

Chapter 6 Registers and Counters

CHAPTER OBJECTIVES

1. Understand the use, functionality, and modes of operation of registers, shift registers, and universal shift registers.
2. Know how to properly create the effect of a gated clock.
3. Understand the structure and functionality of a serial adder circuit.
4. Understand the behavior of a (a) ripple counter, (b) synchronous counter, (c) ring counter, and (d) Johnson counter.
5. Be able to write structural and behavioral HDL models of registers, shift registers, universal shift registers, and counters.

6.1 REGISTERS

A clocked sequential circuit consists of a group of flip-flops and combinational gates. The flip-flops are essential because, in their absence, the circuit reduces to a purely combinational circuit (provided that there is no feedback among the gates). A circuit with flip-flops is considered a sequential circuit even in the absence of combinational gates. Circuits that include flip-flops are usually classified by the function they perform rather than by the name of the sequential circuit. Two such circuits are registers and counters.

A *register* is a group of flip-flops, each one of which shares a common clock and is capable of storing one bit of information. An n -bit register consists of a group of n flip-flops capable of storing n bits of binary information. In addition to the flip-flops, a register may have combinational gates that perform certain data-processing tasks. In its broadest definition, a register consists of a group of flip-flops together with gates that affect their operation. The flip-flops hold the binary information, and the gates determine how the information is transferred into the register.

A *counter* is essentially a register that goes through a predetermined sequence of binary states. The gates in the counter are connected in such a way as to produce the prescribed sequence of states. Although counters are a special type of register, it is common to differentiate them by giving them a different name.

Various types of registers are available commercially. The simplest register is one that consists of only flip-flops, without any gates. [Figure 6.1](#) shows such a register constructed with four D -type flip-flops to form a four-bit data storage register. The common clock input triggers all flip-flops on the positive edge of each pulse, and the binary data available at the four inputs are transferred simultaneously into the register. The value of (I_3, I_2, I_1, I_0) immediately before the clock edge determines the value of (A_3, A_2, A_1, A_0) after the clock edge. The four outputs can be sampled at any time to obtain the binary information stored in the register.¹ The input *Clear_b* goes to the active-low R (reset) input of all four flip-flops. When this input goes to 0, all flip-flops are reset asynchronously, that is,

independently of the clock. The *Clear_b* input is useful for clearing the register to all 0's prior to its clocked operation. The *R* inputs must be maintained at logic 1 (i.e., de-asserted) during normal clocked operation. Note that, depending on the flip-flop, either of the labels *Clear*, *Clear_b*, *reset*, or *reset_b* can be used to indicate the transfer of the register to an all 0's state.

¹ In practice, the outputs are sampled only when they are stable.

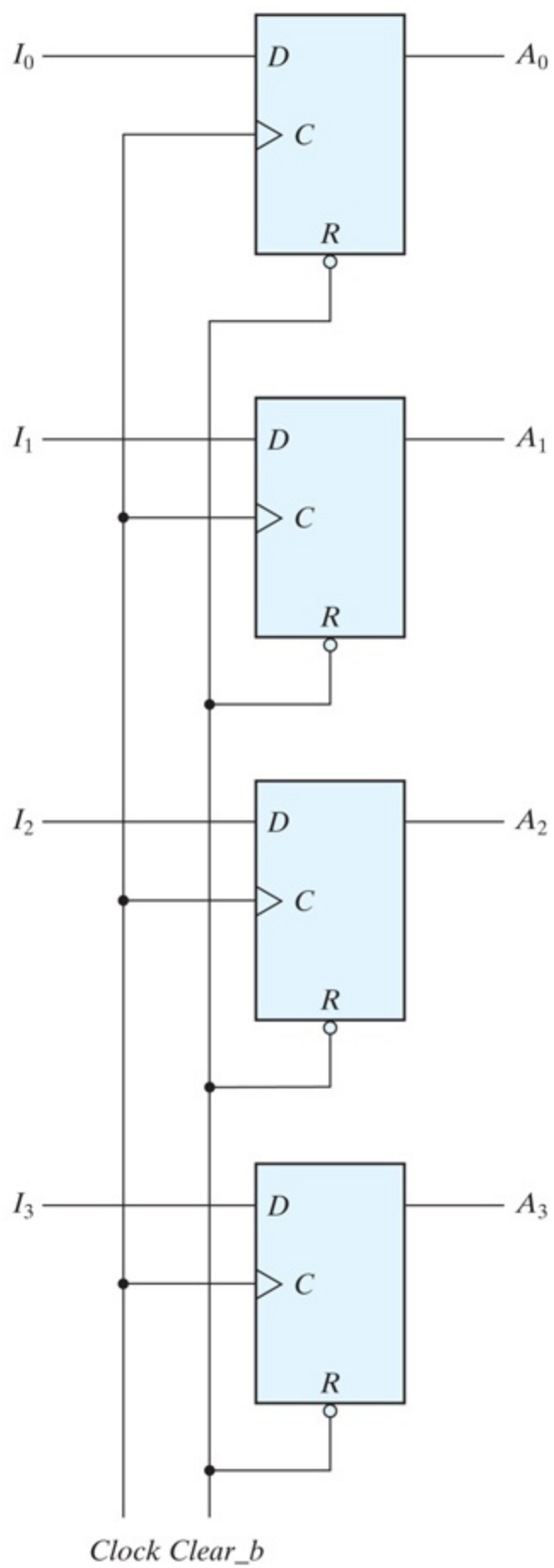


FIGURE 6.1

Four-bit register

Register with Parallel Load

Registers with parallel load are a fundamental building block in digital systems. It is important that you have a thorough understanding of their behavior. Synchronous digital systems have a master clock generator that supplies a continuous train of clock pulses. The pulses are applied simultaneously to all flip-flops and registers in the system. The master clock acts like a drum that supplies a constant beat to all parts of the system. A separate control signal must be used to decide which register operation will execute at each clock pulse. The transfer of new information into a register is referred to as *loading* or *updating* the register. If all the bits of the register are loaded simultaneously with a common clock pulse, we say that the loading is done *in parallel*. A clock edge applied to the *C* inputs of the register of [Fig. 6.1](#) will load all four inputs in parallel. In this configuration, if the contents of the register must be left unchanged, the inputs must be held constant or the clock must be inhibited from the circuit. In the first case, the data bus driving the register would be unavailable for other traffic. In the second case, the clock can be inhibited from reaching the register by controlling the clock input signal with an enabling gate. However, inserting gates into the clock path is ill-advised because it means that logic is performed with clock pulses. The insertion of logic gates in the path of the clock signal produces uneven propagation delays between the master clock and the inputs of flip-flops. To fully synchronize the system, we must ensure that all clock pulses arrive at the same time anywhere in the system, so that all flip-flops trigger simultaneously. Performing logic with clock pulses inserts variable delays and may cause the system to go out of synchronism. For this reason, it is advisable to control the operation of the register with the *D* inputs, rather than controlling the clock in the *C* inputs of the flip-flops. This creates the effect of a gated clock, but without affecting the clock path of the circuit.

A four-bit data-storage register with a load control input that is directed through gates and into the *D* inputs of the flip-flops is shown in [Fig. 6.2](#).

The additional gates implement a two-channel mux whose output drives the input to the register with either the data bus or the output of the register. The *load* input to the register determines the action to be taken with each clock pulse. When the *load* input is 1, the data at the four external inputs are transferred into the register with the next positive edge of the clock. When the *load* input is 0, the outputs of the flip-flops are connected to their respective inputs. The feedback connection from output to input is necessary because a *D* flip-flop does not have a “no change” condition. With each clock edge, the *D* input determines the next state of the register. To leave the output unchanged, it is necessary to make the *D* input equal to the present value of the output (i.e., the output recirculates to the input at each clock pulse). The clock pulses are applied to the *C* inputs without interruption, and the propagation delay of the clock path is unaffected. The load input determines whether the next pulse will accept new information or leave the information in the register intact. In effect, what is commonly referred to as “clock gating” is achieved by gating the datapath of the register. The transfer of information from the data inputs or the outputs of the register is done simultaneously with all four bits in response to a clock edge.

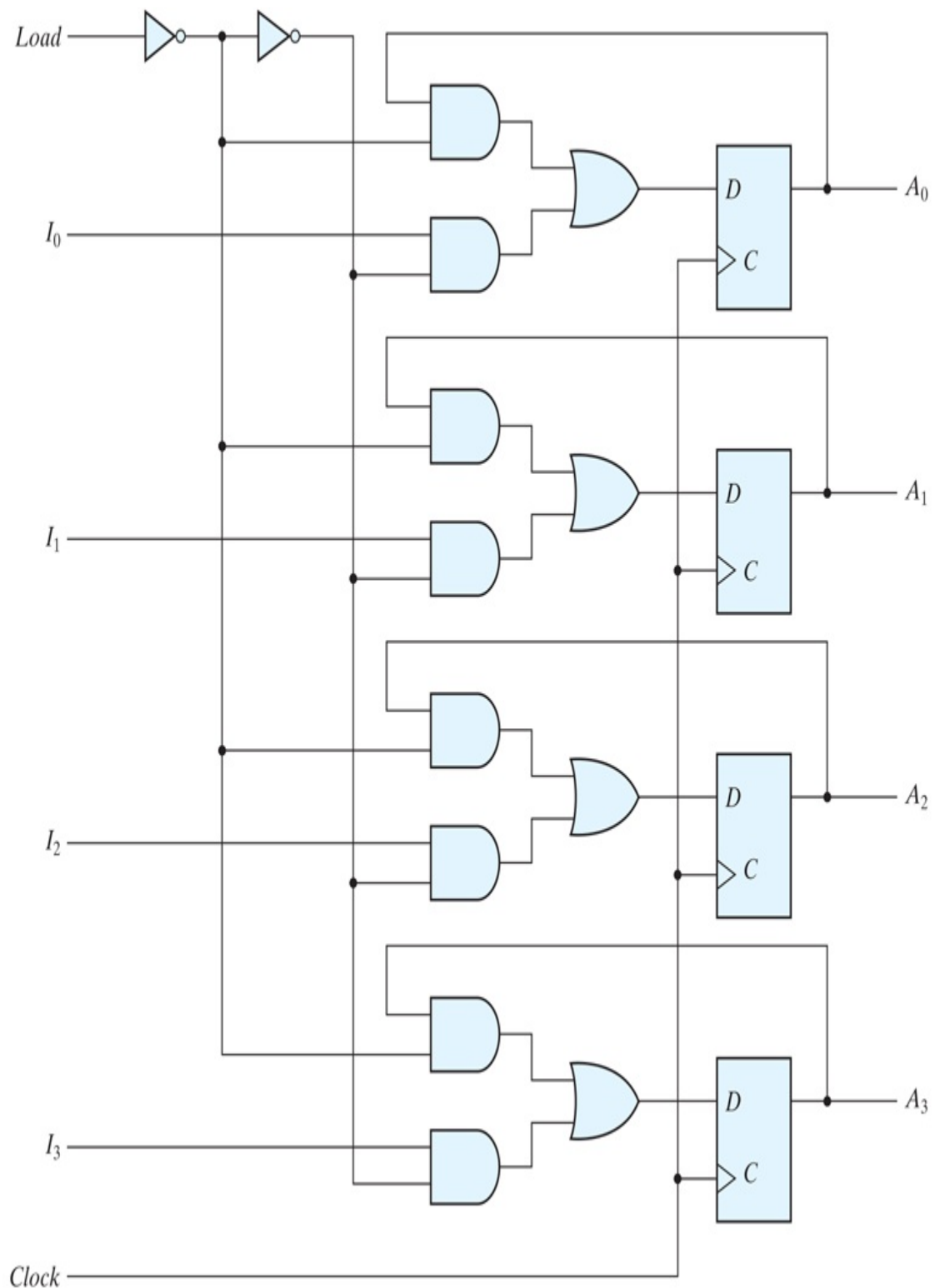


FIGURE 6.2

Four-bit register with parallel load

Description

6.2 SHIFT REGISTERS

A register capable of shifting the binary information held in each cell to its neighboring cell, in a selected direction, is called a *shift register*. The logical configuration of a shift register consists of a chain of flip-flops in cascade, with the output of one flip-flop connected to the data input of the next flip-flop. All flip-flops receive common clock pulses, which activate the shift of data from one stage to the next.

The simplest possible shift register is one that uses only flip-flops, as shown in [Fig. 6.3](#). The output of a given flip-flop is connected to the *D* input of the flip-flop at its right. This shift register is unidirectional (left-to-right). Each clock pulse shifts the contents of the register one bit position to the right. The configuration does not support a left shift. The *serial input* determines what goes into the leftmost flip-flop during the shift. The *serial output* is taken from the output of the rightmost flip-flop. Sometimes it is necessary to control the shift so that it occurs only with certain pulses, but not with others. As with the data register discussed in the previous section, the action of the clock signal can be suppressed by gating the data transfer, achieving the effect of gating the clock, and preventing the register from shifting. This scheme leaves the clock path unchanged, but recirculates the output of each register cell back through a two-channel mux whose output is connected to the input of the cell. When the clock action is not suppressed, the other channel of the mux provides a datapath to the cell.

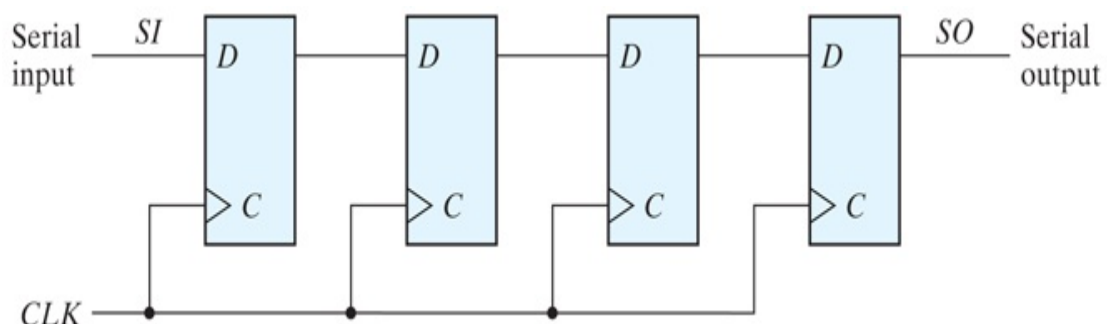


FIGURE 6.3

Four-bit shift register

[Description](#)

It will be shown later that the shift operation can be controlled through the *D* inputs of the flip-flops rather than through the clock input. If, however, the shift register of [Fig. 6.3](#) is used, the shift can be controlled with an input by connecting the clock through an AND gate. This is not a preferred practice because it can lead to timing problems. Note that the simplified schematics ([Fig. 6.2](#), [6.3](#)) do not show a reset signal, but such a signal is required in practical designs.

Practice Exercise 6.1

Objective: Draw the logic diagram of a circuit that suspends the clock action of a D flip-flop without gating its clock. Describe the behavior of the circuit.

Answer:

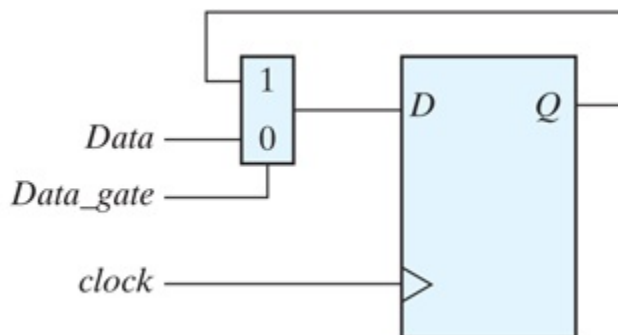


FIGURE PE 6.1

If *Data_gate* is 0, *Data* is transferred to *Q* with each active edge of *clock*. If *Data_gate* is 1, the value of *Q* is recirculated through the mux-flip-flop path, with the effect that *clock* appears to be suspended.

Serial Transfer

The datapath of a digital system is said to operate in serial mode when information is transferred and manipulated one bit at a time. Information is transferred one bit at a time by shifting the bits out of the source register and into the destination register. This type of transfer is in contrast to parallel transfer, whereby all the bits of the register are transferred at the same time.

The serial transfer of information from register *A* to register *B* is done with shift registers, as shown in the block diagram of [Fig. 6.4\(a\)](#). The serial output (*SO*) of register *A* is connected to the serial input (*SI*) of register *B*. To prevent the loss of information stored in the source register, the information in register *A* is made to circulate by connecting the serial output to its serial input. The initial content of register *B* is shifted out through its serial output and is lost unless it is transferred to a third shift register. The shift control input determines when and how many times the registers are shifted. For illustration here, this is done with an AND gate that allows clock pulses to pass into the *CLK* terminals only when the shift control is active. (This practice can be problematic because it may compromise the clock path of the circuit, as discussed earlier.)

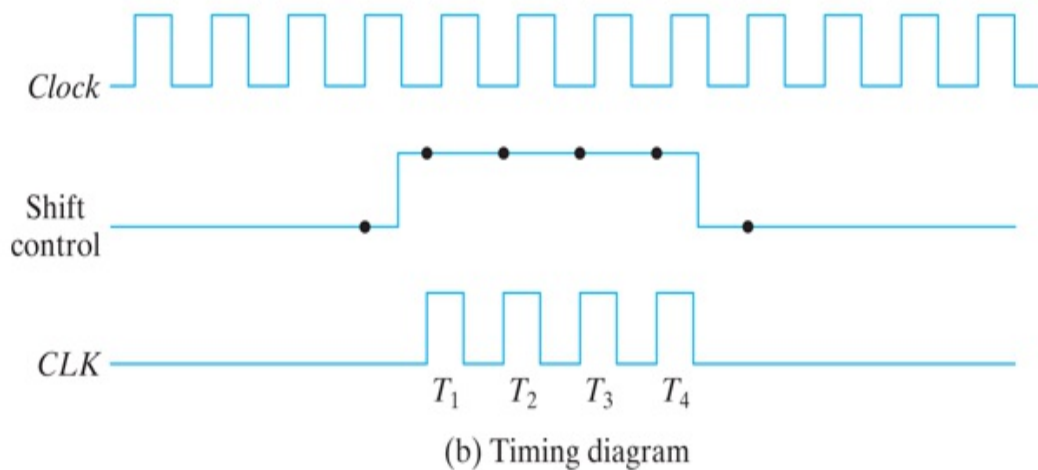
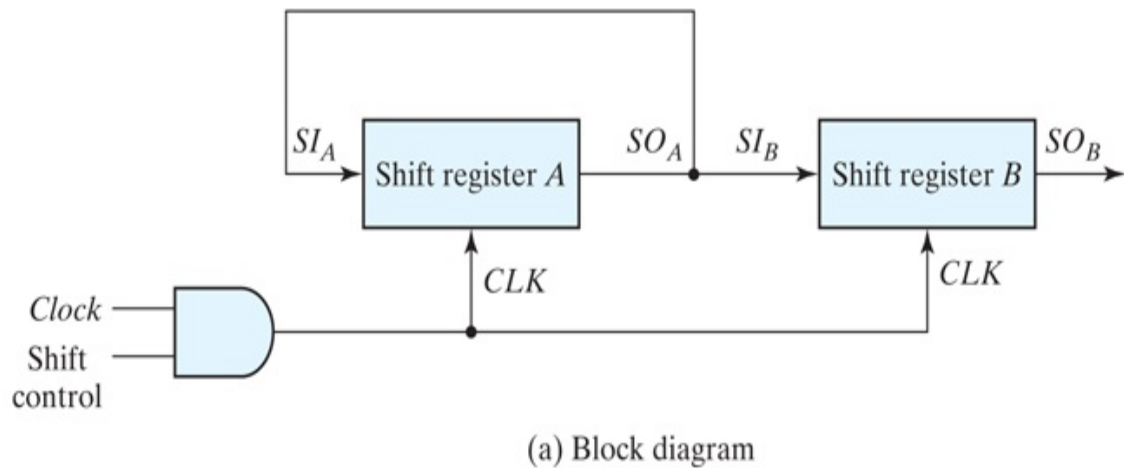


FIGURE 6.4

Serial transfer from register A to register B

Description

Suppose the shift registers in [Fig. 6.4](#) have four bits each. Then the control unit that supervises the transfer of data must be designed in such a way that it enables the shift registers, through the shift control signal, for a fixed time of four clock pulses in order to pass an entire word. This design is shown in the timing diagram of [Fig. 6.4\(b\)](#). The shift control signal is synchronized with the clock and changes value just after the negative edge of the clock. The next four clock pulses find the shift control signal in the active state, so the output of the AND gate connected to the CLK inputs produces four pulses: T_1 , T_2 , T_3 , and T_4 . Each rising edge of the pulse

causes a shift in both registers. The fourth pulse changes the shift control to 0, and the shift registers are disabled.

