Chapter 8: VHDL (Part 2)

In Chap. 5 VHDL was presented as a way 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. In Chap. 7 sequential logic storage devices were presented that did not continuously update their outputs based on the instantaneous values of their inputs. Instead, sequential storage devices only update their outputs based upon an event, most often the edge of a clock signal. The modeling techniques presented in Chap. 5 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 in order 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. This chapter also presents how to create test benches and looks at commonly used packages that increase the capability and accuracy with which VHDL can model modern systems. The goal of this chapter is to give an understanding of the full capability of hardware description languages.

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

8.1 Describe the behavior of a VHDL process and how it is used to model sequential logic circuits.
8.2 Model combinational logic circuits using a process and conditional programming constructs.
8.3 Describe how and why signal attributes are used in VHDL models.
8.4 Design a test bench to verify the functional operation of a system.
8.5 Describe the capabilities provided by the most common VHDL packages.

8.1 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*.

8.1.1 Sensitivity List

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:

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

  -- variable declarations
```


B.J. LaMeres, *Introduction to Logic Circuits & Logic Design with VHDL*,
DOI 10.1007/978-3-319-34195-8_8

```
  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.

8.1.2 The Wait Statement

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.

8.1.3 Sequential Signal Assignments

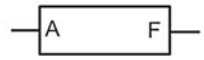
One of the more confusing concepts of a process is how sequential signal assignments behave. The rules of signal assignments within a process are as follows:

- 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 take a look at an example of how signals behave in a process. Example 8.1 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 counter-intuitive 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

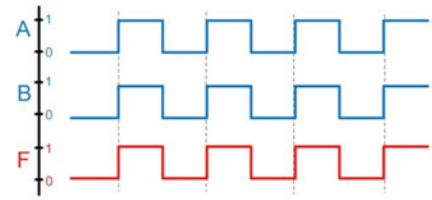
For the following system:




```
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 8.1
Behavior of Sequential Signal Assignments Within a Process

Now let's consider how these assignments behave when executed as concurrent signal assignments. Example 8.2 shows the behavior of the same signal assignments as in Example 8.1, 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.

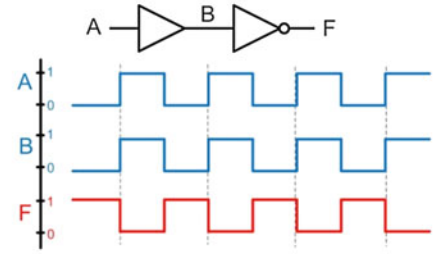
Example: Behavior of Concurrent Signal Assignments outside a Process

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 8.2
Behavior of Concurrent Signal Assignments Outside a Process

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

8.1.4 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 8.1, but this time we'll use a variable in order to accomplish instantaneous signal assignments within the process. Example 8.3 shows this approach to model the behavior where F is the complement of A.

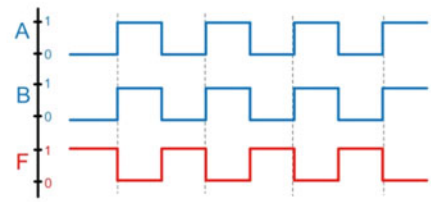
Example: Behavior of Variable Assignments within a Process

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 8.3
Variable Assignment Behavior

CONCEPT CHECK

CC8.1 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.

8.2 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.

8.2.1 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 is 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 8.4 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 8.4
Using If/Then Statements to Model Combinational Logic

8.2.2 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 of 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 8.5 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 8.5
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.

8.2.3 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* provides 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.

8.2.4 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;
```

8.2.5 For Loops

A *for loop* provides the ability to create a loop that will execute a predefined 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 for a VHDL 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

CC8.2 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.

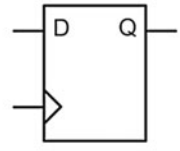
8.3 Signal Attributes

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 are a list of the commonly used, predefined 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
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 8.6 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 8.6
Behavioral Modeling of a Rising Edge-Triggered D-Flip-Flop Using Attributes

CONCEPT CHECK

CC8.3 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'

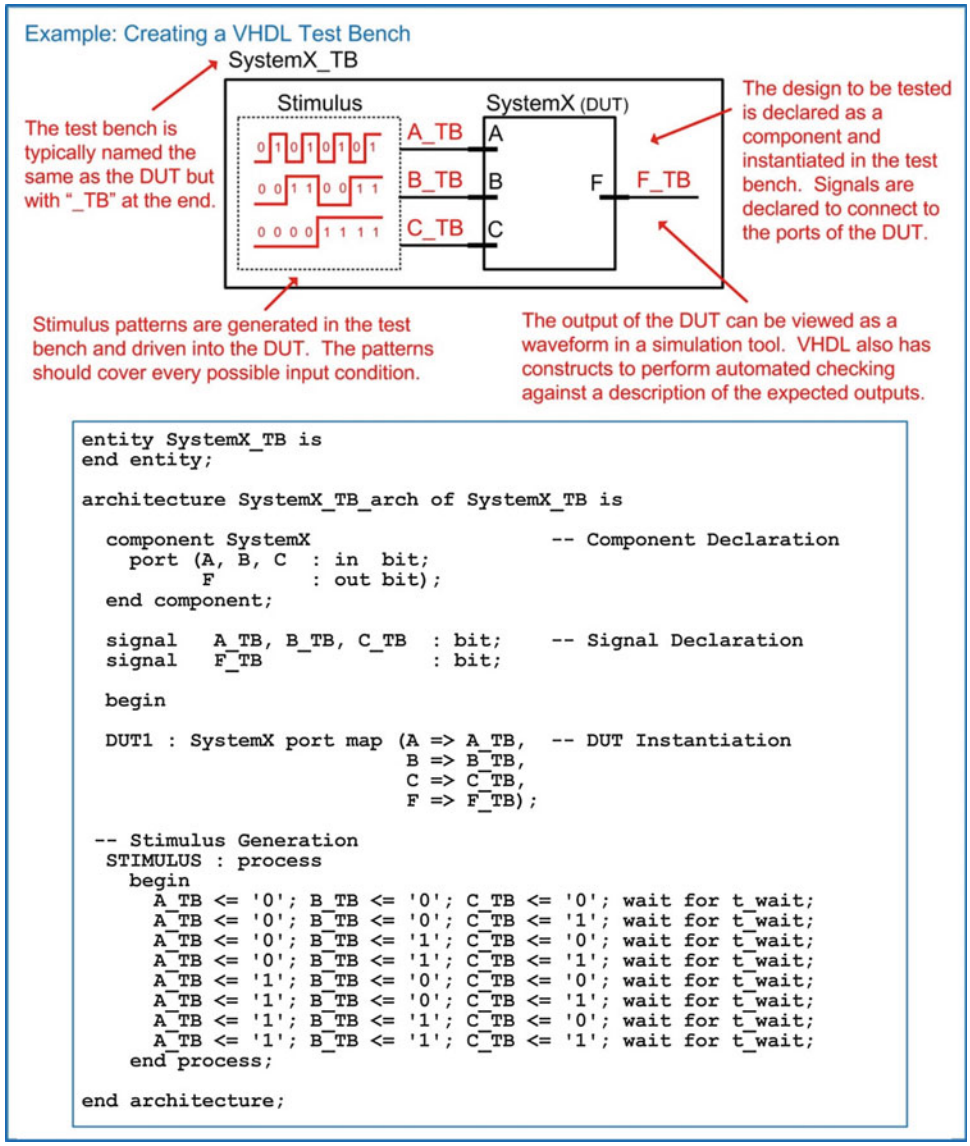
B) Q'active and Q='0'

