VHDL for Sequential Logic

In Unit 10 we learned how to represent combinational logic in VHDL by using concurrent signal assignment statements. In this unit, we will learn how to represent sequential logic by using VHDL processes.

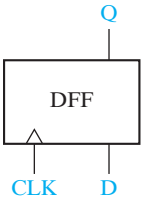
17.1 Modeling Flip-Flops Using VHDL Processes

A flip-flop can change state either on the rising or on the falling edge of the clock input. This type of behavior is modeled in VHDL by a process. For a simple D flip-flop with a Q output that changes on the rising edge of CLK, the corresponding process is given in Figure 17-1.

The expression in parentheses after the word **process** is called a sensitivity list, and the process executes whenever any signal in the sensitivity list changes. For example, if the process begins with **process**(A, B, C), then the process executes whenever any one of A, B, or C changes. When a process finishes executing, it goes back to the beginning and waits for a signal on the sensitivity list to change again.

In Figure 17-1, whenever CLK changes, the process executes once through and, then, waits at the start of the process until CLK changes again. The **if** statement tests for a rising edge of the clock, and Q is set equal to D when a rising edge occurs. The expression CLK'event (read as clock tick event) is TRUE whenever the signal CLK changes. If CLK = '1' is also TRUE, this means that the change was from '0' to '1', which is a rising edge. If the flip-flop has a delay of 5 ns between the rising edge of the clock and the change in the Q output, we would replace the statement Q <= D; with Q <= D **after** 5 ns; in the process in Figure 17-1.

The statements between **begin** and **end** in a process are called sequential statements. In the process in Figure 17-1, Q <= D; is a sequential statement that only

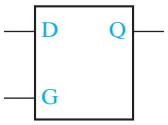



```
process (CLK)
begin
    if CLK'event and CLK = '1' -- rising edge of CLK
        then Q <= D;
    end if;
end process;
```

FIGURE 17-1
VHDL Code for a Simple D Flip-Flop

executes following the rising edge of CLK. In contrast, the concurrent statement Q <= D; executes whenever D changes. If we synthesize the process, the synthesizer infers that Q must be a flip-flop because it only changes on the rising edge of CLK. If we synthesize the concurrent statement Q <= D; the synthesizer will simply connect D to Q with a wire or with a buffer.

In Figure 17-1 note that D is not on the sensitivity list because changing D will not cause the flip-flop to change state. Figure 17-2 shows a transparent latch and its VHDL representation. Both G and D are on the sensitivity list because if G = '1', a change in D causes Q to change. If G changes to '0', the process executes, but Q does not change.




```
process (G,D)
begin
    if G = '1' then Q <= D; end if;
end process;
```

FIGURE 17-2
VHDL Code for a Transparent Latch

If a flip-flop has an active-low asynchronous clear input (ClrN) that resets the flip-flop independently of the clock, then we must modify the process of Figure 17-1 so that it executes when either CLK or ClrN changes. To do this, we add ClrN to the sensitivity list. The VHDL code for a D flip-flop with asynchronous clear is given in Figure 17-3. Because the asynchronous ClrN signal overrides CLK, ClrN is tested first, and the flip-flop is cleared if ClrN is '0'. Otherwise, CLK is tested, and Q is updated if a rising edge has occurred.

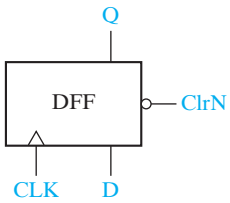
A basic process has the following form:

```
process(sensitivity-list)
begin
   sequential-statements
end process;
```

Whenever one of the signals in the sensitivity list changes, the sequential statements in the process body are executed in sequence one time. The process then goes back to the beginning and waits for a signal in the sensitivity list to change.

In the previous examples, we have used two types of sequential statements—signal assignment statements and **if** statements. The basic **if** statement has the form

```
if condition then
      sequential statements1
   else sequential statements2
end if;
```




```
process (CLK, ClrN)
begin
    if ClrN = '0' then Q <= '0';
        else if CLK'event and CLK = '1'
            then Q <= D;
            end if;
    end if;
end process;
```

FIGURE 17-3
VHDL Code for a D Flip-Flop with Asynchronous Clear

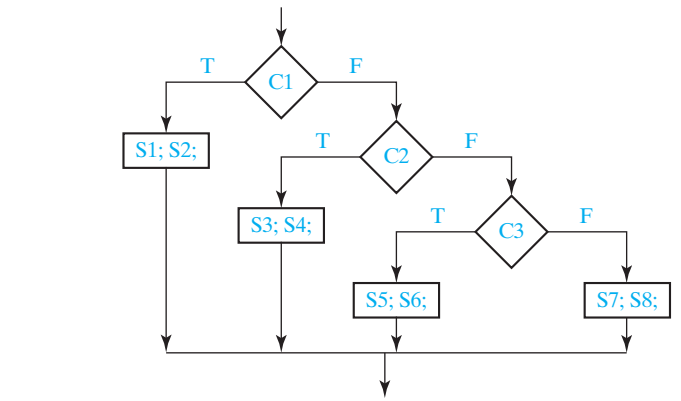
The condition is a Boolean expression which evaluates to TRUE or FALSE. If it is TRUE, sequential statements1 are executed; otherwise, sequential statements2 are executed. VHDL **if** statements are sequential statements that can be used within a process, but they cannot be used as concurrent statements outside of a process. On the other hand, conditional signal assignment statements are concurrent statements that cannot be used within a process.

The most general form of the **if** statement is

```
if condition then
   sequential statements
{elsif condition then
   sequential statements}
   -- 0 or more elsif clauses may be included
[else sequential statements]
end if;
```

The curly brackets indicate that any number of **elsif** clauses may be included, and the square brackets indicate that the **else** clause is optional. The example of Figure 17-4 shows how a flow chart can be represented using nested **if**s or the equivalent using **elsif**s. In this example, C1, C2, and C3 represent conditions that can be TRUE or FALSE, and S1, S2, . . . S8 represent sequential statements. Each **if** requires a corresponding **end if**, but an **elsif** does not.