Assume that the binary content of shift register *A* before the shift is 1011 and that of shift register *B* is 0010. The serial transfer from *A* to *B* occurs in four steps, as shown in [Table 6.1](#). With the first pulse, T1, the rightmost bit of *A* is shifted into the leftmost bit of *B* and is also circulated into the leftmost position of *A*. At the same time, all bits of *A* and *B* are shifted one position to the right. The previous serial output from *B* in the rightmost position is lost, and its value changes from 0 to 1. The next three pulses perform identical operations, shifting the bits of *A* into *B*, one at a time. After the fourth shift, the shift control goes to 0, and registers *A* and *B* both have the value 1011. Thus, the contents of *A* are copied into *B* so that the contents of *A* remain unchanged, that is, the contents of *A* are restored to their original value.

Table 6.1 *Serial-Transfer Example*

Timing Pulse Shift Register *A* Shift Register *B*

Initial value	1	0	1	1	0	0	1	0
After T1	1	1	0	1	1	0	0	1
After T2	1	1	1	0	1	1	0	0
After T3	0	1	1	1	0	1	1	0
After T4	1	0	1	1	1	0	1	1

The difference between the serial and the parallel mode of operation should be apparent from this example. In the parallel mode, information is available from all bits of a register and all bits can be transferred simultaneously during one clock pulse. In the serial mode, the registers have a single serial input and a single serial output. The information is transferred one bit at a time while the registers are shifted in the same direction.

Serial Addition

Operations in digital computers are usually done in parallel because that is a faster mode of operation. Serial operations are slower because a datapath operation takes several clock cycles, but serial operations have the advantage of requiring fewer hardware components. In VLSI circuits, they require less silicon area on a chip and consume less power. To demonstrate the serial mode of operation, we present the design of a serial adder. The parallel counterpart was presented in [Section 4.5](#).

The two binary numbers to be added serially are stored in two shift registers. Beginning with the least significant pair of bits, the circuit adds one pair at a time through a single full-adder (FA) circuit, as shown in [Fig. 6.5](#). The carry out of the full adder is transferred to a *D* flip-flop, the output of which is then used as the carry input for adding the next pair of significant bits. The sum bit from the *S* output of the full adder could be transferred into a third shift register. By shifting the sum into *A* while the bits of *A* are shifted out, it is possible to use one register for storing both the augend and the sum bits. The serial input of register *B* can be used to transfer a new binary number while the addend bits are shifted out during the addition.

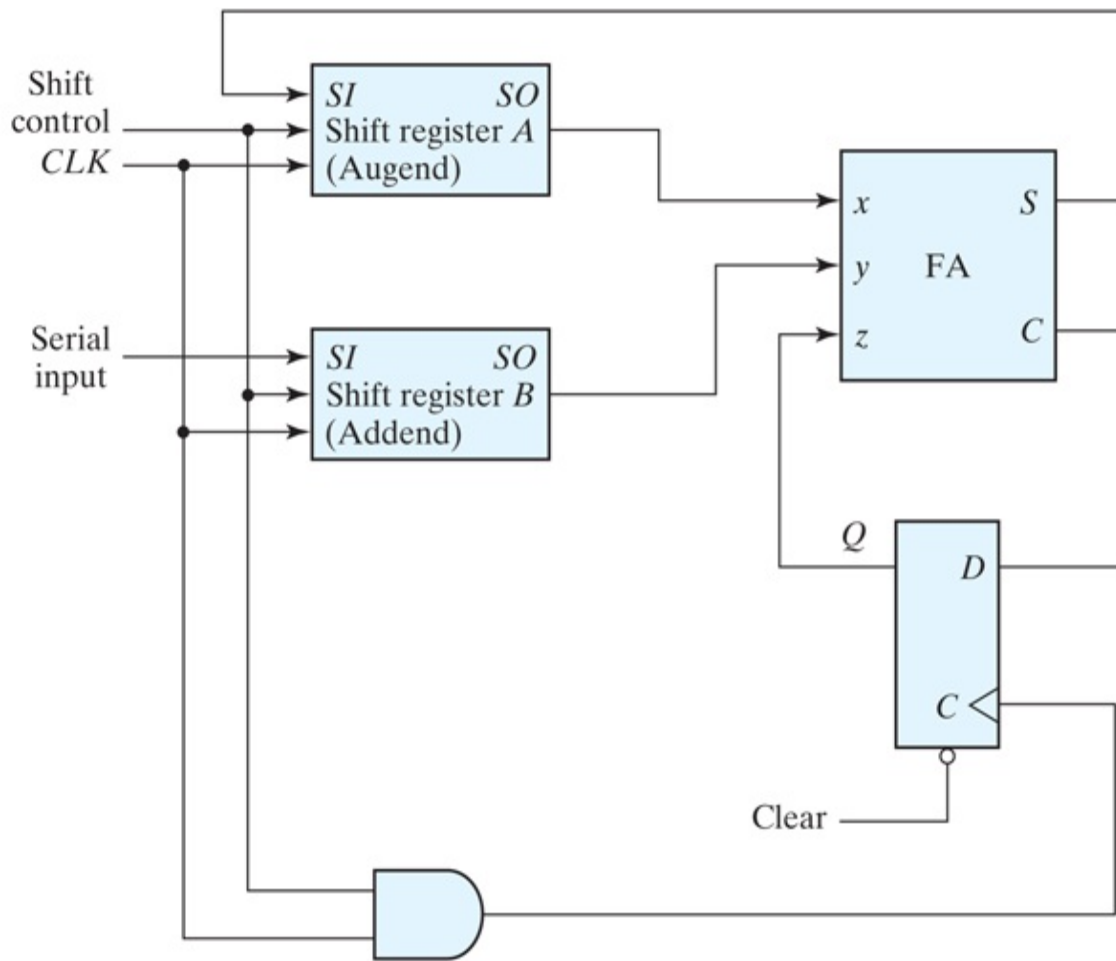


FIGURE 6.5

Serial adder

Description

The operation of the serial adder is as follows: Initially, register A holds the augend, register B holds the addend, and the carry flip-flop is cleared to 0. The outputs (SO) of A and B provide a pair of significant bits for the full adder at x and y. Output Q of the flip-flop provides the input carry at z. The shift control enables both registers and the carry flip-flop, so at the next clock pulse, both registers are shifted once to the right, the sum bit from S enters the leftmost flip-flop of A, and the output carry is transferred into flip-flop Q. The shift control enables the registers for a number of clock pulses equal to the number of bits in the registers. For each succeeding clock pulse, a new sum bit is transferred to A, a new carry is transferred to Q, and both registers are shifted once to the right. This

process continues until the shift control is disabled. Thus, the addition is accomplished by passing each pair of bits together with the previous carry through a single full-adder circuit and transferring the sum, one bit at a time, into register *A*.

Initially, register *A* and the carry flip-flop are cleared to 0, and then the first number is added from *B*. While *B* is shifted through the full adder, a second number is transferred to it through its serial input. The second number is then added to the contents of register *A*, while a third number is transferred serially into register *B*. This can be repeated to perform the addition of two, three, or more four-bit numbers and accumulate their sum in register *A*.

Comparing the serial adder with the parallel adder described in [Section 4.5](#), we note several differences. The parallel adder uses registers with a parallel load, whereas the serial adder uses shift registers. The number of full-adder circuits in the parallel adder is equal to the number of bits in the binary numbers, whereas the serial adder requires only one full-adder circuit and a carry flip-flop. Excluding the registers, the parallel adder is a combinational circuit, whereas the serial adder is a sequential circuit, which consists of a full adder and a flip-flop that stores the output carry. This design is typical in serial operations because the result of a bit-time operation may depend not only on the present inputs but also on previous inputs that must be stored in flip-flops.

To show that serial operations can be designed by means of sequential circuit procedure, we will redesign the serial adder with the use of a state table. First, we assume that two shift registers are available to store the binary numbers to be added serially. The serial outputs from the registers are designated by *x* and *y*. The sequential circuit to be designed will not include the shift registers, but they will be inserted later to show the complete circuit. The sequential circuit proper has the two inputs, *x* and *y*, that provide a pair of significant bits, an output *S* that generates the sum bit, and flip-flop *Q* for storing the carry. The state table that specifies the sequential circuit is listed in [Table 6.2](#). The present state of *Q* is the present value of the carry. The present carry in *Q* is added together with inputs *x* and *y* to produce the sum bit in output *S*. The next state of *Q* is equal to the output carry. Note that the state table entries are identical to the entries in a full-adder truth table, except that the input carry is now the present state of *Q* and the output carry is now the next state of *Q*.

Table 6.2 *State Table for Serial Adder*

Present State Inputs Next State Output Flip-Flop Inputs

Q	x	y	Q	S	JQ	KQ
0	0	0	0	0	0	X
0	0	1	0	1	0	X
0	1	0	0	1	0	X
0	1	1	1	0	1	X
1	0	0	0	1	X	1
1	0	1	1	0	X	0
1	1	0	1	0	X	0
1	1	1	1	1	X	0

If a D flip-flop is used for holding Q , the circuit reduces to the one shown in [Fig. 6.5](#). If a JK flip-flop is used for Q , it is necessary to determine the values of inputs J and K by referring to the excitation table ([Table 5.12](#)). This is done in the last two columns of [Table 6.2](#). The two flip-flop input

equations and the output equation can be simplified by means of maps to

$$J Q = xy \quad K Q = x' y' = (x + y)' \quad S = x \oplus y \oplus Q$$

The circuit diagram is shown in [Fig. 6.6](#). The circuit consists of three gates and a JK flip-flop. The two shift registers are included in the diagram to show the complete serial adder. Note that output S is a function not only of x and y , but also of the present state of Q . The next state of Q is a function of the present state of Q and of the values of x and y that come out of the serial outputs of the shift registers.

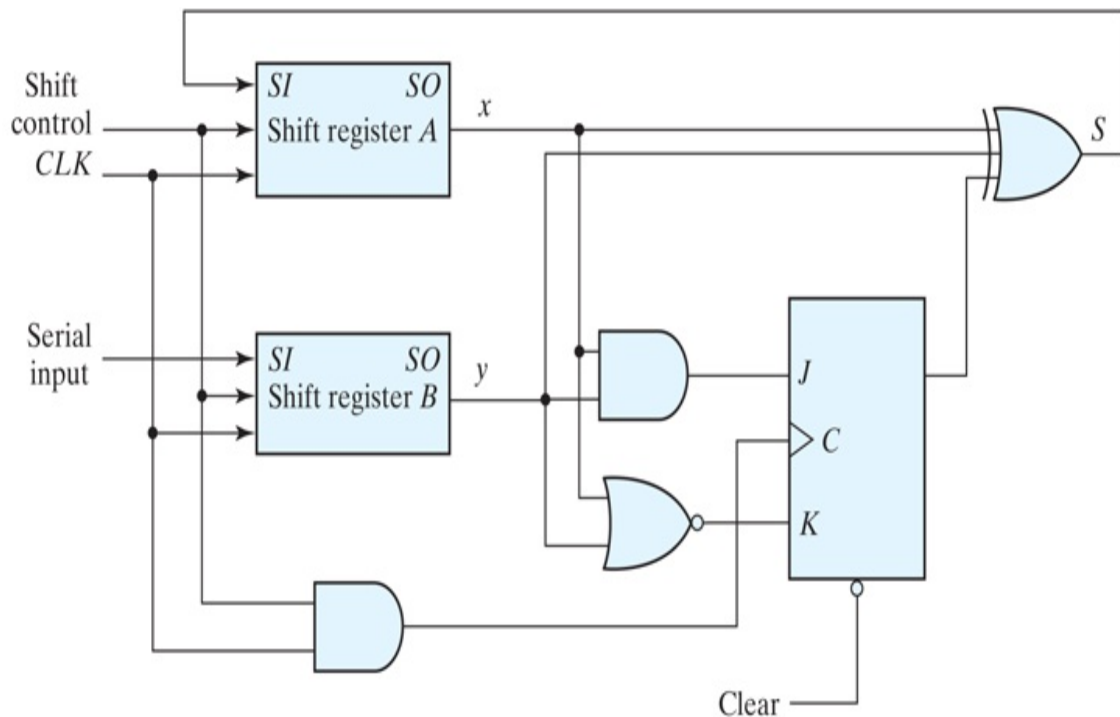


FIGURE 6.6

Second form of serial adder

[Description](#)

Practice Exercise 6.2

1. Explain why a serial adder is a sequential circuit.

Answer: The circuit uses a flip-flop.

Universal Shift Register

If the flip-flop outputs of a shift register are accessible, then information entered serially by shifting can be taken out in parallel from the outputs of the flip-flops. If a parallel load capability is added to a shift register, then data entered in parallel can be taken out in serial fashion by shifting the data stored in the register.

Some shift registers provide the necessary input and output terminals for parallel transfer. They may also have both shift-right and shift-left capabilities. The most general shift register has the following capabilities:

1. A *clear* control to clear the register to 0.
2. A *clock* input to synchronize the operations.
3. A *shift-right* control to enable the shift-right operation and the *serial input* and *output* lines associated with the shift right.
4. A *shift-left* control to enable the shift-left operation and the *serial input* and *output* lines associated with the shift left.
5. A *parallel-load* control to enable a parallel transfer and the n input lines associated with the parallel transfer.
6. n parallel output lines.
7. A control state that leaves the information in the register unchanged in response to the clock.

Other shift registers may have only some of the preceding functions, with at least one shift operation. A register capable of shifting in one direction only is a *unidirectional* shift register. One that can shift in both directions is a *bidirectional* shift register. If the register can shift in both directions and has parallel-load capabilities, it is referred to as a *universal shift register*.

The block diagram symbol and the circuit diagram of a four-bit universal

shift register that has all the capabilities just listed are shown in [Fig. 6.7](#). The circuit consists of four *D* flip-flops and four multiplexers. The four multiplexers have two common selection inputs *s1* and *s0*. Input 0 in each multiplexer is selected when *s1s0*=00, input 1 is selected when *s1s0*=01, and similarly for the other two inputs. The selection inputs control the mode of operation of the register according to the function entries in [Table 6.3](#). When *s1s0*=00, the present value of the register is applied to the *D* inputs of the flip-flops. This condition forms a path from the output of each flip-flop into the input of the same flip-flop so that the output *recirculates* to the input in this mode of operation, creating the effect of a suspended clock. The next clock edge transfers into each flip-flop the binary value it held previously, and no change of state occurs. For example, when *s1s0*=01, terminal 1 of the multiplexer inputs has a path to the *D* inputs of the flip-flops. This causes a shift-right operation, with the serial input transferred into flip-flop A3. When *s1s0*=10, a shift-left operation results, with the other serial input going into flip-flop A0. Finally, when *s1s0*=11, the binary information on the parallel input lines is transferred into the register simultaneously during the next clock edge. Note that data enters *MSB_in* for a shift-right operation and enters *LSB_in* for a shift-left operation. *Clear_b* is an active-low signal that clears all of the flip-flops asynchronously.

Table 6.3 *Function Table for the Register of [Fig. 6.7](#)*

Mode Control

s1	s0	Register Operation
0	0	No change
0	1	Shift right

1	0	Shift left
1	1	Parallel load

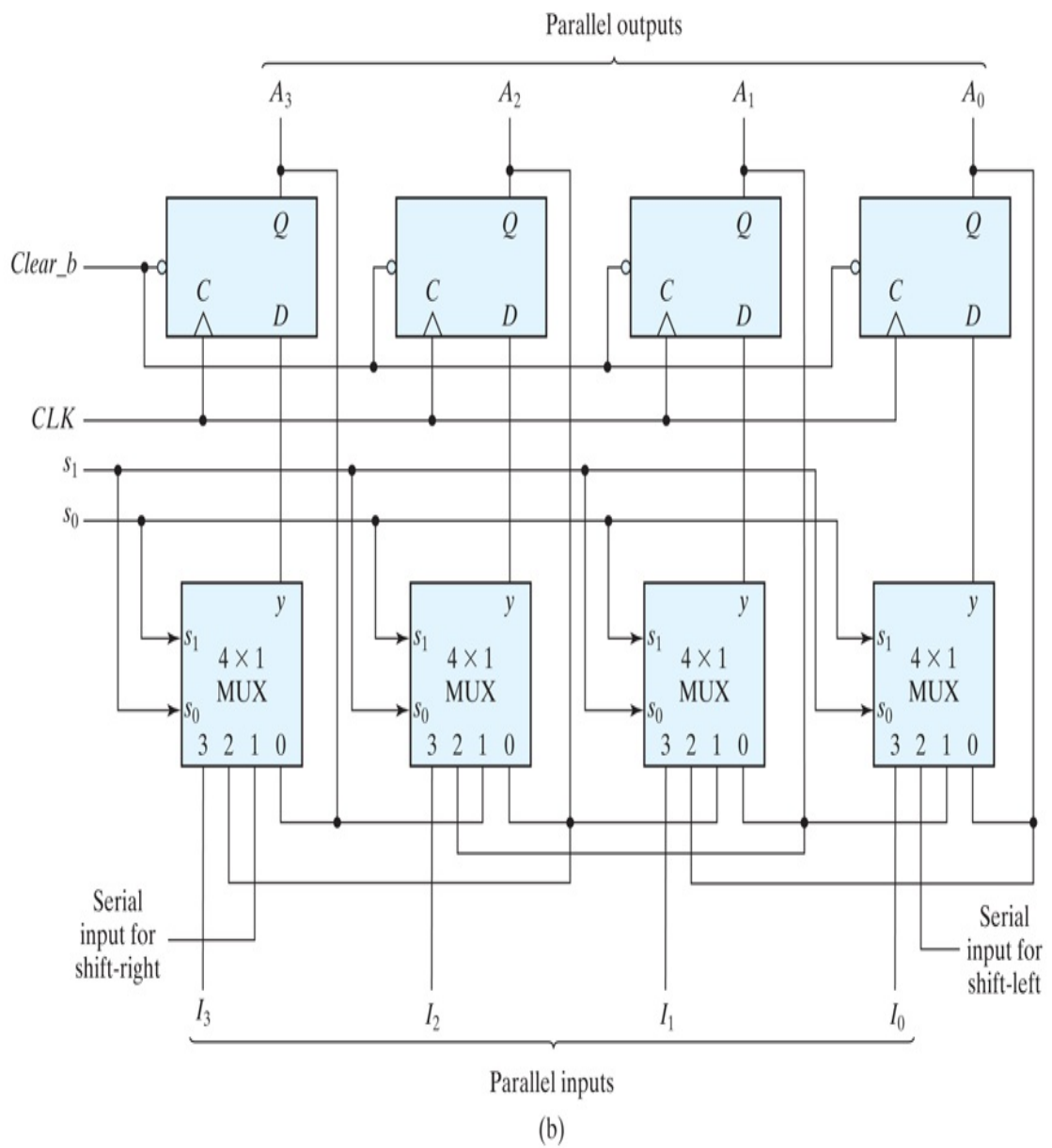
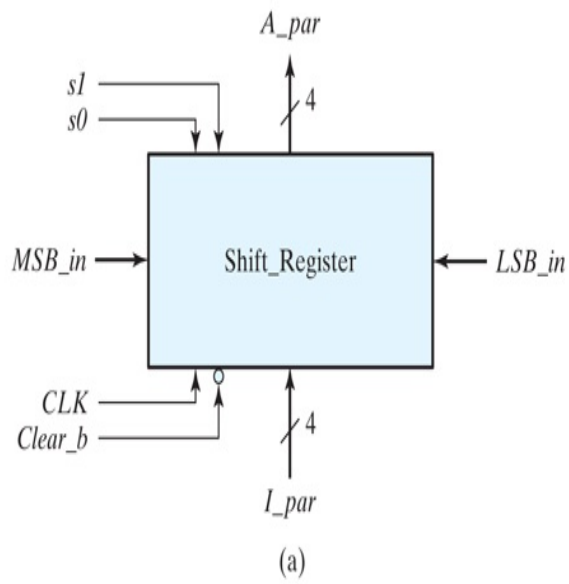


FIGURE 6.7

Four-bit universal shift register

[Description](#)

Shift registers are often used to interface digital systems situated remotely from each other. For example, suppose it is necessary to transmit an n -bit quantity between two points. If the distance is far, it will be expensive to use n lines to transmit the n bits in parallel. It is more economical to use a single line and transmit the information serially, one bit at a time. The transmitter accepts the n -bit data in parallel into a shift register and then transmits the data serially along the common line. The receiver accepts the data serially into a shift register. When all n bits are received, they can be taken from the outputs of the register in parallel. Thus, the transmitter performs a parallel-to-serial conversion of data and the receiver does a serial-to-parallel conversion.

