Chapter 5: Modeling Sequential Functionality

In Chap. 3 techniques were presented to describe the behavior of concurrent systems. The modeling techniques presented were appropriate for combinational logic because these types of circuits have outputs dependent only on the current values of their inputs. This means a model that continuously performs signal assignments provides an accurate model of this circuit behavior. When we start looking at sequential circuits (i.e., D-flip-flops, registers, finite state machine, and counters), these devices only update their outputs based upon an event, most often the edge of a clock signal. The modeling techniques presented in Chap. 3 are unable to accurately describe this type of behavior. In this chapter we describe the VHDL constructs to model signal assignments that are triggered by an event to accurately model sequential logic. We can then use these techniques to describe more complex sequential logic circuits such as finite state machines and register transfer level systems.

Learning Outcomes—After completing this chapter, you will be able to:

5.1 Describe the behavior of a VHDL process and how it is used to model sequential logic circuits.
5.2 Model combinational logic circuits using a process and conditional programming constructs.
5.3 Describe how and why signal attributes are used in VHDL models.

5.1 The Process

VHDL uses a *process* to model signal assignments that are based on an event. A process is a technique to model behavior of a system; thus, a process is placed in the VHDL architecture after the begin statement. The signal assignments within a process have unique characteristics that allow them to accurately model sequential logic. First, the signal assignments do not take place until the process ends or is suspended. Second, the signal assignments will be made only once each time the process is triggered. Finally, the signal assignments will be executed in the order that they appear within the process. This assignment behavior is called a *sequential signal assignment*. Sequential signal assignments allow a process to model register transfer level behavior where a signal can be used as both the operand of an assignment and the destination of a different assignment within the same process. VHDL provides two techniques to trigger a process, the *sensitivity list* and the *wait statement*.

5.1.1 Sensitivity Lists

A *sensitivity list* is a mechanism to control when a process is triggered (or started). A sensitivity list contains a list of signals that the process is sensitive to. If there is a transition on any of the signals in the list, the process will be triggered, and the signal assignments in the process will be made. The following is the syntax for a process that uses a sensitivity list.

B. J. LaMeres, *Quick Start Guide to VHDL*, https://doi.org/10.1007/978-3-030-04516-6_5

```
process_name : process (<signal_name1>, <signal_name2>, ...)

  -- variable declarations

  begin

     sequential_signal_assignment_1
     sequential_signal_assignment_2
                   :
end process;
```

Let's look at a simple model for a flip-flop.

Example:

```
FlipFlop : process (Clock)
  begin
     Q <= D;
end process;
```

In this example, a transition on the signal Clock (LOW to HIGH or HIGH to LOW) will trigger the process. The signal assignment of D to Q will be executed once the process ends. When the signal Clock is not transitioning, the process will not trigger, and no assignments will be made to Q, thus modeling the behavior of Q holding its last value. This behavior is close to modeling the behavior of a real D-flip-flop, but more constructs are needed to model behavior that is sensitive to only a particular type of transition (i.e., rising or falling edge). These constructs will be covered later.

5.1.2 Wait Statements

A *wait statement* is a mechanism to suspend (or stop) a process and allow signal assignments to be executed without the need for the process to end. When using a wait statement, a sensitivity list is not used. Without a sensitivity list, the process will immediately trigger. Within the process, the wait statement is used to stop and start the process. There are three ways in which wait statements can be used. The first is an indefinite wait. In the following example, the process does not contain a sensitivity list, so it will trigger immediately. The keyword **wait** is used to suspend the process. Once this statement is reached, the signal assignments to Y1 and Y2 will be executed, and the process will suspend indefinitely.

Example:

```
Proc_Ex1 : process
  begin
    Y1 <= '0';
    Y2 <= '1';
    wait;
end process;
```

The second technique to use a wait statement to suspend a process is to use it in conjunction with the keyword **for** and a time expression. In the following example, the process will trigger immediately since it does not contain a sensitivity list. Once the process reaches the wait statement, it will suspend and execute the first signal assignment to CLK (CLK <= '0'). After 5 ns, the process will start again. Once it reaches the second wait statement, it will suspend and execute the second signal assignment to CLK (CLK <= '1'). After another 5 ns, the process will start again and immediately end due to the *end process* statement. After the process ends, it will immediately trigger again due to the lack of a sensitivity list and repeat the behavior just described. This behavior will continue indefinitely. This example creates a square wave called CLK with a period of 10 ns.