Next, we will write a VHDL module for a J-K flip-flop (Figure 17-5). This flip-flop has active-low asynchronous preset (SN) and clear (RN) inputs. State changes related to J and K occur on the falling edge of the clock. In this chapter, we use a suffix N to indicate an active-low (negative-logic) signal. For simplicity, we will assume that the condition SN = RN = 0 does not occur.




```
if (C1) then S1; S2;
    else if (C2) then S3; S4;
        else if (C3) then S5; S6;
            else S7; S8;
            end if;
        end if;
end if;
```

```
if (C1) then S1; S2;
    elsif (C2) then S3; S4;
    elsif (C3) then S5; S6;
    else S7; S8;
end if;
```

FIGURE 17-4
Equivalent Representations of a Flow Chart Using Nested Ifs and Elsifs

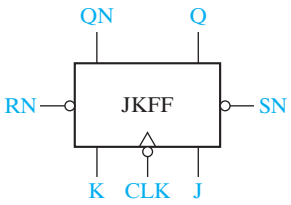


FIGURE 17-5
J-K Flip-Flop

The VHDL code for the J-K flip-flop is given in Figure 17-6. The port declaration in the entity defines the input and output signals. Within the architecture we define a signal Qint that represents the state of the flip-flop internal to the module. The two concurrent statements after **begin** transmit this internal signal to the Q and QN outputs of the flip-flop. We do it this way because an output signal in a port cannot appear on the right side of an assignment statement within the architecture. The flip-flop can change state in response to changes in SN, RN, and CLK, so these three signals are in the sensitivity list of the process. Because RN and SN reset and set the flip-flop independently of the clock, they are tested first. If RN and SN are both '1', then we test for the falling edge of the clock. The condition (CLK'event **and** CLK = '0') is TRUE only if CLK has just changed from '1' to '0'. The next state of the flip-flop is determined by its characteristic equation:

$$Q^+ = JQ' + K'Q$$

The 8-ns delay represents the time it takes to set or clear the flip-flop output after SN or RN changes to 0. The 10-ns delay represents the time it takes for Q to change after the falling edge of the clock.

FIGURE 17-6
J-K Flip-Flop Model


```
1   entity JKFF is
2   port (SN, RN, J, K, CLK: in bit;                    -- inputs
3      Q, QN: out bit);
4   end JKFF;
5   architecture JKFF1 of JKFF is
6   signal Qint: bit;                                   -- internal value of Q
7   begin
8      Q <= Qint;                                       -- output Q and QN to port
9      QN <= not Qint;
10     process (SN, RN, CLK)
11     begin
12        if RN = '0' then Qint <= '0' after 8 ns;      -- RN = '0' will clear the FF
13        elsif SN = '0' then Qint <= '1' after 8 ns;   -- SN = '0' will set the FF
14        elsif CLK'event and CLK = '0' then            -- falling edge of CLK
15           Qint <= (J and not Qint) or (not K and Qint) after 10 ns;
16        end if;
17     end process;
18  end JKFF1;
```

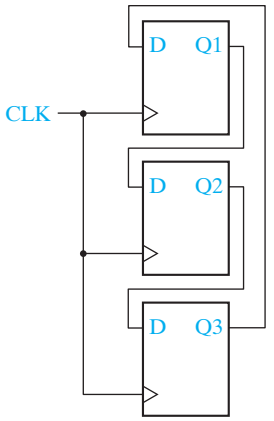
17.2 Modeling Registers and Counters Using VHDL Processes

When several flip-flops change state on the same clock edge, the statements representing these flip-flops can be placed in the same clocked process. Figure 17-7 shows three flip-flops connected as a cyclic shift register. These flip-flops all change state following the rising edge of the clock. We have assumed a 5-ns propagation delay between the clock edge and the output change. Immediately following the clock edge, the three statements in the process execute in sequence with no delay. The new values of the Q's are then scheduled to change after 5 ns. If we omit the delay and replace the sequential statements with

```
Q1 <= Q3;   Q2 <= Q1;   Q3 <= Q2;
```

the operation is basically the same. The three statements execute in sequence in zero time, and, then, the Q's change value after Δ delay. In both cases the old values of Q1, Q2, and Q3 are used to compute the new values. This may seem strange at first, but that is the way the hardware works. At the rising edge of the clock, all of the D inputs are loaded into the flip-flops, but the state change does not occur until after a propagation delay.

Next we will write structural VHDL code for the cyclic shift register using a D flip-flop as a component. In the writing of structural VHDL code, instantiation statements are used to specify how components are connected together. Components may be declared and defined either in a library or within the architecture part of the VHDL code. Each copy of a component requires a separate instantiation statement to specify how it is connected to other components and to the port inputs and outputs. Instantiation statements are concurrent statements, not sequential statements, and therefore they cannot be used within a process. A component can be as simple as a single gate or as complex as a digital system that contains many internal signals, registers, control circuits, and other components. Each instantiation statement represents a copy of a hardware component. The instantiation statement connects the component inputs and outputs, and the component computes new outputs whenever one of its inputs changes. This is exactly how the real hardware component




```
process (CLK)
begin
    if CLK'event and CLK = '1' then
        Q1 <= Q3 after 5 ns;
        Q2 <= Q1 after 5 ns;
        Q3 <= Q2 after 5 ns;
    end if;
end process;
```

FIGURE 17-7
Cyclic Shift Register

works. Instantiating a component is different from calling a function in a computer program. A function returns a new value whenever it is called, but an instantiated component computes a new output value whenever its input changes.

The VHDL code of Figure 17-8 has two modules. The first one models a simple D flip-flop. The second module instantiates three copies of the D flip-flop component to model the cyclic shift register of Figure 17-7. Qout represents a 3-bit output from the register. The internal signals (Q1, Q2, and Q3) that are declared within the architecture are used to connect the flip-flop inputs and outputs. Lines 21, 22, and 23 instantiate three copies of the D flip-flop component. Even though the DFF module has a clock input and internal sequential statements, each instantiation statement is still a concurrent statement and must *not* be placed in a process. If CLK changes, this change is passed to the D flip-flop components, and the effect of the clock change is handled within the components.

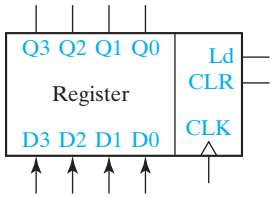
Figure 17-9 shows a simple register that can be loaded or cleared on the rising edge of the clock. If CLR = 1, the register is cleared, and if Ld = 1, the D inputs are loaded into the register. This register is fully synchronous so that the Q outputs only change in response to the clock edge and not in response to a change in Ld or CLR. In the VHDL code for the register, Q and D are bit vectors dimensioned 3 **downto** 0. Because the register outputs can only change on the rising edge of the clock, CLR is not on the