6.3 RIPPLE COUNTERS

A register that goes through a prescribed sequence of states upon the application of input pulses is called a *counter*. The input pulses may be clock pulses, or they may originate from some external source and may occur at a fixed interval of time or at random. The sequence of states may follow the binary number sequence or any other sequence of states. A counter that follows the binary number sequence is called a *binary counter*. An n -bit binary counter consists of n flip-flops and can count in binary from 0 through $2^n - 1$.

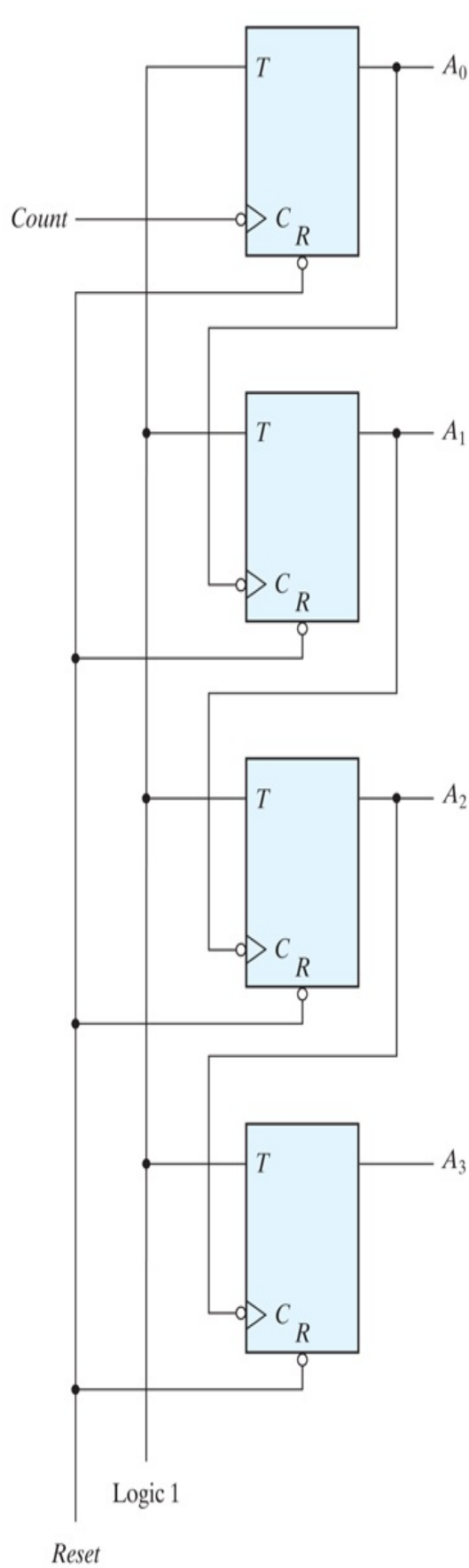
Counters are available in two categories: *ripple* counters and *synchronous* counters. In a ripple counter, a flip-flop output transition serves as a source for triggering other flip-flops. In other words, the *clock* input of some or all flip-flops are triggered, not by the common clock pulses, but rather by the transition that occurs in other flip-flop outputs. In a synchronous counter, the *clock* inputs of all flip-flops receive the common clock. Synchronous counters are presented in the next two sections. First, we present the binary and BCD ripple counters and explain their operation.

Binary Ripple Counter

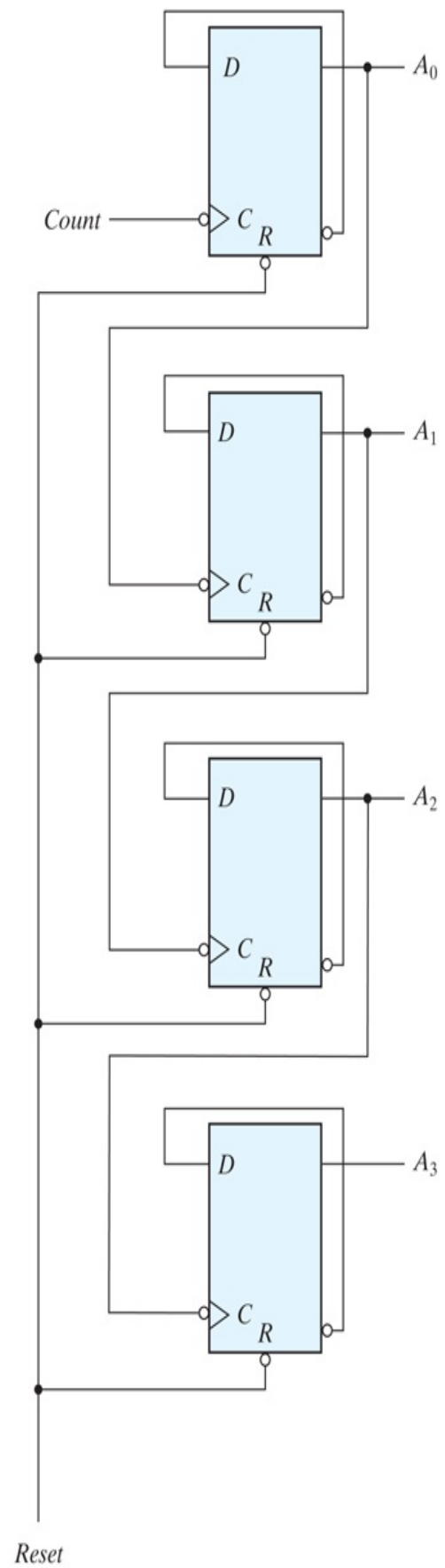
A binary ripple counter consists of a series connection of complementing flip-flops, with the output of each flip-flop connected to the C input of the next higher order flip-flop. The flip-flop holding the least significant bit receives the incoming count pulses. A complementing flip-flop can be obtained from a JK flip-flop with the J and K inputs tied together or from a T flip-flop. A third possibility is to use a D flip-flop with the complement output connected to the D input. In this way, the D input is always the complement of the present state, and the next clock pulse will cause the flip-flop to complement.

The logic diagram of two 4-bit binary ripple counters is shown in [Fig. 6.8](#). The counter is constructed with complementing flip-flops of the T type in part (a) and D type in part (b). The output of each flip-flop is connected to the *clock* input of the next flip-flop in sequence. The flip-flop holding the

least significant bit receives the incoming count pulses. The T inputs of all the flip-flops in (a) are connected to a permanent logic 1, making each flip-flop complement if the signal in its *clock* input goes through a negative transition. The bubble in front of the dynamic indicator symbol next to *clock* indicates that the flip-flops respond to the negative-edge transition of the input. The negative transition occurs when the output of the previous flip-flop to which *the clock* is connected goes from 1 to 0.



(a) With T flip-flops



(b) With D flip-flops

FIGURE 6.8

Four-bit binary ripple counter

Description

To understand the operation of the four-bit binary ripple counter, refer to the first nine binary numbers listed in [Table 6.4](#). The count starts with binary 0 and increments by 1 with each count pulse input. After the count of 15, the counter goes back to 0 to repeat the count. The least significant bit, A0, is complemented with each count pulse input. Every time that A0 goes from 1 to 0, it complements A1. Every time that A1 goes from 1 to 0, it complements A2. Every time that A2 goes from 1 to 0, it complements A3 and so on for any other higher order bits of a ripple counter. For example, consider the transition from count 0011 to 0100. A0 is complemented with the count pulse. Since A0 goes from 1 to 0, it triggers A1 and complements it. As a result, A1 goes from 1 to 0, which in turn complements A2, changing it from 0 to 1. A2 does not trigger A3, because A2 produces a positive transition and the flip-flop responds only to negative transitions. Thus, the count from 0011 to 0100 is achieved by changing the bits one at a time, so the count goes from 0011 to 0010, then to 0000, and finally to 0100. The flip-flops change one at a time in succession, and the signal propagates through the counter in a ripple fashion from one stage to the next.

Table 6.4 *Binary Count Sequence*

A3 A2 A1 A0

0 0 0 0

0 0 0 1

0 0 1 0

0 0 1 1

0 1 0 0

0 1 0 1

0 1 1 0

0 1 1 1

1 0 0 0

A binary counter with a reverse count is called a *binary countdown counter*. In a countdown counter, the binary count is decremented by 1 with every input count pulse. The count of a four-bit countdown counter starts from binary 15 and continues to binary counts 14, 13, 12, ..., 0 and then back to 15. A list of the count sequence of a binary countdown counter shows that the least significant bit is complemented with every count pulse. Any other bit in the sequence is complemented if its previous least significant bit goes from 0 to 1. Therefore, the diagram of a binary countdown counter looks the same as the binary ripple counter in [Fig. 6.8](#), provided that all flip-flops trigger on the positive edge of the clock. (The bubble in the *C* inputs must be absent.) If negative-edge-triggered flip-flops are used, then the *C* input of each flip-flop must be connected to the complemented output of the previous flip-flop. Then, when the true output goes from 0 to 1, the complement will go from 1 to 0 and complement the next flip-flop as required.

BCD Ripple Counter

A decimal counter follows a sequence of 10 states and returns to 0 after the count of 9. Such a counter must have at least four flip-flops to represent each decimal digit, since a decimal digit is represented by a binary code with at least four bits. The sequence of states in a decimal counter is dictated by the binary code used to represent a decimal digit. If the BCD code is used, the sequence of states is as shown in the state diagram of [Fig. 6.9](#). A decimal counter is similar to a binary counter, except that the state after 1001 (the code for decimal digit 9) is 0000 (the code for decimal digit 0).

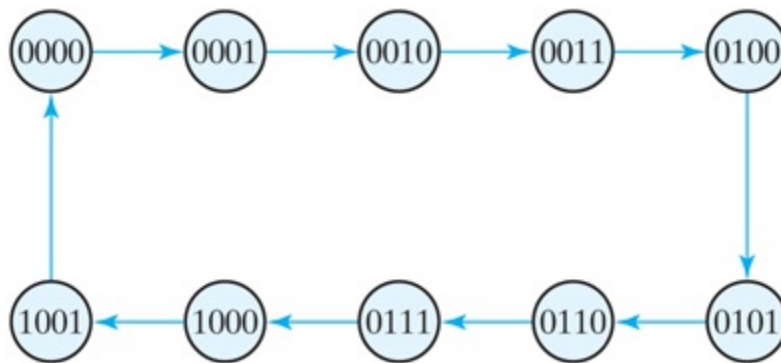


FIGURE 6.9

State diagram of a decimal BCD counter

The logic diagram of a BCD ripple counter using *JK* flip-flops is shown in [Fig. 6.10](#). The four outputs are designated by the letter symbol *Q*, with a numeric subscript equal to the binary weight of the corresponding bit in the BCD code. Note that the output of *Q*₁ is applied to the *C* inputs of both *Q*₂ and *Q*₈ and the output of *Q*₂ is applied to the *C* input of *Q*₄. The *J* and *K* inputs are connected either to a permanent 1 signal or to outputs of other flip-flops.

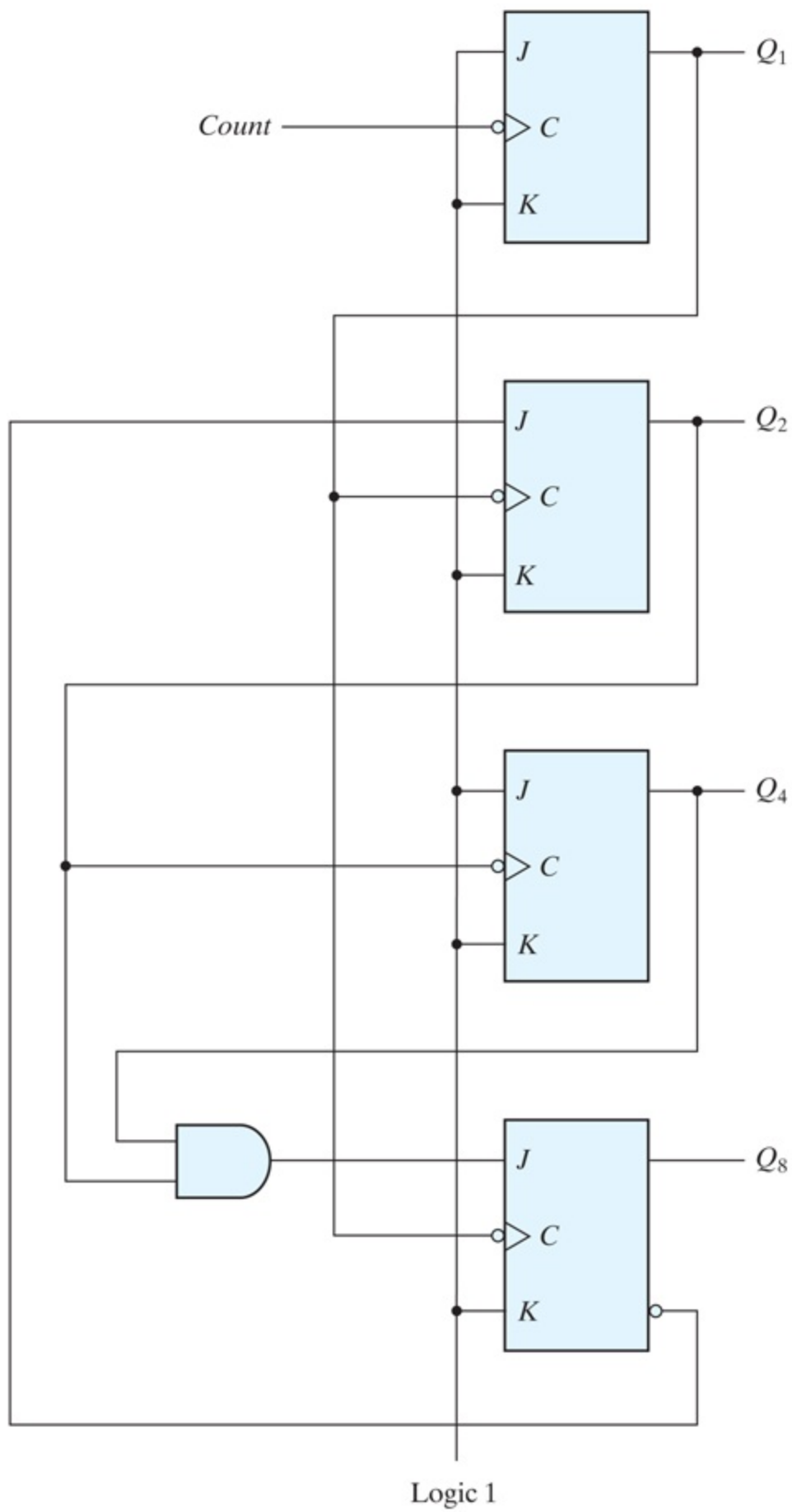


FIGURE 6.10

BCD ripple counter

[Description](#)

A ripple counter is an asynchronous sequential circuit. Its state changes are not synchronized to a common clock. Signals that affect the flip-flop transition depend on the way they change from 1 to 0. The operation of the counter can be explained by a list of conditions for flip-flop transitions. These conditions are derived from the logic diagram and from knowledge of how a *JK* flip-flop operates. Remember that when the *C* input goes from 1 to 0, the flip-flop is set if $J=1$, $K=0$, is cleared if $J=0$ and $K=1$, is complemented if $J=K=1$, and is left unchanged if $J=K=0$.

To verify that these conditions result in the sequence required by a BCD ripple counter, it is necessary to verify that the flip-flop transitions indeed follow a sequence of states as specified by the state diagram of [Fig. 6.9](#). Q_1 changes state after each clock pulse. Q_2 complements every time Q_1 goes from 1 to 0, as long as $Q_8=0$. When Q_8 becomes 1, Q_2 remains at 0. Q_4 complements every time Q_2 goes from 1 to 0. Q_8 remains at 0 as long as Q_2 or Q_4 is 0. When both Q_2 and Q_4 become 1, Q_8 complements when Q_1 goes from 1 to 0. Q_8 is cleared on the next 1-to-0 transition of Q_1 .

The BCD counter of [Fig. 6.10](#) is a *decade* counter, since it counts from 0 to 9. To count in decimal from 0 to 99, we need a two-decade counter. To count from 0 to 999, we need a three-decade counter. Multiple decade counters can be constructed by connecting BCD counters in cascade, one for each decade. A three-decade counter is shown in [Fig. 6.11](#). The inputs to the second and third decades come from Q_8 of the previous decade. When Q_8 in one decade goes from 1 to 0, it triggers the count for the next higher order decade while its own decade goes from 9 to 0.

