRESENT state/NEXT state table. Two key tasks must be accomplished in VHDL: detecting the desired clock edge, and assigning the proper NEXT state to the counter. Recall from our study of flip-flops that a PROCESS can be used to respond to a transition of an input signal. Also, we have learned that a CASE construct can evaluate an expression and, for any valid input value, assign a corresponding value to another signal. The code in Figure 7-37 uses a PROCESS and a CASE construct to implement this counter. The inputs and outputs are defined in the ENTITY declaration, as in the past.

When VHDL is used to describe a counter, we must find a way to "store" the state of the counter between clock pulses (i.e., the action of a flip-flop). This is done in one of two ways: using SIGNALs, or using VARIABLEs. We have used SIGNALs extensively in previous examples that operated concurrently. A SIGNAL in VHDL holds the last value that was assigned to it, very much like a flip-flop. Consequently, we can use a SIGNAL as the data object representing the counter value. This SIGNAL can then be used to

```
ENTITY fig7_37 IS
1
     PORT (
2
3
               clock : IN BIT;
               q :OUT BIT_VECTOR(2 DOWNTO 0)
4
5
           );
     END fig7_37 ;
6
7
     ARCHITECTURE a OF fig7_37
8
                                 TS
9
     BEGIN
        PROCESS (clock)
10
                                                  -- respond to clk input
11
        VARIABLE count: BIT_VECTOR(2 DOWNTO 0);
                                                  -- create a 3-bit register
12
        BEGIN
           IF (clock = '1' AND clock'EVENT) THEN
13
                                                  -- rising edge trigger
14
              CASE count IS
15
                Present
                                    Next
     _ _
16
      _____
                 WHEN "000" =>
17
                                    count := "001";
                 WHEN "001"
                             => count := "010";
18
19
                 WHEN "010"
                             => count := "011";
                 WHEN "011"
                             => count := "100";
20
                 WHEN "100" => count := "000";
21
22
                 WHEN OTHERS => count := "000";
23
              END CASE;
24
           END IF;
25
           q <= count;
                              -- assign register to output pins
26
        END PROCESS;
27
     END a;
```

FIGURE 7-37 VHDL MOD-5 counter.

connect the counter value to any other elements in the architecture description.

In this design, we have chosen to use a VARIABLE instead of a SIGNAL as the data object that stores the counter value. VARIABLEs are not exactly like SIGNALs because they are not used to connect various parts of the design. Instead, they are used as a local place to "store" a value. Variables are considered to be local data objects because they are recognized only within the PROCESS in which they are declared. On line 11 of Figure 7-37, the variable named *count* is declared within the PROCESS before BEGIN. Its type is the same as the output port q. The keyword PROCESS on line 10 is followed by the sensitivity list containing the input signal *clock*. Whenever *clock* changes state, the PROCESS is invoked, and the statements within the PROCESS will be evaluated to produce a result. A 'EVENT (read as "tick-event") attribute will evaluate as TRUE if the signal preceding it has just changed states. Line 13 states that if *clock* has just changed states and right now it is '1', then we know it was a rising edge. To implement the PRESENT state/NEXT state table, a CASE construct is used. For each of the possible values of the variable *count*, we determine the NEXT state of the counter. Notice that the = operator is used to assign a value to a variable. Line 25 assigns the value stored in *count* to the output pins. Because *count* is a local variable, this assignment must be done before END PROCESS on line 26.

Behavioral Description

The **behavioral level of abstraction** is a way to describe a circuit by describing its behavior in terms very similar to the way you might describe its operation in English. Think about the way a counter circuit's operation might be described by someone who knows nothing about flip-flops or logic gates. Perhaps that person's description would sound something like, "When the counter input changes from LOW to HIGH, the number on the output counts up by 1." This level of description deals more with cause-and-effect relationships than with the path of data flow or wiring details. However, we cannot really use just any description in English to describe the circuit's behavior. The proper syntax must be used within the constraints of the HDL.

AHDL

In AHDL, the first important step in this description method is to declare the counter output pins properly. They should be declared as a bit array, with indices decreasing left to right and with 0 as the least significant index in the array, as opposed to individual bits named a, b, c, d, and so on. In this way, the numeric value associated with the bit array's name is interpreted as a binary number upon which certain arithmetic operations can be performed. For example, the bit array *count* shown in Figure 7-38 might contain the bits 1001, as shown. The AHDL compiler interprets this bit pattern as having the value of 9 in decimal.

In order to create our MOD-5 counter in AHDL, we will need a three-bit register that will store the current counter state. This three-bit array, named *count*, is declared using D flip-flops on line 7 in Figure 7-39. Recall from Figure 7-36 that we could name the DFF array the same as the output port q[2..0] and thereby eliminate line 15, but we would also need to change *count[]* to q[] everywhere in the logic section. In other words, the statement on line 7 can be changed to

q[2..0] :DFF;.

If this were done, all references to *count* thereafter would be changed to *q*. This can make the code shorter, but it does not demonstrate universal HDL concepts as clearly. In AHDL, all the clocks can be specified as being tied together and connected to a common clock source using the statement on line 10, *count[].clk* = *clock*. In this example, *count[].clk* refers to the clock input of each flip-flop in the array called *count*.

The behavioral description of this counter is very simple. The current state of the counter is evaluated (count[].q) on line 11, and if it is less than the highest desired count value, it uses the description count[].d = count.q + 1 (line 12). This means that the current state of the *D* inputs must be equal to a value one count greater than the current state of the *Q* outputs. When the current state of the counter has reached the highest desired state (or higher), the *IF* statement test will be false, resulting in a NEXT-state input

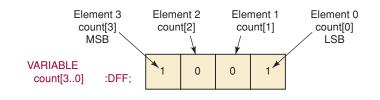


FIGURE 7-38 The elements of a D register storing the number 9.

```
1
      SUBDESIGN fig7_39
2
      (
3
        clock
                 : INPUT;
4
        q[2..0] :OUTPUT; -- declare 3-bit array of output bits
5
6
     VARIABLE
7
        count[2..0] :DFF; -- declare a register of D flip flops.
8
9
     BEGIN
10
        count[].clk = clock; -- connect all clocks to synchronous source
        IF count[].q < 4 THEN -- note; count[] is the same as count[].q
11
12
            count[].d = count[].q + 1; -- increment current value by one
13
        ELSE count[].d = 0; -- recycle to zero: force unused states to 0
14
        END IF;
15
        q[] = count[];
                                 -- transfer register contents to outputs
16
     END;
```

FIGURE 7-39 Behavioral description of a counter in AHDL.

value of zero (line 13), which recycles the counter. The last statement on line 15 simply connects the counter value to the output pins of the device.

VHDL

In VHDL, the first important step in this description method is to declare properly the counter output port, as shown in Figure 7-40. The data type of

```
1
     ENTITY fig7_40 IS
2
     PORT( clock :IN BIT;
3
           q :OUT INTEGER RANGE 0 TO 7 );
 4
     END fig7_40;
5
 6
     ARCHITECTURE a OF fig7_40 IS
 7
     BEGIN
8
        PROCESS (clock)
9
        VARIABLE count: INTEGER RANGE 0 to 7; -- define a numeric VARIABLE
10
            BEGIN
11
              IF (clock = '1' AND clock'EVENT) THEN -- rising edge?
                 IF count < 4 THEN
                                      -- less than max?
12
13
                    count := count + 1;
                                            -- increment value
14
                 ELSE
                                            -- must be at max or bigger
15
                    count := 0;
                                            -- recycle to zero
                 END IF;
16
17
              END IF;
18
        q <= count;
                                -- transfer register contents to outputs
19
        END PROCESS;
20
     END a:
```

FIGURE 7-40 Behavioral description of a counter in VHDL.

the output port (line 3) must match the type of the counter variable (line 9), and it must be a type that allows arithmetic operations. Recall that VHDL treats BIT_VECTORS as just a string of bits, not as a binary numeric quantity. In order to recognize the signal as a numeric quantity, the data object must be typed as an INTEGER. The compiler looks at the RANGE 0 TO 7 clause on line 3 and knows that the counter needs three bits. A similar declaration is needed for the register variable on line 9 that will actually be counting up. This is called *count*. The first statement after BEGIN in the PROCESS responds to the rising edge of the clock as in the previous examples. It then uses behavioral description methods to define the counter's response to the clock edge. If the counter has not reached its maximum (line 12), then it should be incremented (line 13). Otherwise (line 14), it should recycle the counter to zero (line 15). The last statement on line 18 simply connects the counter value to the output pins of the device.

Simulation of Basic Counters

Simulation of any of our MOD-5 counter designs is pretty straightforward. The counters have only one input bit (*clock*) and three output bits (q2 q1 q0) to display in the simulation. The clock frequency has not been specified, so we can use any frequency that we wish for a functional simulation—although we probably should avoid a high-frequency clock unless we want to investigate the effects of propagation delays. About the only decision that we must make is to determine how many clock pulses to apply. Since the counter is a MOD-5 counter, we should apply at least five clock pulses to verify that the HDL design has the correct count sequence and that it recycles. The simulation will start with the initial state 000 because the Altera PLDs have a built-in power-on reset feature. We will not be able to test for any of the unused states because the HDL designs did not provide for a way to preset the counter to any of the unused states. Our simulation results for the HDL design of a MOD-5 counter are shown in Figure 7-41.

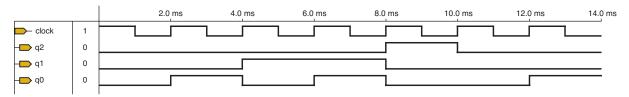


FIGURE 7-41 Simulation results for HDL design of MOD-5 counter.

REVIEW QUESTIONS

- 1. What type of table is used to describe a counter's operation?
- 2. When designing a counter with D flip-flops, what is applied to the D inputs in order to drive it to the NEXT state on the next active clock edge?
- 3. How would you write the HDL description to trigger a storage device (flip-flop) on a falling edge instead of a rising edge of the clock?
- 4. Which method describes the circuit's operation using cause-and-effect relationships?

7-12 FULL-FEATURED COUNTERS IN HDL

The examples we have chosen so far have been very basic counters. All they do is count up to four and then roll over to zero. The standard IC counters that we have examined have many other features that make them very useful for numerous digital applications. For example, consider the 74161 and the 74191 IC counters that were discussed in Section 7-7. These devices have combinations of various features including count enable, up/down counting, parallel loading (preset to any count), and clearing. In addition, these counters have been designed to easily cascade synchronously to create larger counters. In this section, we will explore the techniques that allow us to include these features in an HDL counter. We are going to create a counter that will combine more features than are found in either the 74161 or the 74191. We will use this example to demonstrate the methods of designing a counter with capabilities that specifically suit our needs. When we use HDLs to create digital designs, we are not limited to features that happen to be included with a certain IC.

Let's review the specifications for our more complex counter example. The recycling, MOD-16 binary counter is to change states on the rising edge of the clock input when the counter is enabled with a HIGH level. A direction control input will make the counter count up when it is LOW or count down when it is HIGH. The counter will have an active-HIGH, asynchronous clear to reset the counter immediately when the control input is activated. The counter can be synchronously loaded with a number on the data input pins when the load control is HIGH. The priority of the input control functions, from highest to lowest, will be clearing, loading, and counting. And finally, the counter will also include an active-HIGH output that will detect the terminal state of the counter when the count function is enabled. Remember, the terminal state will be dependent on the count direction. As we will see, the correct operation of these features is determined by the way we write the HDL code, so we will have to pay very close attention to the details.

AHDL FULL-FEATURED COUNTER

The code in Figure 7-42 implements all of the features we have discussed. This is a four-bit counter, but it can easily be expanded in size. Read through the inputs and outputs on lines 3 and 4 to make sure you understand what each one is supposed to do. If you do not, reread the previous paragraphs of this section. Line 7 defines a four-bit register of D flip-flops that will serve as the counter. It should be noted again here that this register could have been named the same as the output variable (q). The code is written with different names to distinguish between ports (inputs and outputs) of the circuit and the devices that are operating within the circuit. The clock input is connected to all the *clk* inputs of all the D flip-flops on line 10. All the active-LOW clear inputs (*clrn*) to the DFF primitive are connected to the complement of the *clear* input signal on line 11. This clears the flip-flops immediately when the *clear* input goes HIGH because the *prn* and *clrn* inputs to the DFF primitive are not dependent on the clock (i.e., they are asynchronous).

In order to make the load function synchronously, the D inputs to the flip-flops must be controlled so that the input data (*din*) is present on the D inputs when the load line is HIGH. This way, when the next active clock edge comes along, the data will be loaded into the counter. This action must happen regardless of whether the counter is enabled or not. Consequently, the first conditional decision (IF) on line 12 evaluates the load input. Recall

```
SUBDESIGN fig7_42
 1
 2
 3
         clock, clear, load, cntenabl, down, din[3..0]
                                                             : INPUT;
 4
         g[3..0], term_ct :OUTPUT; -- declare 4-bit array of output bits
 5
 6
      VARIABLE
 7
         count[3..0]
                        :DFF;
                                     -- declare a register of D flip flops
 8
 9
      BEGIN
10
         count[].clk = clock;
                                     -- connect all clocks to synch source
         count[].clrn= !clear;
                                    -- connect for asynch active HIGH clear
11
12
         IF load THEN count[].d = din[]; -- synchronous load
            ELSIF !cntenabl THEN count[].d = count[].q; -- hold count
13
14
            ELSIF !down THEN count[].d = count[].g + 1; -- increment
            ELSE count[].d = count[].q - 1;
15
                                                         -- decrement
16
         END IF;
17
         IF ((count[],q == 0) \& down # (count[],q == 15) \& !down) \& cntenabl
                 term_ct = VCC;
                                     -- synchronous cascade output signal
         THEN
18
19
         ELSE term_ct = GND;
20
         END IF;
21
         q[] = count[];
                                     -- transfer register contents to outputs
22
      END;
```

FIGURE 7-42 Full-featured counter in AHDL.

from Chapter 4 that the IF/ELSE decision structure gives precedence to the first condition that is found to be true because, once it finds a condition that is true, it does not go on to evaluate the conditions in subsequent ELSE clauses. In this case, it means that if the load line is activated, it does not matter whether the count is enabled, or it is trying to count up or down. It will do a parallel load on the next clock edge.

Assuming that the load line is not active, the ELSIF clause on line 13 is evaluated to see if the count is disabled. In AHDL, it is very important to realize that the Q output must be fed back to the D input so that, on the next clock edge, the register will hold its previous value. Forgetting to insert this clause results in the D inputs defaulting to zero, thus resetting the counter. If the counter is enabled, the ELSIF clause on line 14 is evaluated and either increments *count* (line 14) or decrements *count* (line 15). To summarize these decisions, first decide if it is time to load, next decide if the count should hold or change, then decide whether to count up or down.

The next function described is the detecting (or decoding) of the terminal count. Lines 17–20 decide whether the terminal count has been reached while counting up or down. The double equals (==) operator is the symbol that tests for equality between the expressions on each side of the operator. Which counter state is the terminal state depends on the counting direction. This is determined by ANDing the appropriate terminal state detection of 0 or 15 with the correct expression, *down* or !*down. Term_ct* will output a HIGH if the correct state has been reached, otherwise it will be LOW. Line 21 will connect the output for *count* to the output pins for the SUBDESIGN.

One of the key concepts of using HDLs is that it is generally very easy to expand the size of a logic module. Let us look at the necessary changes to this AHDL design to increase the binary counter modulus to 256. Since $2^8 = 256$,

we will need to increase the number of bits to eight. Only four modifications to Figure 7-42 will be required to make this change in counter modulus:

Line #	Modification
3	din [3 70]
4	q [3 70]
7	count [3 70]
17	(count []. q == 15 255)

VHDL FULL-FEATURED COUNTER

The code in Figure 7-43 implements all the features we have discussed. This is a four-bit counter, but it can easily be expanded in size. Read through the inputs and outputs on lines 2–5 to make sure you understand what each one is supposed to do. If you do not, reread the previous paragraphs of this section. The PROCESS statement on line 10 is the key to all clocked circuits described in VHDL, but it also plays an important role in determining whether the circuit responds synchronously or asynchronously to its inputs. We want

```
1
      ENTITY fig7_43 IS
 2
      PORT( clock, clear, load, cntenabl, down
                                               :IN BIT;
 3
                  :IN INTEGER RANGE 0 TO 15;
            din
 4
                       :OUT INTEGER RANGE 0 TO 15;
            a
 5
                      :OUT BIT);
            term_ct
 6
      END fig7_43;
 7
8
      ARCHITECTURE a OF fig7_43 IS
9
        BEGIN
10
            PROCESS ( clock, clear, down)
11
            VARIABLE count : INTEGER RANGE 0 to 15; -- define a numeric signal
              BEGIN
12
                 IF clear = '1' THEN count := 0; -- asynch clear
13
                  ELSIF (clock = '1' AND clock'EVENT) THEN -- rising edge?
14
                    IF load = '1' THEN count := din; -- parallel load
15
                     ELSIF cntenabl = '1' THEN
16
                                                        -- enabled?
                       IF down = '0' THEN count := count + 1; -- increment
17
                                      count := count - 1; -- decrement
18
                       ELSE
19
                       END IF;
                     END IF;
20
21
                  END IF;
                  IF ((count = 0) AND (down = '1')) OR
22
23
                     ((count = 15) AND (down = '0'))) AND cntenabl = '1'
24
                     THEN term_ct <= '1';
                          term_ct <= '0';
25
                  ELSE
26
                  END IF;
27
                  q <= count;
                               -- transfer register contents to outputs
              END PROCESS;
28
29
          END a;
```

FIGURE 7-43 Full-featured counter in VHDL.

this circuit to respond immediately to transitions on the *clock*, *clear*, and *down* inputs. With these signals in the sensitivity list, we assure that the code inside the PROCESS will be evaluated as soon as any of these inputs change states. The variable *count* is defined on line 11 as an INTEGER so it can be incremented and decremented easily. Variables are declared within the PROCESS and can be used within the PROCESS only.

The *clear* input is given precedence by evaluating it with the first IF statement on line 13. Recall from Chapter 4 that the IF/ELSE decision structure gives precedence to the first condition that is found to be true because it does not go on to evaluate the conditions in subsequent ELSE clauses. In this case, if the *clear* is active, the other conditions will not matter. The output will be zero. In order to make the *load* function operate synchronously, it must be evaluated after detecting the clock edge. The clock edge is detected on line 14, and the circuit checks immediately to see if *load* is active. If *load* is active, the *count* is loaded from *din*, regardless of whether or not the counter is enabled. Consequently, the conditional decision (IF) on line 15 evaluates the *load* input; only if it is inactive does it evaluate line 16 to see if the counter is enabled. If the counter is enabled, the *count* will be incremented or decremented (lines 17 and 18, respectively).

The next issue is detecting the terminal count. Lines 22–25 decide whether the maximum or minimum terminal count has been reached and drive the output to the appropriate level. The decision-making structure here is very important because we want to evaluate this situation, regardless of whether the decision-making process was invoked by *clock*, *clear*, or *down*. Notice that this decision is not another ELSE branch of the previous IF decisions but is evaluated for each signal in the sensitivity list *after* the clearing or counting has occurred. After all these decisions are made, *count* should have the right value in the register, and line 27 effectively connects the register to the output pins.

One of the key concepts of using HDLs is that it is generally very easy to expand the size of a logic module. Let us look at the necessary changes to this VHDL design to increase the binary counter modulus to 256. Only four modifications to Figure 7-43 will be required to make this change in counter modulus:

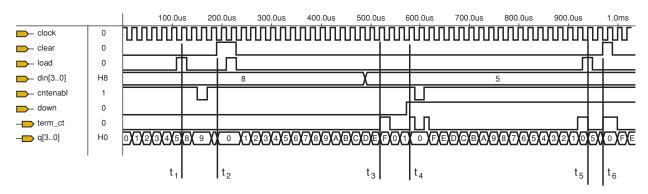
Line #	Modificatio	Modification								
3	RANGE 0	ТО	15 255							
4	RANGE 0	ТО	15 255							
11	RANGE 0	ТО	15 255							
23	(count = 1)	5 255)							

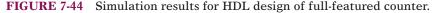
Simulation of Full-Featured Counter

Simulation of our full-featured counter design will require some planning to generate appropriate input waveforms. While it may not be necessary to exhaustively simulate every conceivable input combination, we do need to test enough of the possible input conditions to be convinced that it works properly. This is exactly what we should also do to test our prototype design on the bench. The counter has five different input signals (*clock, clear, load, cntenabl*, and *din*) and two different output signals (*q* and *term_ct*) to display in our simulation. One of the input signals and one of the output signals actually is four bits wide. We will pick a convenient clock frequency since none has been specified for our functional simulation of the counter. We will need to provide enough clock pulses to allow us to look at several operational conditions. The simulation should test the functions of enabling and disabling the counter, counting up and counting down, clearing the counter, loading a value into the counter and counting from that value, and terminal count state detection.

There are some general simulation issues that we should consider in creating our input waveforms. Since the target PLDs have power-on reset, the simulation will start with the initial output state at 0000. Therefore, it would be better to wait until the count has reached another state before applying a clear input so that we can see a change in the output. Likewise, loading in the same value as the counter's NEXT state does not really convince us that *load* is working correctly. Changing input control signals at the same time as the clocking edge occurs may create some setup time problems and produce questionable results. Asynchronous controls should be applied at a time other than the proper clocking edge to show clearly that the resultant circuit action is immediate and not dependent on the clock. In general, we should apply common sense in creating our input waveforms and consider what we are trying to verify with the simulation. Simulation will be valuable in the design process only if we apply appropriate input conditions and evaluate the results critically.

Some simulation results for the full-featured counter are shown in Figure 7-44. The four-bit input *din* and the four-bit output *q* are displayed in hexadecimal. The counter is initially enabled (*cntenabl* = 1) to count up (down =0), and we see the output is incrementing 0, 1, 2, 3, 4, 5. At t_1 , the counter synchronously (i.e., on the PGT of *clock*) responds to the HIGH applied to the *load* input. The counter is preset to the parallel data input (*din*) value of 8. This also shows that loading has priority over counting, since they are both active at the same time. After t_1 , load is LOW again and the counter continues to count up from 8. A LOW input to cntenabl makes the counter hold at state 9 for an extra clock cycle. The count is continued when *cntenabl* goes HIGH again until t_2 , when the counter is asynchronously cleared. Notice the shortened time for the output state A due to the immediate clearing of the counter. We would have to zoom in to actually see that state A is displayed. We can also see that the clear function has the highest priority when all three controls, *clear*, *load*, and *cntenabl*, are simultaneously high. The countup sequence continues and recycles to 0 after state F to verify that the counter is a MOD-16 binary counter. At t_3 , the counter reaches its terminal state F when counting up, and *term_ct* outputs a HIGH. At t_4 , the counter starts counting down because down has been switched to a HIGH. Again, term ct outputs a HIGH since the counter is now at state 0, which is the terminal state when counting down. Notice that, by the action of *term_ct*, the terminal state for the counter depends on its direction of counting, which is





controlled by the input *down*. The count holds at state 0 for an extra clock period when *cntenabl* goes LOW. The output *term_ct* is also disabled while *cntenabl* = 0. The down count sequence continues correctly when *cntenabl* again goes HIGH. At t_5 , the counter synchronously loads the parallel data value 5. At t_6 , the counter is asynchronously cleared. Again the priority of loading or clearing over a down count is verified at t_5 and t_6 . Did we verify that our design operates correctly in comparison to the specifications? We did a pretty good job, but there are a couple of test conditions that could also be added for completeness. Will the counter clear or load when the *cntenabl* is LOW? It appears that we neglected to verify those scenarios. As you can see, complex designs may require a lot of thought to verify their operation adequately by simulation or bench testing. Can you think of any other tests that we should make?