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

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

8.4 Test Benches

The functional verification of VHDL designs is accomplished through simulation using a *test bench*. A test bench is a VHDL system that instantiates the system to be tested as a component and then generates the input patterns and observes the outputs. The system being tested is often called a *device under test (DUT)* or *unit under test (UUT).* Test benches are only used for simulation, so we can use abstract modeling techniques that are unsynthesizable to generate the stimulus patterns. VHDL also contains specific functionality to report on the status of a test and also automatically check that the outputs are correct. Example 8.7 shows how to create a simple test bench to verify the operation of SystemX. The test bench does not have any inputs or outputs; thus there are no ports declared in the entity. SystemX is declared as a component in the test bench and then instantiated (DUT1). Internal signals are declared to connect to the component under test (A_TB, B_TB, C_TB, F_TB). A process is then used to drive the inputs of SystemX. Within the process, wait statements are used to control the execution of the signal assignments; thus the process does not have a sensitivity list. Each possible input code is generated within the process. The output (F_TB) is observed using a simulation tool in either the form of a waveform or a table listing.




```
entity SystemX_TB is
end entity;

architecture SystemX_TB_arch of SystemX_TB is

  component SystemX                        -- Component Declaration
    port (A, B, C  : in  bit;
          F        : out bit);
  end component;

  signal   A_TB, B_TB, C_TB  : bit;     -- Signal Declaration
  signal   F_TB              : bit;

  begin

  DUT1 : SystemX port map (A => A_TB,   -- DUT Instantiation
                           B => B_TB,
                           C => C_TB,
                           F => F_TB);

 -- Stimulus Generation
  STIMULUS : process
    begin
      A_TB <= '0'; B_TB <= '0'; C_TB <= '0'; wait for t_wait;
      A_TB <= '0'; B_TB <= '0'; C_TB <= '1'; wait for t_wait;
      A_TB <= '0'; B_TB <= '1'; C_TB <= '0'; wait for t_wait;
      A_TB <= '0'; B_TB <= '1'; C_TB <= '1'; wait for t_wait;
      A_TB <= '1'; B_TB <= '0'; C_TB <= '0'; wait for t_wait;
      A_TB <= '1'; B_TB <= '0'; C_TB <= '1'; wait for t_wait;
      A_TB <= '1'; B_TB <= '1'; C_TB <= '0'; wait for t_wait;
      A_TB <= '1'; B_TB <= '1'; C_TB <= '1'; wait for t_wait;
    end process;

end architecture;
```

Example 8.7
Creating a VHDL Test Bench

8.4.1 Report Statement

The keyword **report** can be used within a test bench in order to provide the status of the current test. A report statement will print a string to the transcript window of the simulation tool. The report output also contains an optional severity level. There are four levels of severity (ERROR, WARNING, NOTE, and FAILURE). The severity level *FAILURE* will halt a simulation while the levels *ERROR*, *WARNING*, and *NOTE* will allow the simulation to continue. If the severity level is omitted, the report is assumed to be a severity level of NOTE. The syntax for using a report statement is as follows:

```
report "string to be printed" severity <level>;
```

Let's look at how we could use the report function within the example test bench to print the current value of the input pattern to the transcript window of the simulator. Example 8.8 shows the new process and resulting transcript output of the simulator when using report statements.

Example: Using Report Statements in a VHDL Test Bench

Report statements are inserted in the process to indicate the current stimulus pattern.

```
STIMULUS : process
  begin
    A_TB <= '0'; B_TB <= '0'; C_TB <= '0'; wait for 125 ns;
    report "Inputting Pattern 000" severity NOTE;
    A_TB <= '0'; B_TB <= '0'; C_TB <= '1'; wait for 125 ns;
    report "Inputting Pattern 001" severity NOTE;
    A_TB <= '0'; B_TB <= '1'; C_TB <= '0'; wait for 125 ns;
    report "Inputting Pattern 010" severity NOTE;
    A_TB <= '0'; B_TB <= '1'; C_TB <= '1'; wait for 125 ns;
    report "Inputting Pattern 011" severity NOTE;
    A_TB <= '1'; B_TB <= '0'; C_TB <= '0'; wait for 125 ns;
    report "Inputting Pattern 100" severity NOTE;
    A_TB <= '1'; B_TB <= '0'; C_TB <= '1'; wait for 125 ns;
    report "Inputting Pattern 101" severity NOTE;
    A_TB <= '1'; B_TB <= '1'; C_TB <= '0'; wait for 125 ns;
    report "Inputting Pattern 110" severity NOTE;
    A_TB <= '1'; B_TB <= '1'; C_TB <= '1'; wait for 125 ns;
    report "Inputting Pattern 111" severity NOTE;
  end process;
```

The following is the transcript showing the results of the report statements.

```
Transcript
VSIM 4> run
# ** Note: Inputting Pattern 000
#    Time: 125 ns  Iteration: 0  Instance: /systemx_tb
# ** Note: Inputting Pattern 001
#    Time: 250 ns  Iteration: 0  Instance: /systemx_tb
# ** Note: Inputting Pattern 010
#    Time: 375 ns  Iteration: 0  Instance: /systemx_tb
# ** Note: Inputting Pattern 011
#    Time: 500 ns  Iteration: 0  Instance: /systemx_tb
# ** Note: Inputting Pattern 100
#    Time: 625 ns  Iteration: 0  Instance: /systemx_tb
# ** Note: Inputting Pattern 101
#    Time: 750 ns  Iteration: 0  Instance: /systemx_tb
# ** Note: Inputting Pattern 110
#    Time: 875 ns  Iteration: 0  Instance: /systemx_tb
# ** Note: Inputting Pattern 111
#    Time: 1 us  Iteration: 0  Instance: /systemx_tb

VSIM 5>
```

Example 8.8
Using Report Statements in a VHDL Test Bench

8.4.2 Assert Statement

The **assert** statement provides a mechanism to check a Boolean condition before using the report statement. This allows report outputs to be selectively printed based on the values of signals in the system under test. This can be used to print either the successful operation or the failure of a system. If the Boolean condition associated with the assert statement is evaluated true, it *will not* execute the subsequent report statement. If the Boolean condition is evaluated false, it will execute the subsequent report statement. The assert statement is always used in conjunction with the report statement. The following is the syntax for the assert statement:

```
assert boolean_condition report "string" severity <level>;
```

Let's look at how we could use the assert function within the example test bench to check whether the output (F_TB) is correct. In the example in Example 8.9, the system passes the first pattern but fails the second.

Example: Using Assert Statements in a VHDL Test Bench

Assert statements are used to check the correctness of the system outputs.

```
STIMULUS : process
  begin
    A_TB <= '0'; B_TB <= '0'; C_TB <= '0'; wait for 125 ns;
    assert (F_TB='1') report "Failed test at 000" severity FAILURE;
    assert (F_TB='0') report "Passed test at 000" severity NOTE;

    A_TB <= '0'; B_TB <= '0'; C_TB <= '1'; wait for 125 ns;
    assert (F_TB='1') report "Failed test at 001" severity FAILURE;
    assert (F_TB='0') report "Passed test at 001" severity NOTE;
                                  :
  end process;
```

An intentional failure was introduced at the second input pattern to show how the simulation will end if a report statement is issued with a severity level of FAILURE. The following is the output of the transcript for this case.

Transcript

```
VSIM 6> run
# ** Note: Passed test at 000
#    Time: 125 ns  Iteration: 0  Instance: /systemx_tb
# ** Failure: Failed test at 001
#    Time: 250 ns  Iteration: 0  Process: /systemx_tb/STIMULUS File: C:/
Users/lameres/Desktop/EE261_VHDL/ModelSim/Ch08_VHDL_Part2/Test_Bench_Sys
temX/SystemX_TB.vhd
# Break in Process STIMULUS at C:/Users/lameres/Desktop/EE261_VHDL/Model
Sim/Ch08_VHDL_Part2/Test_Bench_SystemX/SystemX_TB.vhd line 35

VSIM 7>
```

Example 8.9
Using Assert Statements in a VHDL Test Bench

CONCEPT CHECK

CC8.4 Could a test bench ever use sensitivity lists exclusively to create its stimulus? Why or why not?

A) Yes. The signal assignments will simply be made when the process ends.

B) No. Since a sensitivity list triggers when there is a change on one or more of the signals listed, the processes in the test bench would never trigger because there is no method to make the initial signal transition.

8.5 Packages

One of the drawbacks of the VHDL standard package is that it provides limited functionality in its synthesizable data types. The bit and bit_vector, while synthesizable, lack the ability to accurately model many of the topologies implemented in modern digital systems. Of primary interest are topologies that involve multiple drivers connected to a single wire. The standard package will not permit this type of connection; however, this type of topology is a common way to interface multiple nodes on a shared interconnection. Furthermore, the standard package does not provide many useful features for these

types, such as don't cares, arithmetic using the + and – operators, type conversion functions, or the ability to read/write external files. To increase the functionality of VHDL, packages are included in the design.

8.5.1 STD_LOGIC_1164

In the late 1980s, the IEEE 1164 standard was released that added functionality to VHDL to allow a multi-valued logic system (i.e., a signal can take on more values than just 0 and 1). This standard also provided a mechanism for multiple drivers to be connected to the same signal. An updated release in 1993 called IEEE 1164-1993 was the most significant update to this standard and contains the majority of functionality used in VHDL today. Nearly all systems described in VHDL include the 1164 standard as a package. This package is included by adding the following syntax at the beginning of the VHDL file:

```
library IEEE;
use IEEE.std_logic_1164.all;
```

This package defines four new data types: **std_ulogic**, **std_ulogic_vector**, **std_logic**, and **std_logic_vector**. The std_ulogic and std_logic are enumerated, scalar types that can provide a multi-valued logic system. The types std_ulogic_vector and std_logic_vector are vector types containing a linear array of scalar types std_ulogic and std_logic, respectively. The scalar types can take on nine different values as described below:

Value	Description	Notes
U	Uninitialized	Default initial value
X	Forcing unknown	
0	Forcing 0	
1	Forcing 1	
Z	High impedance	
W	Weak unknown	
L	Weak 0	Pull-down
H	Weak 1	Pull-up
–	Don't care	Used for synthesis only

These values can be assigned to signals by enclosing them in single quotes (scalars) or double quotes (vectors).