Example:

```
Proc_Ex2 : process
  begin
     CLK <= '0'; wait for 5 ns;
     CLK <= '1';  wait for 5 ns;
end process;
```

The third technique to use a wait statement to suspend a process is to use it in conjunction with the keyword **until** and a Boolean condition. In the following example, the process will again trigger immediately because there is not a sensitivity list present. The process will then immediately suspend and only resume once a Boolean condition becomes true (i.e., Counter > 15). Once this condition is true, the process will start again. Once it reaches the second wait statement, it will execute the first signal assignment to RollOver (RollOver <= '1'). After 1 ns, the process will resume. Once the process ends, it will execute the second signal assignment to RollOver (RollOver <= '0').

Example:

```
Proc_Ex3 : process
  begin
     wait until (Counter > 15);          -- first wait statement
     RollOver <= '1'; wait for 1 ns;     -- second wait statement
     RollOver <= '0';
end process;
```

Wait statements are typically not synthesizable and are most often used for creating stimulus patterns in test benches.

5.1.3 Sequential Signal Assignments

One of the more confusing concepts of a process is how sequential signal assignments behave. The rules of signal assignments within a process are as follows:

- Signals cannot be declared within a process.
- Signal assignments do not take place until the process ends or suspends.
- Signal assignments are executed in the sequence they appear in the process (once the process ends or process suspends).

Let's look at an example of how signals behave in a process. Example 5.1 shows the behavior of sequential signal assignments when executed within a process. Intuitively, we would assume that F will be the complement of A; however, due to the way that sequential signal assignments are performed within a process, this is not the case. In order to understand this behavior, let's look at the situation where A transitions from a 0 to a 1 with B = 0 and F = 0 initially. This transition triggers the process since A is listed in the sensitivity list. When the process triggers, A = 1 since this is where the input resides after the triggering transition. The first signal assignment (B <= A) will cause B = 1, but this assignment occurs only after the process ends. This means that when the second signal assignment is evaluated (F <= not B), it uses the initial value of B from when the process triggered (B = 0) since B is not updated to a 1 until the process ends. The second assignment yields F = 1. When the process ends, A = 1, B = 1, and F = 1. The behavior of this process will always result in A = B = F. This is counterintuitive because the statement F <= not B leads us to believe that F will always be the complement of A and B; however, this is not the case due to the way that signal assignments are only updated in a process upon suspension or when the process ends.

Example: Behavior of Sequential Signal Assignments within a Process

For the following system:

```
entity Ex is
  port (A   : in  bit;
        F   : out bit);
end entity;
```

The output F will match the input A when modeled with the following process:

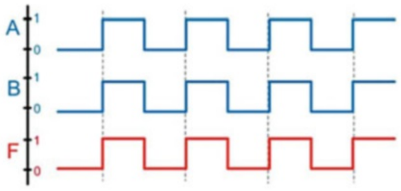
```
architecture Ex_arch of Ex is

  signal B : bit;

begin

  Proc_Ex : process (A)
    begin
      B <= A;
      F <= not B;
  end process;

end architecture;
```



Example 5.1
Behavior of sequential signal assignments within a process

Now let's consider how these assignments behave when executed as concurrent signal assignments. Example 5.2 shows the behavior of the same signal assignments as in Example 5.1, but this time outside of a process. In this model, the statements are executed concurrently and produce the expected behavior of F being the complement of A.

Example: Behavior of Concurrent Signal Assignments outside a Process

For the following system:

```
entity Ex is
  port (A   : in  bit;
        F   : out bit);
end entity;
```

The output F will be the complement of input A when the assignments are executed concurrently.