REVIEW QUESTIONS

- 1. What is the difference between asynchronous clear and synchronous load?
- 2. How do you create an asynchronous clear function in an HDL?
- 3. How do you create functions priority in an HDL description of a counter?

7-13 WIRING HDL MODULES TOGETHER

In the previous two sections we have looked at how to implement common counter features using an HDL. We should also investigate how we can connect these counter circuits to other digital modules to create larger systems. Designing large digital systems becomes much easier if the system is subdivided into smaller, more manageable modules that are then interconnected. This is the essence of the concept of **hierarchical design**, and we will readily see its benefits with example projects in Chapter 10. Let us now look at the basic techniques for wiring modules together.

DECODING THE AHDL MOD-5 COUNTER

We looked briefly at the idea of decoding a counter in Section 7-8. You should recall that a decoding circuit detects a counter's state by the unique bit pattern for that state. Let's see how to connect a decoder circuit to the MOD-5 counter design in Figure 7-35 (or Figure 7-36). We will rename the counter SUBDESIGN mod5 to be a bit more descriptive in the block diagram for the overall circuit that we will draw later. Since the counter does not produce all eight possible states for a three-bit counter, our decoder design shown in Figure 7-45 will only decode the states that are used, 000 through 100. The three input bits (c = MSB) declared on line 3 will be connected later to the MOD-5 counter's outputs. The five outputs for the decoder are named state0 through state4 on line 4. A CASE statement (lines 7-14) describes the behavior of the decoder by checking the *c b a* input combination to determine which one of the decoder outputs should be HIGH. When the c b a input is 000, only the *state0* output will be HIGH or, when c b a is 001, only the state1 output will be HIGH, and so on. Any input value greater than 100, which is covered by OTHERS and actually should not occur in this application, will produce LOWs on all outputs.

1 2	SUBDESIGN decode5	
3	c, b, a	: INPUT;
-		,
4	state[04]	: OUTPUT;
5)	
6	BEGIN	
7	CASE (c,b,a) IS	decode binary value
8	WHEN B"000"	<pre>=> state[] = B"10000";</pre>
9	WHEN B"001"	<pre>=> state[] = B"01000";</pre>
10	WHEN B"010"	<pre>=> state[] = B"00100";</pre>
11	WHEN B"011"	<pre>=> state[] = B"00010";</pre>
12	WHEN B"100"	<pre>=> state[] = B"00001";</pre>
13	WHEN OTHERS	<pre>=> state[] = B"00000";</pre>
14	END CASE;	
15	END;	

FIGURE 7-45 AHDL MOD-5 counter decoder module.

We will instruct the Altera software to create symbols for our two design files, mod5 and decode5. This will allow us to draw a block diagram (see Figure 7-46) for our complete circuit that consists of these two modules, input and output ports, and the wiring between them. Each symbol is labeled with its respective SUBDESIGN name mod5 or decode5. Notice that some of the wiring is drawn with heavier-weight lines. This is to represent a bus, which is a collection of signal lines. The lighter-weight lines are individual signals. The symbols created by Altera will automatically have ports drawn to indicate whether they represent individual signals or buses. This will be determined by the signal declarations in the SUBDESIGN section. Ports with group names will be drawn as buses. Since the counter output port is a bus but the decoder input ports are individual signals, it will be necessary to split the bus into individual signal lines to wire the two modules together. Whenever a bus is split, you must label both the group signal name of the bus and the individual signals that are being used. Our block diagram has a bus labeled *q*[2..0] and the corresponding individual signals *q*2, *q*1, and *q*0. The simulation results for this counter and decoder circuit are shown in Figure 7-47.

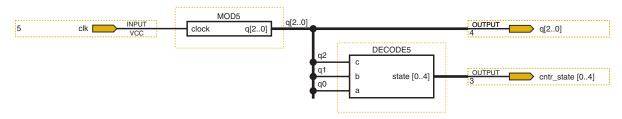


FIGURE 7-46 Block diagram design for the MOD-5 counter and decoder circuit.

		1.0us 2	.0us	3.0us	4.0us	5.0us	6.0us	7.0us	8.0us	9.0us	10.0	Dus	11.0us
D clk	0												
- — q[20]	B 000	000	χ	001		010		011		100		000	C
- cntr_state[04]	B 10000	10000	χ	01000		00100		00010		00001		10000	



DECODING THE VHDL MOD-5 COUNTER

We looked briefly at the idea of decoding a counter in Section 7-8. You should recall that a decoding circuit detects a counter's state by the unique bit pattern for that state. Let's see how to connect a decoder circuit to the MOD-5 counter design in Figure 7-37. We will rename the counter ENTITY mod5 to make it easier to identify the module in our overall circuit. Since the counter does not produce all eight possible states for a three-bit counter, our decoder design shown in Figure 7-48 will only decode the states that are used, 000 through 100. The three input bits (c = MSB) declared on line 3 will be connected later to the MOD-5 counter's outputs. The five outputs for the decoder are named state, a bit vector, on line 4. An internal bit vector signal named *input* is declared on line 9. Then line 11 combines the three input port bits (c b a) together as a bit vector called *input*, which then can be evaluated by the CASE statement on lines 14–21. If any of the input bits changes logic level, the PROCESS will be invoked to determine the resultant output. The CASE statement describes the behavior of the decoder by checking the *input* combination (representing c b a) to determine which one of the decoder outputs should be HIGH. When the *input* is 000, only the *state(0)* output will be HIGH; when *input* is 001, only the *state(1)* output will be HIGH; and so on. Any *input* value greater than 100, which is covered by OTHERS and actually should not occur in this application, will produce LOWs on all outputs.

Since we are using the Altera PLD Development software, we can connect the two modules graphically. To do this, you will need to instruct the software to create symbols for our two design files, mod5 and decode5. This will allow us to draw a block diagram (see Figure 7-46) for our complete circuit that consists of these two modules, input and output ports, and the wiring between them. Notice that some of the wiring is drawn with heavier-weight lines. This is to represent a bus, which is a collection of signal lines. The lighter-weight lines are individual signals. The symbols created by Altera will automatically have ports drawn to indicate whether they represent individual signals or

```
1
      ENTITY
              decode5 IS
 2
      PORT (
 3
                c, b, a : IN BIT;
 4
                state
                          : OUT BIT_VECTOR (0 TO 4)
 5
      ) :
 6
      END decode5;
 7
 8
      ARCHITECTURE a OF decode5 IS
 9
      SIGNAL
                input : BIT_VECTOR (2 DOWNTO 0);
10
      BEGIN
11
           input <= (c & b & a);
                                      -- combine inputs into bit vector
12
           PROCESS (c, b, a)
13
           BEGIN
14
                CASE input IS
15
                      WHEN "000" =>
                                         state <= "10000";
16
                      WHEN "001" =>
                                           state <= "01000";</pre>
                      WHEN "010" =>
                                           state <= "00100";</pre>
17
                      WHEN "011" =>
                                           state <= "00010";</pre>
18
                      WHEN "100" =>
                                          state <= "00001";
19
                                           state <= "00000";
20
                      WHEN OTHERS =>
21
                END CASE;
22
           END PROCESS;
23
      END a;
```

FIGURE 7-48 VHDL MOD-5 counter decoder module.

V H D L

buses. This will be determined by the data type declarations for each port of the ENTITY. BIT_VECTOR ports will be drawn as buses and BIT type ports will be drawn as individual signal lines. Since the counter output port is a bus but the decoder input ports are individual signals, it will be necessary to split the bus into individual signal lines to wire the two modules together. Whenever a bus is split, you must label both the group signal name of the bus and the individual signals that are being used. Our block diagram has a bus labeled q[2..0] and the corresponding individual signals q2, q1, and q0. The simulation results for this counter and decoder circuit are shown in Figure 7-47.

The standard VHDL technique (and an alternative with Altera's software) to connect design modules is to use VHDL to describe the connections between the modules in a text file. The desired modules are instantiated in a higher-level design file using COMPONENTs in which the module's PORTs are declared. The wiring connections for each instance where the module is utilized are listed in a PORT MAP. A VHDL file that connects the mod5 and decode5 modules together is shown in Figure 7-49. Even though q is an output port for our top-level design file, it is typed as a BUFFER on line 4 due to the fact that it is necessary to "read" the bit vector array for an input to the decode5 COMPONENT in its PORT MAP (line 25). VHDL does not permit output ports to be used as inputs. The BUFFER data type declaration provides a port that can be used for both input and output. The mod5 module is declared on lines 10–15 and the decode5 module is declared on lines 16–21. The mod5 and decode5 ENTITY/ARCHITECTURE descriptions may be included within the top-level design file, or instead they may be saved in the same folder as the top-level file as was done here. The PORT MAP for each instance of the modules is listed on lines 23 and 24-25. The word to the left of the colon is a unique label for each instance and the module name is on the right, then the keywords PORT MAP, and finally, in parentheses, are the named associations between the design signals and ports. The => operator indicates which module ports (on the left side) are connected to which

```
1
      ENTITY mod5decoded1 IS
      PORT (
 2
 3
           clk
                            :IN BIT;
                            :BUFFER BIT_VECTOR (2 DOWNTO 0);
 4
           q
 5
           cntr_state
                            :OUT BIT VECTOR (0 TO 4)
 6
           );
 7
      END mod5decoded1;
 8
 9
      ARCHITECTURE toplevel OF mod5decoded1 IS
10
      COMPONENT mod5
11
           PORT (
                 clock
                            :IN BIT;
12
13
                            :OUT BIT_VECTOR (2 DOWNTO 0)
                 q
14
                 );
15
      END COMPONENT;
16
      COMPONENT decode5
17
      PORT (
18
                 c, b, a
                            :IN BIT;
                            :OUT BIT_VECTOR (0 TO 4)
19
                 state
20
                 );
21
      END COMPONENT;
22
      BEGIN
23
      counter: mod5
                            PORT MAP (clock => clk, q => q);
24
      decoder: decode5
                            PORT MAP
25
            (c \Rightarrow q(2), b \Rightarrow q(1), a \Rightarrow q(0), state \Rightarrow cntr_state);
26
      END toplevel;
```

FIGURE 7-49 Higher-level VHDL file to connect mod5 and decode5 together.

higher-level system signals (on the right side). This circuit produces the simulation results shown in Figure 7-47.

MOD-100 BCD Counter

We wish to design a recycling, MOD-100 BCD counter that has a synchronous clear. Creating a MOD-10 BCD counter module and synchronously cascading two of these modules together in a higher-level design file is the easiest way to do this. The clock inputs to the two MOD-10 modules will both be connected to the system clock to achieve synchronous cascading of the two counter modules. Remember, there are significant benefits to using synchronous counter design rather than asynchronous clocking techniques. Also, if we did not employ synchronous clocking, the synchronous clear would not work properly. Even though the design specifications did not require a count enable or terminal count detection for the MOD-100 counter, it will be necessary to include these features in our design. In order to synchronously cascade two counters, the enable and decoding features will be needed. The count enable input causes the counter to ignore clock edges unless it is enabled. The terminal count output indicates that the counting sequence has reached its limit and will roll over on the next clock. To synchronously cascade counter stages together, the terminal count output is connected to the next higher-order stage's enable input. By using the count enable to also control the decoding of the terminal count, our MOD-10 module can be used to create even larger BCD counters.

CASCADING AHDL BCD COUNTERS

Our MOD-10 BCD counter SUBDESIGN is shown in Figure 7-50. The terminal state for a BCD counter is 9. Lines 10–13 will detect this terminal state only when the counter is enabled with a HIGH. ANDing the *enable* control

```
1
      SUBDESIGN mod10
 2
      (
 3
           clock, enable, clear
                                          : INPUT:
           counter[3..0], tc
 4
                                          :OUTPUT;
 5
 6
     VARIABLE
 7
          counter[3..0]
                                          :DFF;
 8
     BEGIN
 9
           counter[].clk = clock;
10
           IF counter[].q == 9 & enable == VCC THEN
                    tc = VCC;
11
                                               -- detect terminal count
12
           ELSE
                     tc = GND;
13
           END IF;
14
           IF clear THEN
               counter[].d = B"0000";
15
                                               -- synchronous clear
16
           ELSIF enable THEN
                                               -- clear has priority
17
                IF counter[].q == 9 THEN
                                               -- check for last state
                     counter[].d = B"0000";
18
19
                ELSE
20
                     counter[].d = counter[].g + 1;
                                                          -- increment
21
                END IF;
2.2
           ELSE
                                           -- hold count when disabled
23
                counter[].d = counter[].q;
24
           END IF;
25
     END;
```

FIGURE 7-50 MOD-10 BCD counter in AHDL.

in the decoding function will allow more than two counter modules to be cascaded synchronously if necessary and makes our mod10 design more versatile. The *clear* function will operate synchronously in AHDL by including it in the IF statement as shown on lines 14–15. If *clear* is inactive, we next check to see if the counter is enabled (line 16). If enable is HIGH, the counter checks, using a nested IF on lines 17–21, to see if the last state 9 has been reached. After state 9, the counter synchronously recycles to 0. Otherwise, the count will be incremented. If the counter is disabled, lines 22-23 will hold the current count value by feeding the current output back to the counter's input. This holding action will be necessary in the cascaded MOD-100 counter for the 10s digit to hold its current state while the 1s digit progresses through its count sequence. An appropriate design strategy would be for us to simulate this module to determine if it functions correctly before we use it in a more complex circuit application. From the simulation results for mod10, given in Figure 7-51, we see that the count sequence is correct, the *clear* is synchronous and has priority, and *enable* controls both the count function and the decoding output tc.

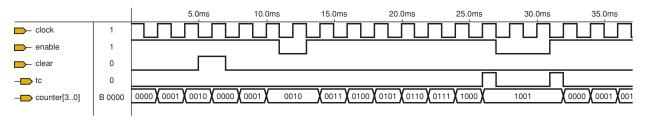


FIGURE 7-51 MOD-10 simulation results.

After creating a default symbol for our mod10 counter module, we can now draw the block diagram for the MOD-100 BCD counter application. The input ports, output ports, and wiring have also been added to create the design in Figure 7-52. Notice that the counter outputs representing the 1s and 10s digits are drawn as buses. The mod10 modules are clocked synchronously. They are cascaded by using the terminal count output from the 1s digit to control the enable input on the 10s digit. The *en* input port controls the enabling/disabling of the entire MOD-100 counter circuit. The BCD counter design can be easily expanded with an additional mod10 stage by connecting the *tc* output to the next *enable* input for each digit needed. A sample of simulation results can be seen in Figure 7-53. The simulation shows that the MOD-100 counter has a correct BCD count sequence and can be synchronously cleared.

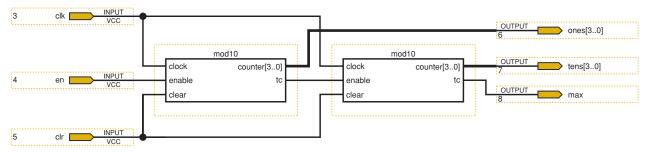
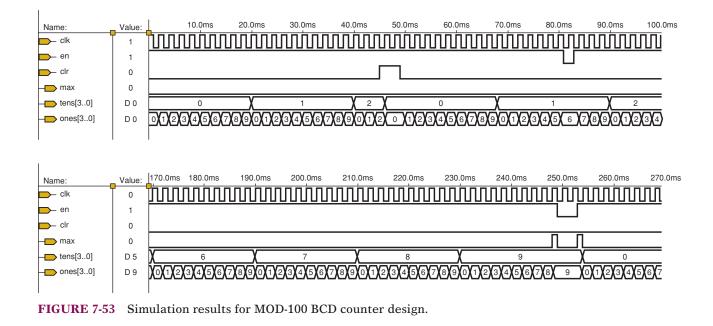


FIGURE 7-52 Block diagram design for a MOD-100 BCD counter.



CASCADING VHDL BCD COUNTERS

The ENTITY and ARCHITECTURE for our MOD-10 BCD counter is shown in lines 26–51 of Figure 7-54. The terminal state for a BCD counter is 9. Lines 38–40 will detect this terminal state only when the counter is enabled with a HIGH. ANDing the enable control in the decoding function will allow more than two counter modules to be cascaded synchronously if necessary and makes our mod10 design more versatile. The *clear* function will be synchronous in VHDL by placing it in the nested IF statement (line 42) after the clock edge has been detected in line 41. If *clear* is inactive, we next check to see if the counter is enabled (line 43). If enable is HIGH, the counter checks, using another nested IF on lines 44-46, to see if the last state 9 has been reached. After state 9, the counter synchronously recycles to 0. Otherwise, the count will be incremented. If the counter is disabled, VHDL will automatically hold the current count value. This holding action will be necessary in the cascaded MOD-100 counter for the 10s digit to hold its current state while the 1s digit progresses through its count sequence. An appropriate design strategy would be for us to simulate this module as a separate ENTITY to determine if it functions correctly before we use it in a more complex circuit application. Simulation results for the mod10 ENTITY, given in Figure 7-51, show that the count sequence is correct, the *clear* is synchronous and has priority, and *enable* controls both the count function and the decoding output.

We have two choices for implementing the MOD-100 counter. One technique is to represent the design graphically in a block diagram as seen in Figure 7-52. The mod10 counter modules, input ports, output ports, and wiring have also been added to create the MOD-100 counter. Notice that the counter outputs representing the 1s and 10s digits are drawn as buses. The mod10 modules are clocked synchronously. They are cascaded by using the terminal count output from the 1s digit to control the enable input on the 10s digit. The *en* input port controls the enabling/disabling of the entire MOD-100 counter circuit. The BCD counter design can be easily expanded with an additional mod10 stage by connecting the *tc* output to the next *enable* input for each digit needed. A sample of simulation results can be seen in Figure 7-53.

```
1
     ENTITY mod100 IS
 2
     PORT (
 3
            clk, en, clr
                                           :IN BIT;
 4
                                           :OUT INTEGER RANGE 0 TO 15;
            ones
 5
                                           :OUT INTEGER RANGE 0 TO 15;
            tens
 6
            max
                                           :OUT BIT
 7
     );
 8
     END mod100;
 9
     ARCHITECTURE toplevel OF mod100 IS
10
     COMPONENT mod10
           PORT (
11
12
                  clock, enable, clear
                                           :IN BIT;
13
                                           :OUT INTEGER RANGE 0 TO 15;
                  q
14
                  tc
                                           :OUT BIT
15
                  );
16
     END COMPONENT;
17
     SIGNAL rco
                                           :BIT;
     BEGIN
18
19
     digit1: mod10 PORT MAP (clock => clk, enable => en,
20
                             clear => clr, q => ones, tc => rco);
21
     digit2: mod10 PORT MAP (clock => clk, enable => rco,
22
                              clear => clr, g => tens, tc => max);
23
     END toplevel;
24
25
     ENTITY mod10 IS
26
27
     PORT (
28
           clock, enable, clear
                                          :IN BIT;
29
                                           :OUT INTEGER RANGE 0 TO 15;
           α
30
                                           :OUT BIT
            tc
31
     );
32
     END mod10;
33
     ARCHITECTURE lowerblk OF mod10 IS
34
     BEGIN
35
            PROCESS (clock, enable)
36
                  VARIABLE counter
                                         :INTEGER RANGE 0 TO 15;
37
            BEGIN
38
                  IF ((counter = 9) AND (enable = '1')) THEN tc <= '1';
                  ELSE tc <= '0';
39
                  END IF;
40
41
                  IF (clock'EVENT AND clock = '1') THEN
42
                        IF (clear = '1') THEN counter := 0;
43
                        ELSIF (enable = '1') THEN
44
                              IF (counter = 9) THEN counter := 0;
45
                              ELSE
                                     counter := counter + 1;
46
                              END IF;
47
                        END IF;
48
                  END IF;
49
                  q <= counter;
50
            END PROCESS;
51
     END lowerblk;
```

FIGURE 7-54 MOD-100 BCD counter in VHDL.

The simulation shows that the MOD-100 counter has a correct BCD count sequence and can be synchronously cleared.

The second technique for creating the MOD-100 counter is to make the necessary connections between design modules by describing the circuit structure with VHDL. The listing for this system design file is given in Figure 7-54. The ENTITY/ARCHITECTURE description for the mod10 sub-block is contained within the overall mod100 design file (but could be in a separate file within this project's folder). The mod100 design file would be the top

level for the hierarchical design of this system. It contains lower-level subblocks, which are actually two copies of the lower-level mod10 counter. The mod10 COMPONENT is declared in this higher-level design file (lines 10–16). The wiring connections for each instance where the module is utilized are listed in a PORT MAP. Since we need two instances of mod10, there is a PORT MAP for each instance (lines 19–20 and 21–22). Each instance must have a unique label (*digit1* or *digit2*) to distinguish them from each other. The PORT MAPs contain named associations between the lower-level module ports, given on the left, and the higher-level signals to which they are connected, given on the right. This circuit produces the same simulation results shown in Figure 7-53.

REVIEW QUESTIONS

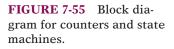
- 1. Describe how to connect HDL modules together to create a digital system.
 - 2. What is a bus and how is it represented in a graphical block diagram design file in Altera?
 - 3. What counter features must be included to synchronously cascade counter modules together?

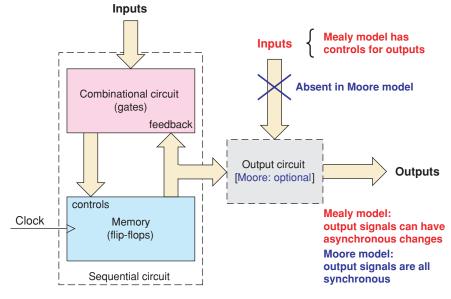
7-14 STATE MACHINES

The term state machine refers to a circuit that sequences through a set of predetermined states controlled by a clock and other input signals. So the counter circuits we have been studying so far in Chapter 7 are state machines. Generally, we use the term *counter* for sequential circuits that have a regular numeric count sequence. They may count up or count down, they may have a full 2^N modules or they may have a $< 2^N$ modulus, or they may recycle or stop automatically at some predetermined state. A counter, as its name implies, is used to count things. The things that are counted are actually called clock pulses, but the pulses may represent many kinds of events. The pulses may be the cycles of a signal for frequency division or they may be seconds, minutes, and hours of a day for a digital clock. They may indicate that an item has moved down the conveyer in a factory or that a car has passed a particular spot on the highway.