FIGURE 17-8
Structural VHDL Code for Cyclic Shift Register


```
1     entity DFF is                                          --simple DFF
2        port (D, CLK: in bit, q: out bit);
3     end DFF;
4     architecture DFF_simple of DFF is
5     begin
6     process (CLK)
7     begin
8        if CLK'event and CLK = '1' then
9           Q <= D after 5 ns; end if;
10    end process;
11    end DFF_simple;

12    entity cyclicSR is                                     -- 3-bit cyclic shift register
13       port (CLK: in bit; Qout: out bit_vector(1 to 3) ) ;
14    end cyclicSR;
15    architecture cyclicSR3 of cyclicSR is
16    component DFF
17       port (D, CLK: in bit; Q: out bit);
18    end component;
19    signal Q1, Q2, Q3: bit;
20    begin
21       FF1: DFF port map (Q3, CLK, Q1);
22       FF2: DFF port map (Q1, CLK, Q2);
23       FF3: DFF port map (Q2, CLK, Q3);
24       Qout <= Q1&Q2&Q3;
25    end cyclicSR3;
```

FIGURE 17-9
Register with Synchronous Clear and Load



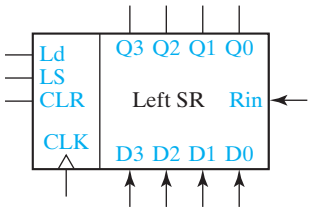

```
process (CLK)
begin
    if CLK'event and CLK = '1' then
        if CLR = '1' then Q <= "0000";
            elsif Ld = '1' then Q <= D;
        end if;
    end if;
end process;
```

sensitivity list. It is tested after the rising edge of the clock instead of being tested first as in Figure 17-3. If CLR = Ld = '0', Q does not change. Because CLR is tested before Ld, if CLR = '1', the **elsif** prevents Ld from being tested and CLR overrides Ld.

Next, we will model a left-shift register using a VHDL process. The register in Figure 17-10 is similar to that in Figure 17-9, except we have added a left-shift control input (LS). When LS is '1', the contents of the register are shifted left and the rightmost bit is set equal to Rin. The shifting is accomplished by taking the rightmost three bits of Q, Q(2 **downto** 0) and concatenating them with Rin. For example, if Q = "1101" and Rin = '0', then Q(2 **downto** 0) & Rin = "1010", and this value is loaded back into the Q register on the rising edge of CLK. The code implies that if CLR = Ld = LS = '0', then Q remains unchanged.

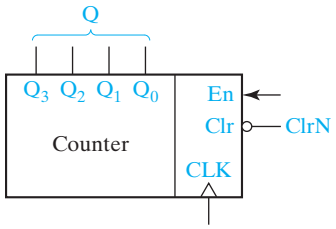
Figure 17-11 shows a simple synchronous counter. On the rising edge of the clock, the counter is cleared when ClrN = '0', and it is incremented when ClrN = En = '1'. In this example, the signal Q represents the 4-bit value stored in the counter. Because addition is not defined for bit_vectors, we have declared Q to be of type std_logic_vector. Then, we can increment the counter using the overloaded "+" operator that is defined in the ieee.std_logic_unsigned package. The statement Q <= Q + 1; increments the counter. When the counter is in state "1111", the next increment takes it back to state "0000".

FIGURE 17-10
Left-Shift Register with Synchronous Clear and Load




```
process (CLK)
begin
  if CLK'event and CLK = '1' then
    if CLR = '1' then Q <= "0000";
      elsif Ld = '1' then Q <= D;
      elsif LS = '1' then Q <= Q(2 downto 0) & Rin;
    end if;
  end if;
end process;
```

FIGURE 17-11
VHDL Code for a Simple Synchronous Counter




```
signal Q: std_logic_vector(3 downto 0);
------------
process (CLK)
begin
    if CLK'event and CLK = '1' then
        if ClrN = '0' then Q <= "0000";
            elsif En = '1' then Q <= Q + 1;
        end if;
    end if;
end process;
```

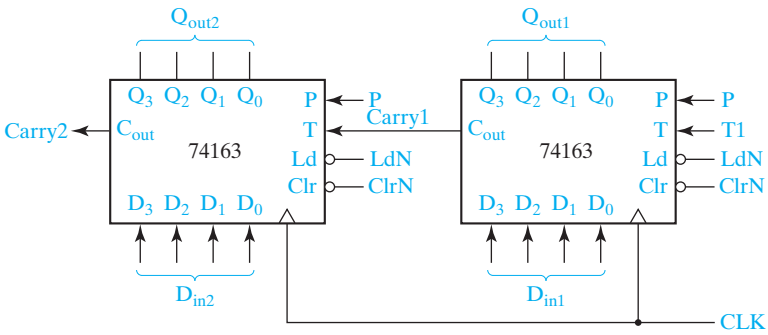


FIGURE 17-12
Two 74163 Counters Cascaded to Form an 8-Bit Counter

The 74163 (see Figure 17-12) is a 4-bit fully synchronous binary counter which is available in both TTL and CMOS logic families. Although rarely used in new designs at present, it represents a general type of counter that is found in many CAD design libraries. In addition to performing the counting function, it can be cleared or loaded in parallel. All operations are synchronized by the clock, and all state changes take place following the rising edge of the clock input.

This counter has four control inputs: ClrN, LdN, P, and T. Inputs P and T are used to enable the counting function. Operation of the counter is as follows:

1. If ClrN = 0, all flip-flops are set to 0 following the rising clock edge.
2. If ClrN = 1 and LdN = 0, the D inputs are transferred in parallel to the flip-flops following the rising clock edge.
3. If ClrN = LdN = 1 and P = T = 1, the count is enabled and the counter state will be incremented by 1 following the rising clock edge.

If T = 1, the counter generates a carry (C_{out}) in state 15, so

$$C_{out} = Q_3\, Q_2\, Q_1\, Q_0\, T$$

Table 17-1 summarizes the operation of the counter. Note that ClrN overrides the load and count functions in the sense that when ClrN = 0, clearing occurs regardless of the values of LdN, P, and T. Similarly, LdN overrides the count function. The ClrN input on the 74163 is referred to as a *synchronous* clear input because it clears the counter in synchronization with the clock, and no clearing can occur if a clock pulse is not present.

The VHDL description of the counter is shown in Figure 17-13. Q represents the four flip-flops that make up the counter. The counter output, Q_{out}, changes whenever Q changes. The carry output is computed whenever Q or T changes. The first **if** statement in the process tests for a rising edge of CLK. Because clear overrides load and count, the next **if** statement tests ClrN first. Because load overrides count, LdN is tested next.