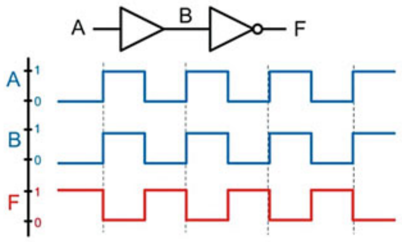
```
architecture Ex_arch of Ex is

  signal B : bit;

begin

     B <= A;
     F <= not B;

end architecture;
```



Example 5.2
Behavior of concurrent signal assignments outside a process

While the behavior of the sequential signal assignments initially seems counterintuitive, it is necessary to model the behavior of sequential storage devices and will become clear once more VHDL constructs have been introduced.

5.1.4 Variables

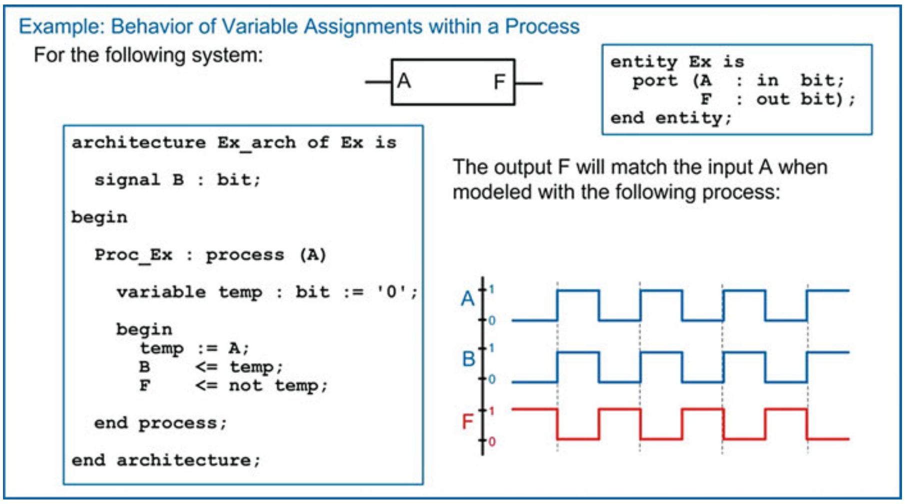
There are situations inside of processes in which it is desired for assignments to be made instantaneously instead of when the process suspends. For these situations, VHDL provides the concept of a *variable.* A variable has the following characteristics:

- Variables only exist within a process.
- Variables are defined in a process before the begin statement.
- Once the process ends, variables are removed from the system. This means that assignments to variables cannot be made by systems outside of the process.
- Assignments to variables are made using the ":=" operator.
- Assignments to variables are made instantaneously.

A variable is declared before the begin statement in a process. The syntax for declaring a variable is as follows:

```
variable variable_name : <type> := <initial_value>;
```

Let's reconsider the example in Example 5.1, but this time we'll use a variable in order to accomplish instantaneous signal assignments within the process. Example 5.3 shows this approach to model the behavior where F is the complement of A.



Example: Behavior of Variable Assignments within a Process

For the following system:

```
entity Ex is
  port (A  : in  bit;
        F  : out bit);
end entity;
```

```
architecture Ex_arch of Ex is

  signal B : bit;

begin

  Proc_Ex : process (A)

    variable temp : bit := '0';

    begin
      temp := A;
      B    <= temp;
      F    <= not temp;

  end process;

end architecture;
```

The output F will match the input A when modeled with the following process:

Example 5.3
Variable assignment behavior

Concept Check

CC5.1 If a model of a combinational logic circuit excludes one of its inputs from the sensitivity list, what is the implied behavior?

(A) A storage element because the output will be held at its last value when the unlisted input transitions.

(B) An infinite loop.

(C) A don't care will be used to form the minimal logic expression.

(D) Not applicable because this syntax will not compile.

5.2 Conditional Programming Constructs

One of the more powerful features that processes provide in VHDL is the ability to use conditional programming constructs such as if/then clauses, case statements, and loops. These constructs are only available within a process, but their use is not limited to modeling sequential logic. As we'll see, the characteristics of a process also support modeling of combinational logic circuits, so these conditional constructs are a very useful tool in VHDL. This provides the ability to model both combinational and sequential logic using the more familiar programming language constructs.