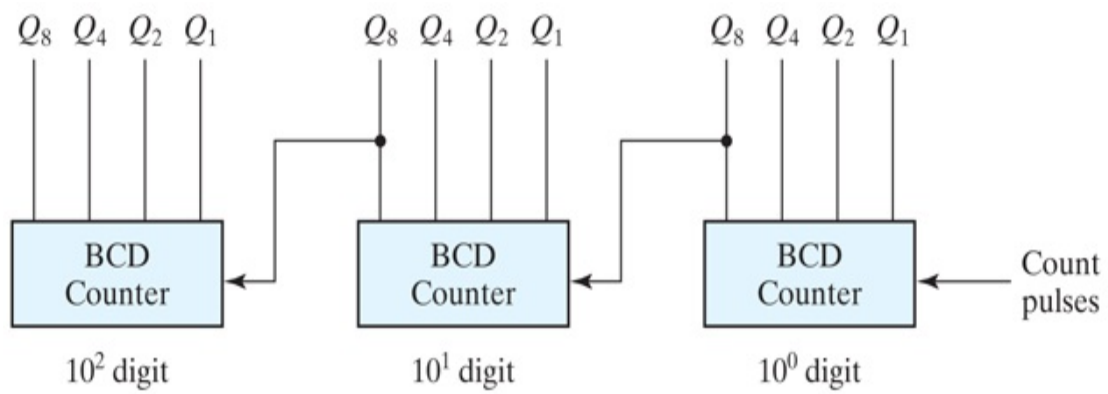


FIGURE 6.11

Block diagram of a three-decade decimal BCD counter

[Description](#)

6.4 SYNCHRONOUS COUNTERS

Synchronous counters are different from ripple counters in that clock pulses are applied to the inputs of all flip-flops. A common clock triggers all flip-flops simultaneously, rather than one at a time in succession as in a ripple counter. The decision whether a flip-flop is to be complemented is determined from the values of the data inputs, such as T or J and K at the time of the clock edge. If $T=0$ or $J=K=0$, the flip-flop does not change state. If $T=1$ or $J=K=1$, the flip-flop complements.

The design procedure for synchronous counters was presented in [Section 5.8](#), and the design of a three-bit binary counter was carried out in conjunction with [Fig. 5.32](#). In this section, we present some typical synchronous counters and explain their operation.

Binary Counter

The design of a synchronous binary counter is so simple that there is no need to go through a sequential logic design process. In a synchronous binary counter, the flip-flop in the least significant position is complemented with every pulse. *A flip-flop in any other position is complemented when all the bits in the lower significant positions are equal to 1.* For example, if the present state of a four-bit counter is $A_3A_2A_1A_0=0011$, the next count is 0100. A_0 is always complemented. A_1 is complemented because the present state of $A_0=1$. A_2 is complemented because the present state of $A_1A_0=11$. However, A_3 is not complemented, because the present state of $A_2A_1A_0=011$, which does not give an all-1's condition.

Synchronous binary counters have a regular pattern of hardware elements and can be constructed with complementing flip-flops and gates. The regular pattern can be seen from the four-bit counter depicted in [Fig. 6.12](#). The C (clock) inputs of all flip-flops are connected to a common clock. The counter is enabled by *Count_enable*. If the enable input is 0, all J and

K inputs are equal to 0 and the clock does not change the state of the counter. The first stage, A_0 , has its J and K inputs equal to 1 if the counter is enabled. The other J and K inputs are equal to 1 only if all previous least significant stages are equal to 1 and the count is enabled. The chain of AND gates generates the required logic for the J and K inputs in each stage. The counter can be extended to any number of stages, with each stage having an additional flip-flop and an AND gate that gives an output of 1 only if all previous flip-flop outputs are 1.

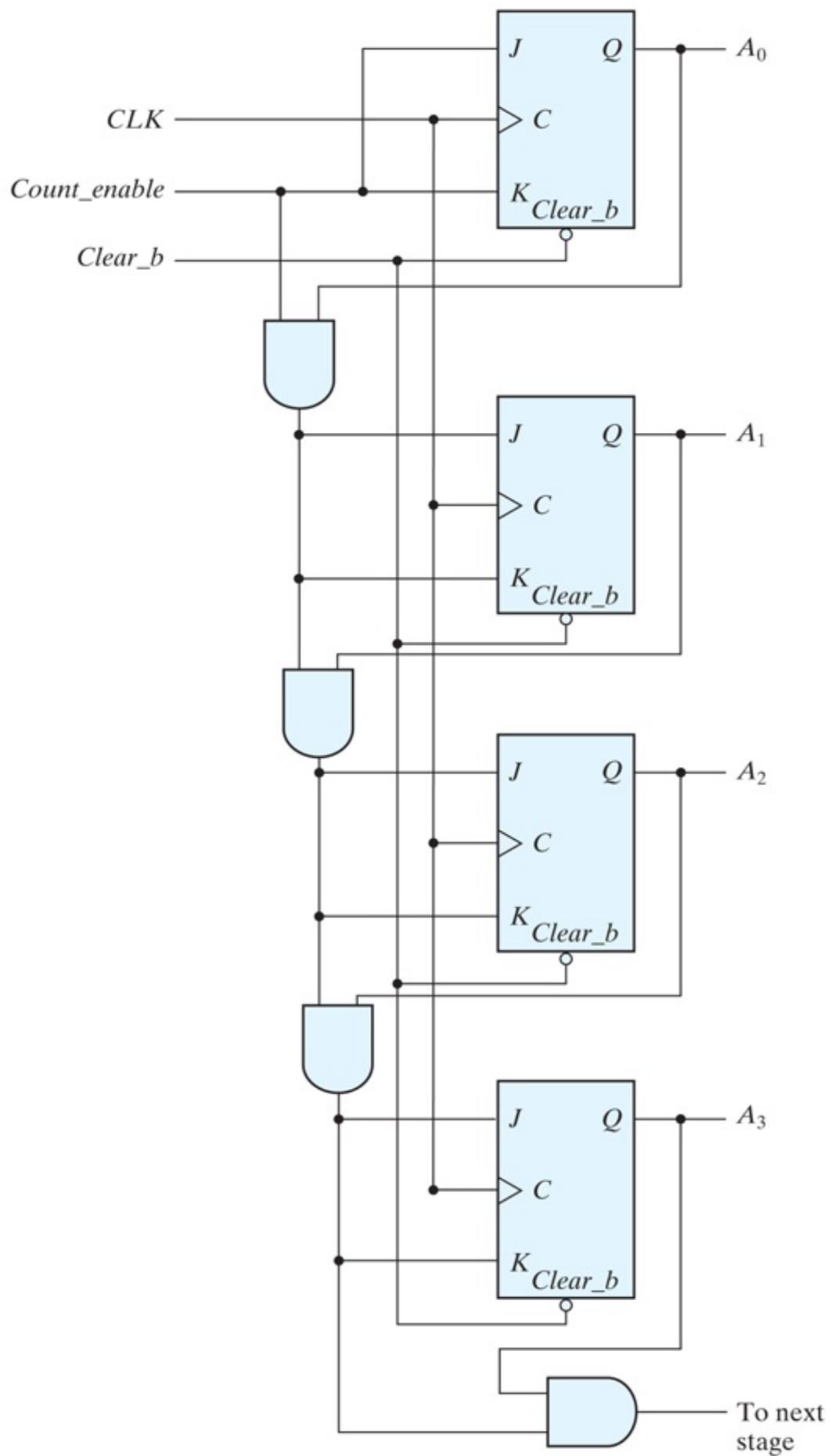


FIGURE 6.12

Four-bit synchronous binary counter

[Description](#)

Note that the flip-flops trigger on the positive edge of the clock. The polarity of the clock is not essential here, but it is with the ripple counter. The synchronous counter can be triggered with either the positive or the negative clock edge. The complementing flip-flops in a binary counter can be of either the *JK* type, the *T* type, or the *D* type with XOR gates. The equivalency of the three types is indicated in [Fig. 5.13](#).

Up–Down Binary Counter

A synchronous countdown binary counter goes through the binary states in reverse order, from 1111 down to 0000 and back to 1111 to repeat the count. It is possible to design a countdown counter in the usual manner, but the result is predictable by inspection of the downward binary count. The bit in the least significant position is complemented with each pulse. *A bit in any other position is complemented if all lower significant bits are equal to 0.* For example, the next state after the present state of 0100 is 0011. The least significant bit is always complemented. The second significant bit is complemented because the first bit is 0. The third significant bit is complemented because the first two bits are equal to 0. But the fourth bit does not change, because not all lower significant bits are equal to 0.

A countdown binary counter can be constructed as shown in [Fig. 6.12](#), except that the inputs to the AND gates must come from the complemented outputs, instead of the normal outputs, of the previous flip-flops. The two operations can be combined in one circuit to form a counter capable of counting either up or down. The circuit of an up–down binary counter using *T* flip-flops is shown in [Fig. 6.13](#). It has an up control input and a down control input. When the *up* input is 1, the circuit counts up, since the *T* inputs receive their signals from the values of the previous normal outputs of the flip-flops. When the *down* input is 1 and the *up* input

is 0, the circuit counts down, since the complemented outputs of the previous flip-flops are applied to the T inputs. When the *up* and *down* inputs are both 0, the circuit does not change state and remains in the same count. When the *up* and *down* inputs are both 1, the circuit counts up. This set of conditions ensures that only one operation is performed at any given time. Note that the *up* input has priority over the *down* input.

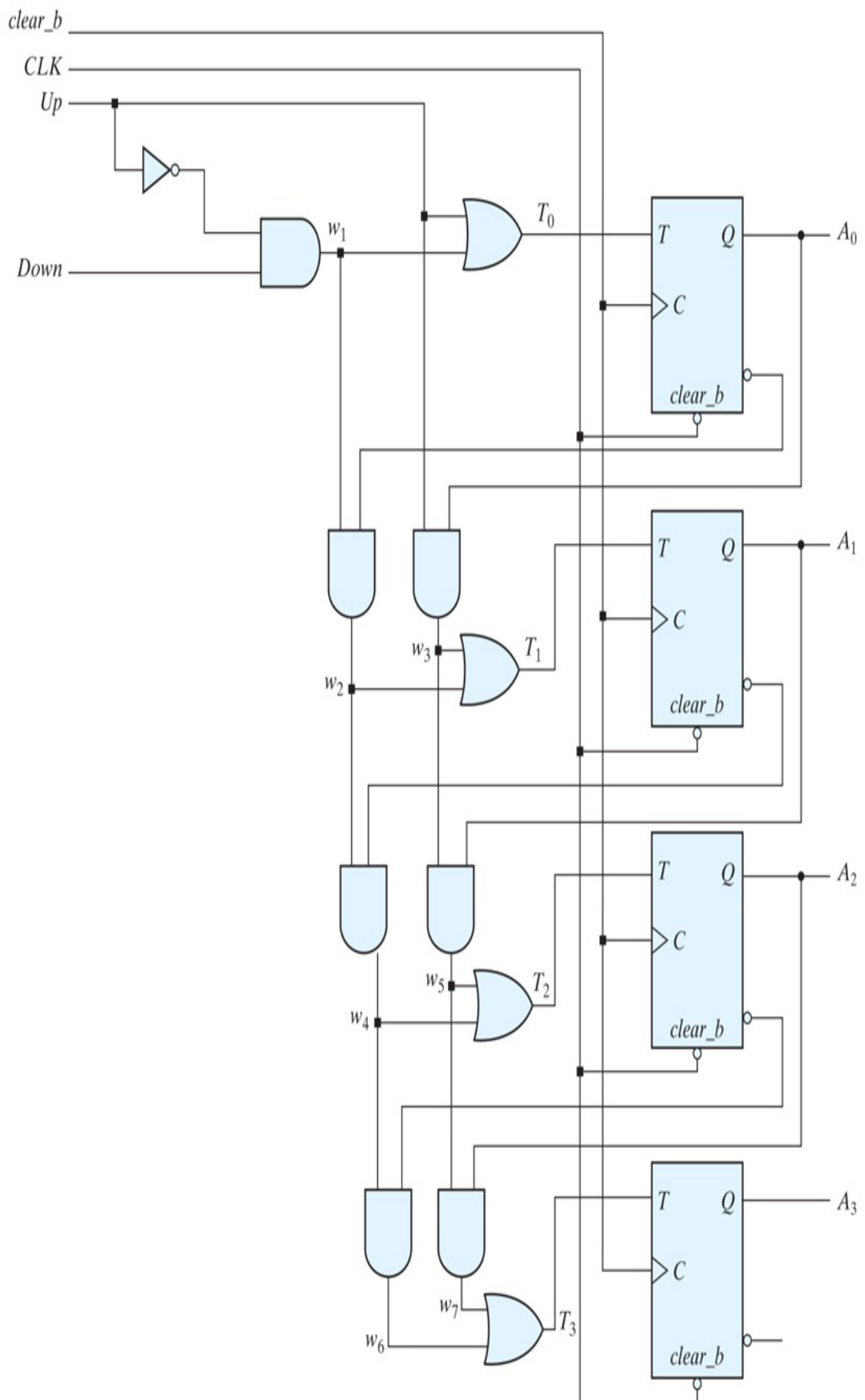


FIGURE 6.13

Four-bit up–down binary counter

[Description](#)

BCD Counter

A BCD counter counts in binary-coded decimal from 0000 to 1001 and back to 0000. Because of the return to 0 after a count of 9, a BCD counter does not have a regular pattern, unlike a straight binary count. To derive the circuit of a BCD synchronous counter, it is necessary to go through a sequential circuit design procedure.

The state table of a BCD counter is listed in [Table 6.5](#). The input conditions for the *T* flip-flops are obtained from the present- and next-state conditions. Also shown in the table is an output *y*, which is equal to 1 when the present state is 1001. In this way, *y* can enable the count of the next-higher significant decade while the same pulse switches the present decade from 1001 to 0000.

Table 6.5 *State Table for BCD Counter*

Present State				Next State				Output	Flip-Flop Inputs			
Q8	Q4	Q2	Q1	Q8	Q4	Q2	Q1	<i>y</i>	TQ8	TQ4	TQ2	TQ1
0	0	0	0	0	0	0	1	0	0	0	0	1
0	0	0	1	0	0	1	0	0	0	0	1	1

0	0	1	0	0	0	1	1	0	0	0	0	1
0	0	1	1	0	1	0	0	0	0	1	1	1
0	1	0	0	0	1	0	1	0	0	0	0	1
0	1	0	1	0	1	1	0	0	0	0	1	1
0	1	1	0	0	1	1	1	0	0	0	0	1
0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	0	0	1	0	0	0	0	1
1	0	0	1	0	0	0	0	1	1	0	0	1

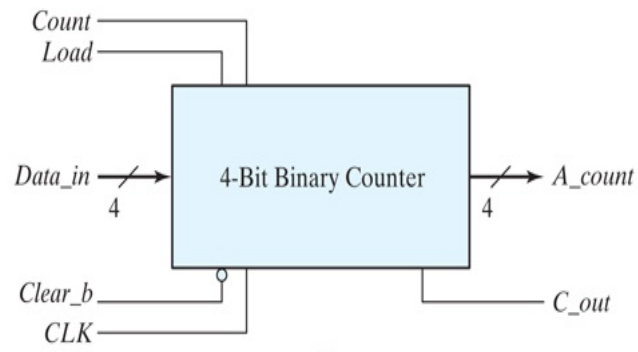
The flip-flop input equations can be simplified by means of maps. The unused states for minterms 10 to 15 are taken as don't-care terms. The simplified functions are

$$TQ1 = 1 \quad TQ2 = Q' \quad TQ4 = Q2Q1 \quad TQ8 = Q8Q1 + Q4Q2Q1 \quad y = Q8Q1$$

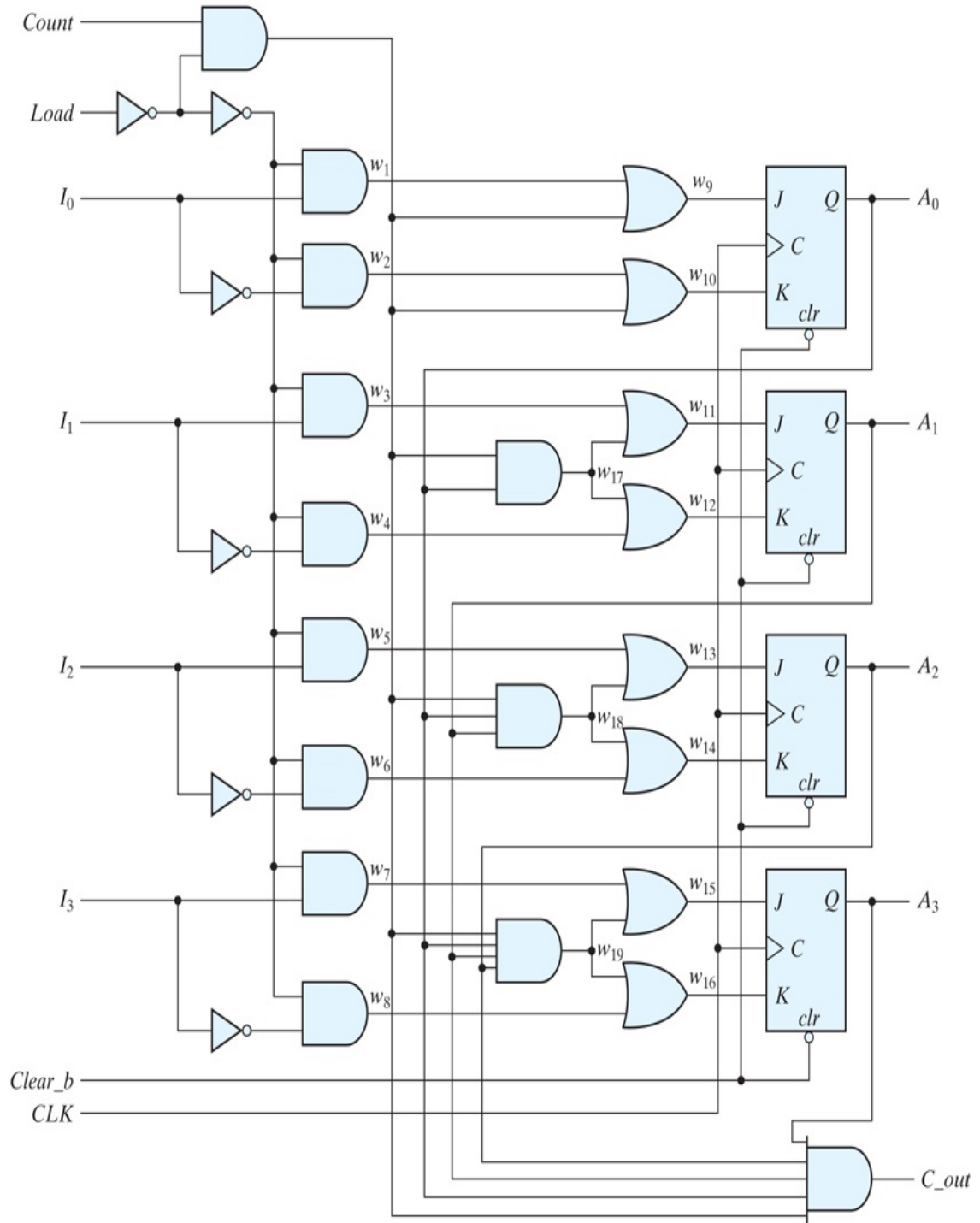
The circuit can easily be drawn with four *T* flip-flops, five AND gates, and one OR gate. Synchronous BCD counters can be cascaded to form a counter for decimal numbers of any length. The cascading is done as in [Fig. 6.11](#), except that output *y* must be connected to the *count* input of the next-higher significant decade.