The term *state machine* is more often used to describe other kinds of sequential circuits. They may have an irregular counting pattern like our stepper motor control circuit in Section 7-10. The objective for that design was to drive a stepper motor so that it would rotate in precise angular steps. The control circuit had to produce the required specific sequence of states for that movement, rather than count numerically. There are also many applications where we do not care about the specific binary value for each state because we will use appropriate decoding logic to identify specific states of interest and to generate desired output signals. The general distinction between the two terms is that a *counter* is commonly used to count events, while a *state machine* is commonly used to control events. The correct descriptive term depends on how we wish to use the sequential circuit.

The block diagram shown in Figure 7-55 may represent a state machine or a counter. In Section 7-10 we found out that the classic sequential circuit design process was to figure out how many flip-flops would be needed and then determine the necessary combinational circuit to produce the desired sequence. The output produced by a counter or a state machine may come





directly from the flip-flop outputs or there may be some gating circuitry needed, as indicated in the block diagram. The two variations are described as either a **Mealy model** for a sequential circuit or a **Moore model**. In the Mealy model the output signals are also controlled by additional input signals, while the Moore model does not have any external controls for the generated output signals. The Moore output is a function only of the current flipflop state. An example of a Moore-type design would be the decoded MOD-5 circuit in Section 7-13. On the other hand, the BCD counter design in the same section would be a Mealy-type design because of the external input (*enable*) that controls the terminal state decoding output (*tc*). One significant consequence of this subtle design variation is that Moore-type circuit outputs will be completely synchronous to the circuit's clock, while outputs produced by a Mealy-type circuit can change asynchronously. The enable input is not synchronized to the system clock in our MOD-10 design.

HDLs, of course, can make state machines easy and intuitive to describe. As an oversimplified example that everyone can relate to, the following hardware description deals with four states through which a typical washing machine might progress. Although a real washing machine is more complex than this example, it will serve to demonstrate the techniques. This washing machine is idle until the start button is pressed, then it fills with water until the tub is full, then it runs the agitator until a timer expires, and finally it spins the tub until the water is spun out, at which time it goes back to idle. The point of this example focuses on the use of a set of named states for which no binary values are defined. The name of the counter variable is *wash*, which can be in any of the named states: *idle, fill, agitate,* or *spin*.

The AHDL code in Figure 7-56 shows the syntax for declaring a counter with named states on lines 6 and 7. The name of this counter is *cycle*. The keyword **MACHINE** is used in AHDL to define *cycle* as a state machine. The number of bits needed for this counter to produce the named states will be determined by the compiler. Notice that in line 7 the states are named, but the binary value for each state is also left for the compiler to determine. The

FIGURE 7-56 State machine example using AHDL.

```
1
      SUBDESIGN fig7_56
2
      ( clock, start, full, timesup, dry
                                             : INPUT;
 3
         water_valve, ag_mode, sp_mode
                                             :OUTPUT;
 4
      )
 5
      VARIABLE
      cycle: MACHINE
 6
 7
              WITH STATES (idle, fill, agitate, spin);
8
      BEGIN
9
      cycle.clk = clock;
10
         CASE cycle IS
11
12
            WHEN idle =>IF start THEN cycle = fill;
13
                       ELSE cycle = idle;
14
                        END IF;
15
            WHEN fill =>IF full THEN cycle = agitate;
                        ELSE
                                cycle = fill;
16
17
                        END IF;
            WHEN agitate=> IF timesup THEN cycle = spin;
18
19
                        ELSE
                                cycle = agitate;
20
                        END IF;
21
            WHEN spin => IF dry THEN cycle = idle;
22
                        ELSE
                                cycle = spin;
23
                        END IF;
            WHEN OTHERS => cycle = idle;
24
25
         END CASE;
26
         TABLE
27
                    => water_valve,
28
            cycle
                                       ag_mode, sp_mode;
            idle
                                       GND,
                                                 GND;
29
                    => GND,
            fill => VCC,
30
                                       GND,
                                                 GND;
31
            agitate => GND,
                                        VCC,
                                                 GND;
32
            spin
                    => GND,
                                        GND,
                                                 VCC;
33
         END TABLE;
34
      END;
```

designer does not need to worry about this level of detail. The CASE structure on lines 11–25 and the decoding logic that drives the outputs (lines 27–33) refer to the states by name. This makes the description easy to read and allows the compiler more freedom to minimize the circuitry. If the design requires the state machine also to be connected to an output port, then line 6 can be changed to:

cycle: MACHINE OF BITS (st [1..0])

and the output port st[1..0] can be added to the SUBDESIGN section. A second state machine option that is available is the ability for the designer to define a binary value for each state. This can be accomplished in this example by changing line 7 to:

WITH STATES (idle = B''00'', fill = B''01'', agitate = B''11'', spin = B''10'');

SIMPLE VHDL STATE MACHINE

The VHDL code in Figure 7-57 shows the syntax for declaring a counter with named states. On line 6, a data object is declared named *state_machine*. Notice the keyword TYPE. This is called an **enumerated type** in VHDL, in which the designer lists by symbolic names all possible values that a signal, variable, or port that is declared to be of that type is allowed to have. Notice also that on line 6, the states are named, but the binary value for each state is left for the compiler to determine. The designer does not need to worry about this level of detail. The CASE structure on lines 12–29 and the decoding logic that drives the outputs (lines 31–36) refer to the states by name. This makes the description easy to read and allows the compiler more freedom to minimize the circuitry.

Using the simulator to verify our HDL designs produces the results given in Figure 7-58. The Altera simulator allows us to also simulate intermediate nodes in our design modules. The "buried" state machine named *cycle* has been included in the simulation in order to confirm that it operates correctly. Note that the results for *cycle* are given twice, since it will be displayed differently

```
ENTITY fig7_57 IS
1
            clock, start, full, timesup, dry
     PORT (
2
                                                     :IN BIT:
3
           water_valve, ag_mode, sp_mode
                                                     :OUT BIT);
 4
     END fig7 57;
     ARCHITECTURE vhdl OF fig7_57 IS
 5
 6
     TYPE state_machine IS (idle, fill, agitate, spin);
7
     BEGIN
8
        PROCESS (clock)
9
        VARIABLE cycle
                                   :state machine;
10
        BEGIN
11
        IF (clock'EVENT AND clock = '1') THEN
12
           CASE cycle IS
13
              WHEN idle =>
                 IF start = '1' THEN cycle := fill;
14
                 ELSE
                                         cycle := idle;
15
                 END IF;
16
17
              WHEN fill =>
                 IF full = '1' THEN
18
                                         cycle := agitate;
19
                 ELSE
                                         cycle := fill;
20
                 END IF;
              WHEN agitate =>
21
                 IF timesup = '1' THEN cycle := spin;
2.2
23
                 ELSE
                                         cycle := agitate;
24
                 END IF;
25
              WHEN spin =>
                 IF dry = '1' THEN
26
                                         cycle := idle;
27
                 ELSE
                                         cycle := spin;
                 END IF;
28
           END CASE;
29
30
        END IF;
31
        CASE cycle IS
           WHEN idle => water_valve <= '0'; ag_mode <= '0'; sp_mode <= '0';</pre>
32
33
           WHEN fill => water_valve <= '1'; ag_mode <= '0'; sp_mode <= '0';
           WHEN agitate => water_valve <= '0'; ag_mode <= '1'; sp_mode <= '0';
34
35
          WHEN spin => water_valve <= '0'; ag_mode <= '0'; sp_mode <= '1';
36
        END CASE:
37
        END PROCESS;
38
     END vhdl;
```

FIGURE 7-57 State machine example using VHDL.

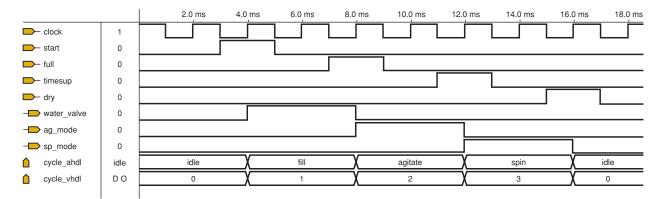


FIGURE 7-58 Simulation of washing machine HDL design example for a state machine.

for the two HDLs. The simulator cannot actually show the simulations for both AHDL and VHDL together. The second buried node information has been merely copied and pasted for a composite figure here. In AHDL the machine state names are displayed, while in VHDL the compiler-assigned values for the enumerated state names are displayed instead.

Traffic Light Controller State Machine

Let us investigate a state machine design that is a little more complicated, a traffic light controller. The block diagram is shown in Figure 7-59. Our simple controller is designed to control the flow of traffic at the intersection of a main road with a less busy side road. Traffic will flow uninterrupted on the main road with a green light, until a car is sensed on the side road (indicated by the input labeled *car*). After a time delay that is set by the five-bit binary input labeled *tmaingrn*, the main road light will change to yellow. The *tmaingrn* time delay ensures that the main road will receive a green light for

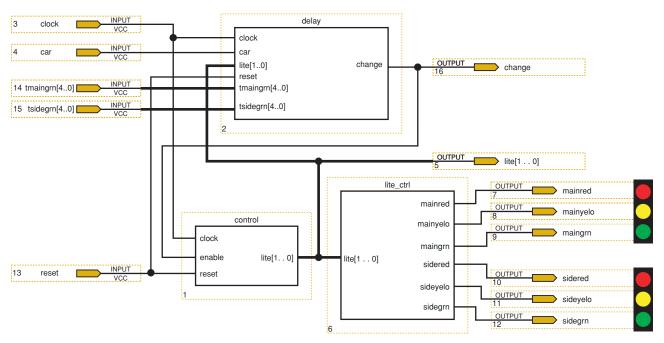


FIGURE 7-59 Traffic light controller.

at least this length of time during each cycling of the lights. The yellow light will last for a fixed amount of time that is set in the HDL design and then transition to red. When the main road light is red, the side road light turns to green. The side road light will be green for a time that is set by the five-bit binary input labeled *tsidegrn*. Again the yellow light will last for the same fixed length of time and then the side road will return to a red light and the main road light will be green again. The delay module will control the time periods for each of the lights. The actual time delays will be the period of the system clock multiplied times the delay factor. The control module determines the state of the traffic controller. There are four light combinations—maingreen/side-red, main-yellow/side-red, main-red/side-green, and main-red/sideyellow—so control will need four states. The traffic light states are translated into the proper on–off patterns for each of the six pairs of lights by the lite_ctrl module. The outputs labeled *change* and *lite* are provided for diagnostic purposes. *Reset* is used to initialize each of the two sequential circuits.

AHDL TRAFFIC LIGHT CONTROLLER

The three design modules for our AHDL traffic light controller are listed together in Figure 7-60. They are actually three separate design files that are interconnected with the block diagram design shown in Figure 7-59. The delay module (lines 1-23) is basically a buried down counter (line 20) named *mach*, which waits at zero when the main road has a green light (lite = 0) until it is triggered by the car sensor (line 13) to load the delay factor tmaingrn-1 on line 14. Since the counter decrements all the way to zero, one is subtracted from each delay factor to make the delay counter's modulus equal to the value of the delay factor. For example, if we wish to have a delay factor of 25, the counter must count from 24 down to 0. The actual length of time represented by the delay factors depends on the clock frequency. With a 1-Hz clock frequency, the period would be 1 s, and the delay factors would then be in seconds. Line 22 defines an output signal called *change* that detects when mach is equal to one. Change will be HIGH to indicate that the test condition is true, which in turn will enable the state machine in the control module to move to its next state (lite = 1) when clocked to indicate a yellow light on the main road. As the delay counter mach counts down and reaches zero, CASE determines that *lite* has a new value and the fixed time delay factor of 5 for a yellow light is loaded (actually loading one less than 5, as previously discussed) into mach (line 16) on the next clock. The count down continues from this new delay time, with *change* again enabling the control module to move to its next state (lite = 2) when mach is equal to 1, resulting in a green light for the side road. When *mach* again reaches zero, the time delay (tsidegrn-1) for a green light on the side road will be loaded into the down counter (line 17). When change again goes active, lite will advance to state 3 for a yellow light on the side road. Mach will recycle to the value 4 (5-1) on line 18 for the fixed time delay for a yellow light. When change goes active this time, the control module will return to the *lite* = 0 state (green light on main). When mach decrements to its terminal state (zero) this time, lines 13–15 will determine by the status of the *car* sensor input whether to wait for another car or to load in the delay factor for a green light on main (tmaingrn-1) to start the cycle over again. The main road will receive a green light for at least this length of time, even if there is a continuous stream of cars on the side road. It is obvious that we could make improvements to this design, but that, of course, would also complicate the design further.

```
1
    SUBDESIGN delay
 2
    ( clock, car, lite[1..0], reset
                                            : INPUT;
 3
       tmaingrn[4..0], tsidegrn[4..0]
                                            : INPUT;
 4
                                            :OUTPUT; )
       change
 5
    VARIABLE
 6
       mach[4..0]
                                            :DFF;
 7
    BEGIN
 8
       mach[].clk = clock;
                                                   -- with 1 Hz clock, times in seconds
 9
       mach[].clrn = reset;
       IF mach[] == 0 THEN
10
          CASE lite[] IS
                                                   -- check state of light controller
11
12
              WHEN 0 =>
13
                 IF !car THEN mach[].d = 0;
                                                 -- wait for car on side road
14
                 ELSE mach[].d = tmaingrn[] - 1; -- set time for main's green
15
                 END IF;
              WHEN 1 = mach[].d = 5 - 1;
16
                                                  -- set time for main's yellow
17
              WHEN 2 => mach[].d = tsidegrn[] - 1; -- set time for side's green
18
              WHEN 3 => mach[].d = 5 - 1;
                                                   -- set time for side's yellow
19
           END CASE;
20
        ELSE mach[].d = mach[].q - 1;
                                                   -- decrement timer counter
21
       END IF;
2.2
       change = mach[] == 1;
                                                   -- change lights on control module
23
    END:
24
                                 _____
25
    SUBDESIGN control
26
    ( clock, enable, reset :INPUT;
27
       lite[1..0]
                              :OUTPUT; )
28
    VARTABLE
29
       light:
                MACHINE OF BITS (lite[1..0]) -- need 4 states for light combinations
                 WITH STATES (mgrn = B"00", myel = B"01", sgrn = B"10", syel = B"11");
30
31
    BEGIN
32
       light.clk = clock;
33
       light.reset = !reset;
                                     -- MACHINEs have asynchronous, active-high reset
34
       CASE light IS
                               -- wait for enable to change light states
          WHEN mgrn => IF enable THEN light = myel; ELSE light = mgrn; END IF;
35
          WHEN myel => IF enable THEN light = sgrn; ELSE light = myel; END IF;
36
          WHEN sgrn=>IF enable THEN light = syel;ELSE light = sgrn;END IF;WHEN syel=>IF enable THEN light = mgrn;ELSE light = syel;END IF;
37
38
39
       END CASE;
40
    END:
41
                       _____
    SUBDESIGN lite_ctrl
42
43
    ( lite[1..0]
                                     : INPUT;
       mainred, mainyelo, maingrn
sidered, sideyelo, sidegrn
44
                                     :OUTPUT;
45
                                     :OUTPUT;
                                               )
46
    BEGIN
47
       CASE lite[]
                       IS
                                     -- determine which lights to turn on
48
          WHEN B"00"
                      =>
                             maingrn = VCC; mainyelo = GND; mainred = GND;
49
                              sidegrn = GND; sideyelo = GND; sidered = VCC;
50
          WHEN B"01"
                             maingrn = GND; mainyelo = VCC; mainred = GND;
                        =>
51
                              sidegrn = GND; sideyelo = GND; sidered = VCC;
52
          WHEN B"10"
                      =>
                              maingrn = GND; mainyelo = GND; mainred = VCC;
53
                               sidegrn = VCC; sideyelo = GND; sidered = GND;
54
          WHEN B"11"
                      =>
                              maingrn = GND; mainyelo = GND; mainred = VCC;
55
                               sidegrn = GND; sideyelo = VCC; sidered = GND;
56
       END CASE;
57
    END;
```

FIGURE 7-60 AHDL design files for traffic light controller.

The control module (lines 25–40) contains a state machine named *light* that will sequence through the four states for the traffic light combinations. The bits for the state machine are named and connected as an output port for this module (lines 27 and 29). The four states for *light* are named *mgrn*, *myel*, *sgrn*, and *syel* on line 30. Each state represents which road, main or side, is to receive a green or yellow light. The other road will have a red light. The values for each state of the control module have also been specified on line 30 so that we can identify them as inputs to the other two modules, delay and lite_ctrl. The *enable* input is connected to the *change* output signal produced by the delay module. When enabled, the *light* state machine will advance to the next state when clocked as described by the CASE and nested IF statements on lines 34–39. Otherwise, *light* will hold at the current state.

The lite_ctrl module (lines 42–57) inputs *lite[1..0]*, which represents the state of the *light* state machine from the control module, and will output the signals that will turn on the proper combinations of green, yellow, and red lights for the main and side roads. Each output from the lite_ctrl module will actually be connected to lamp driver circuits to control the higher voltages and currents necessary for real lamps in a traffic light. The CASE statement on lines 47–55 determines which main road/side road light combination to turn on for each state of *light*. The function of the lite_ctrl module is very much like a decoder. It essentially decodes each state combination of *lite* to turn on a green or yellow light for one road and a red light for the other road. A unique output combination is produced for each input state.

VHDL TRAFFIC LIGHT CONTROLLER

The VHDL design for the traffic light controller is listed in Figure 7-61. The top level of the design is described structurally on lines 1–34. There are three COMPONENT modules to declare (lines 10–24). The PORT MAPs giving the wiring interconnects between each module and the top level design are listed on lines 26–33.

The delay module (lines 36–66) is basically a buried down counter (line 59) created with the integer variable mach that waits at zero when the main road has a green light (lite = "00") until it is triggered by the car sensor (line 52) to load the delay factor tmaingrn-1 on line 53. Since the counter decrements all the way to zero, one is subtracted from each delay factor to make the delay counter's modulus equal to the value of the delay factor. For example, if we wish to have a delay factor of 25, the counter must count from 24 down to 0. The actual length of time represented by the delay factors depends on the clock frequency. With a 1-Hz clock frequency, the period would be 1 s, and the delay factors would then be in seconds. Lines 62-64 define an output signal called *change* that detects when *mach* is equal to one. *Change* will be HIGH to indicate that the test condition is true, which in turn will enable the state machine in the control module to move to its next state (lite = "01") when clocked to indicate a yellow light on the main road. When mach reaches zero now, CASE determines that lite has a new value and the fixed time delay factor of 5 for a yellow light is loaded (actually loading one less, as previously discussed) into mach (line 55) on the next clock. The count down continues from this new delay time, with *change* again enabling the control module to move to its next state (lite = "10"), resulting in a green light for the side road. When mach again reaches zero, the time delay (tsidegrn-1) for a green light on the side road will be loaded into the down

ENTITY traffic IS 1 2 PORT (clock, car, reset :IN BIT; 3 tmaingrn, tsidegrn :IN INTEGER RANGE 0 TO 31; 4 lite :BUFFER INTEGER RANGE 0 TO 3; 5 change :BUFFER BIT; mainred, mainyelo, maingrn 6 :OUT BIT; 7 sidered, sideyelo, sidegrn :OUT BIT); 8 END traffic; 9 ARCHITECTURE toplevel OF traffic IS 10 COMPONENT delay 11 PORT (clock, car, reset :IN BIT; 12 lite :IN INTEGER RANGE 0 TO 3; tmaingrn, tsidegrn :IN INTEGER RANGE 0 TO 31; 13 14 change :OUT BIT); 15 END COMPONENT; 16 COMPONENT control 17 PORT (clock, enable, reset :IN BIT; :OUT INTEGER RANGE 0 TO 3); 18 lite END COMPONENT; 19 COMPONENT lite ctrl 20 21 PORT (lite :IN INTEGER RANGE 0 TO 3; mainred, mainyelo, maingrn 2.2 :OUT BIT; 23 sidered, sideyelo, sidegrn :OUT BIT); 24 END COMPONENT; 25 BEGIN module1: delay PORT MAP (clock => clock, car => car, reset => reset, 26 27 lite => lite, tmaingrn => tmaingrn, tsidegrn => tsidegrn, 2.8 change => change); 29 module2: control PORT MAP (clock => clock, enable => change, reset => reset, 30 lite => lite); 31 module3: lite ctrl PORT MAP (lite => lite, mainred => mainred, mainyelo => mainyelo, 32 maingrn => maingrn, sidered => sidered, sideyelo => sideyelo, 33 sidegrn => sidegrn); 34 END toplevel; 35 36 ENTITY delay IS PORT (clock, car, reset 37 :IN BIT; 38 lite :IN BIT VECTOR (1 DOWNTO 0); :IN INTEGER RANGE 0 TO 31; 39 tmaingrn, tsidegrn 40 change :OUT BIT); 41 END delay; ARCHITECTURE time OF delay IS 42 BEGIN 43 44 PROCESS (clock, reset) 45 VARIABLE mach :INTEGER RANGE 0 TO 31; 46 BEGIN 47 IF reset = '0' THEN mach := 0; ELSIF (clock = '1' AND clock'EVENT) THEN -- with 1 Hz clock, times in seconds 48 IF mach = 0 THEN 49 50 CASE lite IS WHEN "00" 51 IF car = '0' THEN mach := 0;52 -- wait for car on side road mach := tmaingrn - 1; -- set time for main's green 53 ELSE 54 END IF; WHEN "01" 55 => mach := 5 - 1;-- set time for main's yellow WHEN "10" 56 => mach := tsidegrn - 1; -- set time for side's green WHEN "11" -- set time for side's yellow 57 mach := 5 - 1;=> 58 END CASE: 59 ELSE mach := mach - 1; -- decrement timer counter 60 END IF; 61 END IF;

FIGURE 7-61 VHDL design for traffic light controller.

```
IF mach = 1 THEN change <= '1';
 62
                                                                                               -- change lights on control
              ELSE change <= '0';
 63
 64
              END IF;
              END PROCESS;
 65
 66
         END time;
 67
         _ _ _ _ _ _ .
         ENTITY control IS
 68
         PORT ( clock, enable, reset : IN BIT;
 69
 70
                         lite
                                                             :OUT BIT VECTOR (1 DOWNTO 0));
 71
         END control;
         ARCHITECTURE a OF control IS
 72
 73
         TYPE enumerated IS (mgrn, myel, sgrn, syel); -- need 4 states for light combinations
 74
         BEGIN
 75
              PROCESS (clock, reset)
 76
              VARIABLE lights :enumerated;
              BEGIN
 77
                   IF reset = '0' THEN lights := mgrn;
 78
                   ELSIF (clock = '1' AND clock'EVENT) THEN
 79
                        IF enable = '1' THEN -- wait for enable to change light states
 80
                              CASE lights IS
 81
 82
                                    WHEN mgrn
                                                              =>
                                                                       lights := myel;
                                                             => lights := myel;
=> lights := sgrn;
=> lights := syel;
=> lights := mgrn;
                                    WHEN myel
 83
                                   WHEN sgrn
WHEN syel
 84
 85
 86
                              END CASE;
 87
                        END IF;
 88
                  END IF;
                   CASE lights IS
                                                                        -- patterns for light states
 89

      WHEN mgrn=>
      lite <= "00";</td>

      WHEN myel=>
      lite <= "01";</td>

      WHEN sgrn=>
      lite <= "10";</td>

      WHEN syel=>
      lite <= "11";</td>

 90
 91
 92
 93
 94
                   END CASE;
 95
           END PROCESS;
 96
         END a;
 97
         _ _ _ _ .
                                               ENTITY lite ctrl IS
 98
                       lite:IN BIT_VECTOR (1 DOWNTO 0);mainred, mainyelo, maingrn:OUT BIT;sidered, sideyelo, sidegrn:OUT BIT);
 99
         PORT ( lite
100
101
102
         END lite ctrl;
103
         ARCHITECTURE patterns OF lite_ctrl IS
104
         BEGIN
105
              PROCESS (lite)
106
              BEGIN
              CASE lite IS -- control state determines which lights to turn on/off
107
                  SE fifte is -- control state determines which lights to turn on/off
WHEN "00" => maingrn <= '1'; mainyelo <= '0'; sidered <= '0';
WHEN "01" => maingrn <= '0'; mainyelo <= '1'; mainred <= '0';
sidegrn <= '0'; sideyelo <= '0'; sidered <= '1';
WHEN "10" => maingrn <= '0'; mainyelo <= '0'; sidered <= '1';
WHEN "11" => maingrn <= '0'; mainyelo <= '0'; sidered <= '1';
WHEN "11" => maingrn <= '0'; mainyelo <= '0'; sidered <= '0';
WHEN "11" => maingrn <= '0'; sideyelo <= '0'; sidered <= '1';
WHEN "11" => maingrn <= '0'; sideyelo <= '0'; sidered <= '1';
WHEN "11" => maingrn <= '0'; sideyelo <= '0'; sidered <= '0';
WHEN "11" => maingrn <= '0'; mainyelo <= '0'; sidered <= '0';</pre>
108
109
110
111
112
113
114
115
116
              END CASE;
117
              END PROCESS;
118
         END patterns;
```

counter (line 56). When *change* again goes active, *lite* will advance to "11" for a yellow light on the side road. *Mach* will recycle to the value 4(5-1) on line 57 for the fixed time delay for a yellow light. When *change* goes active this time, the control module will return to *lite* = "00" (green light on main). When *mach* decrements to its terminal state (zero) this time, lines 52–54 will determine by the status of the *car* sensor input whether to wait for another car or to load in the delay factor for a green light on main (*tmaingrn*-1) to start the cycle over again. The main road will receive a green light for at least this length of time, even if there is a continuous stream of cars on the side road. It is obvious that we could make improvements to this design, but that, of course, would also complicate the design further.