5.2.1 If/Then Statements

An *if/then* statement provides a way to make conditional signal assignments based on Boolean conditions. The **if** portion of statement is followed by a Boolean condition that if evaluated TRUE will cause the signal assignment after the **then** statement to be performed. If the Boolean condition is evaluated FALSE, no assignment is made. VHDL provides multiple variants of the if/then statement. An *if/then/else* statement provides a final signal assignment that will be made if the Boolean condition is evaluated false. An *if/then/elsif* statement allows multiple Boolean conditions to be used. The syntax for the various forms of the VHDL if/then statement are as follows:

```
if boolean_condition then sequential_statement
end if;

if boolean_condition then sequential_statement_1
else sequential_statement_2
end if;

if boolean_condition_1 then sequential_statement_1
elsif boolean_condition_2 then sequential_statement_2
  :
  :
elsif boolean_condition_n then sequential_statement_n
end if;

if boolean_condition_1 then sequential_statement_1
elsif boolean_condition_2 then sequential_statement_2
  :
  :
elsif boolean_condition_n then sequential_statement_n
else sequential_statement_n+1
end if;
```

Let's take a look at using an if/then statement to describe the behavior of a combinational logic circuit. Recall that a combinational logic circuit is one in which the output depends on the instantaneous values of the inputs. This behavior can be modeled by placing all of the inputs to the circuit in the sensitivity list of a process. A change on any of the inputs in the sensitivity list will trigger the process and cause the output to be updated. Example 5.4 shows how to model a 3-input combinational logic circuit using if/then statements within a process.

Example: Using If/Then Statements to Model Combinational Logic

Implement the following truth table using an if/then statement within a process.

A	B	C	F
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

```
entity SystemX is
   port (A, B, C  :  in  bit;
         F        :  out bit);
end entity;
```

Recall that an if/then statement is only legal within a process. In order to create a process that models combinational logic, we need to list each of the inputs to the circuit in the sensitivity list. This will cause the process to trigger and make an assignment to the output whenever there is a change on any of the inputs.

```
architecture SystemX_arch of SystemX is

begin

  SystemX_Proc : process (A, B, C)
    begin
      if    (A='0' and B='0' and C='0') then F <= '1';
      elsif (A='0' and B='0' and C='1') then F <= '0';
      elsif (A='0' and B='1' and C='0') then F <= '1';
      elsif (A='0' and B='1' and C='1') then F <= '0';
      elsif (A='1' and B='0' and C='0') then F <= '0';
      elsif (A='1' and B='0' and C='1') then F <= '0';
      elsif (A='1' and B='1' and C='0') then F <= '1';
      elsif (A='1' and B='1' and C='1') then F <= '0';
      end if;
  end process;

end architecture;
```

A more compact version of this behavior can be created by taking advantage of the else clause. In this model, only Boolean conditions are listed for outputs corresponding to 1's.

```
architecture SystemX_arch of SystemX is

begin

  SystemX_Proc : process (A, B, C)
    begin
      if    (A='0' and B='0' and C='0') then F <= '1';
      elsif (A='0' and B='1' and C='0') then F <= '1';
      elsif (A='1' and B='1' and C='0') then F <= '1';
      else F <= '0';
      end if;
  end process;

end architecture;
```

Example 5.4
Using if/then statements to model combinational logic

5.2.2 Case Statements

A *case* statement is another technique to model signal assignments based on Boolean conditions. As with the if/then statement, a case statement can only be used inside of a process. The statement begins with the keyword **case** followed by the input signal name that assignments will be based off of. The input signal name can be optionally enclosed in parentheses for readability. The keyword **when** is used to specify a particular value (or choice) of the input signal that will result in associated sequential signal assignments. The assignments are listed after the **=>** symbol. The following is the syntax for a case statement.

```
case (input_name) is
    when choice_1 => sequential_statement(s);
    when choice_2 => sequential_statement(s);
                                 :
                                 :
    when choice_n => sequential_statement(s);
end case;
```

When not all the possible input conditions (or choices) are specified, a **when others** clause is used to provide signal assignments for all other input conditions. The following is the syntax for a case statement that uses a *when others* clause.

```
case (input_name) is
    when choice_1 => sequential_statement(s);
    when choice_2 => sequential_statement(s);
                                 :
                                 :
    when others  => sequential_statement(s);
end case;
```

Multiple choices that correspond to the same signal assignments can be pipe-delimited in the case statement. The following is the syntax for a case statement with pipe-delimited choices.

```
case (input_name) is
    when choice_1 | choice_2 => sequential_statement(s);
    when others              => sequential_statement(s);
end case;
```

The input signal for a case statement must be a single signal name. If multiple scalars are to be used as the input expression for a case statement, they should be concatenated either outside of the process resulting in a new signal vector or within the process resulting in a new variable vector. Example 5.5 shows how to model a 3-input combinational logic circuit using case statements within a process.