TABLE 17-1
74163 Counter Operation

Control Signals			Next State				
ClrN	LdN	*PT*	Q_3^+	Q_2^+	Q_1^+	Q_0^+	
0	X	X	0	0	0	0	(Clear)
1	0	X	D_3	D_2	D_1	D_0	(Parallel load)
1	1	0	Q_3	Q_2	Q_1	Q_0	(No change)
1	1	1	Present state + 1				(Increment count)

FIGURE 17-13
74163 Counter Model


```
-- 74163 FULLY SYNCHRONOUS COUNTER
1     library IEEE;
2     use IEEE.STD_LOGIC_1164.ALL;
3     use IEEE.STD_LOGIC_ARITH.ALL;
4     use IEEE.STD_LOGIC_UNSIGNED.ALL;
5     entity c74163 is
6        port(LdN, ClrN, P, T, CLK: in std_logic;
7             D: in std_logic_vector(3 downto 0);
8             Cout: out std_logic; Qout: out std_logic_vector(3 downto 0) );
9     end c74163;
10    architecture b74163 of c74163 is
11    signal Q: std_logic_vector(3 downto 0);          -- Q is the counter register
12    begin
13       Qout <= Q;
14       Cout <= Q(3) and Q(2) and Q(1) and Q(0) and T;
15       process (CLK)
16       begin
17          if CLK'event and CLK = '1' then            -- change state on rising edge
18             if ClrN = '0' then Q <= "0000";
19                elsif LdN = '0' then Q <= D;
20                elsif (P and T) = '1' then Q <= Q + 1;
21             end if;
22          end if;
23       end process;
24    end b74163;
```

Finally, the counter is incremented if both P and T are 1. Because Q is type std_logic_vector, we can use the overloaded "+" operator from the ieee.std_logic_unsigned library to add 1 to increment the counter. The expression Q + 1 would not be legal if Q were a bit_vector because addition is not defined for bit_vectors.

To test the counter, we have cascaded two 74163's to form an 8-bit counter (Figure 17-12). When the counter on the right is in state 1111 and T1 = 1, the T input to the left counter is Carry1 = 1. Then, if P = 1, on the next clock the right counter is incremented to 0000 at the same time the left counter is incremented. Figure 17-14 shows the VHDL code for the 8-bit counter. In this code we have used the c74163 model as a component and instantiated two copies of it. For convenience in reading the output, we have defined a signal Count which is the integer equivalent of the 8-bit counter value. The function Conv_integer converts a std_logic_vector to an integer.

The two instantiation statements (lines 21 and 22) connect the inputs and outputs of two copies of the 4-bit counter component. Each of these concurrent statements will execute when one of the counter inputs changes, and then the corresponding counter module computes new values of the counter outputs. Although the 4-bit

FIGURE 17-14
VHDL for 8-Bit Counter


```
-- Test module for 74163 counter

1    library IEEE;
2    use IEEE.STD_LOGIC_1164.ALL;
3    use IEEE.STD_LOGIC_ARITH.ALL;
4    use IEEE.STD_LOGIC_UNSIGNED.ALL;

5    entity c74163test is
6       port(ClrN, LdN, P, T1, CLK: in std_logic;
7          Din1, Din2: in std_logic_vector (3 downto 0);
8          Count: out integer range 0 to 255;
9          Carry2: out std_logic);
10   end c74163test;

11   architecture tester of c74163test is
12      component c74163
13         port(LdN, ClrN, P, T, CLK: in std_logic;
14            D: in std_logic_vector(3 downto 0);
15         Cout: out std_logic; Qout: out std_logic_vector (3 downto 0) );
16      end component;
17      signal Carry1: std_logic;
18      signal Qout1, Qout2: std_logic_vector (3 downto 0);
19   begin
20      ct1: c74163 port map (LdN, ClrN, P, T1, CLK, Din1, Carry1, Qout1);
21      ct2: c74163 port map (LdN, ClrN, P, Carry1, CLK, Din2, Carry2, Qout2);
22      Count <= Conv_integer(Qout2 & Qout1);
23   end tester;
```

counter module (Figure 17-13) contains a process and sequential statements, each statement that instantiates a counter module is nevertheless a concurrent statement and cannot be placed within a process.

17.3 Modeling Combinational Logic Using VHDL Processes

Although processes are most useful for modeling sequential logic, they can also be used to model combinational logic. The circuit of Figure 10-1 can be modeled by the process shown in Figure 17-15.

For a combinational process, every signal that appears on the right side of a signal assignment must appear on the sensitivity list. Suppose that initially A = 1, and B = C = D = E = 0. If B changes to 1 at time = 4 ns, the process executes, and the two sequential assignment statements execute in sequence. The new value of

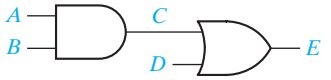


FIGURE 17-15
VHDL Code for Gate Circuit


```
process (A, B, C, D)
begin
    C <= A and B after 5 ns;
    E <= C or D after 5 ns;
end process;
```

C is computed to be '1', and C is scheduled to change 5 ns later. Meanwhile, E is immediately computed using the old value of C, but it does not change because C has not yet changed. After 5 ns, C changes, and because it is on the sensitivity list, the process executes again, and the sequential statements again execute in sequence. This time C does not change, but E is scheduled to change after 5 ns. Because E is not on the sensitivity list, no further execution of the process occurs. The following listing summarizes the operation:

time	A	B	C	D	E	
0	1	0	0	0	0	
4	1	1	0	0	0	
	process executes (C ← 1 after 5 ns; E ← 0, no change)					
9	1	1	1	0	0	
	process executes (C ← 1, no change; E ← 1 after 5 ns)					
14	1	1	1	0	1	
	no further execution until A, B, C, or D changes					

In Section 10.2, we modeled a MUX using a conditional signal assignment statement and a selected signal assignment statement. Because these are concurrent statements, they cannot be used inside a process. However, the **case** statement is a sequential statement that can be used to model a MUX within a process. The 4-to-1 MUX of Figure 10-7 can be modeled as follows:

```
signal sel: bit_vector(0 to 1);
-------------------------------------------------------
sel <= A&B;                -- a concurrent statement, outside of the process
process (sel, I0, I1, I2, I3)
begin
   case sel is             -- a sequential statement in the process
      when "00" =>     F <= I0;
      when "01" =>     F <= I1;
      when "10" =>     F <= I2;
      when "11" =>     F <= I3;
      when others => null;          -- required if sel is a std_logic_vector;
                                    -- omit if sel is a bit_vector
   end case;
end process;
```

The **case** statement has the general form:

```
case expression is
   when choice1 => sequential statements1
   when choice2 => sequential statements2
   . . .
   [when others => sequential statements]
end case;
```

The "expression" is evaluated first. If it is equal to "choice1", then "sequential statements1" are executed; if it is equal to "choice2", then "sequential statements2" are executed, etc. All possible values of the expression must be included in the choices. If all values are not explicitly given, a "**when others**" clause is required in the case statement. If no action is specified for the other choices, the clause should be

```
when others => null;
```

17.4 Modeling a Sequential Machine

In this section we will discuss several ways of writing VHDL descriptions for sequential machines. First, we will write a behavioral model for a Mealy sequential circuit based on the state table of Table 17-2. This table is the same as Table 16-3 with the states renamed. It represents a BCD to excess-3 code converter with inputs and outputs LSB first.