The control module (lines 68–96) contains a state machine named *lights* that will sequence through four enumerated states for the traffic light combinations. The four enumerated states for *lights* are *mgrn*, *myel*, *sgrn*, and *syel* (lines 73 and 76). Each state represents which road, main or side, is to receive a green or yellow light. The other road will have a red light. The *enable* input is connected to the *change* output signal produced by the delay module. When enabled, the *lights* state machine will advance to the next state when clocked, as described by the nested IF and CASE statements on lines 79–88. Otherwise, *lights* will hold at the current state. The bit patterns for output port *lite* have been specified for each state of *lights* with the CASE statement on lines 89–94 so that we can identify them as inputs to the other two modules, delay and lite_ctrl.

The lite_ctrl module (lines 98–118) inputs *lite*, which represents the state of the *lights* state machine from the control module, and will output the signals that will turn on the proper combinations of green, yellow, and red lights for the main and side roads. Each output from the lite_ctrl module will actually be connected to lamp driver circuits to control the higher voltages and currents necessary for real lamps in a traffic light. The CASE statement on lines 107–116, invoked by the PROCESS when the *lite* input changes, determines which main road/side road light combination to turn on for each state of *lights*. The function of the lite_ctrl module is very much like a decoder. It essentially decodes each state combination of *lite* to turn on a green or yellow light for one road and a red light for the other road. A unique output combination is produced for each input state.

By this time, you may be wondering why there are so many ways to describe logic circuits. If one way is easier than the others, why not just study that one? The answer, of course, is that each level of abstraction offers advantages over the others in certain cases. The structural method provides the most complete control over interconnections. The use of Boolean equations, truth tables, and PRESENT state/NEXT state tables allows us to describe the way data flows through the circuit using HDL. Finally, the behavioral method allows a more abstract description of the circuit's operation in terms of cause and effect. In practice, each source file may have portions that can be categorized under each level of abstraction. Choosing the right level when writing code is not an issue of right and wrong as much as it is an issue of style and preference.

There are also several ways to approach any task from a standpoint of choosing control structures. Should we use selected signal assignments or Boolean equations, IF/ELSE or CASE, sequential processes or concurrent statements, macrofunctions or megafunctions? Or should we write our own code? The answers to these questions ultimately define your personal strategy in solving the problem. Your preferences and the advantages you find in using one method over another will be established with practice and experience.

REVIEW QUESTIONS

- 1. What is the fundamental difference between a counter and a state machine?
- 2. What is the difference between describing a counter and describing a state machine in an HDL?
- 3. If the actual binary states for a state machine are not defined in the HDL code, how are they assigned?
- 4. What is the advantage of using state machine description?

PART 1 SUMMARY

- 1. In asynchronous (ripple) counters, the clock signal is applied to the LSB FF, and all other FFs are clocked by the output of the preceding FF.
- 2. A counter's MOD number is the number of stable states in its counting cycle; it is also the maximum frequency-division ratio.
- 3. The normal (maximum) MOD number of a counter is 2^{*N*}. One way to modify a counter's MOD number is to add circuitry that will cause it to recycle before it reaches its normal last count.
- 4. Counters can be cascaded (chained together) to produce greater counting ranges and frequency-division ratios.
- 5. In a synchronous (parallel) counter, all of the FFs are simultaneously clocked from the input clock signal.
- 6. The maximum clock frequency for an asynchronous counter, f_{max} , decreases as the number of bits increases. For a synchronous counter, f_{max} remains the same, regardless of the number of bits.
- 7. A decade counter is any MOD-10 counter. A BCD counter is a decade counter that sequences through the 10 BCD codes (0–9).
- 8. A presettable counter can be loaded with any desired starting count.
- 9. An up/down counter can be commanded to count up or count down.
- 10. Logic gates can be used to decode (detect) any or all states of a counter.
- 11. The count sequence for a synchronous counter can be easily determined by using a PRESENT state/NEXT state table that lists all possible states, the flip-flop input control information, and the resulting NEXT states.
- 12. Synchronous counters with arbitrary counting sequences can be implemented by following a standard design procedure.
- Counters can be described in many different ways using HDL, including structural wiring descriptions, PRESENT state/NEXT state tables, and behavioral descriptions.
- 14. All the features available on the various standard IC counter chips, such as asynchronous or synchronous loading or clearing, count enabling, and terminal count decoding, can be described using HDL. HDL counters can be easily modified for higher MOD numbers or changes in the active levels for controls.
- 15. Digital systems can be subdivided into smaller modules or blocks that can be interconnected as a hierarchical design.
- 16. State machines can be represented in HDL using descriptive names for each state rather than specifying a numeric sequence of states.

PART 1 IMPORTANT TERMS

asynchronous (ripple)
counter
MOD number
glitches
synchronous
(parallel) counters
decade counter
BCD counter
up counter
down counter
up/down counters
presettable counters

parallel load count enable multistage counters cascading decoding PRESENT state/NEXT state table self-correcting counter sequential circuit design J-K excitation table circuit excitation table VARIABLE behavioral level of abstraction hierarchical design state machine mealy model Moore model MACHINE enumerated type

PART 2 7-15 INTEGRATED-CIRCUIT REGISTERS

The various types of registers can be classified according to the manner in which data can be entered into the register for storage and the manner in which data are outputted from the register. The various classifications are listed below.

- 1. Parallel in/parallel out (PIPO)
- 2. Serial in/serial out (SISO)
- 3. Parallel in/serial out (PISO)
- 4. Serial in/parallel out (SIPO)

Each of these types and several variations are available in IC form so that a logic designer can usually find exactly what is required for a given application. In the following sections, we will examine a representative IC from each of the above categories.

7-16 PARALLEL IN/PARALLEL OUT—THE 74ALS174/74HC174

A group of flip-flops that can store multiple bits simultaneously and in which all bits of the stored binary value are directly available is referred to as a **parallel in/parallel out** register. Figure 7-62(a) shows the logic diagram for the 74ALS174 (also the 74HC174), a six-bit register that has parallel inputs D_5 through D_0 and parallel outputs Q_5 through Q_0 . Parallel data are loaded into the register on the PGT of the clock input *CP*. A master reset input \overline{MR} can be used to reset asynchronously all of the register FFs to 0. The logic symbol for the 74ALS174 is shown in Figure 7-62(b). This symbol is used in circuit diagrams to represent the circuitry of Figure 7-62(a).

The 74ALS174 is normally used for synchronous parallel data transfer whereby the logic levels present at the *D* inputs are transferred to the corresponding *Q* outputs when a PGT occurs at the clock *CP*. This IC, however, can be wired for serial data transfer, as the following examples will show.

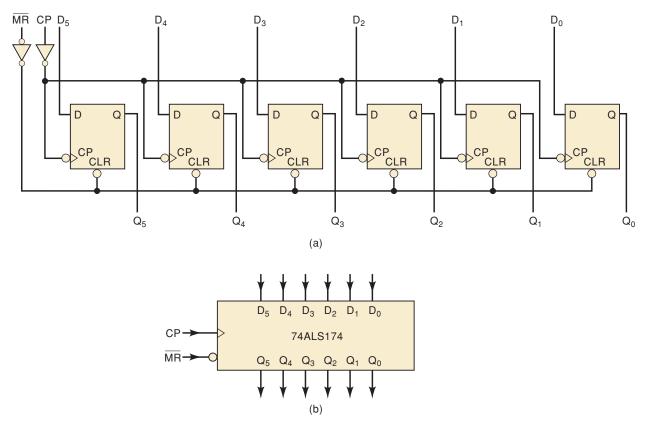


FIGURE 7-62 (a) Circuit diagram of the 74ALS174; (b) logic symbol.

EXAMPLE 7-16

Show how to connect the 74ALS174 so that it operates as a serial shift register with data shifting on each PGT of *CP* as follows: Serial input $\rightarrow Q_5 \rightarrow Q_4 \rightarrow Q_3 \rightarrow Q_2 \rightarrow Q_1 \rightarrow Q_0$. In other words, serial data will enter at D_5 and will output at Q_0 .

Solution

Looking at Figure 7-62(a), we can see that to connect the six FFs as a serial shift register, we have to connect the Q output of one to the D input of the next so that data is transferred in the required manner. Figure 7-63 shows how this is accomplished. Note that data shifts left to right, with input data applied at D_5 and output data appearing at Q_0 .

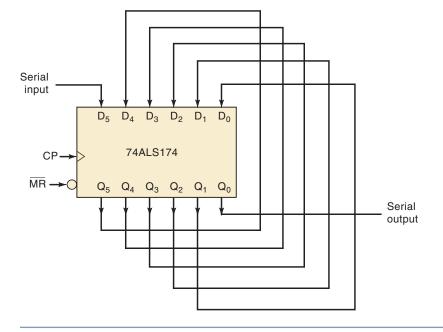
EXAMPLE 7-17

How would you connect two 74ALS174s to operate as a 12-bit shift register?

Solution

Connect a second 74ALS174 IC as a shift register, and connect Q_0 from the first IC to D_5 of the second IC. Connect the *CP* inputs of both ICs so that they will be clocked from the same signal. Also connect the MR inputs together if using the asynchronous reset.





7-17 SERIAL IN/SERIAL OUT—THE 74ALS166/74HC166

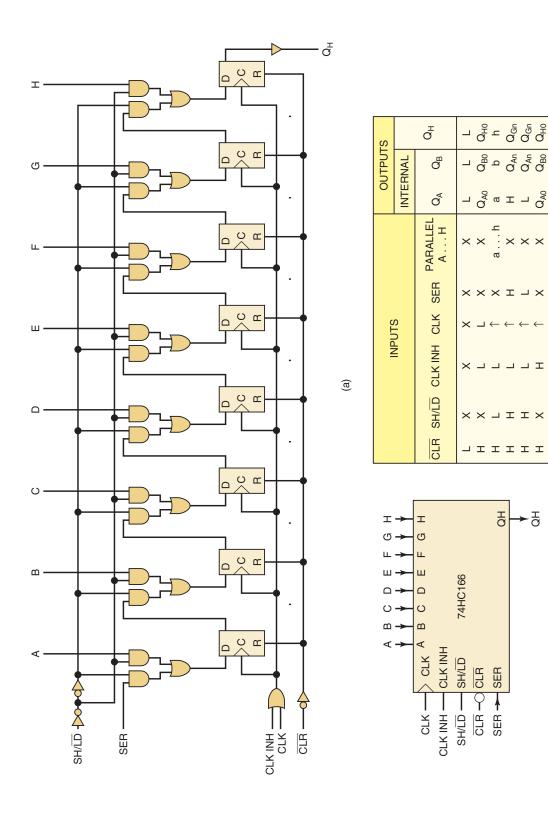
A serial in/serial out shift register will have data loaded into it one bit at a time. The data will move one bit at a time with each clock pulse through the set of flip-flops toward the other end of the register. With continued clocking, the data will then exit the register one bit at a time in the same order as it was originally loaded. The 74HC166 (and also the 74ALS166) can be used as a serial in/serial out register. The logic diagram and schematic symbol for the 74HC166 is shown in Figure 7-64. It is an eight-bit shift register of which only FF QH is accessible. The serial data is input on *SER* and will be stored in FF QA. The serial output is obtained at the other end of the shift register on Q_H . As can be seen from the function table for this shift register in Figure 7-64(c), parallel data can also be synchronously loaded into it. If SH/ $\overline{\text{LD}} = 1$, the register function will be serial shifting, while a LOW will instead parallel load data via the *A* through *H* inputs. The synchronous serial shifting and parallel loading functions can be inhibited (disabled) by applying a HIGH to the *CLK INH* control input. The register also has an active-LOW, asynchronous clear input (*CLR*).

EXAMPLE 7-18

A shift register is often used as a way to delay a digital signal by an integral number of clock cycles. The digital signal is applied to the shift register's serial input and is shifted through the shift register by successive clock pulses until it reaches the end of the shift register, where it appears as the output signal. This method for delaying the effect of a digital signal is common in the digital communications field. For instance, the digital signal might be the digitized version of an audio signal that is to be delayed before it is transmitted. The input waveforms given in Figure 7-65 are applied to a 74HC166. Determine the resultant output waveform.

Solution

QH starts at a LOW, since all flip-flops are initially cleared by the LOW applied to the asynchronous *CLR* input at the beginning of the timing diagram.





a...h

 $\times \times \times$

 $\leftarrow \leftarrow$

 \leftarrow

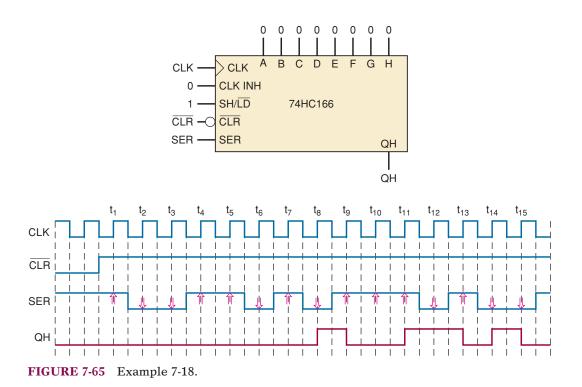
 \neg \neg \neg \top \top

 \neg \pm \pm \times

ВH

SER

►B



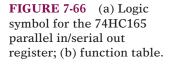
At t_1 , the shift register will input the current bit applied to *SER*. This will be stored in QA. At t_2 , the first bit will move to QB and a second bit on *SER* will be stored in QA. At t_3 , the first bit will now move to QC and a third bit on *SER* will be stored in QA. The first data input bit will finally show up at the output QH at t_8 . Each successive input bit on *SER* will follow at QH delayed by eight clock cycles.

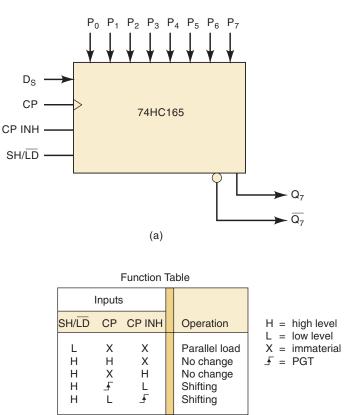
7-18 PARALLEL IN/SERIAL OUT—THE 74ALS165/74HC165

The logic symbol for the 74HC165 is shown in Figure 7-66(a). This IC is an eight-bit **parallel in/serial out** register. It actually has serial data entry via D_S and asynchronous parallel data entry via P_0 through P_7 . The register contains eight FFs— Q_0 through Q_7 —internally connected as a shift register, but the only accessible FF outputs are Q_7 and \overline{Q}_7 . *CP* is the clock input used for the shifting operation. The clock inhibit input, *CP INH* is used to inhibit the effect of the *CP* input. The shift/load input, *SH*/ \overline{LD} , controls which operation is taking place—shifting or parallel loading. The function table in Figure 7-66(b) shows how the various input combinations determine what operation, if any, is being performed. Parallel loading is asynchronous and serial shifting is synchronous. Note that the serial shifting function will always be synchronous, since the clock is required to ensure that the input data moves only one bit at a time with each appropriate clocking edge.

EXAMPLE 7-19

Examine the 74HC165 function table and determine (a) the conditions necessary to load the register with parallel data; (b) the conditions necessary for the shifting operation.





(b)

Solution

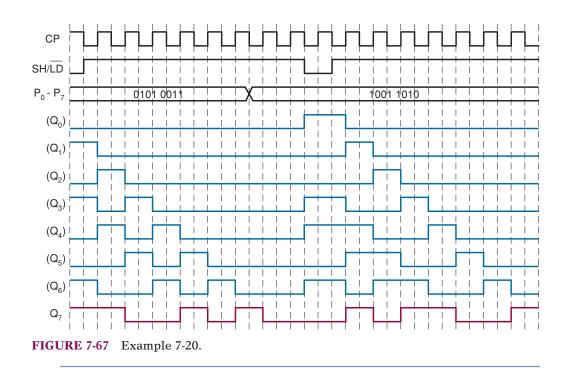
- (a) The first entry in the table shows that the SH/LD input has to be LOW for the parallel load operation. When this input is LOW, the data present at the *P* inputs are *asynchronously* loaded into the register FFs, independent of the *CP* and the *CP INH* inputs. Of course, only the outputs from the last FF are externally available.
- (b) The shifting operation cannot take place unless the SH/LD input is HIGH and a PGT occurs at CP while CP INH is LOW [see the fourth table entry in Figure 7-66(b)]. A HIGH at CP INH will inhibit the effect of any clock pulses. Note that the roles of the CP and CP INH inputs can be reversed, as indicated by the last table entry, because these two signals are ORed together inside the IC.

EXAMPLE 7-20

Determine the output signal at Q_7 if we connect a 74HC165 with $D_S = 0$ and CP INH = 0 and then apply the input waveforms given in Figure 7-67. P_0-P_7 represent the parallel data on $P_0 P_1 P_2 P_3 P_4 P_5 P_6 P_7$.

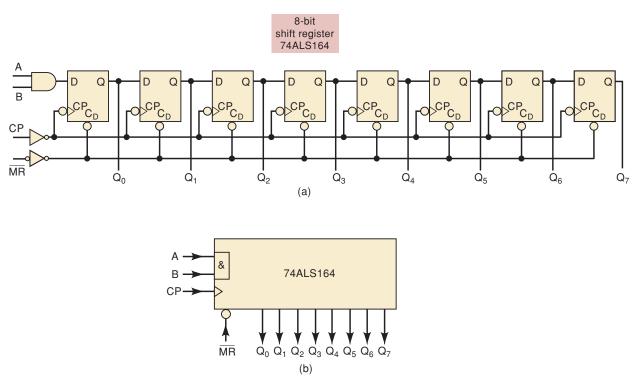
Solution

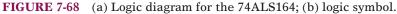
We have drawn the timing diagram for all eight FFs so that we could track their contents over time even though only Q_7 will be accessible. The parallel load is asynchronous and will occur as soon as SH/LD goes LOW. After SH/LD returns to a HIGH, the data stored in the register will move one FF to the right (toward Q_7) with each PGT on CP.



7-19 SERIAL IN/PARALLEL OUT—THE 74ALS164/74HC164

The logic diagram for the 74ALS164 is shown in Figure 7-68(a). It is an eightbit serial in/parallel out shift register with each FF output externally accessible. Instead of a single serial input, an AND gate combines inputs A and Bto produce the serial input to flip-flop Q_0 .



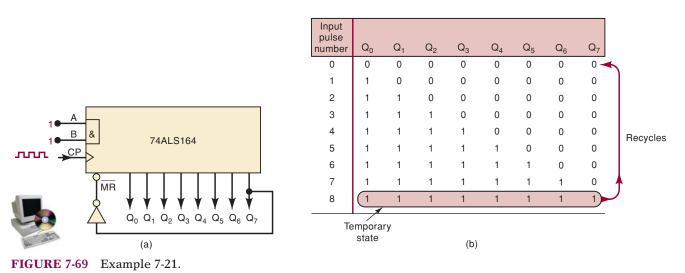


The shift operation occurs on the PGTs of the clock input *CP*. The *MR* input provides asynchronous resetting of all FFs on a LOW level.

The logic symbol for the 74ALS164 is shown in Figure 7-68(b). Note that the & symbol is used inside the block to indicate that the A and B inputs are ANDed inside the IC and the result is applied to the D input of Q_0 .

EXAMPLE 7-21

Assume that the initial contents of the 74ALS164 register in Figure 7-69(a) are 00000000. Determine the sequence of states as clock pulses are applied.



Solution

The correct sequence is given in Figure 7-69(b). With A = B = 1, the serial input is 1, so that 1s will shift into the register on each PGT of *CP*. Because Q_7 is initially at 0, the \overline{MR} input is inactive.

On the eighth pulse, the register tries to go to the 11111111 state as the 1 from Q_6 shifts into Q_7 . This state occurs only momentarily because $Q_7 = 1$ produces a LOW at \overline{MR} that immediately resets the register back to 00000000. The sequence is then repeated on the next eight clock pulses.

The following is a list of some other register ICs that are variations on those already presented:

- 74194/ALS194/HC194. This is a four-bit *bidirectional universal shift-register* IC that can perform shift-left, shift-right, parallel in, and parallel out operations. These operations are selected by a two-bit mode-select code applied as inputs to the device. (Problem 7-71 will provide you with a chance to find out more about this versatile chip.)
- 74373/ALS373/HC373/HCT373. This is an eight-bit (octal) parallel in/parallel out register containing eight D latches with *tristate* outputs. A tristate output is a special type of logic circuit output that allows device outputs to be tied together safely. We will cover the characteristics of tristate devices such as the 74373 in the next chapter.
- 74374/ALS374/HC374/HCT374. This is an eight-bit (octal) parallel in/parallel out register containing eight edge-triggered D flip-flops with tristate outputs.

The IC registers that have been presented here are representative of the various types that are commercially available. Although there are many variations on these basic registers, most of them should now be relatively easy to understand from the manufacturers' data sheets.