Example: Using Case Statements to Model Combinational Logic

Implement the following truth table using a case statement within a process.

A	B	C	F
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

```
entity SystemX is
   port (A, B, C  :  in  bit;
         F        :  out bit);
end entity;
```

A case statement is only legal within a process. In the following example, the three input scalars (A,B,C) are concatenated into a new variable for use as the input signal to the case statement.

```
architecture SystemX_arch of SystemX is

begin

  SystemX_Proc : process (A, B, C)

    variable ABC : bit_vector (2 downto 0) := "000";

    begin

      ABC := A & B & C;

      case (ABC) is
        when "000" => F <= '1';
        when "001" => F <= '0';
        when "010" => F <= '1';
        when "011" => F <= '0';
        when "100" => F <= '0';
        when "101" => F <= '0';
        when "110" => F <= '1';
        when "111" => F <= '0';
      end case;

  end process;

end architecture;
```

More compact forms of the case statement can be created using the *when others* clause and pipe delimited inputs.

```
case (ABC) is
  when "000"   => F <= '1';
  when "010"   => F <= '1';
  when "110"   => F <= '1';
  when others => F <= '0';
end case;
```

```
case (ABC) is
 when "000" | "010" | "110"   => F <= '1';
 when others                  => F <= '0';
end case;
```

Example 5.5
Using case statements to model combinational logic

If/then statements can be embedded within a case statement, and, conversely, case statements can be embedded within an if/then statement.

5.2.3 Infinite Loops

A *loop* within VHDL provides a mechanism to perform repetitive assignments infinitely. This is useful in test benches for creating stimulus such as clocks or other periodic waveforms. A loop can only be used within a process. The keyword **loop** is used to signify the beginning of the loop. Sequential signal assignments are then inserted. The end of the loop is signified with the keywords **end loop**. Within the loop, the *wait for*, *wait until*, and *after* statements are all legal. Signal assignments within a loop will be executed repeatedly forever unless an **exit** or **next** statement is encountered. The *exit* clause provides a Boolean condition that will force the loop to end if the condition is evaluated true. When using the exit statement, an additional signal assignment is typically placed after the loop to provide the desired behavior when the loop is not active. Using flow control statements such as *wait for* and *wait after* provide a means to avoid having the loop immediately executed again after exiting. The *next* clause provides a way to skip the remaining signal assignments and begin the next iteration of the loop. The following is the syntax for an infinite loop in VHDL.

```
loop
  exit when boolean_condition;     -- optional exit statement
  next when boolean_condition;     -- optional next statement
  sequential_statement(s);
end loop;
```

Consider the following example of an infinite loop that generates a clock signal (CLK) with a period of 100 ns. In this example, the process does not contain a sensitivity list, so a wait statement must be used to control the signal assignments. This process in this example will trigger immediately and then enter the infinite loop and never exit.

Example:

```
Clock_Proc1 : process
  begin
    loop
      CLK <= not CLK;
      wait for 50 ns;
    end loop;
  end process;
```

Now consider the following loop example that will generate a clock signal with a period of 100 ns with an enable (EN) line. This loop will produce a periodic clock signal as long as EN = 1. When EN = 0, the clock output will remain at CLK = 0. An exit condition is placed at the beginning of the loop to check if EN = 0. If this condition is true, the loop will exit, and the clock signal will be assigned a 0. The process will then wait until EN = 1. Once EN = 1, the process will end and then immediately trigger again and reenter the loop. When EN = 1, the clock signal will be toggled (CLK <= not CLK) and then wait for 50 ns. This toggling behavior will repeat as long as EN = 1.