As shown in Figure 17-16, a Mealy machine consists of a combinational circuit and a state register. The VHDL model of Figure 17-17 uses two processes to represent these two parts of the circuit. Because X and Z are external signals, they are declared in the port. State and Nextstate are internal signals that represent the state and next state of the sequential circuit, so they are declared at the start of the architecture. At the behavioral level, we represent the state and next state of the circuit by integer signals with a range of 0 to 6.

TABLE 17-2
State Table for Code Converter

	NS		Z	
PS	$X = 0$	$X = 1$	$X = 0$	$X = 1$
S0	S1	S2	1	0
S1	S3	S4	1	0
S2	S4	S4	0	1
S3	S5	S5	0	1
S4	S5	S6	1	0
S5	S0	S0	0	1
S6	S0	–	1	–

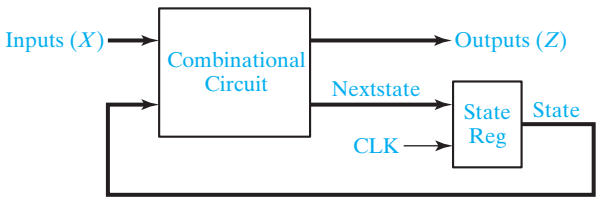


FIGURE 17-16
General Model of Mealy Sequential Machine

The first process represents the combinational circuit of Figure 17-16. Because the circuit outputs, *Z* and Nextstate, can change when either the State or *X* changes, the sensitivity list includes both State and *X*. The case statement tests the value of State, and then for each state, the **if** statement tests *X* to determine the new values of *Z* and Nextstate. For state *S*6, we assigned values to the don't-cares so that *Z* and Nextstate are independent of *X*. The second process represents the state register. Whenever the rising edge of the clock occurs, the State is updated to the Nextstate value, so CLK appears in the sensitivity list. A typical sequence of execution for the two processes is as follows:

1. *X* changes and the first process executes. New values of *Z* and NextState are computed.
2. The clock falls, and the second process executes. Because CLK = '0', nothing happens.
3. The clock rises, and the second process executes again. Because CLK = '1', State is set equal to the Nextstate value.
4. If State changes, the first process executes again. New values of *Z* and Nextstate are computed.

A simulator command file which can be used to test Figure 17-17 follows:

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

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

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

signal_name gets the value v1 at time t1, the value v2 at time t2, etc. *X* is '0' at time 0 ns, changes to '1' at time 350 ns, changes to '0' at time 550 ns, etc. The *X* input corresponds to the sequence 0010 1001, and only the times at which *X* changes are

FIGURE 17-17
Behavioral Model for Table 17-2

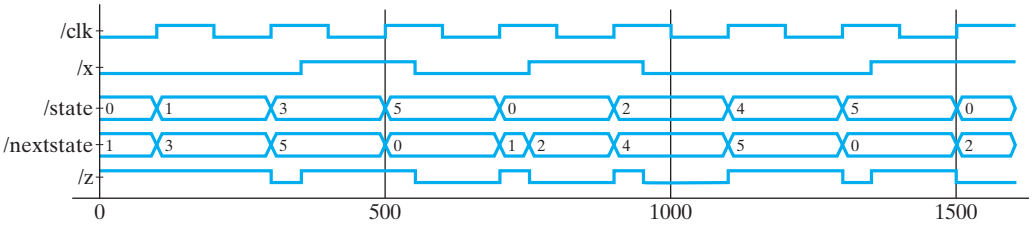

```
-- This is a behavioral model of a Mealy state machine (Table 17-2) based on its state
-- table. The output (Z) and next state are computed before the active edge of the clock.
-- The state change occurs on the rising edge of the clock.

1   entity SM17_2 is
2     port (X, CLK: in bit;
3       Z: out bit);
4   end SM17_2;

5   architecture Table of SM17_2 is
6     signal State, Nextstate: integer range 0 to 6 := 0;
7   begin
8     process(State, X)       -- Combinational Circuit
9     begin
10      case State is
11      when 0 = >
12        if X = '0' then Z <= '1'; Nextstate <= 1;
13        else Z <= '0'; Nextstate <= 2; end if;
14      when 1 =>
15        if X = '0' then Z <= '1'; Nextstate <= 3;
16        else Z <= '0'; Nextstate <= 4; end if;
17      when 2 =>
18        if X = '0' then Z <= '0'; Nextstate <= 4;
19        else Z <= '1'; Nextstate <= 4; end if;
20      when 3 =>
21        if X = '0' then Z <= '0'; Nextstate <= 5;
22        else Z <= '1'; Nextstate <= 5; end if;
23      when 4 =>
24        if X = '0' then Z <= '1'; Nextstate <= 5;
25        else Z <= '0'; Nextstate <= 6; end if;
26      when 5 =>
27        if X = '0' then Z <= '0'; Nextstate <= 0;
28        else Z <= '1'; Nextstate <= 0; end if;
29      when 6 =>
30        Z <= '1'; Nextstate <= 0;
31      end case;
32    end process;

33    process (CLK)       -- State Register
34    begin
35      if CLK'event and CLK = '1' then       -- rising edge of clock
36        State <= Nextstate;
37      end if;
38    end process;
39  end Table;
```

FIGURE 17-18 Waveforms for Figure 17-17



specified. Execution of the preceding command file produces the waveforms shown in Figure 17-18.

The behavioral VHDL model of Figure 17-17 is based on the state table. After we have derived the next-state and output equations from the state table, we can write a data flow VHDL model based on these equations. The VHDL model of Figure 17-19 is based on the next-state and output equations that are derived in Figure 16-3 using the state assignment of Figure 16-2. The flip-flops are updated in a process which is

FIGURE 17-19 Sequential Machine Model Using Equations


```
-- The following is a description of the sequential machine of Table 17-2 in terms
-- of its next-state equations. The following state assignment was used:
-- S0--> 0; S1--> 4; S2--> 5; S3--> 7; S4--> 6; S5--> 3; S6--> 2

1     entity SM17_2 is
2        port (X, CLK: in bit;
3              Z: out bit);
4     end SM17_2;

5     architecture Equations1_4 of SM17_2 is
6        signal Q1, Q2, Q3: bit;
7     begin
8        process(CLK)
9        begin
10          if CLK'event and CLK = '1' then        -- rising edge of clock
11             Q1 <= not Q2 after 10 ns;
12             Q2 <= Q1 after 10 ns;
13             Q3 <= (Q1 and Q2 and Q3) or (not X and Q1 and not Q3) or
14                          (X and not Q1 and not Q2) after 10 ns;
15          end if;
16       end process;
17       Z <= (not X and not Q3) or (X and Q3) after 20 ns;
18    end Equations1_4;
```

sensitive to CLK. When the rising edge of the clock occurs, Q1, Q2, and Q3 are all assigned new values. A 10-ns delay is included to represent the propagation delay between the active edge of the clock and the change of the flip-flop outputs. Even though the assignment statements in the process are executed sequentially, Q1, Q2, and Q3 are all scheduled to be updated at the same time, T + 10 ns, where T is the time at which the rising edge of the clock occurred. Thus, the old value of Q1 is used to compute $Q2^+$, and the old values of Q1, Q2, and Q3 are used to compute $Q3^+$. The concurrent assignment statement for Z causes Z to be updated whenever a change in X or Q3 occurs. The 20-ns delay represents two gate delays.