Example:

```
A <= 'X';      -- assignment to a scalar (std_ulogic or std_logic)
V <= "01ZH";   -- assignment to a 4-bit vector (std_ulogic_vector or
               --                 std_logic_vector)
```

The type std_ulogic is *unresolved* (note: the "u" standard for "unresolved"). This means that if a signal is being driven by two circuits with type std_ulogic, the VHDL simulator will not be able to *resolve* the conflict and it will result in a compiler error. The std_logic type is *resolved*. This means that if a signal is being driven by two circuits with type std_logic, the VHDL simulator *will* be able to resolve the conflict and will allow the simulation to continue. Figure 8.1 shows an example of a shared signal topology and how conflicts are handled when using various data types.

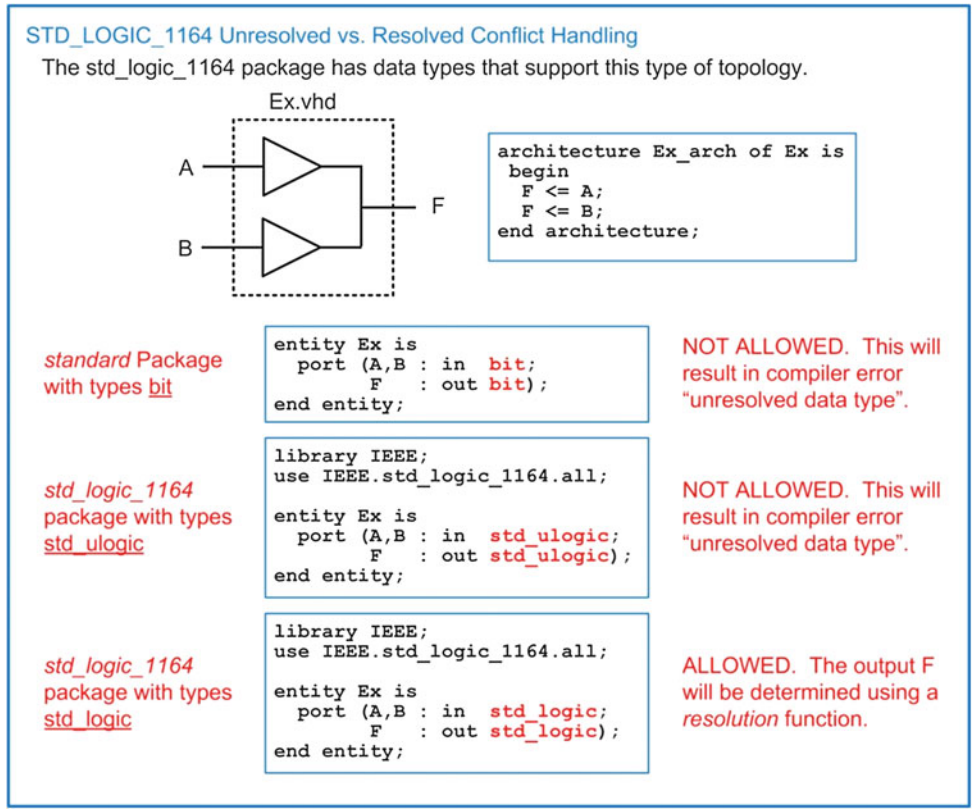


Fig. 8.1
STD_LOGIC_1164 unresolved vs. resolved conflict handling

8.5.1.1 STD_LOGIC Resolution Function

The std_logic_1164 will resolve signal conflict of type std_logic using a **resolution function**. The nine allowed values each has a relative drive strength that allows a resolution to be made in the event of conflict. Whenever there is a conflict, the simulator will consult the resolution function to determine the value of the signal. Figure 8.2 shows the relative drive strengths of the nine possible signal values provided by the std_logic_1164 package and the resolution function table.

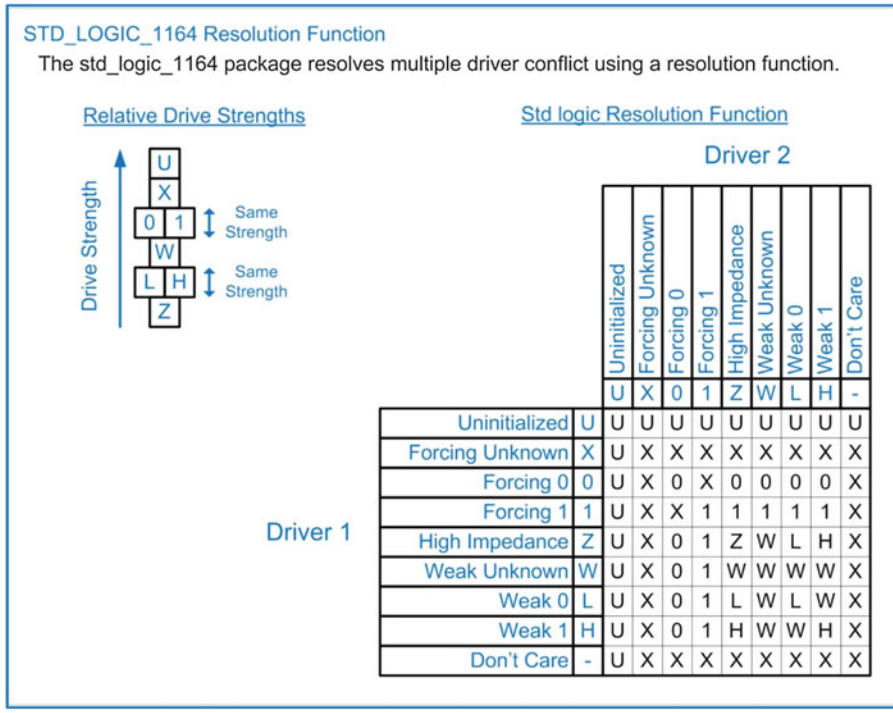


Fig. 8.2
STD_LOGIC_1164 resolution function

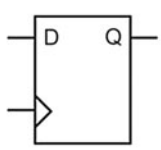
8.5.1.2 STD_LOGIC_1164 Logical Operators

The std_logic_1164 also contains new definitions for all of the logical operators (**and, nand, or, nor, xor, xnor, not)** for types std_ulogic and std_logic. These are required since these data types can take on more logic values than just a 0 or 1; thus the logical operator definitions from the standard package are not sufficient.

8.5.1.3 STD_LOGIC_1164 Edge Detection Functions

The std_logic_1164 also provides functions for the detection of rising or falling transitions on a signal. The functions **rising_edge()** and **falling_edge()** provide a more readable form of this functionality compared to the (Clock'event and Clock = "1") approach. Example 8.10 shows the use of the rising_edge() function to model the behavior of a rising edge-triggered D-flip-flop.

Example: Behavioral Modeling of a D-Flip-Flop using the rising_edge() Function




```
library IEEE;
use IEEE.std_logic_1164.all;

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

Equivalent Models

```
architecture Dflipflop_arch of Dflipflop is

  begin

     D_FLIP_FLOP  : process (Clock)
       begin
          if (rising_edge(Clock)) then
             Q <= D;
          end if;
       end process;

end architecture;
```

Example 8.10
Behavioral Modeling of a D-Flip-Flop Using the rising_edge() Function

8.5.1.4 STD_LOGIC-Type Conversion Functions

The std_logic_1164 package also provides functions to convert between data types. Functions exist to convert between bit, std_ulogic, and std_logic. Functions also exist to convert between these types' vector forms (bit_vector, std_ulogic_vector, and std_logic_vector). The functions are listed below.

Name	Input type	Return type
To_bit()	std_ulogic	bit
To_bitvector()	std_ulogic_vector	bit_vector
To_bitvector()	std_logic_vector	bit_vector
To_StdULogic()	bit	std_ulogic
To_StdULogicVector()	bit_vector	std_ulogic_vector
To_StdULogicVector()	std_logic_vector	std_ulogic_vector
To_StdLogicVector()	bit_vector	std_logic_vector
To_StdLogicVector()	std_ulogic_vector	std_logic_vector

When using these functions, the function name and input signal are placed to the right of the assignment operator and the target signal is placed on the left.

Example:

```
A <= To_bit(B);                 -- B is type std_ulogic, A is type bit
V <= To_StdLogicVector(C);      -- C is type bit_vector, V is std_logic_vector
```

When identical function names exist that can have different input data types, the VHDL compiler will automatically decide which function to use based on the input argument type. For example, the function "To_bitvector" exists for an input of std_ulogic_vector *and* std_logic_vector. When using this function, the compiler will automatically detect which input type is being used and select the corresponding function variant. No additional syntax is required by the designer in this situation.

8.5.2 NUMERIC_STD

The *numeric_std* package provides numerical computation for types std_logic and std_logic_vector. When performing binary arithmetic, the results of arithmetic operations and comparisons vary greatly depending on whether the binary number is unsigned or signed. As a result, the numeric_std package defines two new data types, **unsigned** and **signed**. An unsigned type is defined to have its MSB in the leftmost position of the vector, and the LSB in the rightmost position of the vector. A signed number uses two's complement representation with the leftmost bit of the vector being the sign bit. When declaring a signal to be one of these types, it is implied that these represent the encoding of an underlying native type of std_logic/std_logic_vector. The use of unsigned/signed types provides the interpretation of how arithmetic, logical, and comparison operators will perform. This also implies that the numeric_std package requires the std_logic_1164 to always be included. While the numeric_std package includes an inclusion call of the std_logic_1164 package, it is common to explicitly include both the std_logic_1164 and the numeric_std packages in the main VHDL file. The VHDL compiler will ignore redundant package statements. The syntax for including these packages is as follows:

```
library IEEE;
use IEEE.std_logic_1164.all;   -- defines types std_ulogic and std_logic
use IEEE.numeric_std.all;      -- defines types unsigned and signed
```

8.5.2.1 NUMERIC_STD Arithmetic Functions

The numeric_std package provides support for a variety of arithmetic functions for the types unsigned and signed. These include **+**, −, *****, **/**, **mod**, **rem**, and **abs** functions. These arithmetic operations behave differently for the unsigned versus signed types, but the VHDL compiler will automatically use the correct operation based on the types of the input arguments.

Most synthesis tools support the addition, subtraction, and multiplication operators in this package. This provides a higher level of abstraction when modeling arithmetic circuitry. Recall that the VHDL standard package does not support addition, subtraction, and multiplication of types bit/bit_vector using the +, −, and * operators. Using the numeric_std package gives the ability to model these arithmetic operations with a synthesizable data type using the more familiar mathematical operators. The division, modulo, remainder, and absolute value functions are not synthesizable directly from this package.

Example:

```
F <= A + B;         -- A, B, F are type unsigned(3 downto 0)
F <= A - B;
```

8.5.2.2 NUMERIC_STD Logical Functions

The numeric_std package provides support for all of the logical operators (**and, nand, or, nor, xor, xnor, not)** for types unsigned and signed. It also provides two new shift functions **shift_left()** and

shift_right(). These shift functions will fill the vacant position in the vector after the shift with a 0; thus these are *logical shifts*. This package also provides two new rotate functions **rotate_left()** and **rotate_right()**.

8.5.2.3 NUMERIC_STD Comparison Functions

The numeric_std package provides support for all of the comparison functions for types unsigned and signed. These include >, <, <=, >=, =, and /=. These comparisons return type Boolean.

Example: (A = "0000", B = "1111")

```
if (A < B) then  -- This condition is TRUE if A and B are UNSIGNED
   :

if (A < B) then  -- This condition is FALSE if A and B are SIGNED
```

8.5.2.4 NUMERIC_STD Edge Detection Functions

The numeric_std also provides the functions **rising_edge()** and **falling_edge()** for the detection of rising or falling edge transition detection for types unsigned and signed.

8.5.2.5 NUMERIC_STD Conversion Functions

The numeric_std package contains a variety of useful conversion functions. Of particular usefulness are functions between the type *integer* and to/from *unsigned/signed*. This allows behavioral models for counters, adders, and subtractors to be implemented using the more readable type integer. After the functionally has been described, a conversion can be used to turn the result into types unsigned or signed to provide a synthesizable output. When converting an integer to a vector, a *size* argument is included. The size argument is of type integer and provides the number of bits in the vector that the integer will be converted to:

Name	Input type	Return type
To_integer()	Unsigned	Integer
To_integer()	Signed	Integer
To_unsigned()	integer, <size>	Unsigned (size-1 downto 0)
To_signed()	Integer, <size>	Signed (size-1 downto 0)

8.5.2.6 NUMERIC_STD Type Casting

VHDL contains a set of built-in *type casting* operations that are commonly used with the numeric_std package to convert between *std_logic_vector* and *unsigned/signed*. Since the types unsigned/signed are based on the underlying type std_logic_vector, the conversion is simply known as casting. The following are the built-in type casting capabilities in VHDL.:

Name	Input type	Return type
std_logic_vector()	Unsigned	std_logic_vector
std_logic_vector()	Signed	std_logic_vector
unsigned()	std_logic_vector	Unsigned
signed()	std_logic_vector	Signed

When using these type casts, they are placed on the right-hand side of the assignment exactly as a conversion function.

Example:

```
A <= std_logic_vector(B); -- B is unsigned, A is std_logic_vector
C <= unsigned(D);         -- D is std_logic_vector, C is unsigned
```

Type casts and conversion functions can be compounded in order to perform multiple conversions in one assignment. This is useful when converting between types that do not have a direct cast or conversion function. Let's look at the example of converting an integer to an 8-bit std_logic_vector where the number being represented is unsigned. The first step is to convert the integer to an unsigned type. This can be accomplished with the *to_unsigned* function defined in the numeric_std package. This can be embedded in a second cast from unsigned to std_logic_vector. In the following example, E is the target of the operation and is of type std_logic vector. F is the argument of assignment and is of type integer. Recall that the to_unsigned conversions require both the input integer name and the size of the unsigned vector being converted to.

Example:

```
E <= std_logic_vector(to_unsigned(F, 8));
```

8.5.3 NUMERIC_STD_UNSIGNED

When using the numeric_std package, the data types unsigned and signed must be used in order to get access to the numeric operators. While this provides ultimate control over the behavior of the signal operations and comparisons, many designs may only use unsigned types. In order to provide a mechanism to treat all vectors as unsigned while leaving their type as std_logic_vector, the numeric_std_unsigned package was created. When this package is used, it will treat all std_logic_vectors in the design as unsigned. This package requires the std_logic_1164 and numeric_std packages to be previously included. When used, all signals and ports can be declared as std_logic/std_logic_vector and they will be treated as unsigned when performing arithmetic operations and comparisons. The following is an example of how to include this package:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.numeric_std_unsigned.all;
```

8.5.3.1 NUMERIC_STD_UNSIGNED Conversion Functions

The numeric_std_unsigned package contains a few more type conversions beyond the numeric_std package. These additional conversions are as follows:

Name	Input type	Return type
To_Integer	std_logic_vector	Integer
To_StdLogicVector	Unsigned	std_logic_vector

8.5.4 NUMERIC_BIT

The *numeric_bit* package provides numerical computation for types bit and bit_vector. Since the vast majority of VHDL designs today use types std_logic/std_logic_vector instead of bit/bit_vector, this package is rarely used. This package is included by adding the following syntax at the beginning of the VHDL file in the design:

```
library IEEE;
use IEEE.numeric_bit.all;   -- defines types unsigned and signed
```

The numeric_bit package is nearly identical to numeric_std. It defines data types **unsigned** and **signed**, which provide information on the encoding style of the underlying data types bit and bit_vector. All of the arithmetic, logical, and comparison functions defined in numeric_std are supported in numeric_bit (**+**, −, *****, **/**, **mod**, **rem, abs, and, nand, or, nor, xor, xnor, not,** >, <, <=, >=, =, **/**=) for types unsigned and signed. This package also provides the same edge detection (**rising_edge()**, **falling_edge()**), shift (**shift_left()**, **shift_right()**), and rotate (**rotate_left()**, **rotate_right()**) functions for types unsigned and signed.

The primary difference between numeric_bit and numeric_std is that numeric_bit also provides support for the shift/rotate operators from the standard package (**sll**, **srl**, **rol**, **ror**). Also, the conversion functions are defined only for conversions between integer, unsigned, and signed.