Example:

```
Clock_Proc2 : process
  begin
    loop
      exit when EN='0';
      CLK <= not CLK;
      wait for 50 ns;
    end loop;

    CLK <= '0';
    wait until EN='1';

  end process;
```

It is important to keep in mind that infinite loops that continuously make signal assignments without the use of sensitivity lists or wait statements will cause logic simulators to hang.

5.2.4 While Loops

A *while loop* provides a looping structure with a Boolean condition that controls its execution. The loop will only execute as long as its condition is evaluated true. The following is the syntax for a VHDL while loop.

```
while boolean_condition loop
  sequential_statement(s);
end loop;
```

Let's implement the previous example of a loop that generates a clock signal (CLK) with a period of 100 ns as long as EN = 1. The Boolean condition for the while loop is EN = 1. When EN = 1, the loop will be executed indefinitely. When EN = 0, the while loop will be skipped. In this case, an additional signal assignment is necessary to model the desired behavior when the loop is not used (i.e., CLK = 0).

Example:

```
Clock_Proc3 : process
  begin
   while (EN='1') loop
    CLK <= not CLK;
    wait for 50 ns;
   end loop;

   CLK <= '0';
   wait until EN='1';

  end process;
```

5.2.5 For Loops

A *for loop* provides the ability to create a loop that will execute a pre-defined number of times. The range of the loop is specified with integers (*min, max*) at the beginning of the for loop. A *loop variable* is implicitly declared in the loop that will increment (or decrement) from *min* to *max* of the range. The loop variable is of type integer. If it is desired to have the loop variable increment from min to max, the keyword **to** is used when specifying the range of the loop. If it is desired to have the loop variable decrement max to min, the keyword **downto** is used when specifying the range of the loop. The loop variable can be used within the loop as an index for vectors; thus the for loop is useful for automatically accessing and assigning multiple signals within a single loop structure. The following is the syntax for a VHDL for loop in which the loop variable will increment from min to max of the range:

```
for loop_variable in min to max loop
   sequential_statement(s);
end loop;
```

The following is the syntax of a for loop in which the loop variable will decrement from max to min of the range:

```
for loop_variable in max downto min loop
   sequential_statement(s);
end loop;
```

For loops are useful for test benches in which a range of patterns are to be created. For loops are also synthesizable as long as the complete behavior of the desired system is described by the loop. The following is an example of creating a simple counter using the loop variable. The signal Count_Out in this example is of type integer. This allows the loop variable *i* to be assigned to Count_Out each time through the loop since the loop variable is also of type integer. This counter will count from 0 to 15 and then repeat. The count will increment every 50 ns.

Example:

```
Counter_Proc : process
  begin
    for i in 0 to 15 loop
      Count_Out <= i;
      wait for 50 ns;
    end loop;
  end process;
```

Concept Check

CC5.2 When using an if/then statement to model a combinational logic circuit, is using the else clause the same as using don't cares when minimizing a logic expression with a K-map?

(A) Yes. The else clause allows the synthesizer to assign whatever output values are necessary in order to create the most minimal circuit.

(B) No. The else clause explicitly states the output values for all input codes not listed in the if/elsif portion of the if/then construct. This is the same as filling in the truth table with specific values for all input codes covered by the else clause and the synthesizer will create the logic expression accordingly.

5.3 Signal Attributes

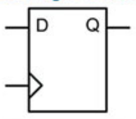
There are situations where we want to describe behavior that is based on more than just the current value of a signal. For example, a real D-flip-flop will only update its outputs on a particular type of transition (i.e., rising or falling). In order to model this behavior, we need to specify more information about the signal. This is accomplished by using *attributes.* Attributes provide additional information about a signal other than just its present value. An attribute can provide information such as past values, whether an assignment was made to a signal or when the last time an assignment resulted in a value change. A signal attribute is implemented by placing an apostrophe (‘) after the signal name and then listing the VHDL attribute keyword. Different attributes will result in different output types. Attributes that yield Boolean output types can be used as inputs to Boolean decision conditions for other VHDL constructs. Other attributes can be used to define the range of new vectors by referencing the size of existing vectors or automatically defining the number of iterations in a loop. Finally, some attributes can be used to create self-checking test benches that monitor the impact of circuit delays on the functionality of a system. The following is a list of the commonly used, pre-defined VHDL signal attributes. The example signal name *A* is used to illustrate how scalar attributes operate. The example signal **B** is used to illustrate how vector attributes operate with type bit_vector (7 downto 0).