After we have designed a sequential circuit using components such as gates and flip-flops, we can write a structural VHDL model based on the actual interconnection of these components. Figure 17-20 shows a structural VHDL representation of the circuit of Figure 16-4. Seven NAND gates, three D flip-flops, and one inverter are used. All of these components are defined in a library named BITLIB.

FIGURE 17-20
Structural Model of Sequential Machine


```
-- The following is a STRUCTURAL VHDL description of the circuit of Figure 16-4.

1     library BITLIB;
2     use BITLIB.bit_pack.all;

3     entity SM16_4 is
4        port (X, CLK: in bit;
5              Z: out bit);
6     end SM16_4;

7     architecture Structure of SM16_4 is
8        signal A1, A2, A3, A5, A6, D3: bit: = '0';
9        signal Q1, Q2, Q3: bit: = '0';
10       signal Q1N, Q2N, Q3N, XN: bit: = '1';
11    begin
12       I1: Inverter port map (X, XN);
13       G1: Nand3 port map (Q1, Q2, Q3, A1);
14       G2: Nand3 port map (Q1, Q3N, XN, A2);
15       G3: Nand3 port map (X, Q1N, Q2N, A3);
16       G4: Nand3 port map (A1, A2, A3, D3);
17       FF1: DFF port map (Q2N, CLK, Q1, Q1N);
18       FF2: DFF port map (Q1, CLK, Q2, Q2N);
19       FF3: DFF port map (D3, CLK, Q3, Q3N);
20       G5: Nand2 port map (X, Q3, A5);
21       G6: Nand2 port map (XN, Q3N, A6);
22       G7: Nand2 port map (A5, A6, Z);
23    end Structure;
```

The component declarations and definitions are contained in a package called bit_pack. The **library** and **use** statements are explained in Section 10.7. Because the NAND gates and D flip-flops are declared as components in bit_pack, they are not explicitly declared in the VHDL code. Because Q1, Q2, and Q3 are initialized to '0', the complementary flip-flop outputs (Q1N, Q2N, and Q3N) are initialized to '1'. G1 is a 3-input NAND gate with inputs Q1, Q2, Q3, and output A1. FF1 is a D flip-flop (see Figure 17-1) with the D input connected to Q2N. All of the gates and flip-flops in the bit_pack have a default delay of 10 ns. Executing the following simulator command file produces the waveforms of Figure 17-21.

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

Next, we will implement the state machine of Table 16-6(a) using a ROM, as shown in Figure 16-10. In the VHDL code (Figure 17-22), we have used packages from the IEEE library and IEEE Standard Logic because synthesis tools often use std_logic and std_logic_vector as default types. The constant array ROM1 represents the truth table of Table 16-6(c), which is stored in the ROM. Reading data from the ROM is accomplished by four concurrent statements. First, the ROM address, which is the index into the array, is formed by concatenating *X* and Q to form a 4-bit vector. The index is converted from a std_logic_vector to an integer by calling the conv_integer function. The ROM1 output is split into the D vector that represents the next state and the *Z* output. The process updates the state register on the rising edge of the clock.

Next, we will write behavioral VHDL code for the state table given in Table 13-4. We will use a two-process model as we did in Figure 17-17. We will use nested case statements instead of using if-then-else because the state table has more columns. Figure 17-23 shows a portion of the VHDL code for the combinational part of the circuit. The first case statement branches on the state, and the nested case statement for each state defines the Nextstate and outputs by branching on X12 (= X1&X2). The second process (not shown) that updates the state register is identical to the one in Figure 17-17.

FIGURE 17-21 Waveforms for Figure 16-4

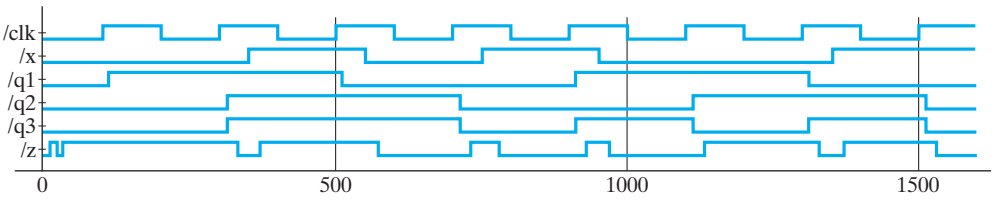


FIGURE 17-22
Sequential Machine Using a ROM


```
1   library IEEE;
2   use IEEE.STD_LOGIC_1164.ALL;
3   use IEEE.STD_LOGIC_ARITH.ALL;
4   use IEEE.STD_LOGIC_UNSIGNED.ALL;

5   entity SM16_6 is
6      Port ( X : in std_logic;
7         CLK : in std_logic;
8         Z : out std_logic);
9   end SM16_6;

10  architecture ROM of SM16_6 is
11  type ROM16X4 is array (0 to 15) of std_logic_vector (0 to 3);
12  constant ROM1: ROM16X4 : = ("1001", "1011", "0100", "0101",
13                              "1101", "0000", "1000", "0000",
14                              "0010", "0100", "1100", "1101",
15                              "0110", "1000", "0000", "0000");
16  signal Q, D: std_logic_vector (1 to 3) := "000";
17  signal Index, Romout: std_logic_vector (0 to 3);
18  begin
19  Index <= X&Q; -- X&Q is a 4-bit vector: X Q1 Q2 Q3
20  Romout <=ROM1 (conv_integer(Index));
            -- this statement reads the output from the ROM
            -- conv_integer converts Index to an Integer
21  Z <=Romout(0);
22  D <=Romout(1 to 3);

23  process (CLK)
24     begin
25        if CLK'event and CLK = '1' then Q <= D; end if;
26     end process;
27  end ROM;
```

A Moore machine can be modeled using two processes just like a Mealy machine. For example, the first row of the Moore table of Table 14-3 could be modeled within the combinational process as follows:

```
case state is
   when 0 =>
      Z <= '0';
      if X = '0' then Nextstate <= 0; else Nextstate <= 1; end if;
   . . .
```

Note that the Z output is specified before X is tested because the Moore output only depends on the state and not on the input.