Name	Input type	Return type
To_integer	Unsigned	Integer
To_integer	Signed	Integer
To_unsigned	Integer, <size>	Unsigned (size-1 downto 0)
To_signed	Integer, <size>	Signed (size-1 downto 0)

8.5.5 NUMERIC_BIT_UNSIGNED

The numeric_bit_unsigned package provides a way to treat all bit/bit_vectors in a design as unsigned numbers. The syntax for including the numeric_bit_unsigned package is shown below. In this example, all bit/bit_vectors will be treated as unsigned numbers for all arithmetic operations and comparisons:

```
library IEEE;
use IEEE.numeric_bit.all;
use IEEE.numeric_bit_unsigned.all;
```

8.5.5.1 NUMERIC_BIT_UNSIGNED Conversion Functions

The numeric_bit_unsigned package contains a few more type conversions beyond the numeric_bit package. These additional conversions are as follows:

Name	Input type	Return type
To_integer	std_logic_vector	Integer
To_BitVector	Unsigned	bit_vector

8.5.6 MATH_REAL

The *math_real* package provides numerical computation for the type *real*. The type *real* is the VHDL type used to describe a 32-bit floating point number. None of the operators provided in the math_real package are synthesizable. This package is primarily used for test benches. This package is included by adding the following syntax at the beginning of the VHDL file in the design:

```
library IEEE;
use IEEE.math_real.all;
```

The math_real package defines a set of commonly used constants, which are shown below.

Constant name	Type	Value	Description
MATH_E	Real	2.718	Value of e
MATH_1_E	Real	0.367	Value of 1/e
MATH_PI	Real	3.141	Value of pi
MATH_1_PI	Real	0.318	Value of 1/pi
MATH_LOG_OF_2	Real	0.693	Natural log of 2
MATH_LOG_OF_10	Real	2.302	Natural log of10
MATH_LOG2_OF_E	Real	1.442	log base 2 of e
MATH_LOG10_OF_E	Real	0.434	log base 10 of e
MATH_SQRT2	Real	1.414	sqrt of 2
MATH_SQRT1_2	Real	0.707	sqrt of 1/2
MATH_SQRT_PI	Real	1.772	sqrt of pi
MATH_DEG_TO_RAD	Real	0.017	Conversion factor from degree to radian
MATH_RAD_TO_DEG	Real	57.295	Conversion factor from radian to degree

Only three digits of accuracy are shown in this table; however, the constants defined in the math_real package have full 32-bit accuracy. The math_real package provides a set of commonly used floating point operators for the type real.

Function name	Return type	Description
SIGN	Real	Returns sign of input
CEIL	Real	Returns smallest integer value
FLOOR	Real	Returns largest integer value
ROUND	Real	Rounds input up/down to whole number
FMAX	Real	Returns largest of two inputs
FMIN	Real	Returns smallest of two inputs
SQRT	Real	Returns square root of input
CBRT	Real	Returns cube root of input
**	Real	Raise to power of (X**Y)
EXP	Real	e^X
LOG	Real	log(X)
SIN	Real	sin(X)
COS	Real	cos(X)
TAN	Real	tan(X)
ASIN	Real	asin(X)
ACOS	Real	acos(X)
ATAN	Real	atan(X)
ATAN2	Real	atan(X/Y)
SINH	Real	sinh(X)
COSH	Real	cosh(X)
TANH	Real	tanh(X)
ASINH	Real	asinh(X)
ACOSH	Real	acosh(X)
ATANH	Real	atanh(X)

8.5.7 MATH_COMPLEX

The *math_complex* package provides numerical computation for complex numbers. Again, nothing in this package is synthesizable and is typically used only for test benches. This package is included by adding the following syntax at the beginning of the VHDL file in the design:

```
library IEEE;
use IEEE.math_complex.all;
```

This package defines three new data types, **complex**, **complex_vector**, and **complex_polar**. The type *complex* is defined with two fields, *real* and *imaginary*. The type *complex_vector* is a linear array of type complex. The type *complex_polar* is defined with two fields, *magnitude* and *angle*. This package provides a set of common operations for use with complex numbers. This package also supports the arithmetic operators **+**, −, *****, and **/** for the type complex.

Function name	Return type	Description
CABS	Real	Absolute value of complex number
CARG	Real (radians)	Returns angle of complex number
CMPLX	Complex	Returns complex number form of input
CONJ	Complex or complex_polar	Returns complex conjugate
CSQRT	Real	Returns square root
CEXP	Real	Returns e^Z of complex input
COMPLEX_TO_POLAR	complex_polar	Convert complex to complex_polar
POLAR_TO_COMPLEX	Complex	Convert complex_polar to complex

8.5.8 TEXTIO and STD_LOGIC_TEXTIO

The *textio* package provides the ability to read and write to/from external input/output (I/O). External I/O refers to items such as files or the standard input/output of a computer. This package contains functions that allow the values of signals and variables to be read and written in addition to strings. This allows more sophisticated output messages to be created compared to the *report* statement alone, which can only output strings. The ability to read in values from a file allows sophisticated test patterns to be created outside of VHDL and then read in during simulation for testing a system. It is important to keep in mind that the term “I/O” refers to external files or the transcript window, not the inputs and outputs of a system model. The textio package is not synthesizable and is only used in test benches. The textio package is within the STD library and is included in a VHDL design using the following syntax:

```
library STD;
use STD.textio.all;
```

This package by itself only supports reading and writing types bit, bit_vector, integer, character, and string. Since the majority of synthesizable designs use types std_logic and std_logic_vector, an additional package was created that added support for these types. The package is called *std_logic_textio* and is located within the IEEE library. The syntax for including this package is below:

```
library IEEE;
use IEEE.std_logic_textio.all;
```

The textio package defines two new types for interfacing with external I/O. These types are **file** and **line**. The type *file* is used to identify or create a file for reading/writing within the VHDL design. The syntax for declaring a file is as follows:

```
file file_handle : <file_type> open <file_mode> is <"filename">;
```

Declaring a file will automatically open the file and keep it open until the end of the process that is using it. The *file_handle* is a unique identifier for the file that is used in subsequent procedures. The file handle name is user defined. A file handle eliminates the need to specify the entire file name each time a file access procedure is called. The *file_type* describes the information within the file. There are two supported file types, **TEXT** and **INTF**. A *TEXT* file is one that contains strings of characters. This is the most common type of file used as there are functions that can convert between types string, bit/bit_vector and std_logic/std_logic_vector. This allows all of the information in the file to be stored as characters, which makes the file readable by other programs. An *INTF* file type contains only integer values and the information is stored as a 32-bit, signed binary number. The *file_mode* describes whether the file will be read from or written to. There are two supported modes, **WRITE_MODE** and **READ_MODE**. The *filename* is given within double quotes and is user defined. It is common to enter an extension on the file so that it can be opened by other programs (e.g., output.txt). Declaring a file always takes place within a process before the process begins statement. The following are examples of how to declare files.:

```
file Fout: TEXT open WRITE_MODE is "output_file.txt";
file Fin: TEXT open READ_MODE is "input_file.txt";
```

The information within a file is accessed (either read or written) using the concept of a *line*. In the textio package, a file is interpreted as a sequence of lines, each containing either a string of characters or an integer value. The type *line* is used as a temporary buffer when accessing a line within the file. When accessing a file, a variable is created of type *line*. This variable is then used to either hold information that is *read* from a line in the file or to hold the information that is to be *written* to a line in the file. A variable is necessary for this behavior since assignments to/from the file must be made immediately. As such, a line variable is always declared within a process before the process begins statement. The syntax for declaring a variable of type line is as follows:

```
variable <line_variable_name> : line;
```

There are two procedures that allow information to be transferred between a line variable in VHDL and a line in a file. These procedures are **readline()** and **writeline()**. Their syntax is as follows:

```
readline(<file_handle>, <line_variable_name>);
writeline(<file_handle>, <line_variable_name>);
```

The transfer of information between a line variable and a line in a file using these procedures is accomplished on the entire line. There is no mechanism to read or write only a portion of the line in a file. Once a file is opened/created using a file declaration, the lines are accessed in the order they appear in the file. The first procedure called (either readline() or writeline()) will access the first line of the file. The next time a procedure is called, it will access the second line of the file. This will continue until all of the lines have been accessed. The textio package provides a function to indicate when the end of the file has been reached when performing a readline(). This function is called **endfile()** and returns type Boolean. This function will return true once the end of the file has been reached. Figure 8.3 shows a graphical representation of how the textio package handles external file access.