Attribute	Information returned	Type returned
A‘**event**	True when signal A changes, false otherwise	boolean
A‘**active**	True when an assignment is made to A, false otherwise	boolean
A‘**last_event**	Time when signal A last changed	time
A‘**last_active**	Time when signal A was last assigned to	time
A‘**last_value**	The previous value of A	same type as A

Attribute	Information returned	Type returned
B'**length**	Size of the vector (e.g., 8)	integer
B'**left**	Left bound of the vector (e.g., 7)	integer
B'**right**	Right bound of the vector (e.g., 0)	integer
B'**range**	Range of the vector "(7 downto 0)"	string

Signal attributes can be used to model edge sensitive behavior. Let's look at the model for a simple D-flip-flop. A process is used to model the synchronous behavior of the D-flip-flop. The sensitivity list contains only the *Clock* input. The *D* input is not included in the sensitivity list because a change on D should not trigger the process. Attributes and logical operators are not allowed in the sensitivity list of a process. As a result, the process will trigger on every edge of the clock signal. Within the process, an if/then statement is used with the Boolean condition **(Clock'event and Clock = '1')** in order to make signal assignments only on a rising edge of the clock. The syntax for this Boolean condition is understood and is synthesizable by all CAD tools. An else clause is not included in the if/then statement. This implies that when there is not a rising edge, no assignments will be made to the outputs and they will simply hold their last value. Example 5.6 shows how to model a simple D-flip-flop using attributes. Note that this example does not model the reset behavior of a real D-flip-flop.

Example: Behavioral Modeling of a Rising Edge Triggered D-Flip-Flop Using Attributes



Clk	D	Q	
0	X	Last Q	Store
1	X	Last Q	Store
↑	0	0	Update
↑	1	1	Update

```
entity Dflipflop is
    port    (Clock      : in    bit;
             D          : in    bit;
             Q          : out   bit);
end entity;

architecture Dflipflop_arch of Dflipflop is

  begin

     D_FLIP_FLOP   : process (Clock)
       begin
          if (Clock'event and Clock='1') then
             Q <= D;
          end if;
       end process;

end architecture;
```

Example 5.6
Behavioral modeling of a rising edge-triggered D-flip-flop using attributes

CONCEPT CHECK

CC5.3 If the D input to a D-flip-flop is tied to a 0, which of the following conditions will return true on every triggering edge of the clock?

(A) Q'event and Q = '0'

(B) Q'active and Q = '0'

(C) Q'last_event = '0' and Q = '0'

(D) Q'last_active = '0' and Q = '0'

Summary

- ❖ To model sequential logic, an HDL needs to be able to trigger signal assignments based on a triggering event. This is accomplished in VHDL using a *process*.
- ❖ A *sensitivity* list is a way to control when a VHDL process is triggered. A sensitivity list contains a list of signals. If any of the signals in the sensitivity list transition, it will cause the process to trigger. If a sensitivity list is omitted, the process will trigger immediately.
- ❖ Signal assignments are made when a process suspends. There are two techniques to suspend a process. The first is using the *wait* statement. The second is simply ending the process.
- ❖ Sensitivity lists and wait statements are never used at the same time. Sensitivity lists are used to model synthesizable logic, while wait statements are used for test benches.
- ❖ When signal assignments are made in a process, they are made in the order they are listed in the process. If assignments are made to the same signal within a process, only the last assignment will take place when the process suspends.
- ❖ If assignments are needed to occur prior to the process suspending, a *variable* is used. In VHDL, variables only exist within a process. Variables are defined when a process triggers and deleted when the process ends.
- ❖ Processes also allow more advanced modeling constructs in VHDL. These include *if/then statements*, *case statements*, *infinite loops*, *while loops*, and *for loops*.
- ❖ *Signal attributes* allow additional information to be observed about a signal other than its value.