FIGURE 17-23
Partial VHDL Code for the Table of Figure 13-4


```
1     entity Table_13_4 is
2        port(X1, X2, CLK: in bit; Z1, Z2: out bit);
3     end Table_13_4;

4     architecture T1 of Table_13_4 is
5     signal State, Nextstate: integer range 0 to 3: = 0;
6     signal X12: bit_vector(0 to 1);
7     begin
8        X12 <= X1&X2;
9        process(State, X12)
10       begin
11          case State is
12          when 0 =>
13             case X12 is
14                when "00" => Nextstate <= 3; Z1 <= '0'; Z2 <= '0';
15                when "01" => Nextstate <= 2; Z1 <= '1'; Z2 <= '0';
16                when "10" => Nextstate <= 1; Z1 <= '1'; Z2 <= '1';
17                when "11" => Nextstate <= 0; Z1 <= '0'; Z2 <= '1';
18                when others =>  null;              -- not required since X is a bit_vector
19             end case;
20          when 1 = >                              -- code for state 1 goes here, etc.
```

17.5 Synthesis of VHDL Code

The synthesis software for VHDL translates the VHDL code to a circuit description that specifies the needed components and the connections between the components. When writing VHDL code, you should always keep in mind that you are designing hardware, not simply writing a computer program. Each VHDL statement implies certain hardware requirements. So poorly written VHDL code may result in poorly designed hardware. Even if VHDL code gives the correct result when simulated, it may not result in hardware that works correctly when synthesized. Timing problems may prevent the hardware from working properly even though the simulation results are correct.

The synthesis software tries to infer the components needed by "looking" at the VHDL code. In order for code to synthesize correctly, certain conventions must be followed. In order to infer flip-flops or registers that change state on the rising edge of a clock signal, an **if** clause of the form

if clock'event **and** clock = '1' **then . . . end if**;

is required by most synthesizers. For every assignment statement between **then** and **end if** in the preceding statement, a signal on the left side of the assignment will cause

creation of a register or flip-flop. The moral to this story is: If you do not want to create unnecessary flip-flops, do not put the signal assignments in a clocked process. If clock' event is omitted, the synthesizer may produce latches instead of flip-flops.

Before synthesis is started, we must specify a target device so that the synthesizer knows what components are available. We will assume that the target is a CPLD or FPGA that has D flip-flops with clock enable (D-CE flip-flops). We will synthesize the VHDL code for a left-shift register (Figure 17-10). Q and D are 4-bit vectors. Because updates to Q follow "CLK'event and CLK = '1' then", this infers that Q must be a register composed of four flip-flops, which we will label Q3, Q2, Q1, and Q0. Because the flip-flops can change state when Clr, Ld, or Ls is '1', we connect the clock enables to an OR gate whose output is Clr + Ld + Ls. Then, we connect gates to the D inputs to select the data to be loaded into the flip-flops. If Clr = 0 and Ld = 1, D is loaded into the register on the rising clock edge. If Clr = Ld = 0 and Ls = 1, then Q2 is loaded into Q3, Q1 is loaded into Q2, etc. Figure 17-24 shows the logic circuit for the first two flip-flops. If Clr = 1, the D flip-flop inputs are 0, and the register is cleared.

A VHDL synthesizer cannot synthesize delays. Clauses of the form "**after** time-expression" will be ignored by most synthesizers, but some synthesizers require that **after** clauses be removed. Although the initial values for signals may be specified in port and signal declarations, these initial values are ignored by the synthesizer. A reset signal should be provided if the hardware must be set to a specific initial state. Otherwise, the initial state of the hardware may be unknown, and the hardware may malfunction. When an integer signal is synthesized, the integer is represented in hardware by its binary equivalent. If the range of an integer is not specified, the synthesizer will assume the maximum number of bits, usually 32. Thus,

```
signal count: integer range 0 to 7;
```

would result in a 3-bit counter, but

```
signal count: integer;
```

could result in a 32-bit counter.

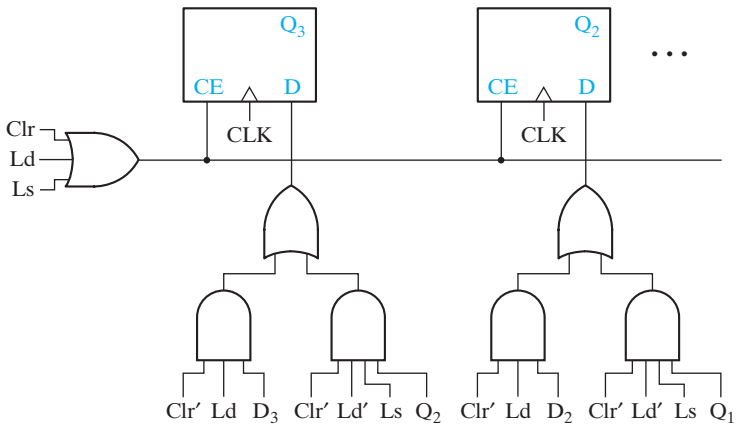


FIGURE 17-24
Synthesis of VHDL Code From Figure 17-10

VHDL signals retain their current values until they are changed. This can result in the creation of unwanted latches when the code is synthesized. For example, in a combinational process, the statement

```
if X = '1' then B <= 1; end if;
```

would create latches to hold the value of B when *X* changed to '0'. To avoid the creation of unwanted latches in a combinational process, always include an **else** clause in every if statement. For example,

```
if X = '1' then B <= 1 else B <= 2; end if;
```

would create a MUX to switch the value of B from 1 to 2.