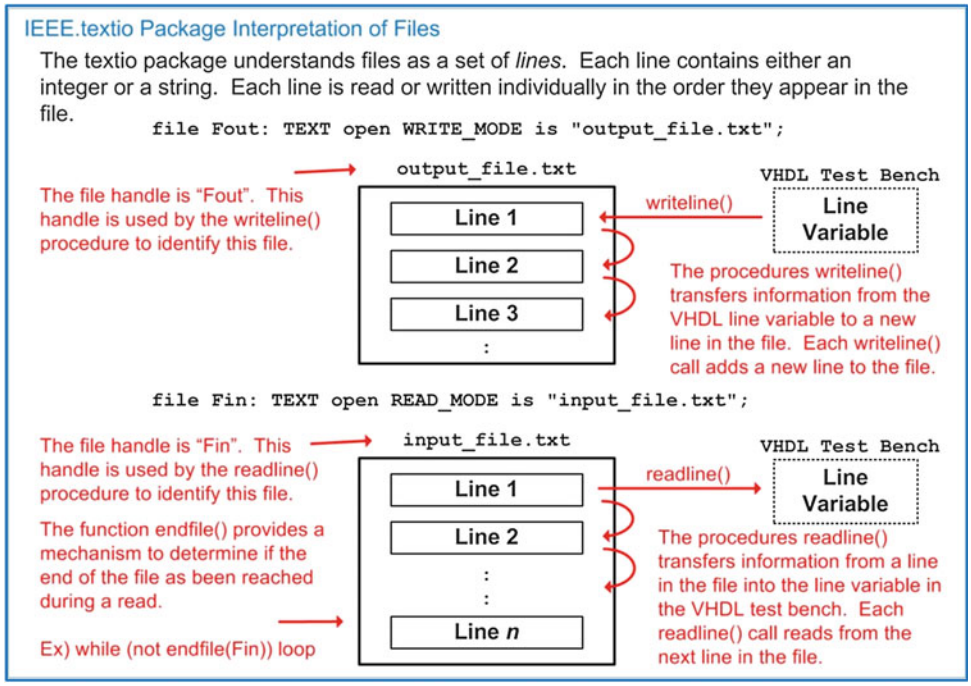


Fig. 8.3
IEEE.textio package interpretation of files

Two additional procedures are provided to add or retrieve information to/from the line variable within the VHDL test bench. These procedures are **read()** and **write()**. The syntax for these procedures is as follows:

```
read(<line_variable_name>, <destination_variable>);
write(<line_variable_name>, <source_variable>);
```

When using the read() procedure, the information in the line variable is treated as *space delimited*. This means that each read() procedure will retrieve the information from the line variable until it reaches a white space. This allows multiple read() procedures to be used in order to parse the information into separate *destination_variable* names. The destination_variable must be of the appropriate type and size of the information being read from the file. For example, if the field in the line being read is a four-character string ("wxyz"), then a destination variable must be defined of type *string(1 to 4)*. If the field being read is a 2-bit std_logic_vector, then a destination variable must be defined of type *std_logic_vector(1 downto 0)*. The read() procedure will ignore the delimiting white space character.

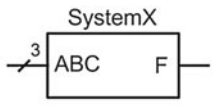
When using the write() procedure, the *source_destination* is assumed to be of type bit, bit_vector, integer, std_logic, or std_logic_vector. If it is desired to enter a text string directly, then the function **string** is used with the format *string'<"characters. . .">*. Multiple write() procedures can be used to insert information into the line variable. Each subsequent write procedure appends the information to the end of the string. This allows different types of information to be interleaved (e.g., text, signal value, text).

8.5.8.1 Example: Writing to an External File from a Test Bench

Let's look at an example of a test bench that writes information about the tests being conducted to an external file. Example 8.11 shows the model for the system to be tested (SystemX) and an overview of the test bench approach (SystemX_TB).

Example: Writing to an External File from a Test Bench (Part 1)

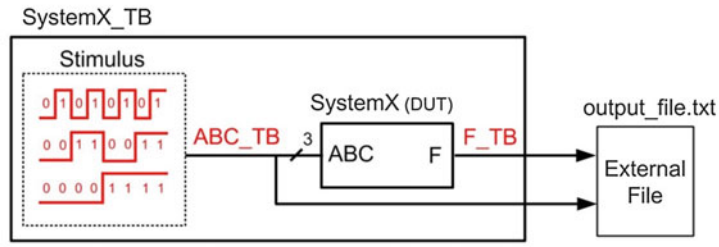
The following combinational logic circuit is implemented as follows:



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

```
library IEEE;
use IEEE.std_logic_1164.all;

entity SystemX is
  port (ABC : in  std_logic_vector(2 downto 0);
        F   : out std_logic);
end entity;

architecture SystemX_arch of SystemX is
begin
  SystemX_Proc : process (ABC)
    begin
        case (ABC) is
          when "000"|"010"|"110" => F <= '1';
          when others            => F <= '0';
       end case;
  end process;
end architecture;
```

A test bench is created to drive in all possible binary codes into the system to test its functionality. The input codes (ABC_TB) and the DUT output (F_TB) will be written to an external file called "output_file.txt".



Example 8.11
Writing to an External File from a Test Bench (Part 1)

Example 8.12 shows the details of the test bench model. In this test bench, a file is declared in order to create "output_file.txt". This file is given the handle *Fout*. A line variable is also declared called *current_line* to act as a temporary buffer to hold information that will be written to the file. The procedure write() is used to add information to the line variable. The first write() procedure is used to create a text message ("Beginning Test. . ."). Notice that since the information to be written to the line variable is of type string, a conversion function must be used within the write() procedure (e.g.,. string'(Beginning Test. . .")). This message is written as the first line in the file using the writeline() procedure. After an input vector has been applied to the DUT, a new line is constructed containing descriptive text, the input vector value, and the output value from the DUT. This message is repeated for each input code in the test bench.

Example: Writing to an External File from a Test Bench (Part 2)

The following test bench is created to perform the testing on SystemX.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_textio.all;

library STD;
use STD.textio.all;

entity SystemX_TB is
end entity;

architecture SystemX_TB_arch of SystemX_TB is

  component SystemX
    port (ABC  : in  std_logic_vector(2 downto 0);
          F    : out std_logic);
  end component;

  signal   ABC_TB  : std_logic_vector(2 downto 0);
  signal   F_TB    : std_logic;

 begin

  DUT : SystemX port map (ABC => ABC_TB, F => F_TB);

  STIMULUS : process

    file Fout: TEXT open WRITE_MODE is "output_file.txt";
    variable current_line : line;

    begin

      write(current_line, string'("Beginning Test (Input=ABC, Output=F)"));
      writeline(Fout, current_line);

      ABC_TB <= "000"; wait for 125 ns;

      write(current_line, string'("ABC="));
      write(current_line, ABC_TB);
      write(current_line, string'(", F="));
      write(current_line, F_TB);
      writeline(Fout, current_line);


      ABC_TB <= "001"; wait for t_wait;

      write(current_line, string'("ABC="));
      write(current_line, ABC_TB);
      write(current_line, string'(", F="));
      write(current_line, F_TB);
      writeline(Fout, current_line);

                :

      wait;

    end process;