Binary Counter with Parallel Load

Counters employed in digital systems quite often require a parallel-load capability for transferring an initial binary number into the counter prior to the count operation. [Figure 6.14](#) shows the top-level block diagram symbol and the logic diagram of a four-bit register that has a parallel load capability and can operate as a binary counter. When equal to 1, the input *Load* disables the count operation and causes a transfer of data from the four data inputs into the four flip-flops. If inputs *Load* and *Count* are both 0, clock pulses do not change the state of the register (because *J* and *K* are also both 0).



(a)



(b)

FIGURE 6.14

Four-bit binary counter with parallel load

[Description](#)

The carry output becomes a 1 if all the flip-flops are equal to 1 while *Count* is enabled (i.e., *Count*=1 and *Load*=0). This is the condition for complementing the flip-flop that holds the next significant bit. The carry output is useful for expanding the counter to more than four bits. The speed of the counter is increased when the carry is generated directly from the outputs of all four flip-flops, because the delay to generate the carry bit is reduced. In going from state 1111 to 0000, only one gate delay occurs, whereas four gate delays occur in the AND gate chain shown in [Fig. 6.12](#). Similarly, in the faster counter the output of each flip-flop is directed to an AND gate that receives all previous flip-flop outputs directly instead of connecting the AND gates in a chain.

The operation of the counter is summarized in [Table 6.6](#). The four control inputs—*Clear_b*, *CLK*, *Load*, and *Count*—determine the next state. The *Clear_b* input is active-low, asynchronous and, when equal to 0, causes the counter to be cleared regardless of the presence of clock pulses or other inputs. This relationship is indicated in the table by the X entries in the first row of the table, which symbolize don't-care conditions for the other inputs. The *Clear_b* input must be 1 (de-asserted) for all other operations. With the *Load* and *Count* inputs both at 0, the outputs do not change, even when clock pulses are applied. A *Load* input of 1 causes a transfer from inputs I0–I3 into the register during a positive edge of *CLK* (i.e., the load action is synchronous). The input data are loaded into the register regardless of the value of the *Count* input, because the *Count* input is inhibited when the *Load* input is enabled. The *Load* input must be 0 for the *Count* input to control the operation of the counter.

Table 6.6 *Function Table for the Counter of [Fig. 6.14](#)*

Clear_b	CLK	Load	Count	Function
0	X	X	X	Clear to 0
1	↑	1	X	Load inputs
1	↑	0	1	Count next binary state
1	↑	0	0	No change

A counter with a parallel load can be used to generate any desired count sequence. [Figure 6.15](#) shows two ways in which a counter with a parallel load is used to generate the BCD count. In each case, the *Count* control is set to 1 to enable the count through the *CLK* input. Also, recall that the *Load* control inhibits the count and that the active-low clear action is independent of other control inputs.

The AND gate in [Fig. 6.15\(a\)](#) detects the occurrence of state 1001(910). At this count *Load* is asserted and 0s are loaded into the register at the next active edge of *CLK*, effectively clearing the counter. Then the *Clear_b* input is set to 1 and the *Count* input is set to 1, so the counter is active at all times. As long as the output of the AND gate is 0, each positive-edge clock increments the counter by 1. When the output reaches the count of 1001, both A0 and A3 become 1, making the output of the AND gate equal to 1. This condition asserts the *Load* input; therefore, on the next clock edge the register does not count, but is loaded from its four inputs. Since all four inputs are connected to logic 0, an all-0's value is loaded into the register following the count of 1001. Thus, the circuit goes through the count from 0000(010) through 1001(910) and back to 0000(010), as is required in a BCD counter.

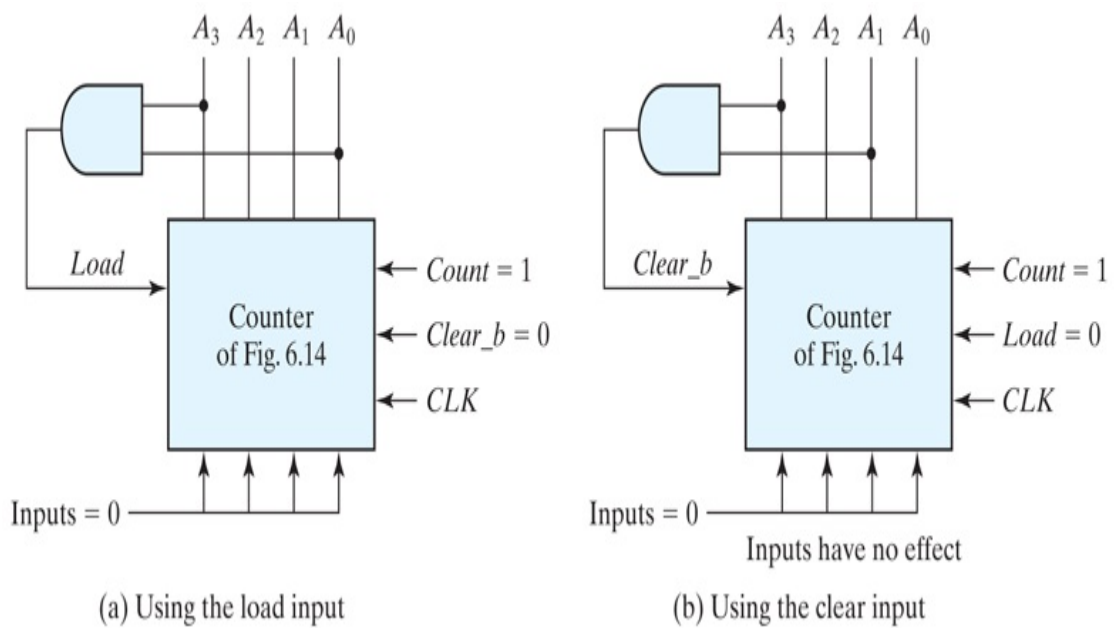


FIGURE 6.15

Two ways to achieve a BCD counter using a counter with parallel load

[Description](#)

In the alternative counter shown in [Fig. 6.15\(b\)](#), the NAND gate detects the count of 1010(1010), but as soon as this count occurs, the register is cleared (asynchronously). The count 1010(1010) has no chance of staying on for any appreciable time, because the register goes immediately to 0. A momentary spike occurs in output A0 as the count goes from 1010(1010) to 1011(1110) and immediately to 0000(010). The spike may be undesirable, and for that reason, this configuration is not recommended. If the counter has a synchronous clear input, it is possible to clear the counter with the clock after an occurrence of the 1001 count.

Practice Exercise 6.3

1. How do a ripple counter and a synchronous counter differ in their behavior?

Answer: All of the flip-flops in a synchronous counter are

synchronized by receiving a common clock pulse; only the first stage of a ripple counter receives a clock pulse. The stages of a synchronous counter are updated simultaneously by a common clock; the stages of a ripple counter are updated one at a time. A synchronous counter operates faster than a ripple counter.

6.5 OTHER COUNTERS

Counters can be designed to generate any desired sequence of states. A divide-by- N counter (also known as a modulo- N counter) is a counter that goes through a repeated sequence of N states. The sequence may follow the binary count or may be any other arbitrary sequence. Counters are used to generate timing signals to control the sequence of operations in a digital system. Counters can also be constructed by means of shift registers. In this section, we present a few examples of nonbinary counters.

Counter with Unused States

A circuit with n flip-flops has 2^n binary states. There are occasions when a sequential circuit uses fewer than this maximum possible number of states. States that are not used in specifying the sequential circuit are not listed in the state table. In simplifying the input equations, the unused states may be treated as don't-care conditions or may be assigned specific next states. It is important to realize that once the circuit is designed and constructed, outside interference during its operation may cause the circuit to enter one of the unused states. In that case, it is necessary to ensure that the circuit eventually goes into one of the valid states so that it can resume normal operation. Otherwise, if the sequential circuit circulates among unused states, there will be no way to bring it back to its intended sequence of state transitions. If the unused states are treated as don't-care conditions, then once the circuit is designed, it must be investigated to determine the effect of the unused states. The next state from an unused state can be determined from the analysis of the circuit after it is designed.

As an illustration, consider the counter specified in [Table 6.7](#). The count has a repeated sequence of six states, with flip-flops B and C repeating the binary count 00, 01, 10, and flip-flop A alternating between 0 and 1 every three counts. The count sequence of the counter is not straight binary, and two states, 011 and 111, are not included in the count. The choice of JK flip-flops results in the flip-flop input conditions listed in the table. Inputs K_B and K_C have only 1's and X's in their columns, so these inputs will always be set to 1. The other flip-flop input equations can be simplified by

using minterms 3 and 7 as don't-care conditions. The simplified equations are

Table 6.7 *State Table for Counter*

Present State			Next State			Flip-Flop Inputs					
A	B	C	A	B	C	JA	KA	JB	KB	JC	KC
0	0	0	0	0	1	0	X	0	X	1	X
0	0	1	0	1	0	0	X	1	X	X	1
0	1	0	1	0	0	1	X	X	1	0	X
1	0	0	1	0	1	X	0	0	X	1	X
1	0	1	1	1	0	X	0	1	X	X	1
1	1	0	0	0	0	X	1	X	1	0	X

$$JA = B \quad KA = B \quad JB = C \quad KB = 1 \quad JC = B' \quad KC = 1$$

The logic diagram of the counter is shown in [Fig. 6.16\(a\)](#). Since there are two unused states, we analyze the circuit to determine their effect. If the circuit happens to be in state 011 because of an error signal, the circuit goes to state 100 after the application of a clock pulse. This action may be determined from an inspection of the logic diagram by noting that when

$B=1$, the next clock edge complements A and clears C to 0, and when $C=1$, the next clock edge complements B . In a similar manner, we can evaluate the next state from present state 111 to be 000.

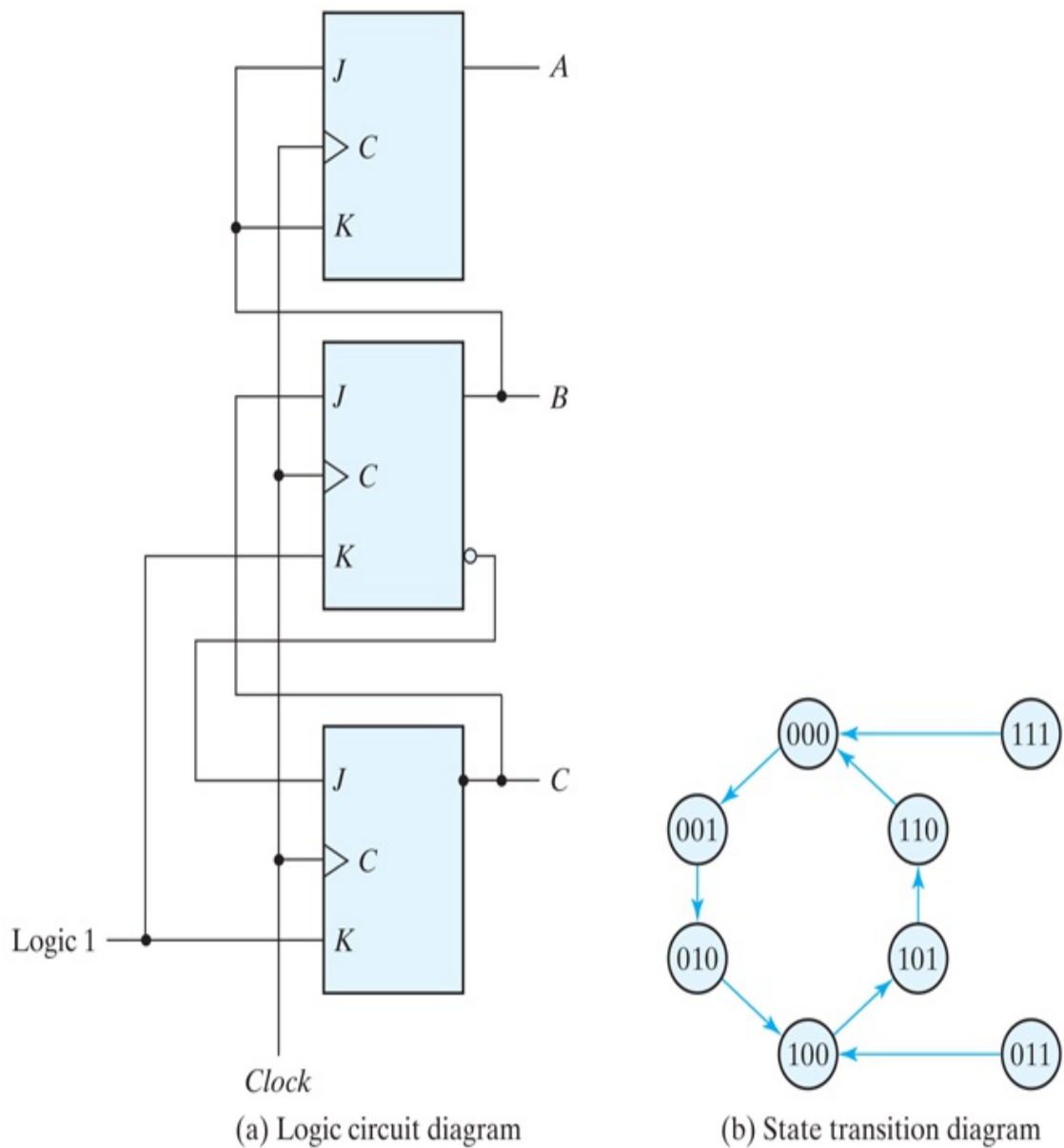


FIGURE 6.16

Counter with unused states

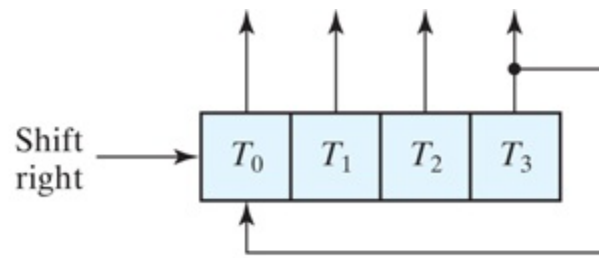
Description

The state diagram including the effect of the unused states is shown in [Fig. 6.16\(b\)](#). If the circuit ever goes to one of the unused states because of

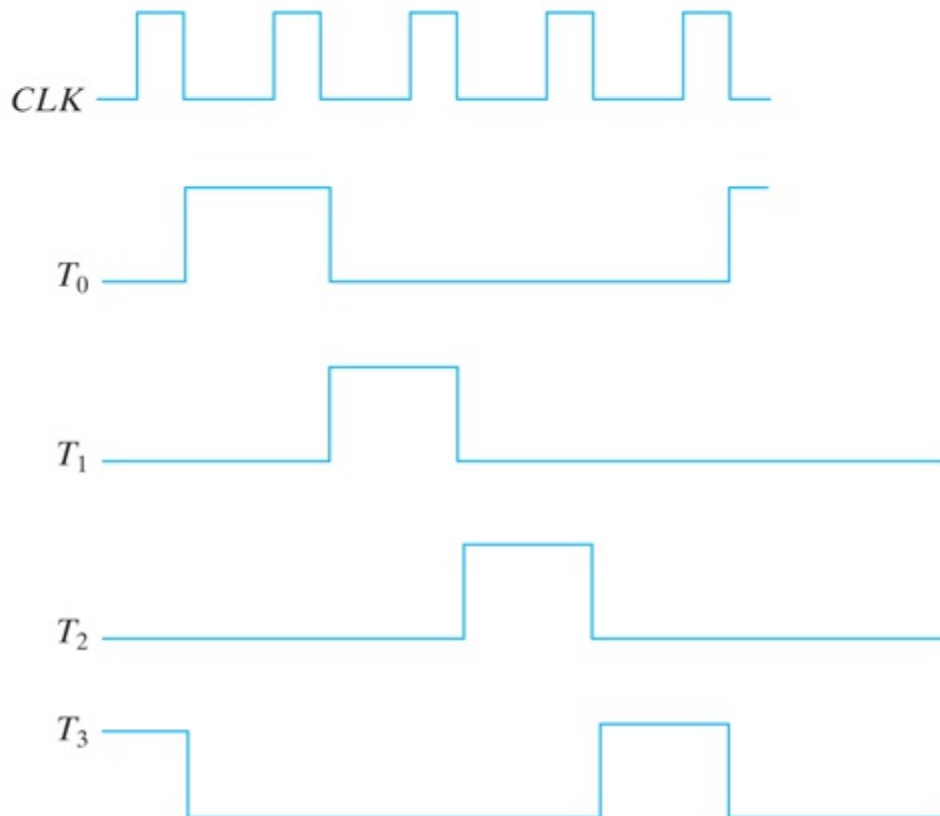
outside interference, the next count pulse transfers it to one of the valid states and the circuit continues to count correctly. Thus, the counter is self-correcting. In a self-correcting counter, if the counter happens to be in one of the unused states, it eventually reaches the normal count sequence after one or more clock pulses. An alternative design could use additional logic to direct every unused state to a specific next state.

Ring Counter

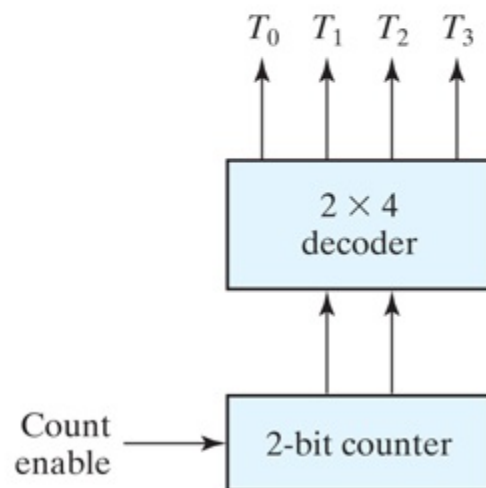
Timing signals that control the sequence of operations in a digital system can be generated by a shift register or by a counter with a decoder. A *ring counter* is a circular shift register with only one flip-flop being set at any particular time; all others are cleared. The single bit is shifted from one flip-flop to the next to produce the sequence of timing signals. [Figure 6.17\(a\)](#) shows a four-bit shift register connected as a 8-4-2-1 ring counter. The initial value of the register is 1000 and requires Preset/Clear flip-flops. The single bit is shifted right with every clock pulse and circulates back from T3 to T0. Each flip-flop is in the 1 state once every four clock cycles and produces one of the four timing signals shown in [Fig. 6.17\(b\)](#). Each output becomes a 1 after the negative-edge transition of a clock pulse and remains 1 during the next clock cycle.



(a) Ring-counter (initial value = 1000)



(b) Sequence of four timing signals



(c) Counter and decoder

FIGURE 6.17

Generation of timing signals

[Description](#)

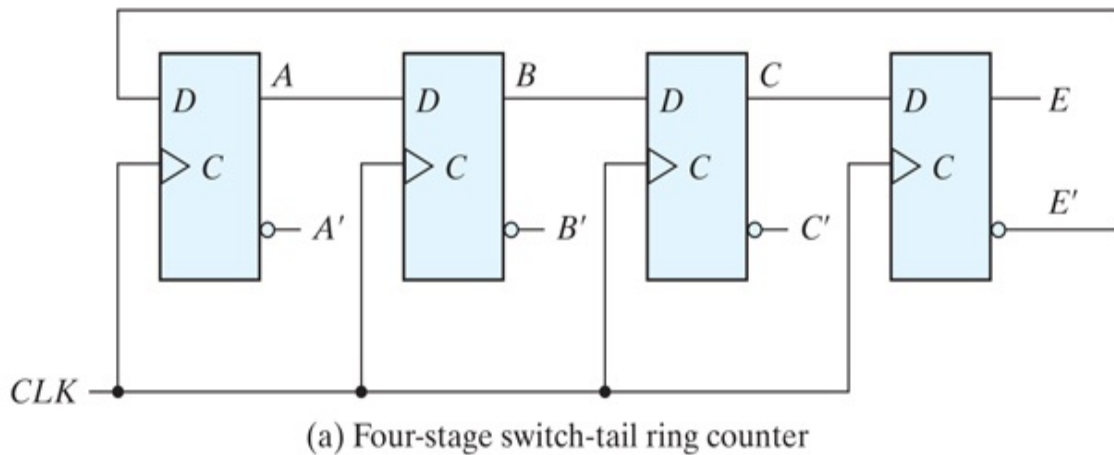
For an alternative design, the timing signals can be generated by a two-bit counter that goes through four distinct states. The decoder shown in [Fig. 6.17\(c\)](#) decodes the four states of the counter and generates the required sequence of timing signals.