We will present several register applications in the end-of-chapter problems and in the material covered in subsequent chapters.

REVIEW QUESTIONS	1. What kind of register can have a complete binary number loaded into it in one operation, and then have it shifted out one bit at a time?
	2. <i>True or false:</i> A serial in/parallel out register can have all of its bits displayed at one time.
	3. What type of register can have data entered into it only one bit at a time, but has all data bits available as outputs?
	4. In what type of register do we store data one bit at a time and have access to only one output bit at a time?
	5. How does the parallel data entry differ for the 74165 and the 74174?

6. How does the CP INH input of the 74ALS165 work?

7-20 SHIFT-REGISTER COUNTERS

In Section 5-18, we saw how to connect FFs in a shift-register arrangement to transfer data left to right, or vice versa, one bit at a time (serially). Shift-register counters use *feedback*, which means that the output of the last FF in the register is connected back to the first FF in some way.

Ring Counter

The simplest shift-register counter is essentially a **circulating shift register** connected so that the last FF shifts its value into the first FF. This arrangement is shown in Figure 7-70 using D-type FFs (J-K flip-flops can also be used). The FFs are connected so that information shifts from left to right and back around from Q_0 to Q_3 . In most instances, only a single 1 is in the register, and it is made to circulate around the register as long as clock pulses are applied. For this reason, it is called a **ring counter**.

The waveforms, sequence table, and state diagram in Figure 7-70 show the various states of the FFs as pulses are applied, assuming a starting state of $Q_3 = 1$ and $Q_2 = Q_1 = Q_0 = 0$. After the first pulse, the 1 has shifted from Q_3 to Q_2 so that the counter is in the 0100 state. The second pulse produces the 0010 state, and the third pulse produces the 0001 state. On the *fourth* clock pulse, the 1 from Q_0 is transferred to Q_3 , resulting in the 1000 state, which is, of course, the initial state. Subsequent pulses cause the sequence to repeat.

This counter functions as a MOD-4 counter because it has *four* distinct states before the sequence repeats. Although this circuit does not progress through the normal binary counting sequence, it is still a counter because each count corresponds to a unique set of FF states. Note that each FF output waveform has a frequency equal to one-fourth of the clock frequency because this is a MOD-4 ring counter.

Ring counters can be constructed for any desired MOD number; a MOD-N ring counter uses N flip-flops connected in the arrangement of Figure 7-70.

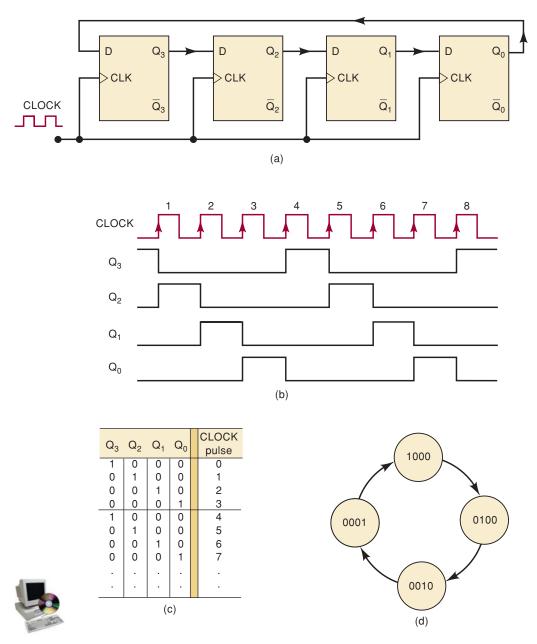


FIGURE 7-70 (a) Four-bit ring counter; (b) waveforms; (c) sequence table; (d) state diagram.

In general, a ring counter requires more FFs than a binary counter for the same MOD number; for example, a MOD-8 ring counter requires eight FFs, while a MOD-8 binary counter requires only three.

Despite the fact that it is less efficient in the use of FFs, a ring counter is still useful because it can be decoded without the use of decoding gates. The decoding signal for each state is obtained at the output of its corresponding FF. Compare the FF waveforms of the ring counter with the decoding waveforms in Figure 7-20. In some cases, a ring counter might be a better choice than a binary counter with its associated decoding gates. This is especially true in applications where the counter is being used to control the sequencing of operations in a system.

Starting a Ring Counter

To operate properly, a ring counter must start off with only one FF in the 1 state and all the others in the 0 state. Because the starting states of the FFs will be unpredictable on power-up, the counter must be preset to the required starting state before clock pulses are applied. One way to do this is to apply a momentary pulse to the asynchronous \overline{PRE} input of one of the FFs (e.g., Q_3 in Figure 7-70) and to the \overline{CLR} input of all other FFs. Another method is shown in Figure 7-71. On power-up, the capacitor will charge up relatively slowly toward $+V_{CC}$. The output of Schmitt-trigger INVERTER 1 will stay HIGH, and the output of INVERTER 2 will remain LOW until the capacitor voltage exceeds the positive-going threshold voltage (V_{T+}) of the INVERTER 1 input of Q_2 , Q_1 , and Q_0 in the LOW state long enough during power-up to ensure that the counter starts at 1000.

Johnson Counter

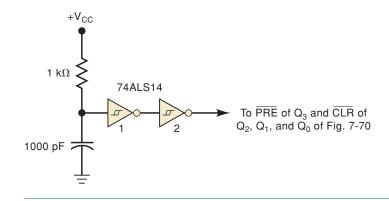
The basic ring counter can be modified slightly to produce another type of shift-register counter, which will have somewhat different properties. The **Johnson** or **twisted-ring counter** is constructed exactly like a normal ring counter except that the *inverted* output of the last FF is connected to the input of the first FF. A three-bit Johnson counter is shown in Figure 7-72. Note that the \overline{Q}_0 output is connected back to the D input of Q_2 , which means that the *inverse* of the level stored in Q_0 will be transferred to Q_2 on the clock pulse.

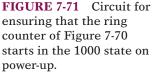
The Johnson-counter operation is easy to analyze if we realize that on each positive clock-pulse transition, the level at Q_2 shifts into Q_1 , the level at Q_1 shifts into Q_0 , and the *inverse* of the level at Q_0 shifts into Q_2 . Using these ideas and assuming that all FFs are initially 0, the waveforms, sequence table, and state diagram of Figure 7-72 can be generated.

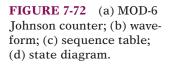
Examination of the waveforms and sequence table reveals the following important points:

- 1. This counter has six distinct states—000, 100, 110, 111, 011, and 001 before it repeats the sequence. Thus, it is a MOD-6 Johnson counter. Note that it does not count in a normal binary sequence.
- 2. The waveform of each FF is a square wave (50 percent duty cycle) at onesixth the frequency of the clock. In addition, the FF waveforms are shifted by one clock period with respect to each other.

The MOD number of a Johnson counter will always be equal to *twice* the number of FFs. For example, if we connect five FFs in the arrangement of









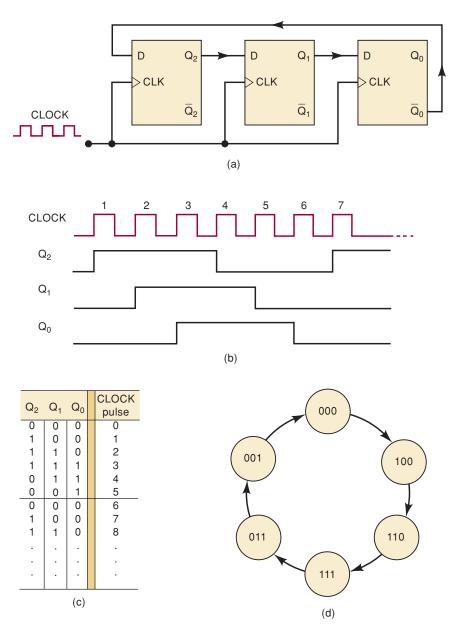
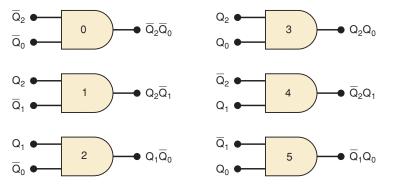


Figure 7-72, the result is a MOD-10 Johnson counter, where each FF output waveform is a square wave at one-tenth the clock frequency. Thus, it is possible to construct a MOD-N counter (where N is an even number) by connecting N/2 flip-flops in a Johnson-counter arrangement.

Decoding a Johnson Counter

For a given MOD number, a Johnson counter requires only half the number of FFs that a ring counter requires. However, a Johnson counter requires decoding gates, whereas a ring counter does not. As in the binary counter, the Johnson counter uses one logic gate to decode for each count, but each gate requires only two inputs, regardless of the number of FFs in the counter. Figure 7-73 shows the decoding gates for the six states of the Johnson counter of Figure 7-72.



			_	
Q ₂	Q_1	Q_0		Active gate
0	0	0		0
1	0	0		1
1	1	0		2
1	1	1		3
0	1	1		4
0	0	1		5
			_	

FIGURE 7-73 Decoding logic for a MOD-6 Johnson counter.

Notice that each decoding gate has only two inputs, even though there are three FFs in the counter, because for each count, two of the three FFs are in a unique combination of states. For example, the combination $Q_2 = Q_0 = 0$ occurs only once in the counting sequence, at the count of 0. Thus, AND gate 0, with inputs \overline{Q}_2 and \overline{Q}_0 , can be used to decode for this count. This same characteristic is shared by all of the other states in the sequence, as the reader can verify. In fact, for *any size* Johnson counter, the decoding gates will have only two inputs.

Johnson counters represent a middle ground between ring counters and binary counters. A Johnson counter requires fewer FFs than a ring counter but generally more than a binary counter; it has more decoding circuitry than a ring counter but less than a binary counter. Thus, it sometimes represents a logical choice for certain applications.

IC Shift-Register Counters

Very few ring counters or Johnson counters are available as ICs because it is relatively simple to take a shift-register IC and to wire it as either a ring counter or a Johnson counter. Some of the CMOS Johnson-counter ICs (74HC4017, 74HC4022) include the complete decoding circuitry on the same chip as the counter.

REVIEW QUESTIONS	1. Which shift-register counter requires the most FFs for a given MOD number?
	2. Which shift-register counter requires the most decoding circuitry?
	3. How can a ring counter be converted to a Johnson counter?
	4. True or false:
	(a) The outputs of a ring counter are always square waves.
	(b) The decoding circuitry for a Johnson counter is simpler than for a binary counter.
	(c) Ring and Johnson counters are synchronous counters.
	5. How many FFs are needed in a MOD-16 ring counter? How many are needed in a MOD-16 Johnson counter?

7-21 TROUBLESHOOTING

Flip-flops, counters, and registers are the major components in **sequential logic systems**. A sequential logic system, because of its storage devices, has the characteristic that its outputs and sequence of operations depend on both the present inputs and the inputs that occurred earlier. Even though sequential logic systems are generally more complex than combinational logic systems, the essential procedures for troubleshooting apply equally well to both types of systems. Sequential systems suffer from the same types of failures (open circuits, shorts, internal IC faults, and the like) as do combinational systems.

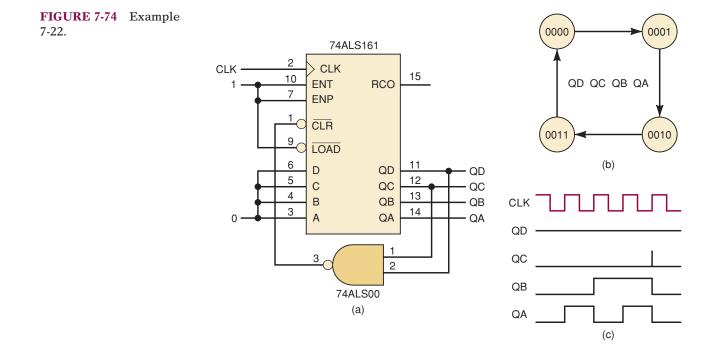
Many of the same steps used to isolate faults in a combinational system can be applied to sequential systems. One of the most effective troubleshooting techniques begins with the troubleshooter observing the system operation and, by analytical reasoning, determining the possible causes of the system malfunction. Then he or she uses available test instruments to isolate the exact fault. The following examples will show the kinds of analytical reasoning that should be the initial step in troubleshooting sequential systems. After studying these examples, you should be ready to tackle the troubleshooting problems at the end of the chapter.

EXAMPLE 7-22

Figure 7-74(a) shows a 74ALS161 wired as a MOD-12 counter, but it produces the count sequence given in Figure 7-74(b). Determine the cause of the incorrect circuit behavior.

Solution

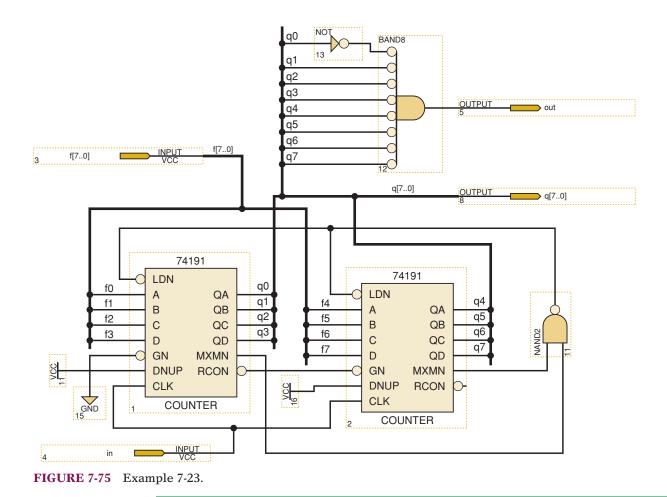
Outputs QB and QA seem to be operating correctly but QC and QD stay LOW. Our first choice for the fault is that QC is shorted to ground, but an ohmmeter check does not confirm this. The 74ALS161 might have an internal fault that prevents it from counting above 0011. We try removing the



7400 NAND chip from its socket and shorting the CLR pin to a HIGH. The counter now counts a regular MOD-16 sequence, so at least the counter's outputs seem to be ok. Next we decide to look at the CLR pin with the NAND reconnected. Using a logic probe with its "pulse capture" turned on shows us that the CLR pin is receiving pulses. Connecting a scope to the outputs, we see that the counter produces the waveforms shown in Figure 7-74(c). A glitch is observed on QC when the counter should be going to state 0100. That indicates that 0100 is a transient state when the transient state should actually be 1100. The QD connection to the NAND gate is now suspected, so we use the logic probe to check pin 2. There is no logic signal at all indicated on pin 2, which now leads us to the conclusion that the fault is an open between the QD output and pin 2 on the NAND. The NAND input is floating HIGH, causing the circuit to detect state 0100 instead of 1100 as it should be doing.

EXAMPLE 7-23

A technician receives a "trouble ticket" for a circuit board that says the variable frequency divider operates "sometimes." Sounds like a dreaded intermittent fault problem—often the hardest problems to find! His first thought is to send it back with the note "Use only when operating correctly!" but he decides to investigate further since he feels up to a good challenge today. The schematic for the circuit block is shown in Figure 7-75. The desired



divide-by factor is applied to input f[7..0] in binary. The eight-bit counter counts down from this number until it reaches zero and then asynchronously loads in f[] again, making zero a transient state. The resulting modulus will be equal to the value on f[]. The output frequency signal is obtained by decoding state 00000001, making the frequency of *out* equal to the frequency of *in* divided by the binary value f[]. In the application, the frequency of *in* is 100 kHz. Change f[] and a new frequency will be output.

Solution

The technician decides that he needs to obtain some test results to look at. He picks some easy divide-by factors to apply to f and records the results listed in Table 7-9.

TABLE 7-9	f[] (decimal)	f[] (binary)	Measured fout	OK?
	255	11111111	398.4 Hz	
	240	11110000	416.7 Hz	1
	200	11001000	500.0 Hz	\checkmark
	100	01100100	1041.7 Hz	
	50	00110010	2000.0 Hz	\checkmark
	25	00011001	4000.0 Hz	\checkmark
	15	00001111	9090.9 Hz	

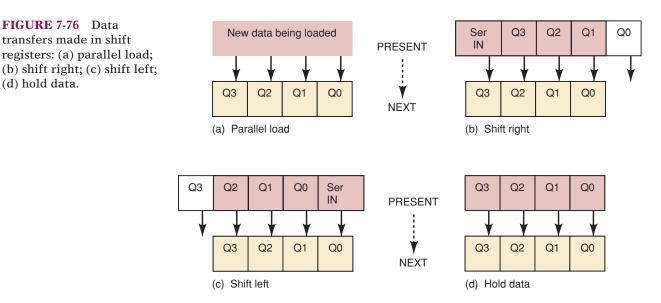
He observes that the circuit produces correct results for some test cases but incorrect results for others. The problem does not seem to be intermittent after all. Instead, it appears to be dependent on the value for f. The technician decides to calculate the relationship between input and output frequencies for the three tests that failed and obtains the following:

100 kHz/398.4 Hz = 251 100 kHz/1041.7 Hz = 96 100 kHz/9090.9 Hz = 11

Each failure seems to be a divide-by factor that is four less than the value that was actually applied to the input. After looking again at the binary representation for f, he notes that every failure occurred when f2 = 1. The weight for that bit, of course, is four. Eureka! That bit doesn't seem to be getting in—time for a logic-probe test on the f2 pin. Sure enough, the logic probe indicates the pin is LOW regardless of the value for f2.

7-22 HDL REGISTERS

The various options of serial and parallel data transfer within registers were described thoroughly in Sections 7-15 through 7-19, and some examples of ICs that perform these operations have also been described. The beauty of using HDL to describe a register is in the fact that a circuit can be given any of these options and as many bits as are needed by simply changing a few words.

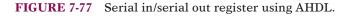


HDL techniques use bit arrays to describe a register's data and to transfer that data in a parallel or serial format. To understand how data are shifted in HDL, consider the diagrams in Figure 7-76, which shows four flipflops performing transfer operations of parallel load, shift right, shift left, and hold data. For all of these diagrams, the bits are transferred synchronously, which means that they all move simultaneously on a single clock edge. In Figure 7-76(a), the data that is to be parallel loaded into the register is presented to the D inputs, and on the next clock pulse, it will be transferred to the q outputs. Shifting data right means that each bit is transferred to the bit location to its immediate right, while a new bit is transferred in on the left end and the last bit on the right end is lost. This situation is depicted in Figure 7-76(b). Notice that the data set that we want in the NEXT state is made up of the new serial input and three of the four bits in the PRESENT state array. This data simply needs to move over and overwrite the four data bits of the register. The same operation occurs in Figure 7-76(c), but it is moving data to the left. The key to shifting the contents of the register to the right or left is to group the appropriate three PRESENT state data bits in correct order with the serial input bit so that these four bits can be loaded in parallel into the register. Concatenation (grouping together in a specific sequence) of the desired set of data bits can be used to describe the necessary data movement for serial shifting in either direction. The last possibility is called the hold data mode and is shown in Figure 7-76(d). It may seem unnecessary because registers (flip-flops) hold data by their very nature. We must consider, however, what must be done to a register in order to hold its value as it is clocked. The Q outputs must be tied back to the D inputs for each flip-flop so that the old data is reloaded on each clock. Let's look at some example HDL shift register circuits.

AHDL SISO REGISTER

A four-bit serial in/serial out (SISO) register in AHDL is listed in Figure 7-77. An array of four D flip-flops is instantiated in line 7 and the serial output is obtained from the last FF q0 (line 10). If the *shift* control is HIGH, *serial_in* will be shifted into the register and the other bits will move to the right (lines 11-15). Concatenating *serial_in* and FF output bits q3, q2, and q1

```
SUBDESIGN
 1
                  fig7_77
 2
 3
            clk, shift, serial in
                                              : INPUT;
 4
            serial out
                                              :OUTPUT;
 5
 6
      VARIABLE
 7
            q[3..0]
                                              :DFF;
 8
      BEGIN
 9
            q[].clk = clk;
10
            serial_out = q0.q;
                                                            -- output last register bit
            IF (shift == VCC)
                                THEN
11
12
                   q[3..0].d = (serial_in, q[3..1].q);
                                                           -- concatenates for shift
            ELSE
13
                   q[3..0].d = (q[3..0].q);
                                                            -- hold data
14
15
            END IF;
16
      END;
```



together in that order creates the proper shift-right data input bit pattern (line 12). If the *shift* control is LOW, the register will hold the current data (line 14). The simulation results are shown in Figure 7-78.

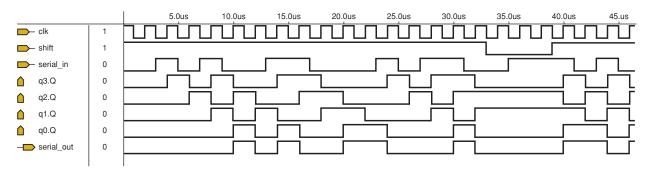


FIGURE 7-78 SISO register simulation.

VHDL SISO REGISTER

A four-bit serial in/serial out (SISO) register in VHDL is listed in Figure 7-79. A register is created with the declaration of the variable q on line 8 and the serial output is obtained from the register's last bit or q(0) (line 10). If the *shift* control is HIGH, *serial_in* will be shifted into the register and the other bits will move to the right (lines 12–14). Concatenating *serial_in* and register bits q(3), q(2), and q(1) together in that order creates the proper shift-right data input bit pattern (line 13). If the shift control is LOW, VHDL will assume that the variable stays the same and will therefore hold the current data. Simulation results are shown in Figure 7-78.

```
V H D L
```

```
ENTITY fig7 79 IS
 1
 2
     PORT (
                  clk, shift, serial in
                                                  :IN BIT;
 3
                  serial out
                                                  :OUT BIT
                                                               );
 4
     END fiq 7-79;
 5
     ARCHITECTURE vhdl OF fig 7-79 IS
 6
     BEGIN
 7
     PROCESS (clk)
 8
           VARIABLE q
                                            :BIT VECTOR (3 DOWNTO 0);
 9
           BEGIN
10
            serial_out <= q(0);</pre>
                                                         -- output last register bit
            IF (clk'EVENT AND clk = '1')
11
                                            THEN
                  IF (shift = '1') THEN
12
13
                        q := (serial in & q(3 DOWNTO 1)); -- concatenate for shift
                  END IF;
14
                                                         -- otherwise, hold data
15
            END IF;
16
     END PROCESS;
17
     END vhdl;
```

FIGURE 7-79 Serial in/serial out register using VHDL.

AHDL PISO REGISTER

A four-bit parallel in/serial out (PISO) register in AHDL is listed in Figure 7-80. The register named *q* is created on line 8 using four D FFs, and the serial output from *q0* is described on line 11. The register has separate parallel *load* and serial *shift* controls. The register's functions are defined in lines 12–15. If *load* is HIGH, the external input *data[3..0]* will be synchronously loaded. *Load* has priority and must be LOW to serial-shift the register's contents on each PGT of *clk* when *shift* is HIGH. The pattern for shifting data right is created by concatenation on line 13. Note that a constant LOW will be the serial data input for a shift operation. If neither *load* nor *shift* is HIGH, the register will hold the current data value (line 14). Simulation results are shown in Figure 7-81.

```
1
     SUBDESIGN fig7 80
 2
      (
 3
            clk, shift, load
                                    : INPUT;
 4
            data[3..0]
                                    : INPUT;
 5
            serial out
                                    : OUTPUT;
 6
      )
 7
      VARIABLE
 8
                                    :DFF;
            q[3..0]
 9
      BEGIN
10
            q[].clk = clk;
            serial_out = q0.q;
11
                                                        -- output last register bit
12
            IF (load == VCC) THEN q[3..0].d = data[3..0]; -- parallel load
            ELSIF (shift == VCC) THEN q[3..0].d = (GND, q[3..1].q); -- shift
13
14
            ELSE q[3..0].d = q[3..0].q;
                                                                          -- hold
15
            END IF;
16
      END;
```

FIGURE 7-80 Parallel in/serial out register using AHDL.