end architecture;
```

The std_logic_textio and textio packages are included to support external I/O access.

Declaration of DUT

Declaration of signals to connect to DUT

Instantiation of DUT

Declare file for writing

Declare line variable

This write() procedure adds text to the line variable.

This writeline() procedure writes the contents of the line variable to the file.

Set first input pattern.

Write input pattern, output value and descriptive text to the line variable and then write the line variable to the file using writeline().

Repeat for next pattern.

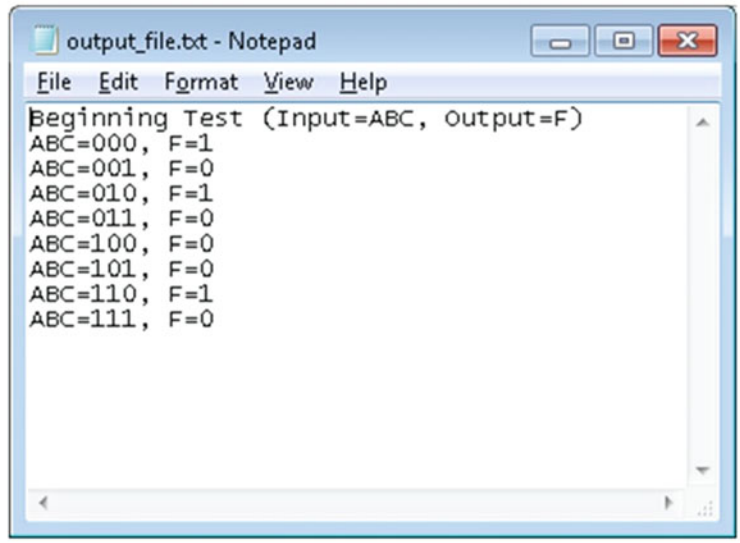
Repeat for all other possible inputs (not shown for brevity).

Example 8.12
Writing to an External File from a Test Bench (Part 2)

Example 8.13 shows the resulting file that is created from this test bench.

Example: Writing to an External File from a Test Bench (Part 3)

The file "output_file.txt" is created with the following contents.



Example 8.13
Writing to an External File from a Test Bench (Part 3)

8.5.8.2 Example: Writing to STD_OUTPUT from a Test Bench

The textio package also provides the ability to write to the standard output of the computer instead of to an external file. The standard output of the computer is typically routed to the transcript window of the simulator. This is accomplished by using a reserved file handle called **OUTPUT**. When using this file handle, a new file does not need to be declared in the test bench since it is already defined as part of the textio package. The reserved file handle name OUTPUT can be used directly in the writeline() procedure.

Let's look at an example of a test bench that outputs information about the test being conducted to STD_OUT. Example 8.14 shows this test bench approach. The test bench is identical as the one used in Example 8.12 with the exception that the writeline() procedure outputs are directed to the STD_OUTPUT of the computer using the reserved file handle name OUTPUT instead of to an external file.

Example: Writing to STD_OUTPUT from a Test Bench (Part 1)

This test bench directs the writeline() outputs to the STD_OUTPUT of the computer by using the reserved file handle "OUTPUT".

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_textio.all;

library STD;
use STD.textio.all;

entity SystemX_TB is
end entity;

architecture SystemX_TB_arch of SystemX_TB is

  component SystemX
    port (ABC  : in  std_logic_vector(2 downto 0);
          F    : out std_logic);
  end component;

  signal   ABC_TB  : std_logic_vector(2 downto 0);
  signal   F_TB    : std_logic;

 begin

  DUT : SystemX port map (ABC => ABC_TB, F => F_TB);

  STIMULUS : process

    variable current_line : line;

    begin

      write(current_line, string'("Beginning Test (Input=ABC, Output=F)"));
      writeline(OUTPUT, current_line);

      ABC_TB <= "000"; wait for 125 ns;

      write(current_line, string'("ABC="));
      write(current_line, ABC_TB);
      write(current_line, string'(", F="));
      write(current_line, F_TB);
      writeline(OUTPUT, current_line);


      ABC_TB <= "001"; wait for t_wait;

      write(current_line, string'("ABC="));
      write(current_line, ABC_TB);
      write(current_line, string'(", F="));
      write(current_line, F_TB);
      writeline(OUTPUT, current_line);

                :

      wait;

    end process;

end architecture;
```

The reserved file handle "OUTPUT" is used to direct the writeline() output to the computer STD_OUTPUT.

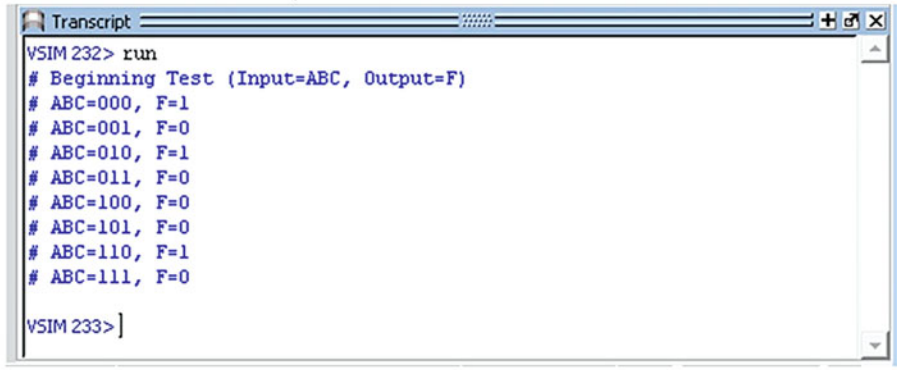
Repeat for all other possible inputs (not shown for brevity).

Example 8.14
Writing to STD_OUT from a Test Bench (Part 1)

Example 8.15 shows the output from the test bench. This output is displayed in the transcript window of the simulation tool.

Example: Writing to STD_OUTPUT from a Test Bench (Part 2)

The results of the writeline() procedure is directed to the STD_OUTPUT of the computer, which is shown in the transcript of the simulator.



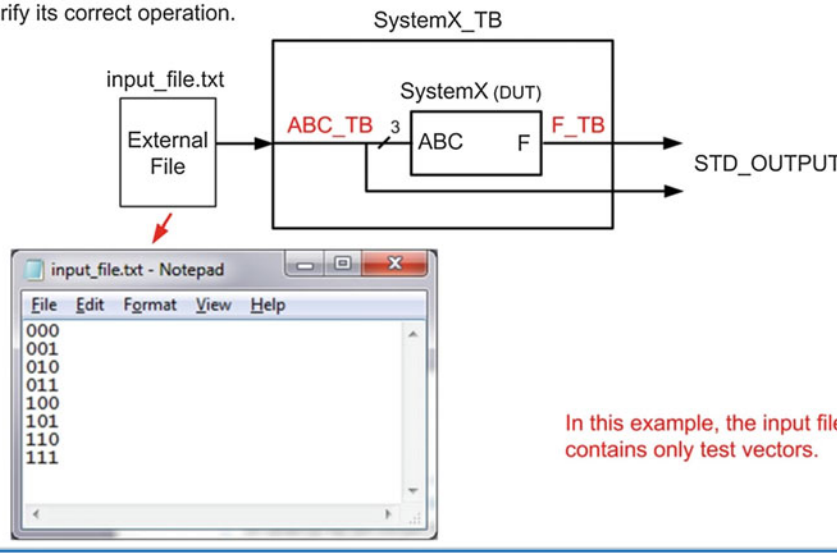
Example 8.15
Writing to STD_OUT from a Test Bench (Part 2)

8.5.8.3 Example: Reading from an External File in a Test Bench

Let's now look at an example of reading test vectors from an external file. Example 8.16 shows the test bench setup. In this example, the SystemX design from the prior example will be tested using vectors provided by an external file (input_file.txt). The test bench will read in each line of the file individually and sequentially. After reading a line, the test bench will drive the DUT with the input vector. In order to verify correct operation, the results will be written to the STD_OUTPUT of the computer.

Example: Reading From an External File in a Test Bench (Part 1)

An external file contains a set of input vectors that will be used to test the functionality of SystemX. The vectors will be read line by line from the file and then sent to the DUT. The input vectors and resulting output of SystemX will be written to STD_OUTPUT to verify its correct operation.



Example 8.16
Reading from an External File in a Test Bench (Part 1)

In order to read the external vectors, a file is declared in READ_MODE. This opens the external file and allows the VHDL test bench to access its lines. A variable is declared to hold the line that is read using the readline() procedure. In this example, the line variable for reading is called “current_read_line”. A variable is also declared that will ultimately hold the vector that is extracted from current_read_line. This variable (called current_read_field) is declared to be of type std_logic_vector(2 downto 0) because the vectors in the file are 3-bit values. Once the line is read from the file using the readline() procedure, the vector can be read from the line variable using the read() procedure. Once the value resides in the *current_read_field* variable, it can be assigned to the DUT input signal vector *ABC_TB*. A set of messages are then written to the STD_OUTPUT of the computer using the reserved file handle OUTPUT. The messages contain descriptive text in addition to the values of the input vector and output value of the DUT. Example 8.17 shows the process to implement this behavior in the test bench.

Example: Reading From an External File in a Test Bench (Part 2)

The following process reads external vectors from a file and drives them into SystemX.

```
STIMULUS : process

  file Fin: TEXT open READ_MODE is "input_file.txt";

  variable current_read_line   : line;
  variable current_read_field  : std_logic_vector(2 downto 0);
  variable current_write_line  : line;

  begin
    while (not endfile(Fin)) loop

      readline(Fin, current_read_line);
      read(current_read_line, current_read_field);

      ABC_TB <= current_read_field; wait for 125 ns;

      write(current_write_line, string'("Input Vector: ABC_TB="));
      write(current_write_line, ABC_TB);
      write(current_write_line, string'(" "));

      write(current_write_line, string'("DUT Output: F_TB="));
      write(current_write_line, F_TB);
      writeline(OUTPUT, current_write_line);

    end loop;

    wait;

end process;
```

Declare file for reading

Declare read line variable

Declare variable for vector read from line variable

Declare write line variable

Read line, read field
Assign field to ABC_TB

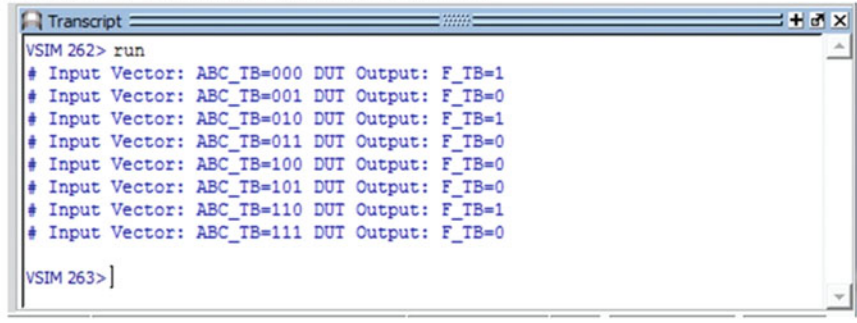
Write results to STD_OUTPUT

Example 8.17
Reading from an External File in a Test Bench (Part 2)

Example 8.18 shows the results of this test bench, which are written to STD_OUTPUT.