To generate $2n$ timing signals, we need either a shift register with $2n$ flip-flops or an n -bit binary counter together with an n -to- $2n$ -line decoder. For example, 16 timing signals can be generated with a 16-bit shift register connected as a ring counter or with a 4-bit binary counter and a 4-to-16-line decoder. In the first case, we need 16 flip-flops. In the second, we need 4 flip-flops and 16 four-input AND gates for the decoder. It is also possible to generate the timing signals with a combination of a shift register and a decoder. That way, the number of flip-flops is less than that in a ring counter, and the decoder requires only two-input gates. This combination is called a *Johnson counter*.

Johnson Counter

A k -bit ring counter circulates a single bit among the flip-flops to provide k distinguishable states. The number of states can be doubled if the shift register is connected as a *switch-tail* ring counter. A switch-tail ring counter is a circular shift register with the complemented output of the last flip-flop connected to the input of the first flip-flop. [Figure 6.18\(a\)](#) shows such a shift register. The circular connection is made from the complemented output of the rightmost flip-flop to the input of the leftmost flip-flop. The register shifts its contents once to the right with every clock pulse, and at the same time, the complemented value of the E flip-flop is transferred into the A flip-flop. Starting from a cleared state, the switch-tail ring counter goes through a sequence of eight states, as listed in [Fig. 6.18\(b\)](#). In general, a k -bit switch-tail ring counter will go through a sequence of $2k$ states. Starting from all 0's, each shift operation inserts 1's

from the left until the register is filled with all 1's. In the next sequences, 0's are inserted from the left until the register is again filled with all 0's.



Sequence number	Flip-flop outputs				AND gate required for output
	A	B	C	E	
1	0	0	0	0	$A'E'$
2	1	0	0	0	AB'
3	1	1	0	0	BC'
4	1	1	1	0	CE'
5	1	1	1	1	AE
6	0	1	1	1	$A'B$
7	0	0	1	1	$B'C$
8	0	0	0	1	$C'E$

(b) Count sequence and required decoding

FIGURE 6.18

Construction of a Johnson counter

[Description](#)

A Johnson counter is a k -bit switch-tail ring counter with $2k$ decoding gates to provide outputs for $2k$ timing signals. The decoding gates are not shown in [Fig. 6.18](#), but are specified in the last column of the table. The eight AND gates listed in the table, when connected to the circuit, will complete the construction of the Johnson counter. Since each gate is enabled during one particular state sequence, the outputs of the gates

generate eight timing signals in succession.

The decoding of a k -bit switch-tail ring counter to obtain $2k$ timing signals follows a regular pattern. The all-0's state is decoded by taking the complement of the two extreme flip-flop outputs. The all-1's state is decoded by taking the normal outputs of the two extreme flip-flops. All other states are decoded from an adjacent 1, 0 or 0, 1 pattern in the sequence. For example, sequence 7 has an adjacent 0, 1 pattern in flip-flops B and C . The decoded output is then obtained by taking the complement of B and the normal output of C , or $B'C$.

One disadvantage of the circuit in [Fig. 6.18\(a\)](#) is that if it finds itself in an unused state, it will persist in moving from one invalid state to another and never find its way to a valid state. The difficulty can be corrected by modifying the circuit to avoid this undesirable condition. One correcting procedure is to disconnect the output from flip-flop B that goes to the D input of flip-flop C and instead enable the input of flip-flop C by the function

$$DC = (A + C)B$$

where DC is the flip-flop input equation for the D input of flip-flop C .

Johnson counters can be constructed for any number of timing sequences. The number of flip-flops needed is one-half the number of timing signals. The number of decoding gates is equal to the number of timing signals, and only two-input gates are needed.

6.6 HDL MODELS OF REGISTERS AND COUNTERS

Registers and counters can be described by an HDL at either the behavioral or the structural level. Behavioral modeling describes only the operations of the register, as prescribed by a function table, without a preconceived structure. A structural-level description shows the circuit in terms of a collection of components such as gates, flip-flops, and multiplexers. The various components are instantiated and connected to form a hierarchical description of the design similar to a representation of a multilevel logic diagram. The examples in this section will illustrate both types of descriptions. Both are useful. When a machine is complex, a hierarchical description creates a physical partition of the machine into simpler and more easily described units.

Shift Register

The universal shift register presented in [Section 6.2](#) is a bidirectional shift register with a parallel load. The four clocked operations that are performed with the register are specified in [Table 6.3](#). The register also can be cleared asynchronously. Our chosen name for a behavioral description of the four-bit universal shift register shown in [Fig. 6.7\(a\)](#), the name *Shift_Register_4_beh*, signifies the behavioral model of the internal detail of the top-level block diagram symbol and distinguishes that model from a structural one. The behavioral model is presented in [HDL Example 6.1](#), and the structural model is given in [HDL Example 6.2](#).

The top-level block diagram symbol in [Fig. 6.7\(a\)](#) indicates that the four-bit universal shift register has a *CLK* input, a *Clear_b* input, two selection inputs (*s1*, *s0*), two serial inputs (*shift_left*, *shift_right*), for controlling the shift register, two serial datapath inputs (*MSB_in* and *LSB_in*), a four-bit parallel input (*I_par*), and a four-bit parallel output (*A_par*). The elements of vector *I_par*[3: 0] correspond to the bits *I3*, . . . , *I0* in [Fig. 6.7](#), and similarly for *A_par*[3: 0]. The **always** block describes the five operations that can be performed with the register. The *Clear_b* input clears the

register asynchronously with an active-low signal. *Clear_b* must be high for the register to respond to the positive edge of the clock. The four clocked operations of the register are determined from the values of the two select inputs in the **case** statement. (*s1* and *s0* are concatenated into a two-bit vector and are used as the expression argument of the **case** statement.) The shifting operation is specified by the concatenation of the serial input and three bits of the register. For example, the statement

```
A_par <= {MSB_in, A_par [3:1]}
```

specifies a concatenation of the serial data input for a shift right operation (*MSB_in*) with bits *A_par*[3: 1] of the output data bus. A reference to a contiguous range of bits within a vector is referred to as a *part select*. The four-bit result of the concatenation is transferred to register *A_par* [3: 0] when the clock pulse triggers the operation. This transfer produces a shift-right operation and updates the register with new information. The shift operation overwrites the contents of *A_par*[0] with the contents of *A_par*[1]. Note that only the functionality of the circuit has been described, irrespective of any particular hardware. A synthesis tool would create a netlist of ASIC cells to implement the shift register in the structure of [Fig. 6.7\(b\)](#).

HDL Example 6.1 (Universal Shift Register—Behavioral Model)

Verilog

```
// Behavioral description of a 4-bit universal shift register
// Fig. 6.7 and Table 6.3
module Shift_Register_4_beh (                                     //
    output reg    [3: 0]  A_par,                                //
    input  [3: 0]  I_par,                                       //
    input  s1, s0,                                             //
    MSB_in, LSB_in,                                           //
    CLK, Clear_b                                             //
);

always @ (posedge CLK, negedge Clear_b)                        //
    if (Clear_b == 0) A_par <= 4'b000;
```

```

else
  case ({s1, s0})
    2'b00: A_par <= A_par;           // No change
    2'b01: A_par <= {MSB_in, A_par[3: 1]}; // Shift right
    2'b10: A_par <= {A_par[2: 0], LSB_in}; // Shift left
    2'b11: A_par <= I_par;           // Parallel loa
  endcase
endmodule

```

Note: *A_par* is a variable of type **reg**, so it retains its value until it is assigned a new value by an assignment statement. Consider the following alternative **case** statement for the shift register model:

```

case ({s1, s0})
  //2'b00: A_par <= A_par;           // No change
  2'b01: A_par <= {MSB_in, A_par [3: 1]}; // Shift righ
  2'b10: A_par <= {A_par [2: 0], LSB_in}; // Shift left
  2'b11: A_par <= I_par;           // Parallel l
endcase

```

Without the case item 2'b00, the **case** statement would not find a match between { s1, s0 } and the case items, so register *A_par* would be left unchanged.

VHDL

```

entity Shift_Register_4_beh_vhdl is
  port (A_par: out Std_Logic_Vector (3 downto 0);
        I_par: in Std_Logic_Vector (3 downto 0);
        s1, s0, MSB_in, LSB_in, CLK, Clear_b: in Std_Logic);
end Shift_Register_4_beh_vhdl;

```

```

architecture Behavioral of Shift_Register_4_beh_vhdl
begin
  process (CLK, Clear_b) begin
    if (Clear_b'event and Clear_b = 0) then A_par <= '0000';
    else case (s1 & s0) is
      when 0 => A_par <= A_par;
      when 1 => A_par <= MSB_in & A_par(3:1);
      when 2 => A_par <= A_par(2: 0) & LSB_in;
      when 3 => A_par <= I_par;
    end case;
  end process;
end Behavioral;

```

HDL Example 6.2 (Universal Shift Register—Structural Model)

Verilog

A structural model of the universal shift register can be described by referring to the logic diagram of [Fig. 6.7\(b\)](#). It shows that the register has four multiplexers and four *D* flip-flops. A mux and flip-flop together are modeled as a stage of the shift register. The stage is a structural model, too, with an instantiation and interconnection of a module for a mux and another for a *D* flip-flop. For simplicity, the lowest-level modules of the structure are behavioral models of the multiplexer and flip-flop. Attention must be paid to the details of connecting the stages correctly. The structural description of the 4-bit universal shift register is shown below. The top-level module declares the inputs and outputs and then instantiates four copies of a stage of the register. The four instantiations specify the interconnections between the four stages and provide the detailed construction of the register as specified in the logic diagram. The behavioral description of the flip-flop uses a single edge-sensitive cyclic behavior (an **always** block). The assignment statements use the nonblocking assignment operator (`<=`) the model of the mux employs a single level-sensitive behavior, and the assignments use the blocking assignment operator (`=`).

```
// Structural description of a 4-bit universal shift register (
module Shift_Register_4_str (                                // V2001, 2005
    output [3: 0]  A_par,                                     // Parallel out
    input  [3: 0]  I_par,                                     // Parallel inp
    input          s1, s0,                                   // Mode select
    input          MSB_in, LSB_in, CLK, Clear_b              // Serial input
);

// bus for mode control
wire [1:0]  select = {s1, s0};

// Instantiate the four stages
stage ST0 (A_par[0], A_par[1], LSB_in, I_par[0], A_par[0], sel
stage ST1 (A_par[1], A_par[2], A_par[0], I_par[1], A_par[1], s
stage ST2 (A_par[2], A_par[3], A_par[1], I_par[2], A_par[2], s
stage ST3 (A_par[3], MSB_in, A_par[2], I_par[3], A_par[3], sel
```

endmodule

// One stage of shift register

module stage (i0, i1, i2, i3, Q, select, CLK, Clr_b);

input i0, // circulation bit selection
 i1, // data from left neighbor or serial input for
 i2, // data from right neighbor or serial input for
 i3; // data from parallel input

output Q;

input [1: 0] select; // stage mode control bus

input CLK, Clr_b; // Clock, Clear for flip-flops

wire mux_out;

// instantiate mux and flip-flop

wire Clr = ~Clr_b // Flip-flop has active-high clear sign
 // but circuit has active-low clear acti

 Mux_4x1 M0 (mux_out, i0, i1, i2, i3, select);

 D_flip_flop M1 (Q, mux_out, CLK, Clr);

endmodule

 // 4x1 multiplexer // behavioral model

module Mux_4x1 (mux_out, i0, i1, i2, i3, select);

output mux_out;

input i0, i1, i2, i3;

input [1: 0] select;

reg mux_out;

always @ (select, i0, i1, i2, i3)

case (select)

 2'b00: mux_out = i0;

 2'b01: mux_out = i1;

 2'b10: mux_out = i2;

 2'b11: mux_out = i3;

endcase

endmodule

// Behavioral model of D flip-flop

module D_flip_flop (Q, D, CLK, Clr);

output Q;

input D, CLK, Clr;

reg Q;

always @ (posedge CLK, posedge Clr)

if (Clr) Q <= 1'b0; **else** Q <= D;

endmodule

VHDL

entity Mux_4x1 **is**

port (mux_out: **out** Std_Logic; i0, i1, i2, i3: **in** Std_Logic;
 select: **in** Std_Logic (1 **downto** 0));

```

end Mux_4x1; <= i0;

architecture Behavioral of Mux_4xa is
begin case select is
  when 0 => mux_out <= i0;
  when 1 => mux_out <= i1;
  when 2 => mux_out <= i2;
  when 3 => mux_out <= i3;
end case;
end Behavioral;

entity D_flip_flop is
  port (Q: out Std_Logic;, CLK, Clr: in Std_Logic);
end D_flip_flop;

  architecture Behavioral of D_flip_flop) is
begin
  process (CLK, Clr) begin
    if Clr'event and Clr = 1 then Q <= 0;
    else if CLK'event and CLK = 1 then Q <= Data;
    end if;
  end process
end Behavioral;

entity stage is
  port (i0, i1, i2, i3: in Std_Logic; Q: out Std_Logic;
    select: in Std_Logic_Vector (1 downto 0); CLK, Clr: in Std_Logic);
end stage;

architecture Structural of stage is
  signal Clr: Std_Logic;
  component Mux_4x1 port (mux_out: out Std_Logic; i0, i1, i2, i3
in Std_Logic_Vector (1 downto 0)); end component;
  component D_flip_flop port (Q: out Std_Logic;, CLK, Clr: in Std_Logic);
begin
  Clr <= not Clr_b          // Flip-flop has active-high clear sign
                          // but circuit has active-low clear act

  M0: Mux_4x1 port map (mux_out, i0, i1, i2, i3, select);
  M1: D_flip_flop port map (Q, mux_out, CLK, Clr);
end Structural;

entity Shift_Register_4_str_vhdl is
  port (A_par: out Std_Logic_Vector (3 downto 0);
    I_par: in Std_Logic_Vector (3 downto 0);
    s1, s0, MSB_in, LSB_in, CLK, Clear_b: in Std_Logic);
end Shift_Register_4_str_vhdl;

architecture Structural of Shift_Register_4_vhdl is
  signal select = s1 & s0;
  signal Clr = not Clr_b;
  component stage port (i0, i1, i2, i3, select, CLK, Clear_b: in

```

```
component;
```

```
begin
```

```
ST0: stage port map (A_par[0], A_par[1], LSB_in, I_par[0], A_p  
ST1: stage port map (A_par[1], A_par[2], A_par[0], I_par[1], A  
ST2: stage port map (A_par[2], A_par[3], A_par[1], I_par[2], A  
ST3: stage port map (A_par[3], MSB_in, A_par[2], I_par[3], A_p  
end Structural;
```

The above examples presented two descriptions of a universal shift register to illustrate the different styles for modeling a digital circuit. A simulation should verify that the models have the same functionality. In practice, a designer develops only the behavioral model, which is then synthesized. The function of the synthesized circuit can be compared with the behavioral description from which it was compiled. Eliminating the need for the designer to develop a structural model produces a huge improvement in the efficiency of the design process.

Synchronous Counter

The following HDL examples present *Binary_Counter_4_Par_Load*, a behavioral model of the synchronous counter with a parallel load from [Fig. 6.14](#). *Count*, *Load*, *CLK*, and *Clear_b* are inputs that determine the operation of the counter according to the function specified in [Table 6.6](#). The counter has four data inputs, four data outputs, and a carry output. The internal data lines (*I3*, *I2*, *I1*, *I0*) are bundled as *Data_in[3: 0]* in the behavioral model. Likewise, the register that holds the bits of the count (*A3*, *A2*, *A1*, *A0*) is *A_count[3: 0]*. It is good practice to have identifiers in the HDL model of a circuit correspond exactly to those in the documentation of the model. That is not always feasible, however, if the circuit-level identifiers are those found in a handbook, for they are often short and cryptic and do not exploit the text that is available with an HDL. The top-level block diagram symbol in [Fig. 6.14\(a\)](#) serves as an interface between the names used in a circuit diagram and the expressive names that can be used in the HDL model. The carry output *C_out* is generated by a combinational circuit and is specified with a continuous assignment in the Verilog model, and signal assignment statement in the VHDL model. *C_out* = 1 when the count reaches 15 and the counter is in the count state. Thus, *C_out* = 1 if *Count*=1, *Load*=0, and *A*=1111; otherwise *C_out* = 0.

HDL Example 6.3 (Synchronous Counter)

Verilog

The **always** block specifies the operation to be performed in the register, depending on the values of *Clear_b*, *Load*, and *Count*. A 0 (active-low signal) at *Clear_b* resets *A* to 0. Otherwise, if *Clear_b* = 1 one out of three operations is triggered by the positive edge of the clock. The **if**, **else if**, and **else** statements establish a precedence among the control signals *Clear*, *Load*, and *Count* corresponding to the specification in [Table 6.6](#). *Clear_b* overrides *Load* and *Count*; *Load* overrides *Count*. A synthesis tool will produce the circuit of [Fig. 6.14\(b\)](#) from the behavioral model.

```
// Four-bit binary counter with parallel load (V2001, 2005)
// See Figure 6.14 and Table 6.6
module Binary_Counter_4_Par_Load (
    output reg [3: 0]      A_count,           // Data
    output                C_out,             // Output carry
    input [3: 0]          Data_in,          // Data input
    input                Count,              // Active high to count
    input                Load,              // Active high to load
    input                CLK,                // Positive-edge sensit
    input                Clear_b             // Active low
);

    assign C_out = Count && (~Load) && (A_count == 4'b1111);
    always @ (posedge CLK, negedge Clear_b)
        if (~Clear_b) A_count <= 4'b0000;
        else if (Load) A_count <= Data_in;
        else if (Count) A_count <= A_count + 1'b1;
        else          A_count <= A_count; // Redundant statement
endmodule
```

VHDL

```
entity Binary_Counter_4_Par_Load is
    port (A_count: out Std_Logic_Vector (3 downto 0); C_out: out
          Data_in: in Std_Logic_Vector (3 downto 0);
          Count, Load, CLK, Clear_b: in Std_Logic);
```

```

end Binary_Counter_4_Par_Load;

architecture Behavioral of Binary_Counter_4_Par_Load is
begin
    C_out <= Count and (not Load) when A_count = '1111';
    process (CLK, Clear_b) begin
        if (not Clear_b) then A_count <= '0000';
        else if (Load) then A_count <= Data_in;
        else if (Count = 1) then A_count <= A_count + '0001';
        else A_count <= A_count; // Redundant statement
        end process;
    end Behavioral;

```

Ripple Counter

The structural description of a four-bit ripple counter is shown in [HDL Example 6.4](#). The top structural block instantiates four internally complementing flip-flops defined as *Comp_D_flip_flop* (*Q*, *CLK*, *Reset*). The clock (input *CLK*) of the first flip-flop is connected to the external control signal, *Count*. (*Count* replaces the *CLK* input of the first flip-flop.) The clock input of the second flip-flop is connected to the output of the first. (*A0* replaces *CLK* in the port of the second flip-flop.) Similarly, the clock of each of the other flip-flops is connected to the output of the previous flip-flop. In this way, the flip-flops are chained together to create a ripple counter as shown in [Fig. 6.8\(b\)](#).