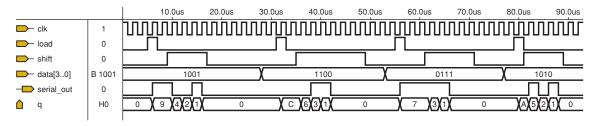


FIGURE 7-81 PISO register simulation.

VHDL PISO REGISTER

A four-bit parallel in/serial out (PISO) register in VHDL is listed in Figure 7-82. The register is created with the variable declaration for q on line 11, and the serial output from q(0) is described on line 13. The register has separate parallel *load* and serial *shift* controls. The register's functions are defined in lines 14–18. If *load* is HIGH, the external input *data* will be synchronously loaded. *Load* has priority and must be LOW to serial-shift the register's contents on each PGT of *clk* when *shift* is HIGH. The pattern for shifting data right is created by concatenation on line 16. Note that a constant LOW will be the serial data input for a shift operation. If neither *load* nor *shift* is HIGH, the register will hold the current data value by VHDL's implied operation. Simulation results are shown in Figure 7-81.

```
1
      ENTITY fig7 82 IS
 2
      PORT (
 3
            clk, shift, load
                                      :IN BIT;
 4
            data
                                      :IN BIT VECTOR (3 DOWNTO 0);
 5
            serial out
                                      :OUT BIT
 6
      );
 7
      END fig 7-82;
 8
      ARCHITECTURE vhdl OF fig 7-82 IS
 9
      BEGIN
10
      PROCESS (clk)
11
            VARIABLE q
                                :BIT VECTOR (3 DOWNTO 0);
12
            BEGIN
13
            serial out <= q(0);</pre>
                                                         -- output last register bit
14
            IF (clk'EVENT AND clk = '1') THEN
                   IF (load = '1') THEN q := data;
15
                                                         -- parallel load
                   ELSIF (shift = '1') THEN q := ('0' & q(3 DOWNTO 1));
16
                                                                               -- shift
17
                   END IF;
                                                          -- otherwise, hold
18
            END IF;
19
      END PROCESS;
2.0
      END vhdl;
```

FIGURE 7-82 Parallel in/serial out register using VHDL.

EXAMPLE 7-24

Suppose we want to design a universal four-bit shift register, using HDL, that has four synchronous modes of operation: Hold Data, Shift Left, Shift Right, and Parallel Load. Two input bits will select the operation that is to be performed on each rising edge of the clock. To implement a shift register, we can use structural code to describe a string of flip-flops. Making the shift register versatile by allowing it to shift right or left or to parallel load would make this file quite long and thus hard to read and understand using structural methods. A much better approach is to use the more abstract and intuitive methods available in HDL to describe the circuit concisely. To do this, we must develop a strategy that will create the shifting action. The concept is very similar to the one presented in Example 7-16, where a D flip-flop register chip (74174) was wired to form a shift register. Rather than thinking of the shift register as a serial string of flip-flops, we consider it as a parallel register whose contents are being transferred in parallel to a set of bits that is offset by one bit position. Figure 7-76 demonstrates the concept of each transfer needed in this design.

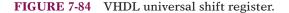
Solution

A very reasonable first step is to define a two-bit input named *mode* with which we can specify mode 0, 1, 2, or 3. The next challenge is deciding how to choose among the four operations using HDL. Several methods can work here. The CASE structure was chosen because it allows us to choose a different set of HDL statements for each and every possible mode value. There is no priority associated with checking for the existing mode settings or overlapping ranges of mode numbers, so we do not need the advantages of the IF/ELSE construct. The HDL solutions are given in Figures 7-83 and 7-84. The same inputs and outputs are defined in each approach: a clock, four bits of parallel load data, a single bit for the serial input to the register, two bits for the mode selection, and four output bits.

```
1
      SUBDESIGN fig7_83
 2
      (
 3
        clock
                   : INPUT;
 4
        din[3..0] :INPUT; -- parallel data in
 5
        ser in
                 :INPUT; -- serial data in from Left or Right
 6
        mode [1..0] :INPUT;
                             -- MODE Select: 0=hold, 1=right, 2=left, 3=load
 7
        q[3..0] :OUTPUT;
 8
      )
 9
     VARIABLE
10
        ff[3..0] :DFF;
                             -- define register set
11
     BEGIN
12
         ff[].clk = clock;
                            -- synchronous clock
13
        CASE mode[] IS
           WHEN 0 => ff[].d
14
                                 = ff[].q;
                                                   -- hold shift
           WHEN 1 => ff[2..0].d = ff[3..1].q);
15
                                                   -- shift right
16
                       ff[3].d
                                 = ser_in;
                                                   -- new data from left
17
           WHEN 2 => ff[3..1].d = ff[2..0].q;
                                                   -- shift left
18
                      ff[0].d
                                 = ser in;
                                                   -- new data bit from right
           WHEN 3 => ff[].d
19
                                  = din[];
                                                   -- parallel load
20
         END CASE;
21
        q[] = ff[];
                                   -- update outputs
22
      END;
```

FIGURE 7-83 AHDL universal shift register.

```
1
      ENTITY fig7_84 IS
 2
      PORT (
 3
         clock
                         :IN BIT;
 4
         din
                          :IN BIT_VECTOR (3 DOWNTO 0); -- parallel data in
         ser_in
                        :IN BIT; -- serial data in L or R
:IN INTEGER RANGE 0 TO 3; -- 0=hold 1=rt 2=lt 3=load
 5
 6
         mode
 7
                         :OUT BIT_VECTOR (3 DOWNTO 0));
         q
 8
      END fig7_84;
 9
      ARCHITECTURE a OF fig7_84 IS
10
      BEGIN
11
         PROCESS (clock)
                                                           -- respond to clock
12
         VARIABLE ff :BIT_VECTOR (3 DOWNTO 0);
13
         BEGIN
14
             IF (clock'EVENT AND clock = '1') THEN
15
                CASE mode IS
16
                   WHEN 0 => ff := ff;
                                                                       -- hold data
17
                   WHEN 1 => ff(2 DOWNTO 0) := ff(3 DOWNTO 1); -- shift right
18
                               ff(3) := ser_in;
19
                   WHEN 2 \implies \text{ff}(3 \text{ DOWNTO } 1) := \text{ff}(2 \text{ DOWNTO } 0);
                                                                      -- shift left
20
                               ff(0) := ser_in;
21
                   WHEN 3 \Rightarrow ff := din;
                                                                       -- parallel load
2.2
                END CASE;
23
            END IF;
24
         q \leq ff;
                                                           -- update outputs
25
         END PROCESS;
26
      END a;
```



AHDL SOLUTION

The AHDL solution of Figure 7-83 uses a register of D flip-flops declared by the name ff on line 10, representing the current state of the register. Because the flip-flops all need to be clocked at the same time (synchronously), all the clock inputs are assigned to *clock* on line 12. The CASE construct selects a different transfer configuration for each value of the *mode* inputs. Mode 0 (hold data) uses a direct parallel transfer from the current state to the same bit positions on the D inputs to produce the identical NEXT state. Mode 1 (shift right), which is described on lines 15 and 16, transfers bits 3, 2, and 1 to bit positions 2, 1, and 0, respectively, and loads bit 3 from the serial input. Mode 2 (shift left) performs a similar operation in the opposite direction (see lines 17 and 18). Mode 3 (parallel load) transfers the value on the parallel data inputs to become the NEXT state of the register. The code creates the circuitry that chooses one of these logical operations on the actual register, and the proper data is transferred to the output pins on the next clock. This code can be shortened by combining lines 15 and 16 into a single statement that concatenates the ser_in with the three data bits and groups them as a set of four bits. The statement that can replace lines 15 and 16 is:

WHEN 1 => ff[].d = (ser_in, ff[3..1].q);

Lines 17 and 18 can also be replaced by:

WHEN 2 => ff[].d = (ff[2..0].q,ser_in);

VHDL SOLUTION

The VHDL solution of Figure 7-84 defines an internal variable by the name ff on line 12, representing the current state of the register. Because all the transfer operations need to take place in response to a rising clock edge, a PROCESS is used, with *clock* specified in the sensitivity list. The CASE construct selects a different transfer configuration for each value of the mode inputs. Mode 0 (hold data) uses a direct parallel transfer from the current state to the same bit positions to produce the identical NEXT state. Mode 1 (shift right) transfers bits 3, 2, and 1 to bit positions 2, 1, and 0, respectively (line 17), and loads bit 3 from the serial input (line 18). Mode 2 (shift left) performs a similar operation in the opposite direction. Mode 3 (parallel load) transfers the value on the parallel data inputs to the NEXT state of the register. After choosing one of these operations on the actual register, the data is transferred to the output pins on line 24. This code can be shortened by combining lines 17 and 18 into a single statement that concatenates the ser in with the three data bits and groups them as a set of four bits. The statement that can replace lines 17 and 18 is:

WHEN 1 => ff := ser_in & ff(3 DOWNTO 1);

Lines 19 and 20 can also be replaced by:

WHEN 2 => ff := ff(2 DOWNTO 0) & ser_in;

REVIEW QUESTIONS

- 1. Write a HDL expression that can implement a shift left of an eight-bit array *reg*[7..0] with serial input *dat*.
- 2. Why is it necessary to reload the current data during the hold data mode on a shift register?

7-23 HDL RING COUNTERS

In Section 7-20 we used a shift register to make a counter that circulates a single active logic level through all of its flip-flops. This was referred to as a ring counter. One characteristic of ring counters is that the modulus is equal to the number of flip-flops in the register and thus there are always many unused and invalid states. We have already discussed ways of describing counters using the CASE construct to specify PRESENT state and NEXT state transitions. In those examples, we took care of invalid states by including them under "others." This method also works for ring counters. In this section, however, we look at a more intuitive way to describe shift counters.

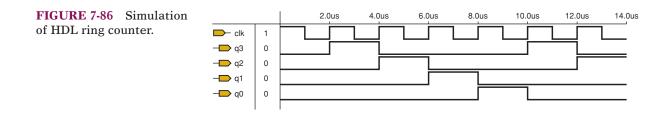
These methods use the same techniques described in Section 7-22 in order to make the register shift one position on each clock. The main feature of this code is the method of completing the "ring" by driving the *ser_in* line of the shift register. With a little planning, we should also be able to ensure that the counter eventually reaches the desired sequence, no matter what state it is in initially. For this example, we re-create the operation of the ring counter whose state diagram is shown in Figure 7-70(d). In order to make this counter self-start without using asynchronous inputs, we control the *ser_in* line of the shift register using an IF/ELSE construct. Any time we detect that the upper three bits are all LOW, we assume the lowest order bit is HIGH, and on the next clock, we want to shift in a HIGH to *ser_in*. For all other states (valid and invalid), we shift in a LOW. Regardless of the state to which the counter is initialized, it eventually fills with zeros; at which time, our logic shifts in a HIGH to start the ring sequence.

AHDL RING COUNTER

The AHDL code shown in Figure 7-85 should look familiar by now. Lines 11 and 12 control the serial input using the strategy we just described. Notice the use of the double equals (==) operator on line 11. This operator evaluates whether the expressions on each side are equal or not. Remember, the single equals (=) operator assigns (i.e., connects) one object to another. Line 14 implements the shift right action that we described in the previous section. Simulation results are shown in Figure 7-86.

FIGURE 7-85 AHDL four-bit ring counter.

-	SUBDESIGN fig7_85
2	(
3	clk :INPUT;
Ŀ	q[30] :OUTPUT;
5)
5	VARIABLE
7	ff[30] :DFF;
3	ser_in :NODE;
)	BEGIN
)	<pre>ff[].clk = clk;</pre>
-	<pre>IF ff[31].q == B"000" THEN ser_in = VCC; self start</pre>
2	ELSE ser_in = GND;
3	END IF;
Ŀ	ff[30].d = (ser_in, ff[31].q); shift right
5	q[] = ff[];
5	END;



HDL

VHDL RING COUNTER

The VHDL code shown in Figure 7-87 should look familiar by now. Lines 12 and 13 control the serial input using the strategy we just described. Line 16 implements the shift right action that we described in the previous section. Simulation results are shown in Figure 7-86.

```
1
      ENTITY fig7_87 IS
2
      PORT (
                clk
                              :IN BIT;
3
                              :OUT BIT_VECTOR (3 DOWNTO 0));
                  q
 4
      END fig7_87;
 5
 6
      ARCHITECTURE vhdl OF fig7_87 IS
7
      SIGNAL ser_in
                      :BIT;
8
      BEGIN
9
      PROCESS (clk)
10
        VARIABLE ff
                             :BIT_VECTOR (3 DOWNTO 0);
11
        BEGIN
12
           IF (ff(3 DOWNTO 1) = "000") THEN ser_in <= '1';
                                                                      -- self-start
13
            ELSE ser_in <= '0';</pre>
14
           END IF;
15
            IF (clk'EVENT AND clk = '1') THEN
16
               ff(3 DOWNTO 0) := (ser_in & ff(3 DOWNTO 1));
                                                                      -- shift right
17
            END IF;
18
         q \leq ff;
19
     END PROCESS;
20
      END vhdl;
```

FIGURE 7-87 VHDL four-bit ring counter.

REVIEW QUESTIONS

- 1. What does it mean for a ring counter to self-start?
- 2. Which lines of Figure 7-85 ensure that the ring counter self-starts?
- 3. Which lines of Figure 7-87 ensure that the ring counter self-starts?

7-24 HDL ONE-SHOTS

Another important circuit that we have studied is the one-shot. We can apply the concept of a counter to implement a **digital one-shot** using HDL. Recall from Chapter 5 that one-shots were devices that produce a pulse of a predefined width every time the trigger input is activated. A nonretriggerable one-shot ignores the trigger input as long as the pulse output is still active. A *retriggerable* one-shot starts a pulse in response to a trigger and restarts the internal pulse timer every time a subsequent trigger edge occurs before the pulse is complete. The first example we investigate is a nonretriggerable, HIGH-level-triggered digital one-shot. The one-shots that we studied in Chapter 5 used a resistor and capacitor as the internal pulse timing mechanism. In order to create a one-shot using HDL techniques, we use a four-bit counter to determine the width of the pulse. The inputs are a clock signal, trigger, clear, and pulse width value. The only output is the pulse out, Q. The idea is quite simple. Whenever a trigger is detected, make the pulse go HIGH and load a down-counter with a number from the pulse width inputs. The larger this number, the longer it will take to count down to zero. The advantage of this one-shot is that the pulse width can be adjusted easily by changing the value loaded into the counter. As you read the sections below, consider the following question: "What makes this circuit nonretriggerable and what makes it leveltriggered?"

SIMPLE AHDL ONE-SHOTS

A nonretriggerable, level-sensitive, one-shot description in AHDL is shown in Figure 7-88. A register of four flip-flops is created on line 8, and it serves as the counter that counts down during the pulse. The *clock* is connected in parallel to all the flip-flops on line 10. The reset function is implemented by connecting the *reset* control line directly to the asynchronous clear input of each flip-flop on line 11. After these assignments, the first condition that is tested is the trigger. If it is activated (HIGH) at any time while the count value is 0 (i.e., the previous pulse is done), then the delay value is loaded into the counter is down to zero. If it is, then the counter should not roll over but rather stay at zero. If the count is not at zero, then it must be counting, so line 15 sets up the flip-flops to decrement on the next clock. Finally, line 17 generates the output pulse. This Boolean expression can be thought of as follows: "Make the pulse (Q) HIGH when the *count* is anything other than zero."

FIGURE 7-88 AHDL nonretriggerable one-shot.

1

2 3

4

5

6 7

8

9

10

11

12

13

14

15 16

17

18

```
SUBDESIGN fig7_88
   clock, trigger, reset
                           : INPUT;
   delay[3..0]
                           : INPUT;
                           : OUTPUT;
   q
VARIABLE
  count[3..0]
                  : DFF;
BEGIN
   count[].clk = clock;
  count[].clrn = reset;
   IF trigger & count[].q == b"0000" THEN
         count[].d = delay[];
   ELSIF count[].g == B"0000" THEN count[].d = B"0000";
   ELSE count[].d = count[].q - 1;
   END IF;
   q = count[].q != B"0000"; -- make output pulse
END;
```

SIMPLE VHDL ONE-SHOTS

A nonretriggerable, level-sensitive, one-shot description in VHDL is shown in Figure 7-89. The inputs and outputs are shown on lines 3–5, as previously described. In the architecture description, a PROCESS is used (line 11) to respond to either of two inputs: the clock, or the reset. Within this PROCESS, a variable is used to represent the value on the counter. The input that should have overriding precedence is the *reset* signal. This is tested first (line 14) and if it is active, the *count* is cleared immediately. If the *reset* is not active, line 15 is evaluated and looks for a rising edge on the *clock*. Line 16 checks for the trigger. If it is activated at any time while the count value is 0 (i.e., the previous pulse is done), then the width value is loaded into the counter. On line 18, it tests to see if the pulse is done by checking to see if the counter is down to zero. If it is, then the counter should not roll over but rather stay at zero. If the

```
1
      ENTITY fig7_89 IS
 2
      PORT (
 3
            clock, trigger, reset
                                      :IN BIT;
 4
            delay
                                       :IN INTEGER RANGE 0 TO 15;
 5
            a
                                       :OUT BIT
 6
            );
 7
      END fig 7_89;
 8
 9
      ARCHITECTURE vhdl OF fig7_89 IS
10
      BEGIN
         PROCESS (clock, reset)
11
12
         VARIABLE count : INTEGER RANGE 0 TO 15;
13
         BEGIN
            IF reset = '0' THEN count := 0;
14
            ELSIF (clock'EVENT AND clock = '1' ) THEN
15
               IF trigger = '1' AND count = 0 THEN
16
17
                  count := delay;
                                                          -- load counter
18
               ELSIF count = 0 THEN count := 0;
19
               ELSE count := count - 1;
20
               END IF;
21
            END IF;
22
            IF count /= 0 THEN q <= '1';
            ELSE q \leq 0';
23
24
            END IF;
25
         END PROCESS;
26
      END vhdl;
```

FIGURE 7-89 VHDL nonretriggerable one-shot.

count is not at zero, then it must be counting, so line 19 sets up the flip-flops to decrement on the next clock. Finally, lines 22 and 23 generate the output pulse. This Boolean expression can be thought of as follows: "Make the pulse (q) HIGH when the count is anything other than zero."

Now that we have reviewed the code that describes this one-shot, let's evaluate its performance. Converting a traditionally analog circuit to digital usually offers some advantages and some disadvantages. On a standard one-shot chip, the output pulse starts immediately after the trigger. For the digital one-shot described here, the output pulse starts on the next clock edge and lasts as long as the counter is greater than zero. This situation is shown in Figure 7-90 within the first ms of the simulation. Notice that the trigger goes high almost 0.5 ms before the q out responds. If another trigger event occurs while it is counting down (like the one just before 3 ms), it is ignored. This is the nonretriggerable characteristic.

Another point to make for this digital one-shot is that the trigger pulse must be long enough to be seen as a HIGH on the rising clock edge. At about the 4.5-ms mark, a pulse occurs on the trigger input but goes LOW before the rising edge of the clock. This circuit does *not* respond to this input event. At just past 5 ms, the trigger goes HIGH and stays there. The pulse lasts exactly 6 ms, but because the trigger input remains HIGH, it responds with another output pulse one clock later. The reason for this situation is that this circuit is level-triggered rather than edge-triggered, like most of the conventional one-shot ICs.

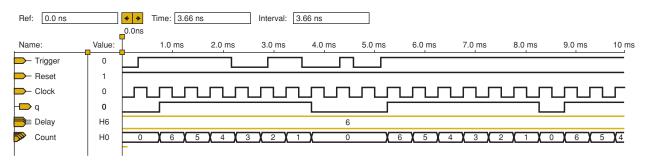


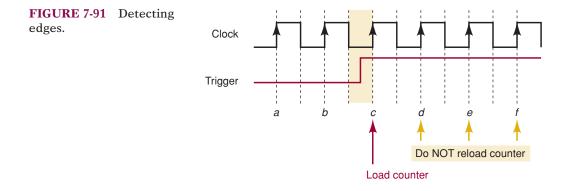
FIGURE 7-90 Simulation of the nonretriggerable one-shots.

Retriggerable, Edge-Triggered One-Shots in HDL

Many applications of one-shots require the circuit to respond to an edge rather than a level. How can HDL code be used to make the circuit respond once to each positive transition on its trigger input? The technique described here is called *edge-trapping* and has been a very useful tool in programming microcontrollers for years. As we will see, it is equally useful for describing edge-triggering for a digital circuit using HDL. This section illustrates an example of a retriggerable one-shot while also demonstrating edge-trapping, which can be useful in many other situations.

The general operation of this retriggerable one-shot requires that it responds to a rising edge of the trigger input. As soon as the edge is detected, it should start timing the pulse. In the digital one-shot, this means that it loads the counter as soon as possible after the trigger edge and starts counting down toward zero. If another trigger event (rising edge) occurs before the pulse is terminated, the counter is immediately reloaded, and the pulse timing starts again from the beginning, thus sustaining the pulse. Activating the clear at any point should force the counter to zero and terminate the pulse. The minimum output pulse width is simply the number applied to the width input multiplied by the clock period.

The strategy behind edge-trapping for a one-shot is demonstrated in Figure 7-91. On each active clock edge are two important pieces of information that are needed. The first is the state of the *trigger* input *now* and the second is the state of the *trigger* input when the last active clock edge occurred. Start with point a on the diagram of Figure 7-91 and determine these two values, then move to point b, and so on. By completing this task, you should have concluded that, at point c, a unique result has been obtained. The *trigger* is HIGH now but it was LOW on the last active clock edge. This is the point where we have detected the *trigger* edge event.



In order to know what the *trigger* was on the last active clock edge, the system must remember the last value that *the trigger had at that point*. This is done by storing the value of the trigger bit in a flip-flop. Recall that we discussed a similar concept in Chapter 5 when we talked about using a flip-flop to detect a sequence. The code for a one-shot is written so that the counter is loaded only after a rising edge is detected on the *trigger* input.