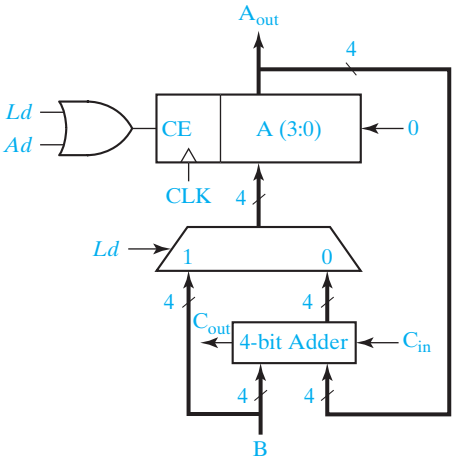
Figure 17-25 shows the VHDL code for a 4-bit adder with accumulator. When the synthesizer analyses this code, it infers the presence of a 4-bit adder with carry in and carry out from line 14. When it analyses the clocked process, it infers from

FIGURE 17-25 VHDL Code and Synthesis Results for 4-Bit Adder with Accumulator


```
1   library IEEE;
2   use IEEE.STD_LOGIC_1164.ALL;
3   use IEEE.STD_LOGIC_UNSIGNED.ALL;

4   entity adder is
5     Port (B: in std_logic_vector(3 downto 0);
6       Ld, Ad, Cin, CLK : in std_logic;
7       Aout : out std_logic_vector(3 downto 0);
8       Cout : out std_logic);
9   end adder;

10  architecture Behavioral of adder is
11  signal A : std_logic_vector(3 downto 0);
12  signal Addout : std_logic_vector(4 downto 0);
13  begin
14    Addout <= ('0' & A) + B + Cin;
15    Cout <= Addout(4);
16    Aout <= A;
17  process(CLK)
18  begin
19    if CLK'event and CLK = '1' then
20      if Ld = '1' then A <= B;
21      elsif Ad = '1'
22        then A<= Addout(3 downto 0);
23      end if;
24    end if;
25  end process;
26  end Behavioral;
```



lines 11, 19, and 20 that A is a 4-bit register that changes state on the rising clock edge. It also infers the presence of a 4-wide 2-to-1 multiplexer to select either B or the adder output to load into A. Because A is loaded when $Ld = 1$ or $Ad = 1$, the CE input to the register is $Ld + Ad$. At this point, a block diagram of the synthesized code resembles that shown in Figure 17-25. The synthesizer output is then optimized and fit into a specific target device.

17.6 More About Processes and Sequential Statements

An alternative form for a process uses wait statements instead of a sensitivity list. A process cannot have both a wait statement and a sensitivity list. A process with wait statements may have the form

```
process
begin
   sequential-statements
   wait-statement
   sequential-statements
   wait-statement
   . . .
end process;
```

This process will execute the sequential-statements until a wait statement is encountered. Then, it will wait until the specified wait condition is satisfied. It will then execute the next set of sequential-statements until another wait is encountered. It will continue in this manner until the end of the process is reached. Then, it will start over again at the beginning of the process.

Wait statements can be of three different forms:

```
wait on sensitivity-list;
wait for time-expression;
wait until Boolean-expression;
```

The first form waits until one of the signals on the sensitivity list changes. For example, **wait on** A,B,C; waits until A, B, or C changes and, then, execution proceeds. The second form waits until the time specified by time expression has lapsed. If **wait for** 5 ns is used, the process waits for 5 ns before continuing. If **wait for** 0 ns is used, the wait is for one Δ time. Wait statements of the form **wait for** xx ns are useful for writing VHDL code for simulation; however, they should not be used when writing VHDL code for synthesis because they are not synthesizable. For the third form of wait statement, the Boolean expression is evaluated whenever one of the signals in the expression changes, and the process continues execution when the expression evaluates to TRUE. For example,

```
wait until A = B;
```

will wait until either A or B changes. Then, A = B is evaluated, and if the result is TRUE, the process will continue, or else the process will continue to wait until A or B changes again and A = B is TRUE.

After a VHDL simulator is initialized, it executes each process with a sensitivity list one time through, and then waits at the beginning of the process for a change in one of the signals on the sensitivity list. If a process has a wait statement, it will initially execute until a wait statement is encountered. Therefore, the following process is equivalent to the one in Figure 17-15:

```
process
begin
   C <= A and B after 5 ns;
   E <= C or D after 5 ns;
   wait on A, B, C, D;
end process;
```

The wait statement at the end of the process replaces the sensitivity list at the beginning. In this way both processes will initially execute the sequential statements one time and, then, wait until A, B, C, or D changes.

The order in which sequential statements are executed in a process is not necessarily the order in which the signals are updated. Consider the following example:

```
process
begin
   wait until CLK'event and CLK = '1';
   A <= E after 10 ns;        -- (1)
   B <= F after 5 ns;         -- (2)
   C <= G;                    -- (3)
   D <= H after 5 ns;         -- (4)
end process;
```

This process waits for a rising clock edge. Suppose the clock rises at time = 20 ns. Statements (1), (2), (3), (4) immediately execute in sequence. A is scheduled to change to E at time = 30 ns; B is scheduled to change to F at time = 25 ns; C is scheduled to change to G at time = 20 + Δ ns; and D is scheduled to change to H at time 25 ns. As simulated time advances, first, C changes. Then, B and D change at time = 25 ns, and finally A changes at time 30 ns. When clk changes to '0', the wait statement is re-evaluated, but it keeps waiting until clk changes to '1', and then the remaining statements execute again.

If several VHDL statements in a process update the same signal at a given time, the last value overrides. For example,

```
process (CLK)
begin
   if CLK'event and CLK = '0' then
      Q <= A; Q <= B; Q <= C;
   end if;
end process;
```

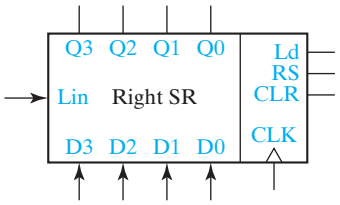
Every time CLK changes from '1' to '0', after Δ time, Q will change to C.

In this unit, we have introduced processes with sensitivity lists and processes with wait statements. The statements within a process are called sequential statements because they execute in sequence, in contrast with concurrent statements that execute only when a signal on the right-hand side changes. Signal assignment statements can be either concurrent or sequential. However, **if** and **case** statements are always sequential, yet conditional signal assignment statements and selected signal assignment statements can only be concurrent.

Problems

17.1 Write VHDL code for a T flip-flop with an active-low asynchronous clear.

17.2 Write VHDL code for the following right-shift register with synchronous clear.



17.3 A 4-bit up/down binary counter with output Q works as follows: All state changes occur on the rising edge of the CLK input, except the asynchronous clear (ClrN). When ClrN = 0, the counter is reset regardless of the values of the other inputs.

If the LOAD input is 0, the data input D is loaded into the counter.
If LOAD = ENT = ENP = UP = 1, the counter is incremented.
If LOAD = ENT = ENP = 1 and UP = 0, the counter is decremented.
If ENT = UP = 1, the carry output (CO) = 1 when the counter is in state 15.
If ENT = 1 and UP = 0, the carry output (CO) = 1 when the counter is in state 0.

(a) Write a VHDL description of the counter.
(b) Draw a block diagram and write a VHDL description of an 8-bit binary up/down counter that uses two of these 4-bit counters.

17.4 Represent the given circuit using a process with a case statement.