The second module describes a complementing flip-flop with delay. The circuit of a complementing flip-flop is constructed by connecting the complement output to the *D* input. A reset input is included with the flip-flop in order to be able to initialize the counter; otherwise the simulator would assign the unknown value (x) to the output of the flip-flop and produce useless results.

HDL Example 6.4 (Ripple Counter)

Verilog

The flip-flop is assigned a delay of two time units from the time that the clock is applied to the time that the flip-flop complements its output. The delay is specified by the statement `Q <= #2 ~Q`. Notice that the delay operator is placed to the right of the nonblocking assignment operator. This form of delay, called *intra-assignment delay*, has the effect of postponing the assignment of the complemented value of `Q` to `Q`. The effect of modeling the delay will be apparent in the simulation results. This style of modeling might be useful in simulation, but it is to be avoided when the model is to be synthesized. The results of synthesis depend on the characteristics of the ASIC cell library that is accessed by the tool, not on any propagation delays that might appear within the model that is to be synthesized.

```
// Ripple counter (see Fig. 6.8(b))
'timescale 1 ns / 100 ps
module Ripple_Counter_4bit (A3, A2, A1, A0, Count, Reset);
    output A3, A2, A1, A0;
    input Count, Reset;
    // Instantiate complementing flip-flop
    Comp_D_flip_flop F0 (A0, Count, Reset);
    Comp_D_flip_flop F1 (A1, A0, Reset);
    Comp_D_flip_flop F2 (A2, A1, Reset);
    Comp_D_flip_flop F3 (A3, A2, Reset);
endmodule
// Complementing flip-flop with delay
// Input to D flip-flop = Q'
module Comp_D_flip_flop (Q, CLK, Reset);
    output Q;
    input CLK, Reset;
    reg Q;
    always @ (negedge CLK, posedge Reset)
        if (Reset) Q <= 1'b0;
        else Q <= #2 ~Q; // intra-assignment delay
endmodule
// Stimulus for testing four-bit ripple counter
module t_Ripple_Counter_4bit;
    reg Count;
    reg Reset;
    wire A0, A1, A2, A3;
    // Instantiate ripple counter
    Ripple_Counter_4bit M0 (A3, A2, A1, A0, Count, Reset);
    always
        #5 Count = ~Count;
    initial
        begin
            Count = 1'b0;
            Reset = 1'b1;
            #4 Reset = 1'b0;
```

```
end  
  
initial #170 $finish;  
  
endmodule
```

The testbench module in [HDL Example 6.4](#) provides a stimulus for simulating and verifying the functionality of the ripple counter. The **always** statement generates a free-running clock with a cycle of 10 time units. The flip-flops trigger on the negative edge of the clock, which occurs at $t=10, 20, 30$, and every 10 time units thereafter. The waveforms obtained from this simulation are shown in [Fig. 6.19](#). The control signal *Count* goes negative every 10 ns. *A0* is complemented with each negative edge of *Count*, but is delayed by 2 ns. Each flip-flop is complemented when its previous flip-flop goes from 1 to 0. After $t=80$ ns, all four flip-flops complement because the counter goes from 0111 to 1000. Each output is delayed by 2 ns, and because of that, *A3* goes from 0 to 1 at $t=88$ ns and from 1 to 0 at 168 ns. Notice how the propagation delays accumulate to the last bit of the counter, resulting in very slow counter action. This limits the practical utility of the counter.

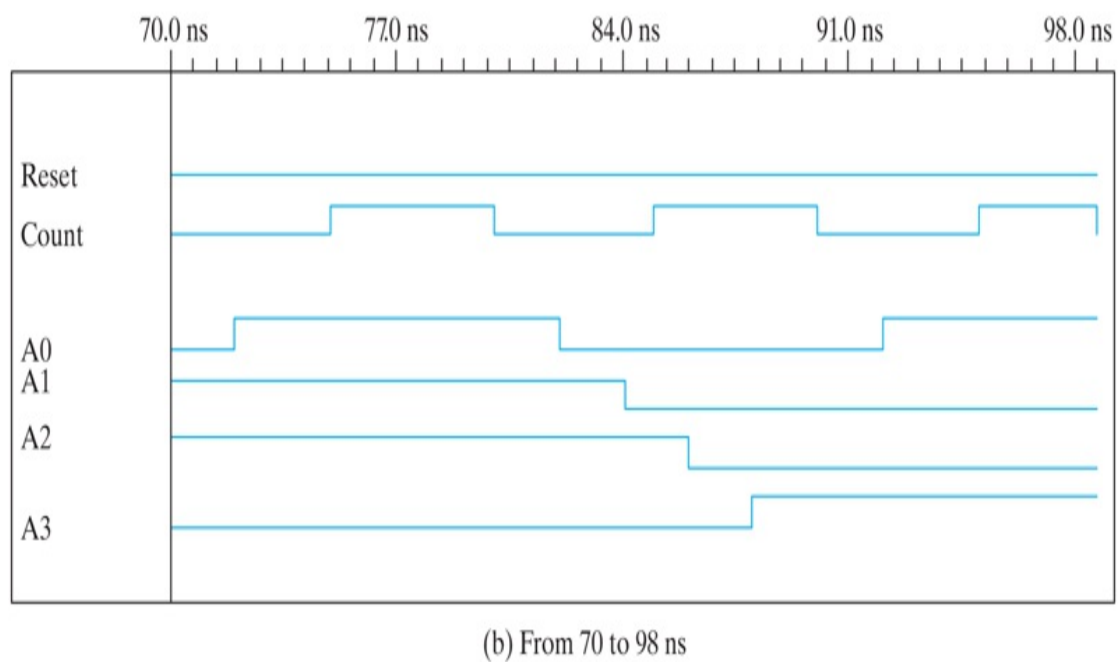
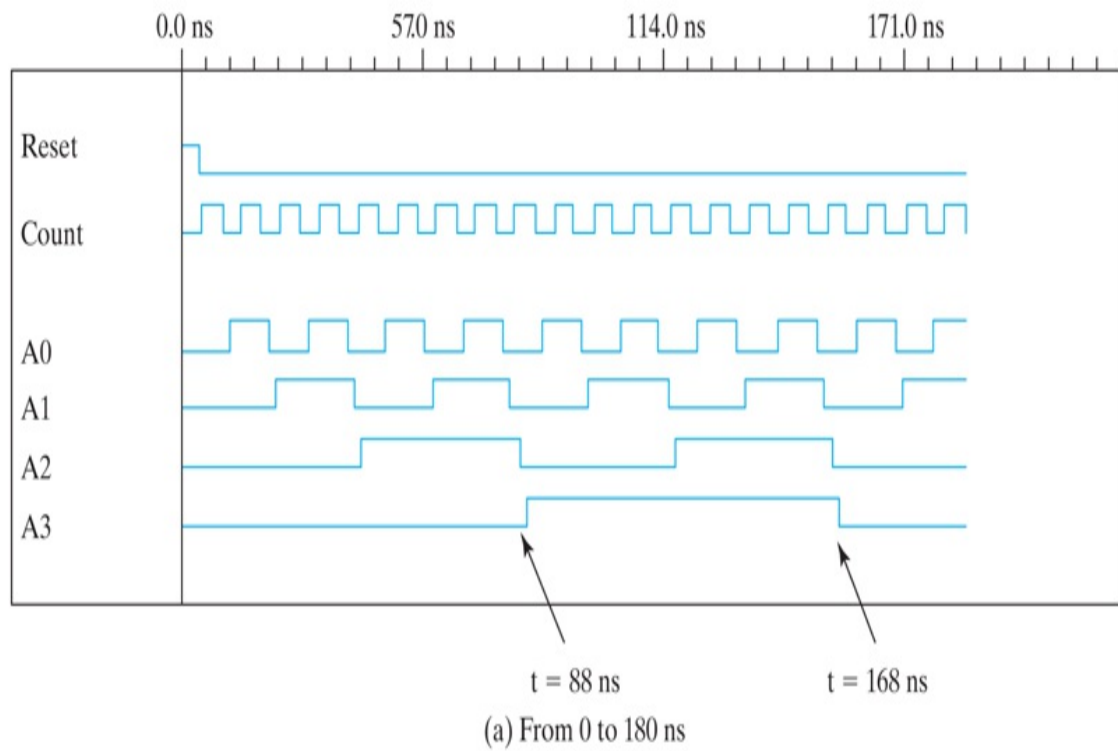


FIGURE 6.19

Simulation output of [HDL Example 6.4](#)

[Description](#)

Practice Exercise 6.3 – Verilog

1. The bits of a four-bit ripple counter are labelled A3, A2, A1, and A0. If the flip-flops of the counter are instantiated and interconnected as shown in the text below, the counter fails to function correctly. Find the error in the code

```
Comp_D_flip_flop F0 (A0, Count, Reset);
Comp_D_flip_flop F1 (A1, A0, Reset);
Comp_D_flip_flop F2 (A2, A3, Reset);
Comp_D_flip_flop F3 (A3, A1, Reset);
```

Answer: Flip-flops *F2* and *F3* are not clocked correctly. In *F2*, replace A3 by A1. In *F3*, replace A1 by A2.

VHDL

```
entity Comp_D_flip_flop is
  port (Q: out Std_Logic; CLK, Reset: in Std_Logic);
end Comp_D_flip_flop;

architecture Behavioral of Comp_D_flip_flop is
begin
  process (CLK, Reset) begin
    if Reset'event and Reset = 1 then Q <= '0';
    else if CLK'event and CLK = 0 then Q <= not Q after 2 ns;
    end if;
  end process;
end Behavioral;

entity Ripple_Counter_4bit is
  port (A3, A2, A1, A0: out Std_Logic; Count, Reset: in Std_Logic);
end Ripple_Counter_4bit;

architecture Structural of Ripple_Counter_4bit is
  component Comp_D_flip_flop port (Q: out Std_Logic; CLK, Reset: in Std_Logic);
  end component;
begin
  F0: Comp_D_flip_flop port map (Q => A0: out CLK => Count, Reset => Reset);
  F1: Comp_D_flip_flop port map (Q => A1: out CLK => A0, Reset => Reset);
  F2: Comp_D_flip_flop port map (Q => A2: out CLK => A1, Reset => Reset);
  F3: Comp_D_flip_flop port map (Q => A3: out CLK => A2, Reset => Reset);
end Structural;

-- stimulus for four-bit ripple counter
```

```

entity t_Ripple_Counter_4bit is
port ();
end t_Ripple_Counter_4bit;

architecture Behavioral of t_Ripple_Counter_4bit is
signal t_A3, t_A2, t_A1, t_A0, t_Count, t_Reset: Std_Logic;
component Ripple_Counter_4bit port (A3, A2, A1, A0: out Std_Log
-- Instantiate UUT
begin
  Ripple_Counter_4bit: UUT port map (A3 => t_A3, A2 => t_A2, A1
process ();
  t_count <= 0;
  t_count <= not t_count after 5 ns;
end process;

process();
  t_Reset = 0;
  wait 4 ns;
  t_Reset = 1;
end process;
end Behavioral;

```

Simulation results are presented in [Fig. 6.19](#).

Practice Exercise 6.3 – VHDL

1. The bits of a four-bit ripple counter are labeled A3, A2, A1, and A0. If the flip-flops of the counter are instantiated and interconnected as shown below, the counter fails to function correctly. Find the error in the code

```

F0: Comp_D_flip_flop port map (Q => A0: out CLK => Count, F
F1: Comp_D_flip_flop port map (Q => A1: out CLK => A0, Rese
F2: Comp_D_flip_flop port map (Q => A2: out CLK => A3, Rese
F3: Comp_D_flip_flop port map (Q => A3: out CLK => A1, Rese

```

Answer: Flip-flops *F2* and *F3* are not clocked correctly. In *F2*, replace A3 by A1. In *F3*, replace A1 by A2.

PROBLEMS

(Answers to problems marked with * appear at the end of the book. Where appropriate, a logic design and its related HDL modeling problem are cross-referenced.) Unless otherwise specified, a HDL model may be written in Verilog or VHDL. Note: For each problem that requires writing and verifying a HDL description, a test plan is to be written to identify which functional features are to be tested during the simulation and how they will be tested. For example, a reset on-the-fly could be tested by asserting the reset signal while the simulated machine is in a state other than the reset state. The test plan is to guide the development of a testbench that will implement the plan. Simulate the model using the testbench and verify that the behavior is correct. If synthesis tools and an ASIC cell library or a field programmable gate array (FPGA) tool suite are available, the HDL descriptions developed for [Problems 6.34–6.51](#) can be assigned as synthesis exercises. The gate-level circuit produced by the synthesis tools should be simulated and compared to the simulation results for the pre-synthesis model.

In some of the HDL problems, there may be a need to deal with the issue of unused states (see the discussion of the **default case** item preceding [HDL Example 4.8](#) in [Chapter 4](#)).

1. 6.1 Include a 2-input NAND gate in the register of [Fig. 6.1](#) and connect the gate output to the *C* inputs of all the flip-flops. One input of the NAND gate receives the clock pulses from the clock generator, and the other input of the NAND gate provides a parallel load control. Explain the operation of the modified register. Explain why this circuit might have operational problems.
2. 6.2 Include a synchronous clear input to the register circuit of [Fig. 6.2](#). The modified register will have a parallel load capability and a synchronous clear capability. The register is cleared synchronously when the clock goes through a positive transition and the clear input is equal to 1. (HDL—see [Problem 6.35\(a\), \(b\)](#))
3. 6.3 What is the difference between serial and parallel transfer? Explain how to convert serial data to parallel and parallel data to

serial. What type of register is needed?

4. 6.4* The content of a four-bit register is initially the 4-bit word 0110. The register is shifted six times to the right with the serial input being 1011100. What is the content of the register after each shift?
5. 6.5 The four-bit universal shift register shown in [Fig. 6.7](#) is enclosed within one IC component package. (HDL—see [Problem 6.52](#))
 1. Draw a block diagram of the IC showing all inputs and outputs. Include two pins for the power supply.
 2. Draw a block diagram using two of these ICs to produce an eight-bit universal shift register.
6. 6.6 Design a four-bit shift register (not a universal shift register) with parallel load using *D* flip-flops. (See [Figs. 6.2](#) and [Fig. 6.3](#).) There are two control inputs: *shift* and *load*. When *shift*=1, the content of the register is shifted toward A3 by one position. New data are transferred into the register when *load*=1 and *shift*=0. If both control inputs are equal to 0, the content of the register does not change. (HDL—see [Problem 6.35\(c\)](#), [\(d\)](#))
7. 6.7 Draw the logic diagram of a four-bit register with four *D* flip-flops and four 4×1 multiplexers with mode selection inputs *s1* and *s0*. The register operates according to the following function table. (HDL—see [Problem 6.35\(e\)](#), [\(f\)](#))

s1 s0	Register Operation
0 0	No change
1 0	Complement the four outputs
0 1	Clear register to 0 (synchronous with the clock)

1 1 Load parallel data

8. 6.8* The serial adder of [Fig. 6.6](#) uses two four-bit registers. Register *A* holds the binary number 0101 and register *B* holds 0111. The carry flip-flop is initially reset to 0. List the binary values in register *A* and the carry flip-flop after each shift. (HDL—see [Problem 6.54](#))
9. 6.9 Two ways for implementing a serial adder ($A+B$) are shown in [Section 6.2](#). It is necessary to modify the circuits to convert them to serial subtractors ($A-B$).
 1. Using the circuit of [Fig. 6.5](#), show the changes needed to perform $A+2$'s complement of *B*. (HDL—see [Problem 6.35\(h\)](#))
 2. * Using the circuit of [Fig. 6.6](#), show the changes needed by modifying [Table 6.2](#) from an adder to a subtractor circuit. (See [Problem 4.12](#).) (HDL—see [Problem 6.35\(i\)](#))
10. 6.10 Design a serial 2's complementer with a shift register and a flip-flop. The binary number is shifted out from one side and its 2's complement shifted into the other side of the shift register. (HDL—see [Problem 6.35\(j\)](#))
11. 6.11 A binary ripple counter uses flip-flops that trigger on the positive-edge of the clock. What will be the count if:
 1. the normal outputs of the flip-flops are connected to the clock; and
 2. the complement outputs of the flip-flops are connected to the clock?
12. 6.12 Draw the logic diagram of a four-bit binary ripple countdown counter using:
 1. flip-flops that trigger on the positive-edge of the clock; and
 2. flip-flops that trigger on the negative-edge of the clock.
13. 6.13 Show that a BCD ripple counter can be constructed using a four-bit binary ripple counter with asynchronous clear and a NAND gate

that detects the occurrence of count 1010. (HDL—see [Problem 6.35\(k\)](#))

14. 6.14 How many flip-flops will be complemented in a 10-bit binary ripple counter to reach the next count after the following counts?
 1. * 1001100111
 2. 1111000111
 3. 0000001111
15. 6.15* A flip-flops has a 3 ns delay from the time the clock edge occurs to the time the output is complemented. What is the maximum delay in a 10-bit binary ripple counter that uses these flip-flops? What is the maximum frequency at which the counter can operate reliably?
16. 6.16* The BCD ripple counter shown in [Fig. 6.10](#) has four flip-flops and 16 states, of which only 10 are used. Modify the logic diagram by adding a reset signal to initialize the counter. Analyze the circuit and determine the next state for each of the other six unused states. What will happen if a noise signal sends the circuit to one of the unused states? (HDL—see [Problem 6.54](#))
17. 6.17* Design a four-bit binary synchronous counter with *D* flip-flops.
18. 6.18 What operation is performed in the up-down counter of [Fig. 6.13](#) when both the up and down inputs are enabled? Modify the circuit so that when both inputs are equal to 1, the counter does not change state. (HDL—see [Problem 6.35\(l\)](#))
19. 6.19 The flip-flop input equations for a BCD counter using *T* flip-flops are given in [Section 6.4](#). Obtain the input equations for a BCD counter that uses (a) *JK* flip-flops and (b)* *D* flip-flops. Compare the three designs to determine which one is the most efficient.
20. 6.20 Enclose the binary counter with parallel load of [Fig. 6.14](#) in a block diagram showing, all inputs and outputs.
 1. Show the connections of four such blocks to produce a 16-bit

counter with parallel load.

2. Construct a binary counter that counts from 0 through binary 127.
21. 6.21* The counter of [Fig. 6.14](#) has two control inputs—*Load (L)* and *Count (C)*, and a data input, (*I*).
1. * Derive the flip-flop input equations for *J* and *K* of the first stage in terms of *L*, *C*, and *I*.
 2. The logic diagram of the first stage of an equivalent circuit is shown in [Fig. P6.21](#). Verify that this circuit is equivalent to the one in (a).