AHDL RETRIGGERABLE, EDGE-TRIGGERED ONE-SHOT

The first five lines of Figure 7-92 are identical to the previous nonretriggerable example. In AHDL, the only way to remember a value obtained in the past is to store the value on a flip-flop. This section uses a flip-flop named *trig_was* (line 9) to store the value that was on the trigger on the last active clock edge. This flip-flop is simply connected so that the trigger is on its *D* input (line 14) and the clock is connected to its *clk* input (line 13). The *Q* output of *trig_was* remembers the value of the *trigger* right up to the next clock edge. At this point, we use line 16 to evaluate if a triggering edge has occurred. If *trigger* is HIGH (now), but *trigger* was LOW (last clock), it is time to load the counter (line 17). Line 18 ensures that, once the *count* reaches zero, it will remain at zero until a new trigger comes along. If the decisions allow line 19 to be evaluated, it means that there is a value loaded into the counter and it is not zero, so it needs to be decremented. Finally, the output pulse is made HIGH any time a value other than 0000 is still on the counter, like we saw previously.

FIGURE 7-92 AHDL retriggerable one-shot with edge trigger.

```
1
      SUBDESIGN fig7_92
 2
 3
         clock, trigger, reset : INPUT;
 4
         delay[3..0]
                                  : INPUT;
 5
                                  : OUTPUT;
         q
 6
 7
      VARIABLE
 8
                   count[3..0]
                                  : DFF;
 9
                   trig_was
                                  : DFF;
      BEGIN
10
11
         count[].clk = clock;
12
         count[].clrn = reset;
13
         trig_was.clk = clock;
14
         trig_was.d = trigger;
15
16
         IF trigger & !trig_was.g THEN
17
                count[].d = delay[];
         ELSIF count[].q == B"0000" THEN count[].d = B"0000";
18
         ELSE count[].d = count[].q - 1;
19
20
         END IF:
21
         q = count[].q != B"0000";
22
      END;
```

VHDL RETRIGGERABLE, EDGE-TRIGGERED ONE-SHOT

The ENTITY description in Figure 7-93 is exactly like the previous nonretriggerable example. In fact, the only differences between this example and the one shown in Figure 7-89 have to do with the logic of the decision process. When we want to remember a value in VHDL, it must be stored in a VARI-ABLE. Recall that we can think of a PROCESS as a description of what happens each time a signal in the sensitivity list changes state. A VARIABLE retains the last value assigned to it between the times the process is invoked. In this sense, it acts like a flip-flop. For the one-shot, we need to store a value that tells us what the trigger was on the last active clock edge. Line 11 declares a variable bit to serve this purpose. The first decision (line 13) is the overriding decision that checks and responds to the *reset* input. Notice that this is an asynchronous control because it is evaluated before the clock edge is detected on line 14. Line 14 determines that a rising clock edge has occurred, and then the main logic of this process is evaluated between lines 15 and 20.

When a clock edge occurs, one of three conditions exists:

- 1. A trigger edge has occurred and we must load the counter.
- 2. The counter is zero and we need to keep it at zero.
- 3. The counter is not zero and we need to count down by one.

```
1
     ENTITY fig7_93 IS
 2
      PORT ( clock, trigger, reset : IN BIT;
 3
           delay
                                      : IN INTEGER RANGE 0 TO 15;
                                       : OUT BIT);
 4
           α
 5
     END fig7_93;
 6
 7
     ARCHITECTURE vhdl OF fig7_93 IS
 8
     BEGIN
 9
        PROCESS (clock, reset)
        VARIABLE count : INTEGER RANGE 0 TO 15;
10
11
        VARIABLE trig_was : BIT;
12
        BEGIN
           IF reset = '0' THEN count := 0;
13
           ELSIF (clock'EVENT AND clock = '1' ) THEN
14
              IF trigger = '1' AND trig_was = '0' THEN
15
                 count := delay;
16
                                              -- load counter
                 trig_was := '1'; -- "remember" edge detected
17
              ELSIF count = 0 THEN count := 0; -- hold @ 0
18
19
              ELSE count := count - 1;
                                               -- decrement
20
              END IF;
              IF trigger = '0' THEN trig_was := '0';
21
22
              END IF;
23
           END IF;
           IF count /= 0 THEN q <= '1';
24
           ELSE q <= '0';
25
26
           END IF;
27
        END PROCESS;
28
     END vhdl;
```

FIGURE 7-93 VHDL retriggerable one-shot with edge trigger.

Recall that it is very important to consider the order in which questions are asked and assignments are made in VHDL PROCESS statements because the *sequence* affects the operation of the circuit we are describing. The code that updates the *trig_was* variable must occur after the evaluation of its previous condition. For this reason, the conditions necessary to detect a rising edge on *trigger* are evaluated on line 15. If an edge occurred, then the counter is loaded (line 16) and the variable is updated (line 17) to remember this for the next time. If a trigger edge has not occurred, the code either holds at zero (line 18) or counts down (line 19). Line 21 makes sure that, as soon as the trigger input goes LOW, the variable *trig_was* remembers this by resetting. Finally, lines 24–25 are used to create the output pulse during the time the counter is not zero.

The two improvements that were made in this one-shot over the last example are the edge-triggering and the retriggerable feature. Figure 7-94 evaluates the new performance features. Notice in the first ms of the timing diagram that a trigger edge is detected, but the response is not immediate. The output pulse goes high on the next clock edge. This is a drawback to the digital one-shot. The retriggerable feature is demonstrated at about the 2ms mark. Notice that *trigger* goes high and on the next clock edge, the *count* starts again at 5, sustaining the output pulse. Also notice that even after the q output pulse is complete and the *trigger* is still HIGH, the one-shot does not fire another pulse because it is not level-triggered but rather rising edge-triggered. At the 6-ms mark, a short trigger pulse occurs but is ignored because it does not stay HIGH until the next clock. On the other hand, an even shorter trigger pulse occurring just after the 7-ms mark does fire the one-shot because it is present during the rising clock edge. The resulting output pulse lasts exactly five clock cycles because no other triggers occur during this period.

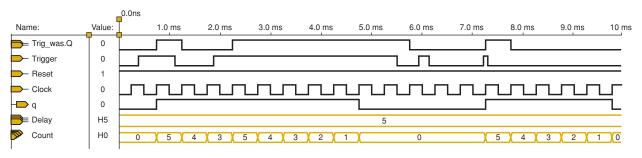


FIGURE 7-94 Simulation of the edge-triggered retriggerable one-shot.

To minimize the effects of delayed response to trigger edges and the possibility of missing trigger edges that are too short, this circuit can be improved quite simply. The clock frequency and the number of bits used to load the delay value can both be increased to provide the same range of pulse widths (with more precise control) while reducing the minimum trigger pulse width. In order to cure this problem completely, the one-shot must respond asynchronously to the trigger input. This is possible in both AHDL and VHDL, but it will always result in a pulse that fluctuates in width by up to one clock period. **REVIEW QUESTIONS**

- 1. Which control input signal holds the highest priority for each of the oneshot descriptions?
- 2. Name two factors that determine how long a pulse from a digital oneshot will last.
- 3. For the one-shots shown in this section, are the counters loaded synchronously or asynchronously?
- 4. What is the advantage of loading a counter synchronously?
- 5. What is the advantage of loading the counter asynchronously?
- 6. What two pieces of information are necessary to detect an edge?

PART 2 SUMMARY

- 1. Numerous IC registers are available and can be classified according to whether their inputs are parallel (all bits entered simultaneously), serial (one bit at a time), or both. Likewise, registers can have outputs that are parallel (all bits available simultaneously) or serial (one bit available at a time).
- 2. A sequential logic system uses FFs, counters, and registers, along with logic gates. Its outputs and sequence of operations depend on present and past inputs.
- 3. Troubleshooting a sequential logic system begins with observation of the system operation, followed by analytical reasoning to determine the possible causes of any malfunction, and finally test measurements to isolate the actual fault.
- 4. A ring counter is actually an *N*-bit shift register that recirculates a single 1 continuously, thereby acting as a MOD-*N* counter. A Johnson counter is a modified ring counter that operates as MOD-2*N* counter.
- 5. Shift registers can be implemented with HDL by writing custom descriptions of their operation.
- 6. An understanding of bit arrays/bit vectors and their notation is very important in describing shift register operations.
- 7. Shift register counters such as Johnson and ring counters can be implemented easily in HDL. Decoding and self-starting features are easy to write into the description.
- 8. Digital one-shots are implemented with a counter loaded with a delay value when the trigger input is detected and counts down to zero. During the countdown time, the output pulse is held HIGH.
- 9. With strategic placement of the hardware description statements, HDL one-shots can be made edge- or level-triggered and retriggerable or non-retriggerable. They produce an output pulse that responds synchronously or asynchronously to the trigger.

PART 2 IMPORTANT TERMS

parallel in/parallel out serial in/serial out parallel in/serial out serial in/parallel out circulating shift register

ring counter Johnson counter (twisted ring counter) sequential logic system concatenation digital one-shot

PROBLEMS

PART 1

SECTION 7-1

- **B** 7-1.*Add another flip-flop, *E*, to the counter of Figure 7-1. The clock signal is an 8-MHz square wave.
 - (a) What will be the frequency at the *E* output? What will be the duty cycle of this signal?
 - (b) Repeat (a) if the clock signal has a 20 percent duty cycle.
 - (c) What will be the frequency at the C output?
 - (d) What is the MOD number of this counter?
- **B** 7-2. Draw a binary counter that will convert a 64-kHz pulse signal into a 1-kHz square wave.
- **B** 7-3.*Assume that a five-bit binary counter starts in the 00000 state. What will be the count after 144 input pulses?
- **B** 7-4. A 10-bit ripple counter has a 256-kHz clock signal applied.
 - (a) What is the MOD number of this counter?
 - (b) What will be the frequency at the MSB output?
 - (c) What will be the duty cycle of the MSB signal?
 - (d) Assume that the counter starts at zero. What will be the count in hexadecimal after 1000 input pulses?

SECTION 7-2

- 7-5.*A four-bit ripple counter is driven by a 20-MHz clock signal. Draw the waveforms at the output of each FF if each FF has $t_{\rm pd} = 20$ ns. Determine which counter states, if any, will not occur because of the propagation delays.
- 7-6. (a) What is the maximum clock frequency that can be used with the counter of Problem 7-5?
 - (b) What would f_{max} be if the counter were expanded to six bits?

SECTIONS 7-3 AND 7-4

- **B** 7-7.*(a) Draw the circuit diagram for a MOD-32 synchronous counter.
 - (b) Determine f_{max} for this counter if each FF has $t_{\text{pd}} = 20$ ns and each gate has $t_{\text{pd}} = 10$ ns.
- **B** 7-8. (a) Draw the circuit diagram for a MOD-64 synchronous counter.
 - (b) Determine f_{max} for this counter if each FF has $t_{\text{pd}} = 20$ ns and each gate has $t_{\text{pd}} = 10$ ns.
- B 7-9.*Draw the waveforms for all the FFs in the decade counter of Figure 7-8(b) in response to a 1-kHz clock frequency. Show any glitches that might appear on any of the FF outputs. Determine the frequency at the D output.
- **B** 7-10. Repeat Problem 7-9 for the counter of Figure 7-8(a).
 - 7-11.* Change the inputs to the NAND gate of Figure 7-9 so that the counter divides the input frequency by 50.
- D 7-12. Draw a synchronous counter that will output a 10-kHz signal when a 1-MHz clock is applied.

^{*}Answers to problems marked with an asterisk can be found in the back of the text.

SECTIONS 7-5 AND 7-6

- **B** 7-13.*Draw a synchronous, MOD-32, down counter.
- **B** 7-14. Draw a synchronous, MOD-16, up/down counter. The count direction is controlled by *dir* (*dir* = 0 to count up).
- C, T 7-15.*Determine the count sequence of the up/down counter in Figure 7-11 if the INVERTER output were stuck HIGH. Assume the counter starts at 000.
 - 7-16. Complete the timing diagram in Figure 7-95 for the presettable counter in Figure 7-12. Note that the initial condition for the counter is given in the timing diagram.

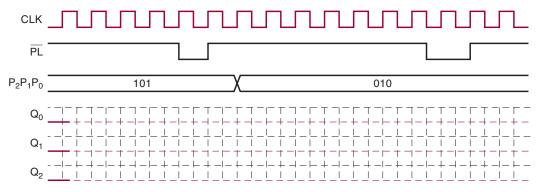
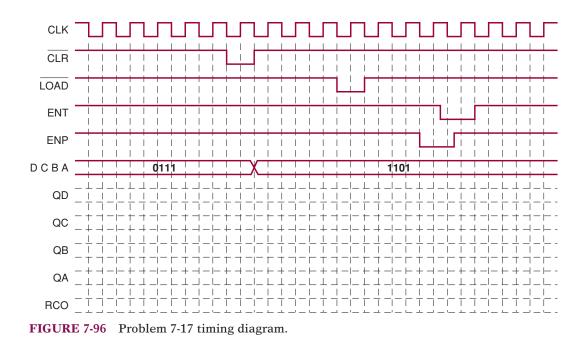
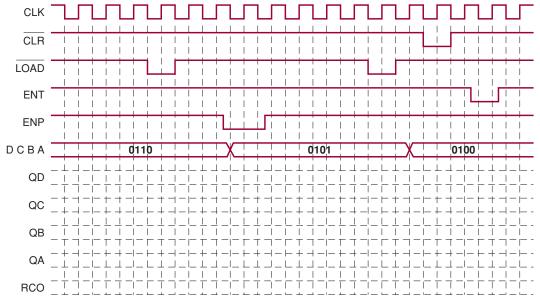


FIGURE 7-95 Problem 7-16 timing diagram.

SECTION 7-7

7-17.*Complete the timing diagram in Figure 7-96 for a 74ALS161 with the indicated input waveforms applied. Assume the initial state is 0000.





7-18. Complete the timing diagram in Figure 7-97 for a 74ALS162 with the indicated input waveforms applied. Assume the initial state is 0000.

FIGURE 7-97 Problem 7-18 timing diagram.

7-19.*Complete the timing diagram in Figure 7-98 for a 74ALS190 with the indicated input waveforms applied. The *DCBA* input is 0101.

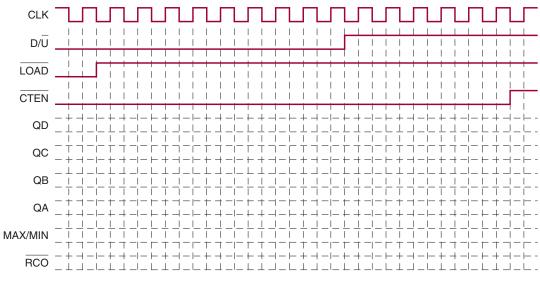
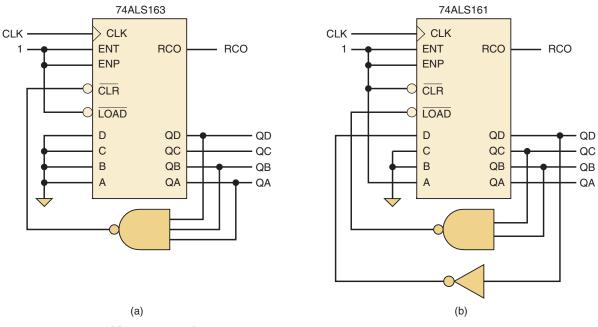
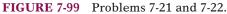


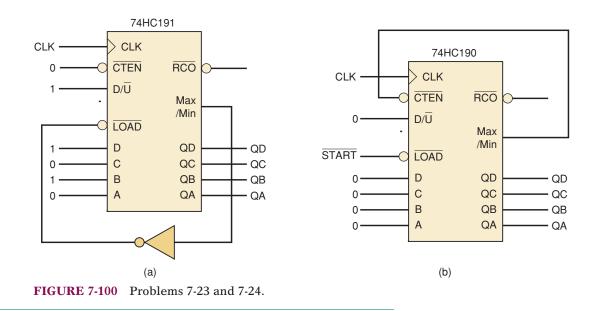
FIGURE 7-98 Problems 7-19 and 7-20 timing diagram.

- 7-20. Repeat Problem 7-19 for a 74ALS191 and a *DCBA* input of 1100.
- **B** 7-21.*Refer to the IC counter circuit in Figure 7-99(a):
 - (a) Draw the state transition diagram for the counter's *QD QC QB QA* outputs.
 - (b) Determine the counter's modulus.

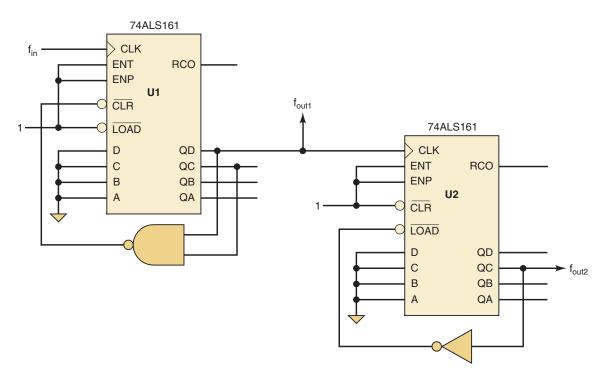




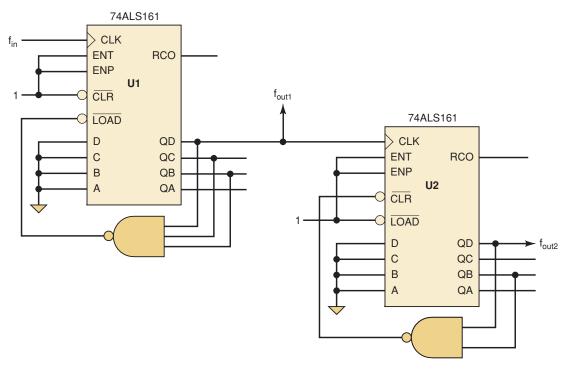
- (c) What is the relationship of the output frequency of the MSB to the input *CLK* frequency?
- (d) What is the duty cycle of the MSB output waveform?
- **B** 7-22. Repeat Problem 7-21 for the IC counter circuit in Figure 7-99(b).
- **B** 7-23.*Refer to the IC counter circuit in Figure 7-100(a).
 - (a) Draw the timing diagram for outputs QD QC QB QA.
 - (b) What is the counter's modulus?
 - (c) What is the count sequence? Does it count up or down?
 - (d) Can we produce the same modulus with a 74HC190? Can we produce the same count sequence with a 74HC190?



- 7-24. Refer to the IC counter circuit in Figure 7-100(b):
 - (a) Describe the counter's output on QD QC QB QA if \overline{START} is LOW.
 - (b) Describe the counter's output on $QD \ QC \ QB \ QA$ if \overline{START} is momentarily pulsed LOW and then returns to a HIGH.
 - (c) What is the counter's modulus? Is this a recycling counter?
- **D** 7-25.*Draw a schematic to create a recycling, MOD-6 counter that uses:
 - (a) the clear control on a 74ALS160
 - (b) the clear control on a 74ALS162
- **D** 7-26. Draw a schematic to create a recycling, MOD-6 counter that produces the count sequence:
 - (a) 1, 2, 3, 4, 5, 6, and repeats with a 74ALS162
 - (b) 5, 4, 3, 2, 1, 0, and repeats with a 74ALS190
 - (c) 6, 5, 4, 3, 2, 1, and repeats with a 74ALS190
- **D** 7-27.* Design a MOD-100, binary counter using either two 74HC161 or two 74HC163 chips and any necessary gates. The IC counter chips are to be synchronously cascaded together to produce the binary count sequence for 0 to 99. The MOD-100 is to have two control inputs, an active-LOW count enable (\overline{EN}) and an active-LOW, asynchronous clear (\overline{CLR}) . Label the counter outputs Q0, Q1, Q2, etc., with Q0 = LSB. Which output is the MSB?
- **D** 7-28. Design a MOD-100, BCD counter using either two 74HC160 or two 74HC162 chips and any necessary gates. The IC counter chips are to be synchronously cascaded together to produce the BCD count sequence for 0 to 99. The MOD-100 is to have two control inputs, an active-HIGH count enable (*EN*) and an active-HIGH, synchronous load (*LD*). Label the counter outputs Q0, Q1, Q2, etc., with Q0 = LSB. Which set of outputs represents the 10s digit?
- **B** 7-29.*With a 6-MHz clock input to a 74ALS163 that has all four control inputs HIGH, determine the output frequency and duty cycle for each of the *five* outputs (including RCO).
- **B** 7-30. With a 6-MHz clock input to a 74ALS162 that has all four control inputs HIGH, determine the output frequency and duty cycle for each of the following outputs: QA, QC, QD, RCO. What is unusual about the waveform pattern that would be produced by the QB output? This pattern characteristic results in an undefined duty cycle.
- **B** 7-31.* The frequency of f_{in} is 6 MHz in Figure 7-101. The two IC counter chips have been cascaded asynchronously so that the output frequency produced by counter U1 is the input frequency for counter U2. Determine the output frequency for f_{out1} and f_{out2} .
- **B** 7-32. The frequency of f_{in} is 1.5 MHz in Figure 7-102. The two IC counter chips have been cascaded asynchronously so that the output frequency produced by counter U1 is the input frequency for counter U2. Determine the output frequency for f_{out1} and f_{out2} .
- 7-33.*Design a frequency divider circuit that will produce the following three output signal frequencies: 1.5 MHz, 150 kHz, and 100 kHz. Use 74HC162 and 74HC163 counter chips and any necessary gates. The input frequency is 12 MHz.
- D 7-34. Design a frequency divider circuit that will produce the following three output signal frequencies: 1 MHz, 800 kHz, and 100 kHz. Use 74HC160 and 74HC161 counter chips and any necessary gates. The input frequency is 12 MHz.



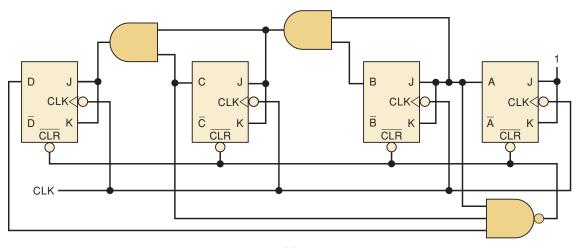






- **B** 7-35.*Draw the gates necessary to decode all of the states of a MOD-16 counter using active-LOW outputs.
- **B** 7-36. Draw the AND gates necessary to decode the 10 states of the BCD counter of Figure 7-8(b).

- **C** 7-37.* Analyze the synchronous counter in Figure 7-103(a). Draw its timing diagram and determine the counter's modulus.
- C 7-38. Repeat Problem 7-37 for Figure 7-103(b).





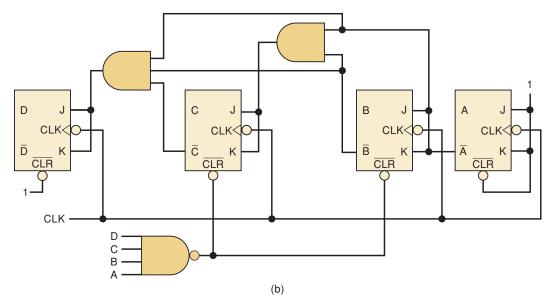
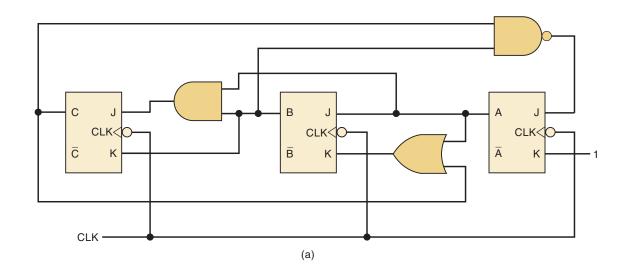


FIGURE 7-103 Problems 7-37 and 7-38.