Exercise Problems

Section 5.1: The Process

5.1.1 When using a sensitivity list in a process, what will cause the process to *trigger*?

5.1.2 When using a sensitivity list in a process, what will cause the process to *suspend*?

5.1.3 When a sensitivity list is not used in a process, when will the process trigger?

5.1.4 Can a sensitivity list and a wait statement be used in the same process at the same time?

5.1.5 Does a wait statement *trigger* or *suspend* a process?

5.1.6 When are signal assignments officially made in a process?

5.1.7 Why are assignments in a process called *sequential signal assignments?*

5.1.8 Can signals be declared in a process?

5.1.9 Are variables declared within a process visible to the rest of the VHDL model (e.g., are they visible outside of the process)?

5.1.10 What happens to a variable when a process ends?

5.1.11 What is the assignment operator for variables?

Section 5.2: Conditional Programming Constructs

5.2.1 Design a VHDL model to implement the behavior described by the 4-input truth table in Fig. 5.1. Use a process and an if/then statement. Use std_logic and std_logic_vector types for your signals. Declare the entity to match the block diagram provided. Hint: Notice that there are far more input codes producing F = 0 than producing F = 1. Can you use this to your advantage to make your VHDL model simpler?

A	B	C	D	F
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

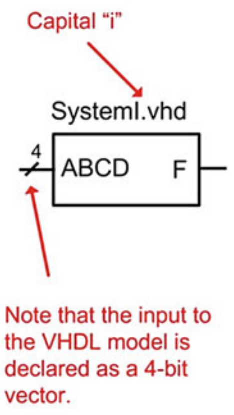

Capital "i"
SystemI.vhd
4
ABCD
F
Note that the input to the VHDL model is declared as a 4-bit vector.

Fig. 5.1
System I functionality

5.2.2 Design a VHDL model to implement the behavior described by the 4-input truth table in Fig. 5.1. Use a process and a case statement. Use std_logic and std_logic_vector types for your signals. Declare the entity to match the block diagram provided.

5.2.3 Design a VHDL model to implement the behavior described by the 4-input minterm list in Fig. 5.2. Use a process and an if/then statement. Use std_logic and std_logic_vector types for your signals. Declare the entity to match the block diagram provided.

$$F = \sum_{A,B,C,D}(4,5,7,12,13,15)$$

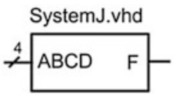


Fig. 5.2
System J functionality

5.2.4 Design a VHDL model to implement the behavior described by the 4-input minterm list in Fig. 5.2. Use a process and a case statement. Use std_logic and std_logic_vector types for your signals. Declare the entity to match the block diagram provided.

5.2.5 Design a VHDL model to implement the behavior described by the 4-input maxterm list in Fig. 5.3. Use a process and an if/then statement. Use std_logic and std_logic_vector types for your signals. Declare the entity to match the block diagram provided.

$$F = \prod_{A,B,C,D}(3,7,11,15)$$

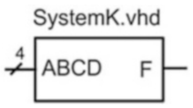


Fig. 5.3
System K functionality

5.2.6 Design a VHDL model to implement the behavior described by the 4-input maxterm list in Fig. 5.3. Use a process and a case statement. Use std_logic and std_logic_vector types for your signals. Declare the entity to match the block diagram provided.

5.2.7 Design a VHDL model to implement the behavior described by the 4-input truth table in Fig. 5.4. Use a process and an if/then statement. Use std_logic and std_logic_vector types for your signals. Declare the entity to match the block diagram provided. Hint: Notice that there are far more input codes producing F = 1 than producing F = 0. Can you use this to your advantage to make your VHDL model simpler?

A	B	C	D	F
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	1
1	1	1	1	1

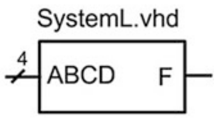


Fig. 5.4
System L functionality

5.2.8 Design a VHDL model to implement the behavior described by the 4-input truth table in Fig. 5.4. Use a process and a case statement. Use std_logic and std_logic_vector types for your signals. Declare the entity to match the block diagram provided.

Section 5.3: Signal Attributes

5.3.1 What is the purpose of a signal attribute?

5.3.2 What is the data type returned when using the signal attribute *'event*?

5.3.3 What is the data type returned when using the signal attribute *'last_event*?

5.3.4 What is the data type returned when using the signal attribute *'length*?