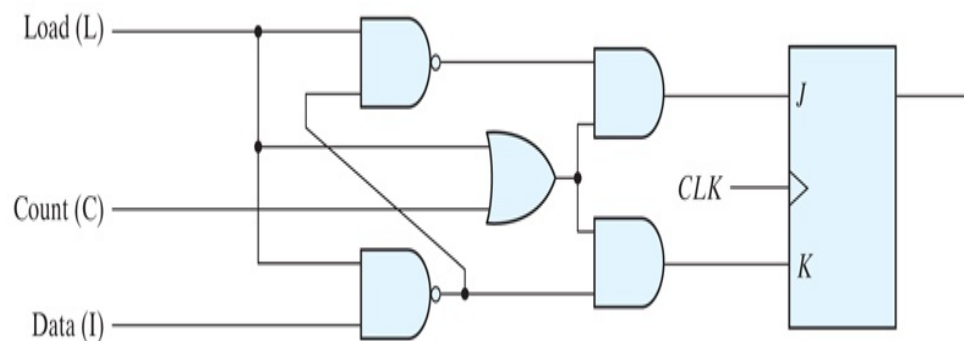


FIGURE P6.21

[Description](#)

22. 6.22 For the circuit of [Fig. 6.14](#), give three alternatives for a mod-10 counter (i.e., the count evolves through a sequence of 10 distinct states).
1. Using an AND gate and the load input.
 2. Using the output carry.
 3. Using a NAND gate and the asynchronous clear input.
23. 6.23 Design a timing circuit that provides an output signal that stays on for exactly 12 clock cycles. A start signal sends the output to the 1

state, and after 12 clock cycles the signal returns to the 0 state. (HDL—see [Problem 6.45](#))

24. 6.24* Design a counter with T flip-flops that goes through the following binary repeated sequence: 0, 1, 3, 7, 6, 4. Show that when binary states 010 and 101 are considered as don't care conditions, the counter may not operate properly. Find a way to correct the design. (HDL—see [Problem 6.55](#))
25. 6.25 It is necessary to generate six repeated timing signals T0 through T5 similar to the ones shown in [Fig. 6.17\(c\)](#). Design the circuit using (HDL—see [Problem 6.46](#)):
 1. Flip-flops only.
 2. A counter and a decoder.
26. 6.26* A digital system has a clock generator that produces pulses at a frequency of 80 MHz. Design a circuit that provides a clock with a cycle time of 50 ns.
27. 6.27 Using JK flip-flops:
 1. Design a counter with the following repeated binary sequence: 0, 1, 2, 3, 4, 5, 6. (HDL—see [Problem 6.50\(a\)](#), 6.51).
 2. Draw the logic diagram of the counter.
28. 6.28 Using D flip-flops:
 1. * Design a counter with the following repeated binary sequence: 0, 1, 2, 4, 6. (HDL—see [Problem 6.50\(b\)](#))
 2. Draw the logic diagram of the counter.
 3. Design a counter with the following repeated binary sequence: 0, 2, 4, 6, 8.
 4. Draw the logic diagram of the counter.
29. 6.29 List the eight unused states in the switch-tail ring counter of [Fig. 6.18\(a\)](#). Determine the next state for each of these states and show

that, if the counter finds itself in an invalid state, it does not return to a valid state. Modify the circuit as recommended in the text and show that the counter produces the same sequence of states and that the circuit reaches a valid state from any one of the unused states.

30. 6.30 Show that a Johnson counter with n flip-flops produces a sequence of $2n$ states. List the 10 states produced with five flip-flops and the Boolean terms of each of the 10 AND gate outputs.
31. 6.31 Write and verify the HDL behavioral and structural descriptions of the four-bit register [Fig. 6.1](#).
32. 6.32
 1. Write and verify an HDL behavioral description of a four-bit register with parallel load and asynchronous clear.
 2. Write and verify the HDL structural description of the four-bit register with parallel load shown in [Fig. 6.2](#). Use a 2×1 multiplexer for the flip-flop inputs. Include an asynchronous clear input.
 3. Verify both descriptions, using a testbench.
33. 6.33 The following stimulus program is used to simulate the binary counter with parallel load described in [HDL Example 6.3](#). Draw waveforms showing the output of the counter and the carry output from $t=0$ to $t=155$ ns.

Verilog

```
// Stimulus for testing the binary counter of Example 6.3
module testcounter( );
  reg t_Count, t_Load, t_CLK, t_Clr;
  reg [3: 0] t_IN;
  wire t_C0;
  wire [3: 0] t_A;
  counter cnt (t_Count, t_Load, t_IN, t_CLK, t_Clr, t_A, t_C0);
always
  #5 t_CLK = ~t_CLK;
initial
  begin
    t_Clr = 0;
    t_CLK = 1;
```

```

    t_Load = 0; t_Count = 1;
#5 t_Clr = 1;
#40 t_Load = 1; t_IN = 4'b1001;
#10 t_Load = 0;
#70 t_Count = 0;
#20 $finish;
end
endmodule

```

VHDL

```

entity testcounter is
port ();
end testcounter;

architecture Behavioral of testcounter is
signal t_count, t_Load, t_CLK, t_Clr, t_CO: Std_Logic; t_A,
Std_Logic_Vector (3 downto 0));
component counter port(A_count: in Std_Logic_Vector (3 downto
C_out: out Std_Logic; Data_in: in Std_Logic_Vector (3 downto
Load, CLK, Clear_b: in Std_Logic);
begin
cnt: counter port map(A_count => t_A; C_out => t_CO, Data_in
Load => t_Load, CLK => t_CLK, Clear_b => t_Clr);

process ();
    t_CLK <= '1';
    t_CLK <= not t_CLK after 5 ns;
    t_CLK <= '0' after 5 ns;
    wait for 5 ns;
end process;

process
    t_Clr <= '0';
    t_Load <= '0';
    t_Count <= '1';
    t_Clear <= '1' after 5 ns;
    t_Load <= '1' after 45 ns;
    t_IN <= '1001' after 45 ns;
    t_Load <= '0' after 55 ns;
    t_count <= '0' after 70 ns;
    wait;
end process;
end Behavioral;

```

34. 6.34* Write and verify the HDL behavioral description of a four-bit shift register (see [Fig. 6.3](#)).
35. 6.35 Write and verify:

1. A structural HDL model for the register described in [Problem 6.2](#).
2. * A behavioral HDL model for the register described in [Problem 6.2](#).
3. A structural HDL model for the register described in [Problem 6.6](#).
4. A behavioral HDL model for the register described in [Problem 6.6](#).
5. A structural HDL model for the register described in [Problem 6.7](#).
6. A behavioral HDL model for the register described in [Problem 6.7](#).
7. A behavioral HDL model of the binary counter described in [Fig. 6.8](#).
8. A behavioral HDL model of the serial subtractor described in [Problem 6.9\(a\)](#).
9. A behavioral HDL model of the serial subtractor described in [Problem 6.9\(b\)](#).
10. A behavioral HDL model of the serial 2's complementer described in [Problem 6.10](#).
11. A behavioral HDL model of the BCD ripple counter described in [Problem 6.13](#).
12. A behavioral HDL model of the up–down counter described in [Problem 6.18](#).
36. 6.36 Write and verify the HDL behavioral and structural descriptions of the four-bit up–down counter whose logic diagram is described by [Fig. 6.13](#), [Table 6.5](#), and [Table 6.6](#).
37. 6.37 Write and verify a behavioral description of the counter described in [Problem 6.24](#).

1. * Using an **if . . . else** statement.
 2. Using a **case** statement.
 3. A finite state machine.
38. 6.38 Write and verify the HDL behavioral description of a four-bit up–down counter with parallel load using the following control inputs:
1. * The counter has three control inputs for the three operations: *Up*, *Down*, and *Load*. The order of precedence is: *Load*, *Up*, and *Down*.
 2. The counter has two selection inputs to specify four operations: *Load*, *Up*, *Down*, and no change.
39. 6.39 Write and verify HDL behavioral and structural descriptions of the counter of [Fig. 6.16](#).
40. 6.40 Write and verify the HDL description of an eight-bit ring-counter similar to the one shown in [Fig. 6.17\(a\)](#).
41. 6.41 Write and verify the HDL description of a four-bit switch-tail ring (Johnson) counter ([Fig. 6.18a](#)).
42. 6.42* The comment with the last clause of the **if** statement in *Binary_Counter_4_Par_Load* in [HDL Example 6.3](#) notes that the statement is redundant. Explain why this statement can be removed without changing the behavior implemented by the description.
43. 6.43 The scheme shown in [Fig. 6.4](#) gates the clock to control the serial transfer of data from shift register A to shift register B. Using multiplexers at the input of each cell of the shift registers, develop a structural model of an alternative circuit that does not alter the clock path. The top level of the design hierarchy is to instantiate the shift registers. The module describing the shift registers is to have instantiations of flip-flops and muxes. Describe the mux and flip-flop modules with behavioral models. Be sure to consider the need to reset the machine. Develop a testbench to simulate the circuit and demonstrate the transfer of data.

44. 6.44 Modify the design of the serial adder shown in [Fig. 6.5](#) by removing the gated clock to the *D* flip-flop and supplying the clock signal to it directly. Augment the *D* flip-flop with a mux to recirculate the contents of the flip-flop when shifting is suspended and provide the carry out of the full adder when shifting is active. The shift registers are to incorporate this feature also, rather than use a gated clock. The top-level of the design is to instantiate modules using behavioral models for the shift registers, full adder, *D* flip-flop, and mux. Assume asynchronous reset. Develop a testbench to simulate the circuit and demonstrate the transfer of data.
45. 6.45* Write and verify a behavioral description of a finite state machine to implement the counter described in [Problem 6.23](#).
46. 6.46 [Problem 6.25](#) specifies an implementation of a circuit to generate timing signals using
1. Only flip-flops.
 2. A counter and a decoder.

As an alternative, write a behavioral description (without consideration of the actual hardware) of a state machine whose output generates the timing signals T0 through T5.

47. 6.47 Write a behavioral description of the circuit shown in [Fig. P6.47](#) and verify that the circuit's output is asserted if successive samples of the input have an odd number of 1s.

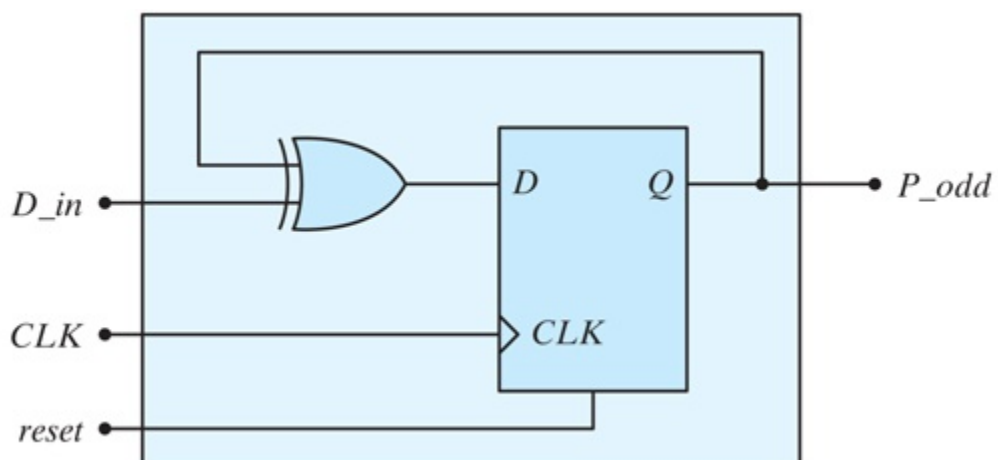


FIGURE P6.47

Circuit for [Problem 6.47](#)

48. 6.48 Write and verify a behavioral description of the counter shown in [Fig. P6.48\(a\)](#); repeat for the counter in [Fig. P6.48\(b\)](#).

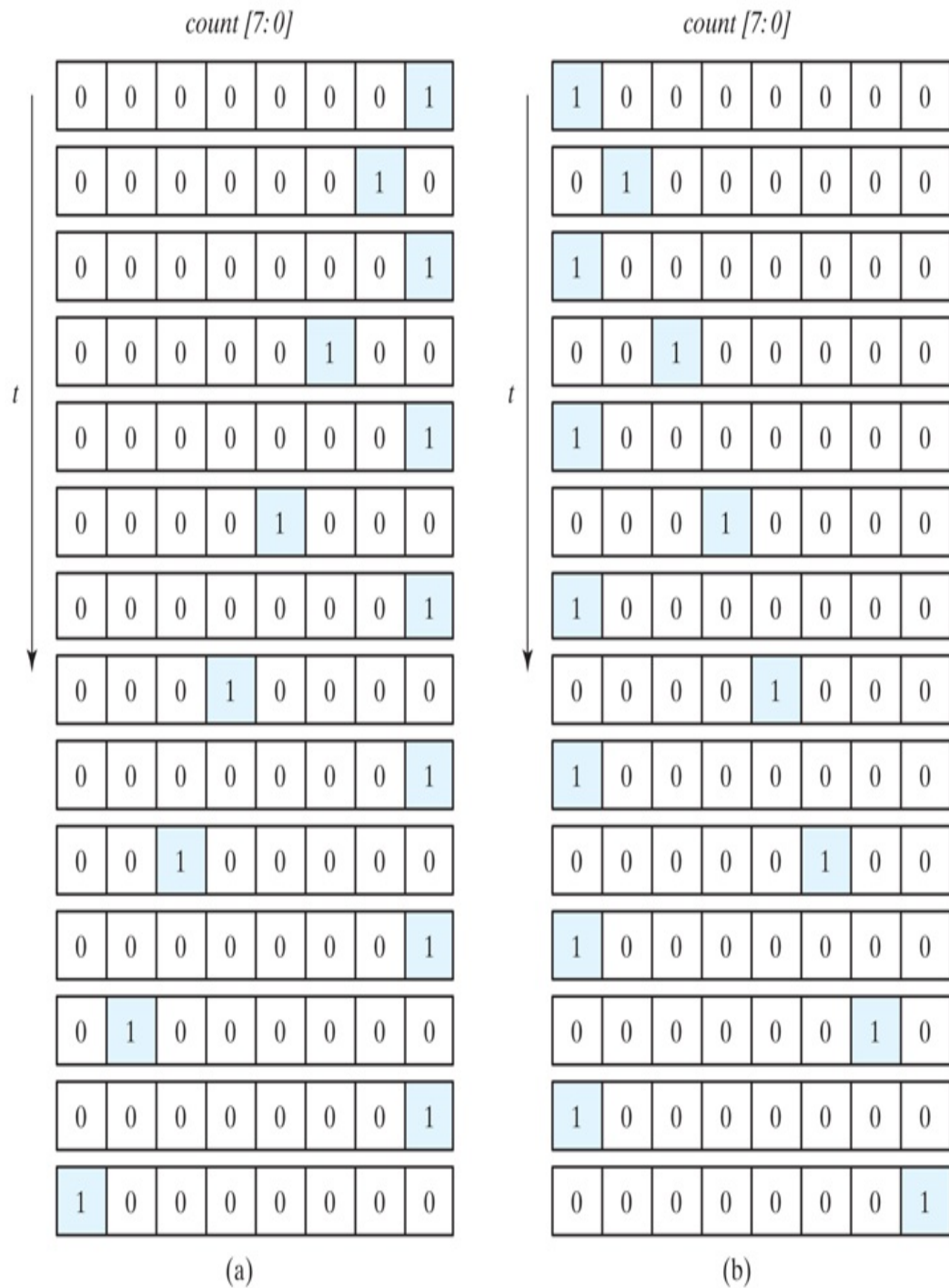


FIGURE P6.48

Circuit for [Problem 6.48](#)

[Description](#)

49. 6.49 Write a test plan for verifying the functionality of the universal shift register described in [HDL Example 6.1](#). Using the test plan, simulate the model given in [HDL Example 6.1](#).
50. 6.50 Write and verify a behavioral model of the counter described in:
 1. [Problem 6.27](#)
 2. [Problem 6.28](#).
51. 6.51 Without requiring a state machine, and using a shift register and additional logic, write and verify a model of an alternative to the sequence detector described in [Fig. 5.27](#). Compare the implementations.
52. 6.52 Write a HDL structural model of the universal shift register in [Fig. 6.7](#). Verify all modes of its operation.
53. 6.53 Verify that the serial adder in [Fig. 6.5](#) operates as an accumulator when words are shifted into the addend register repeatedly.
54. 6.54 Write and verify a structural model of the serial adder in [Fig. 6.6](#).
55. 6.55 Write and verify a structural model of the BCD ripple counter in [Fig. 6.10](#).
56. 6.56 Write and verify a structural model of the synchronous binary counter in [Fig. 6.12](#).
57. 6.57 Write and verify a structural model of the up-down counter in [Fig. 6.13](#).

58. 6.58 Write and verify all modes of operation of:

1. A structural model of the binary counter in [Fig. 6.14](#)
2. A behavioral model of the binary counter in [Fig. 6.14](#).

59. 6.59 Write and verify:

1. A structural model of the switch-tail ring counter in [Fig. 6.18\(a\)](#).
2. A behavioral model of the switch-tail ring counter in [Fig. 6.18\(a\)](#).

REFERENCES

- 1. Mano, M. M. and C. R. Kime. 2007. *Logic and Computer Design Fundamentals*, 4th ed., Upper Saddle River, NJ: Prentice Hall.
- 2. Nelson V. P., H. T. Nagle, J. D. Irwin, and B. D. Carroll. 1995. *Digital Logic Circuit Analysis and Design*. Upper Saddle River, NJ: Prentice Hall.
- 3. Hayes, J. P. 1993. *Introduction to Digital Logic Design*. Reading, MA: Addison-Wesley.
- 4. Wakerly, J. F. 2000. *Digital Design: Principles and Practices*, 3rd ed., Upper Saddle River, NJ: Prentice Hall.
- 5. Dietmeyer, D. L. 1988. *Logic Design of Digital Systems*, 3rd ed., Boston, MA: Allyn Bacon.
- 6. Gajski, D. D. 1997. *Principles of Digital Design*. Upper Saddle River, NJ: Prentice Hall.
- 7. Roth, C. H. 2009. *Fundamentals of Logic Design*, 6th ed., St. Paul, MN: West.
- 8. Katz, R. H. 1994. *Contemporary Logic Design*. Upper Saddle River, NJ: Prentice Hall.
- 9. Ciletti, M. D. 1999. *Modeling, Synthesis, and Rapid Prototyping with Verilog HDL*. Upper Saddle River, NJ: Prentice Hall.
- 10. Bhasker, J. 1997. *A Verilog HDL Primer*. Allentown, PA: Star Galaxy Press.
- 11. Thomas, D. E. and P. R. Moorby. 2002. *The Verilog Hardware Description Language*, 5th ed., Boston, MA: Kluwer Academic Publishers.
- 12. Bhasker, J. 1998. *Verilog HDL Synthesis*. Allentown, PA: Star Galaxy Press.

- 13. Palnitkar, S. 1996. *Verilog HDL: A Guide to Digital Design and Synthesis*. Mountain View, CA: SunSoft Press (a Prentice Hall Title).
- 14. Ciletti, M. D. 2010. *Advanced Digital Design with the Verilog HDL*, 2e. Upper Saddle River, NJ: Prentice Hall.
- 15. Ciletti, M. D. 2004. *Starter's Guide to Verilog 2001*. Upper Saddle River, NJ: Prentice Hall.

WEB SEARCH TOPICS

- BCD counter
- Johnson counter
- Ring counter
- Sequence detector
- Synchronous counter
- Switch-tail ring counter
- Up–down counter