- **C** 7-39.* Analyze the synchronous counter in Figure 7-104(a). Draw its timing diagram and determine the counter's modulus.
- **C** 7-40. Repeat Problem 7-39 for Figure 7-104(b).
- **C** 7-41.* Analyze the synchronous counter in Figure 7-105(a). F is a control input. Draw its state transition diagram and determine the counter's modulus.
- C 7-42. Analyze the synchronous counter in Figure 7-105(b). Draw its complete state transition diagram and determine the counter's modulus. Is the counter self-correcting?



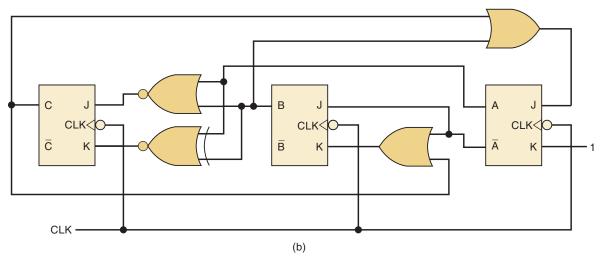
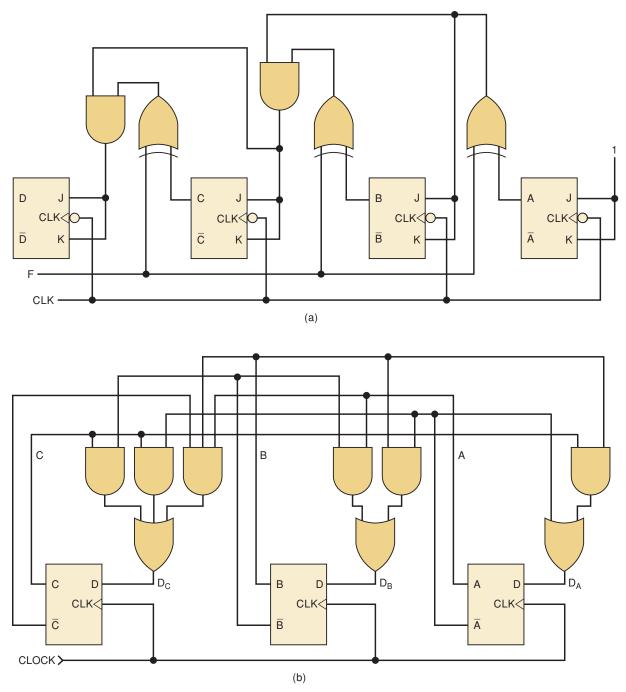
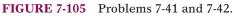


FIGURE 7-104 Problems 7-39 and 7-40.

- **D** 7-43.*(a) Design a synchronous counter using J-K FFs that has the following sequence: 000, 010, 101, 110, and repeat. The undesired (unused) states 001, 011, 100, and 111 must always go to 000 on the next clock pulse.
 - (b) Redesign the counter of part (a) without any requirement on the unused states; that is, their NEXT states can be don't cares. Compare with the design from (a).
- **D** 7-44. Design a synchronous, recycling, MOD-5 down counter that produces the sequence 100, 011, 010, 001, 000, and repeat. Use J-K flip-flops.
 - (a) Force the unused states to 000 on the next clock pulse.
 - (b) Use don't-care NEXT states for the unused states. Is this design self-correcting?
- D 7-45.*Design a synchronous, recycling, BCD down counter with J-K FFs using don't-care NEXT states.
- **D** 7-46. Design a synchronous, recycling, MOD-7 up/down counter with J-K FFs. Use the states 000 through 110 in the counter. Control the count direction with input D (D = 0 to count up and D = 1 to count down).



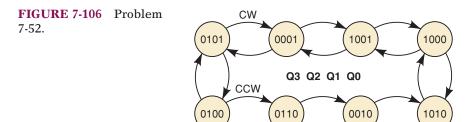


- **D** 7-47.*Design a synchronous, recycling, MOD-8, binary down counter with D flip-flops.
- **D** 7-48. Design a synchronous, recycling, MOD-12 counter with D FFs. Use the states 0000 through 1011 in the counter.

SECTIONS 7-11 AND 7-12

H, D 7-49.* Design a recycling, MOD-13, up counter using an HDL. The count sequence should be 0000 through 1100. Simulate the counter.

- H, D 7-50. Design a recycling, MOD-25, down counter using an HDL. The count sequence should be 11000 through 00000. Simulate the counter.
- H, D 7-51.*Design a recycling, MOD-16 Gray code counter using an HDL. The counter should have an active-HIGH enable (*cnt*). Simulate the counter.
- H, D 7-52. Design a bidirectional, half-step controller for a stepper motor using an HDL. The direction control input (*dir*) will produce a clockwise (CW) pattern when HIGH or counterclockwise when LOW. The sequence is given in Figure 7-106. Simulate the sequential circuit.



- H, D 7-53.* Design a frequency divider circuit to output a 100-kHz signal using an HDL. The input frequency is 5 MHz. Simulate the counter.
- H, D 7-54. Design a frequency divider circuit that will output either of two specified frequency signals using an HDL. The output frequency is selected by the control input *fselect*. The divider will output a frequency of 5 kHz when *fselect* = 0 or 12 kHz when *fselect* = 1. The input frequency is 60 kHz. Simulate the counter.
- **H**, **B** 7-55.*Expand the full-featured HDL counter in Section 7-12 to a MOD-256 counter. Simulate the counter.
- **H**, **B** 7-56. Expand the full-featured HDL counter in Section 7-12 to a MOD-1024 counter. Simulate the counter.
- **H**, **B** 7-57*Design a recycling, MOD-16, down counter using an HDL. The counter should have the following controls (from lowest to highest priority): an active-LOW count enable (en), an active-HIGH synchronous clear (*clr*), and active-LOW synchronous load (\overline{ld}). Decode the terminal count when enabled by en. Simulate the counter.
- H, D 7-58. Design a recycling, MOD-10, up/down counter using an HDL. The counter will count up when up = 1 and counts down when up = 0. The counter should also have the following controls (from lowest to highest priority): an active-HIGH count enable (*enable*), active-HIGH synchronous load (*load*), and an active-LOW asynchronous clear (*clear*). Decode the terminal count when enabled by *enable*. Simulate the counter.

- H 7-59*Create a MOD-1000 BCD counter by cascading together three of the HDL BCD counter modules (described in Section 7-13). Simulate the counter.
- H 7-60. Create a MOD-256 binary counter by cascading together two of the full-featured, MOD-16, HDL counter modules (described in Section 7-12). Simulate the counter.
- H, D 7-61.*Design a synchronous, MOD-50 BCD counter by cascading the HDL designs for a MOD-10 and a MOD-5 counter together. The MOD-50

counter should have an active-HIGH count enable (*enable*) and an active-LOW, synchronous clear (\overline{clrn}). Be sure to include the terminal count detection for the 1s digit to cascade with the 10s digit. Simulate the counter.

 H, D 7-62. Design a synchronous, MOD-100, BCD down counter by cascading two MOD-10 HDL down counter modules together. The MOD-100 counter should have a synchronous parallel load (*load*). Simulate the counter.

SECTION 7-14

- H 7-63[★]Modify the HDL description in Figure 7-56 or Figure 7-57 to add a rinse sequence after the clothes are washed. The new state machine sequence should be *idle* → *wash_fill* → *wash_agitate* → *wash_spin* → *rinse_fill* → *rinse_agitate* → *rinse_spin* → *idle*. Use hot water to wash, and cold water to rinse (add output bits to control two water valves). Simulate the modified HDL design.
- H 7-64. Simulate the HDL traffic light controller design presented in Section 7-14.

PART 2

SECTIONS 7-15 THROUGH 7-19

B 7-65.*A set of 74ALS174 registers is connected as shown in Figure 7-107. What type of data transfer is performed with each register? Determine the output of each register when the \overline{MR} is pulsed momentarily LOW and after each of the indicated clock pulses (CP#) in Table 7-10. How many clock pulses must be applied before data that are input on *I5–I0* are available at *Z5–Z0*?

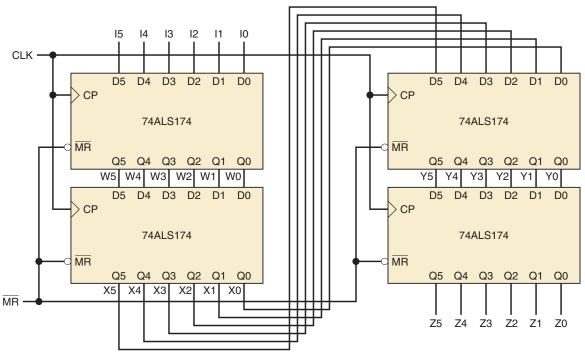


FIGURE 7-107 Problem 7-65.

TADLE 7 10					1		
TABLE 7-10	↑ CLK	MR	l5–l10	W5–W0	X5–X0	Y5–Y0	Z5–Z0
	Х	0	101010				
	CP1	1	101010				
	CP2	1	010101				
	CP3	1	000111				
	CP4	1	111000				
	CP5	1	011011				
	CP6	1	001101				
	CP7	1	000000				
	CP8	1	000000				

B 7-66. Complete the timing diagram in Figure 7-108 for a 74HC174. How does the timing diagram show that the master reset is asynchronous?

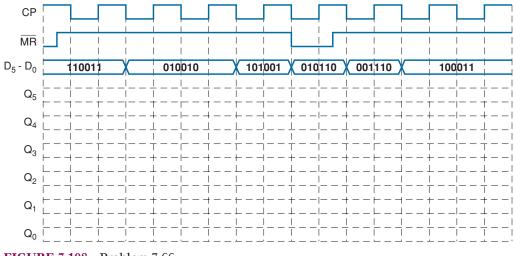
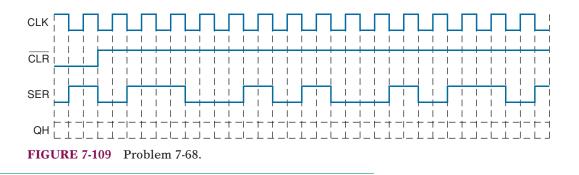
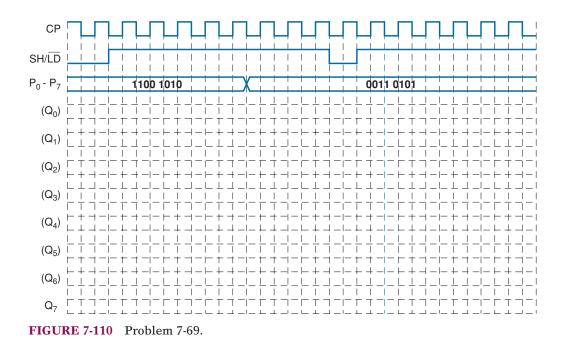


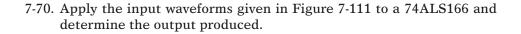
FIGURE 7-108 Problem 7-66.

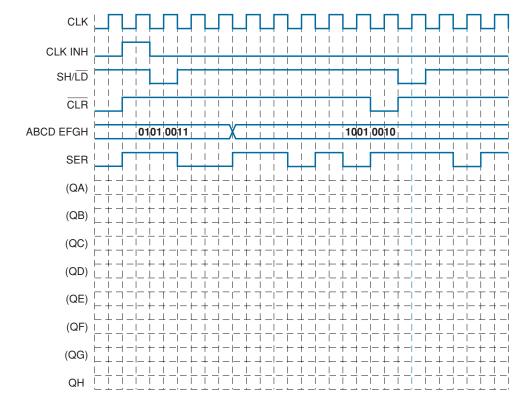
- **B** 7-67.*How many clock pulses will be needed to completely load eight bits of serial data into a 74ALS166? How does this relate to the number of flip-flops contained in the register?
- **B** 7-68. Repeat Example 7-18 for the input waveforms given in Figure 7-109.

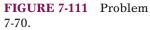




7-69.*Repeat Example 7-20 with $D_{\rm S} = 1$ and the input waveforms given in Figure 7-110.





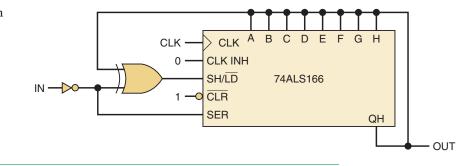


- B 7-71*While examining the schematic for a piece of equipment, a technician or an engineer will often come across an IC that is unfamiliar. In such cases, it is often necessary to consult the manufacturer's data sheets for specifications on the device. Research the data sheet for the 74AS194 bidirectional universal shift register to answer the following questions:
 - (a) Is the \overline{CLR} input asynchronous or synchronous?
 - (b) *True or false:* When *CLK* is LOW, the S_0 and S_1 inputs have no effect on the register.
 - (c) Assume the following conditions:

```
Q_A \ Q_B \ Q_C \ Q_D = 1 \ 0 \ 1 \ 1
A \ B \ C \ D = 0 \ 1 \ 1 \ 0
\overline{CLR} = 1
SR \ SER = 0
SL \ SER = 1
```

If $S_0 = 0$ and $S_1 = 1$, determine the register outputs after one *CLK* pulse. After two *CLK* pulses. After three. After four.

- (d) Use the same conditions except $S_0 = 1$ and $S_1 = 0$ and repeat part (c).
- (e) Repeat part (c) with $S_0 = 1$ and $S_1 = 1$.
- (f) Repeat part (c) with $S_0 = 0$ and $S_1 = 0$.
- (g) Use the same conditions as in part (c), except assume that Q_A is connected to *SL SER*. What will be the register outputs after four *CLK* pulses?
- **C** 7-72. Refer to Figure 7-112 to answer the following questions:
 - (a) Which register function (load or shift) will be performed on the next clock if in = 1 and out = 0? What data value will be input when clocked?
 - (b) Which register function (load or shift) will be performed on the next clock if in = 0 and out = 1? What data value will be input when clocked?
 - (c) Which register function (load or shift) will be performed on the next clock if in = 0 and out = 0? What data value will be input when clocked?
 - (d) Which register function (load or shift) will be performed on the next clock if in = 1 and out = 1? What data value will be input when clocked?
 - (e) What input condition will eventually (after several clock pulses) cause the output to switch states?



- (f) To change the output logic level requires the new input condition
- (g) If the input signal changes levels and then goes back to its original logic level before the number of clock pulses specified in part (f), what happens to the output signal.
- (h) Explain why this circuit can be used to debounce switches.

to last for at least how many clock pulses?

SECTION 7-20

- **B** 7-73.*Draw the diagram for a MOD-5 ring counter using J-K flip-flops. Make sure that the counter will start the proper count sequence when it is turned on.
 - 7-74. Add one more J-K flip-flop to convert the MOD-5 ring counter in Problem 7-73 into a MOD-10 counter. Determine the sequence of states for this counter. This is an example of a decade counter that is not a BCD counter. Draw the decoding circuit for this counter.
- **B** 7-75.*Draw the diagram for a MOD-10 Johnson counter using a 74HC164. Make sure that the counter will start the proper count sequence when it is turned on. Determine the count sequence for this counter and draw the decoding circuit needed to decode each of the 10 states. This is another example of a decade counter that is not a BCD counter.
 - 7-76. The clock input to the Johnson counter in Problem 7-75 is 10 Hz. What is the frequency and duty cycle for each of the counter outputs?

SECTION 7-21

- T 7-77.*The MOD-10 counter in Figure 7-8(b) produces the count sequence 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, and repeats. Identify some possible fault conditions that might produce this result.
- T 7-78. The MOD-10 counter in Figure 7-8(b) produces the count sequence 0000, 0101, 0010, 0111, 1000, 1101, 1010, 1111, and repeats. Identify some possible fault conditions that might produce this result.

SECTIONS 7-22 AND 7-23

- **H** 7-79.*Create an eight-bit SISO shift register using an HDL. The serial input is called *ser* and the serial output is called *qout*. An active-LOW enable (en) controls the shift register. Simulate the design.
- **H** 7-80. Create an eight-bit PIPO shift register using an HDL. The data in is d[7..0] and the outputs are q[7..0]. An active-HIGH enable (*ld*) controls the shift register. Simulate the design.
- **H** 7-81*Create an eight-bit PISO shift register using an HDL. The data in is d[7..0] and the output is q0. The shift register function is controlled by sh_ld ($sh_ld = 0$ to synchronously parallel load and $sh_ld = 1$ to serial shift). The register also should have an active-LOW asynchronous clear (clrn). Simulate the design.
- **H** 7-82. Create an eight-bit SIPO shift register using an HDL. The data in is ser_in and the outputs are q[7..0]. The shift register function is enabled by an active-HIGH control named *shift*. The shift register also has a higher priority active-HIGH synchronous clear (*clear*). Simulate the design.
- **H** 7-83.*Simulate the universal shift register design from Example 7-24.

- H 7-84. Create an eight-bit universal shift register by cascading two of the modules in Example 7-24. Simulate the design.
- H, D 7-85.* Design a MOD-10, self-starting Johnson counter with an active-HIGH, asynchronous reset (*reset*) using an HDL. Simulate the design.
- H, D 7-86. Sometimes a digital application may need a ring counter that recirculates a single zero instead of a single one. The ring counter would then have an active-LOW output instead of an active-HIGH. Design a MOD-8, self-starting ring counter with an active-LOW output using an HDL. The ring counter should also have an active-HIGH *hold* control to disable the counting. Simulate the design.

- H 7-87.*Use Altera's simulator to test the nonretriggerable, level-sensitive, one-shot design example in either Figure 7-88 (AHDL) or 7-89 (VHDL). Use a 1-kHz clock and create a 10-ms output pulse for the simulation. Verify that:
 - (a) The correct pulse width is created when triggered.
 - (b) The output can be terminated early with the reset input.
 - (c) The one-shot design is nonretriggerable and cannot be triggered again until it has timed out.
 - (d) The trigger signal must last long enough for the clock to catch it.
 - (e) The pulse width can be changed to a different value.
- H 7-88. Modify the nonretriggerable, level-sensitive, one-shot design example from either Figure 7-88 (AHDL) or Figure 7-89 (VHDL) so that the one-shot is retriggerable but still level-sensitive. Simulate the design.

DRILL QUESTION

- **B** 7-89.*For each of the following statements, indicate the type(s) of counter being described.
 - (a) Each FF is clocked at the same time.
 - (b) Each FF divides the frequency at its CLK input by 2.
 - (c) The counting sequence is 111, 110, 101, 100, 011, 010, 001, 000.
 - (d) The counter has 10 distinct states.
 - (e) The total switching delay is the sum of the individual FFs' delays.
 - (f) This counter requires no decoding logic.
 - (g) The MOD number is always twice the number of FFs.
 - (h) This counter divides the input frequency by its MOD number.
 - (i) This counter can begin its counting sequence at any desired starting state.
 - (j) This counter can count in any direction.
 - (k) This counter can suffer from decoding glitches due to its propagation delays.
 - (l) This counter only counts from 0 to 9.
 - (m) This counter can be designed to count through arbitrary sequences by determining the logic circuit needed at each flip-flop's synchronous control inputs.

ANSWERS TO SECTION REVIEW QUESTIONS

PART 1

SECTION 7-1

1. False 2. 0000 3. 128

SECTION 7-2

1. Each FF adds its propagation delay to the total counter delay in response to a clock pulse. 2. MOD-256

SECTION 7-3

1. Can operate at higher clock frequencies and has more complex circuitry 2. Six FFs and four AND gates 3. ABCDE

SECTION 7-4

1. D, C, and A 2. True, because a BCD counter has 10 distinct states 3. 5 kHz

SECTION 7-5

1. In an up counter, the count is increased by 1 with each clock pulse; in a down counter, the count is decreased by 1 with each pulse. 2. Change connections to respective inverted outputs instead of *Q*s.

SECTION 7-6

1. It can be preset to any desired starting count. 2. Asynchronous presetting is independent of the clock input, while synchronous presetting occurs on the active edge of the clock signal.

SECTION 7-7

1. $\overline{\text{LOAD}}$ is the control that enables the parallel loading of the data inputs D C B A (A = LSB). 2. $\overline{\text{CLR}}$ is the control that enables the resetting of the counter to 0000. 3. True 4. All control inputs ($\overline{\text{CLR}}$, $\overline{\text{LOAD}}$, ENT, and ENP) on the 74162 must be HIGH. 5. $\overline{\text{LOAD}} = 1$, $\overline{\text{CTEN}} = 0$, and $D/\overline{U} = 1$ to count down. 6. 74HC163: 0 to 65,535; 74ALS190: 0 to 9999 or 9999 to 0.

SECTION 7-8

1. Sixty-four 2. A six-input NAND gate with inputs A, B, C, \overline{D} , E, and \overline{F} .

SECTION 7-9

 We will not have to deal with transient states and possible glitches in output waveforms.
 PRESENT state/NEXT state table
 The gates control the count sequence.
 Unused states all lead back to the count sequence of the counter.

SECTION 7-10

1. See text.2. It associates every possible PRESENT state with its desired NEXTstate.3. It shows the necessary levels at each flip-flop's synchronous input toproduce the counter's state transitions.4. True

SECTION 7-11

1. PRESENT state/NEXT state tables 2. The desired NEXT state 3. AHDL:
ff[].clk = !clock
VHDL:
IF (clock = '0' AND clock' EVENT) THEN
4. Behavioral description

1. Asynchronous clear causes the counter to clear immediately. Synchronous load occurs on the next active clock edge. 2. AHDL: Use .clrn port on FFs; VHDL: Define clear function before checking for clock edge 3. By the order of evaluation in an IF statement.

SECTION 7-13

1. Both HDLs can use a block diagram to connect modules; VHDL can also use a text file that describes the connections between components. 2. A bus is a collection of signal lines; it is represented graphically by a heavy-weight line 3. Count enable and terminal count decoding

SECTION 7-14

 A counter is commonly used to count events, while a state machine is commonly used to control events.
 A state machine can be described using symbols to describe its states rather than actual binary states.
 The compiler assigns the optimal values to minimize the circuitry.
 The description is much easier to write and understand.

PART 2

SECTION 7-19

Parallel in/serial out
 True
 Serial in/parallel out
 Serial in/serial out
 The 74165 uses asynchronous parallel data transfer; the 74174 uses synchronous parallel data transfer.
 A HIGH prevents shifting on CPs.

SECTION 7-20

1. Ring counter2. Johnson counter3. The inverted output of the last FF isconnected to the input of the first FF.3. The inverted output of the last FF is5. Sixteen; eight4. (a) False (b) True (c) True

SECTION 7-22

1. AHDL:
reg [] .d = (reg [6..0], dat)
VHDL:
reg := reg (6 DOWNTO 0) & dat
2. Because the register may continue to receive clock edges during hold

SECTION 7-23

It can start in any state, but it will eventually reach the desired ring sequence.
 Lines 11 and 12
 Lines 12 and 13

SECTION 7-24

The reset input
 The clock frequency and the delay value loaded into the counter
 Synchronously
 The output pulse width is very consistent.
 The output pulse responds to the trigger edge immediately.
 The state of the trigger on the current clock edge and its state on the previous edge.