Example: Reading From an External File in a Test Bench (Part 3)

The STD_OUTPUT provides the status of the test.



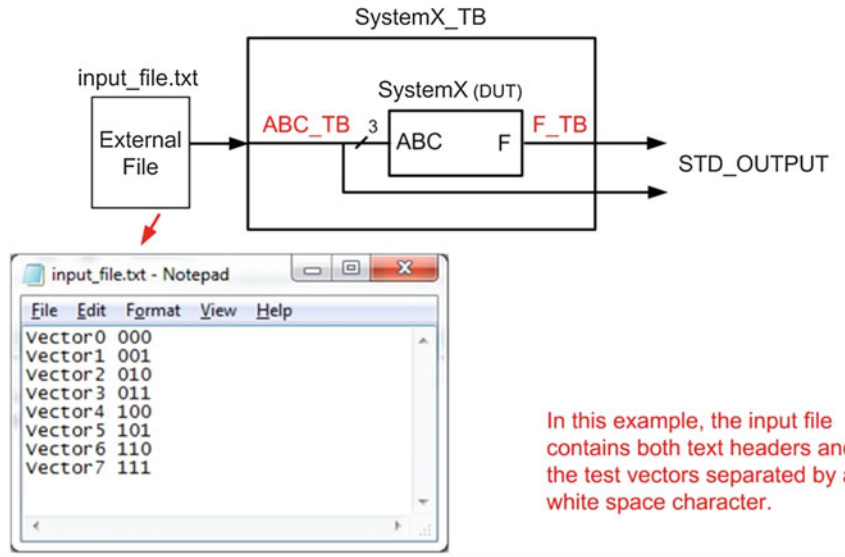
Example 8.18
Reading from an External File in a Test Bench (Part 3)

8.5.8.4 Example: Reading Space-Delimited Data from an External File in a Test Bench

As mentioned earlier, information in a line variable is treated as white space delimited by the read() procedure. This allows more information than just a single vector to be read from a file. When a read() procedure is performed on a line variable, it will extract information until it reaches either a white space or the end-of-line character. If a white space is encountered, the read() procedure will end. Let's take a look at an example of how to read information from a file when it contains both strings and vectors. Example 8.19 shows the test bench setup where an external file is to be read that contains both a text heading and test vector on each line. Since the header and the vector are separated with a white space character, two read() procedures need to be used to independently extract these distinct fields from the line variable.

Example: Reading Space-Delimited Data from an External File in a Test Bench (Part 1)

An external file contains both a text heading and a vector on each line of the file. The vectors will be used to drive the inputs of the DUT. The test bench will need to perform two read() procedures to extract the two separate fields from the line variable.



Example 8.19
Reading Space-Delimited Data from an External File in a Test Bench (Part 1)

The test bench will transfer a line from the file into a line variable using the readline() procedure just as in the previous example; however, this time two different variables will need to be defined in order to read the two separate fields in the line. Each variable must be declared to be the proper type and size for the information in the field. For example, the first field in the file is a string of seven characters. As a result, the first variable declared (current_read_field1) will be of type *string(1 to 7)*. Recall that strings are typically indexed incrementally from left to right starting with the index 1. The second field in the file is a 3-bit vector, so the second variable declared (current_read_field2) will be of type std_logic_vector (2 downto 0). Each time a line is retrieved from the file using the readline() procedure, two subsequent read() procedures can be performed to extract the two fields from the line variable. The second field (i.e., the vector) can be used to drive the input of the DUT. In this example, both fields are written to STD_OUTPUT in addition to the output of the DUT to verify proper functionality. Example 8.20 shows the test bench process which models this behavior.

Example: Reading Space-Delimited Data from an External File in a Test Bench (Part 2)

The following process reads external vectors from a file and drives SystemX.

```
STIMULUS : process

  file Fin: TEXT open READ_MODE is "input_file.txt";

  variable current_read_line   : line;
  variable current_read_field1 : string(1 to 7);
  variable current_read_field2 : std_logic_vector(2 downto 0);
  variable current_write_line  : line;

  begin

    while (not endfile(Fin)) loop

      readline(Fin, current_read_line);
      read(current_read_line, current_read_field1);
      read(current_read_line, current_read_field2);

      ABC_TB <= current_read_field2;

      wait for 125 ns;

      write(current_write_line, current_read_field1);
      write(current_write_line, string'(" "));
      write(current_write_line, current_read_field2);

      write(current_write_line, string'(" "));

      write(current_write_line, string'("DUT Output: F_TB="));
      write(current_write_line, F_TB);
      writeline(OUTPUT, current_write_line);

    end loop;

    wait;

end process;
```

Declare file for reading

Declare read line variable

Declare separate variables to hold the two different fields in the line variable

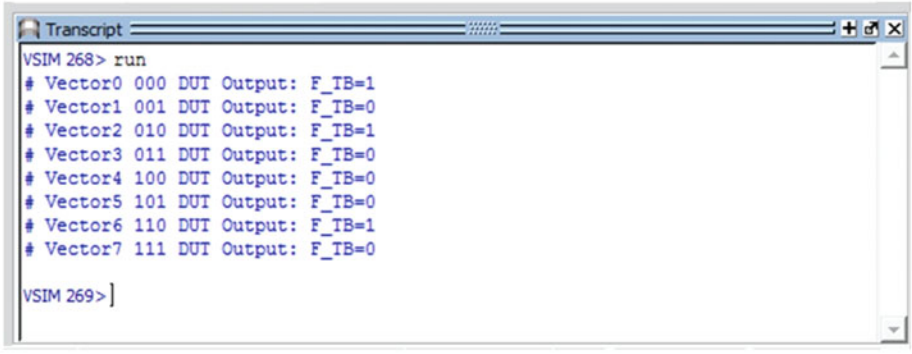
Declare write line variable

Two read() procedures are used to extract both fields from the line variable

Example 8.20
Reading Space-Delimited Data from an External File in a Test Bench (Part 2)

Example 8.21 shows the results of this test bench, which are written to STD_OUTPUT.

Example: Reading Space-Delimited Data from an External File in a Test Bench (Part 3)
The STD_OUTPUT provides the status of the test.



Example 8.21
Reading Space-Delimited Data from an External File in a Test Bench (Part 3)

8.5.9 Legacy Packages (STD_LOGIC_ARITH/UNSIGNED/SIGNED)

Prior to the release of the *numeric_std* package by IEEE, Synopsis, Inc. created a set of packages to provide computational operations for types std_logic and std_logic_vector. Since these arithmetic packages were defined very early in the life of VHDL, they were widely adopted. Unfortunately, due to these packages not being standardized through a governing body such as IEEE, vendors began modifying the packages to meet proprietary needs. This led to a variety of incompatibility issues that have plagued these packages. As a result, all new designs requiring computational operations should be based on the IEEE *numeric_std* package. While the IEEE standard is the recommended numerical package for VHDL, the original Synopsis packages are still commonly found in designs and in design examples, so providing an overview of their functionality is necessary.

Synopsis, Inc. created the **std_logic_arith** package to provide computational operations for types std_logic and std_logic_vector. Just as with the *numeric_std* package, this package defines two new types, **unsigned** and **signed**. Arithmetic, comparison, and shift operators are provided for these types that include **+**, –, *****, **abs**, >, <, <=, >=, =, /=, **shl**, and **shr**. This package also provides a set of conversion functions between types unsigned, signed, std_logic_vector, and integer. The syntax for these conversions is as follows:

Name	Input type	Return type
CONV_INTEGER	Unsigned	Integer
CONV_INTEGER	Signed	Integer
CONV_UNSIGNED	Integer, <size>	Unsigned
CONV_UNSIGNED	Signed	Unsigned
CONV_SIGNED	Integer, <size>	Signed
CONV_SIGNED	Unsigned	Signed
CONV_STD_LOGIC_VECTOR	Integer, <size>	std_logic_vector(size-1 downto 0)
CONV_STD_LOGIC_VECTOR	Unsigned, <size>	std_logic_vector(size-1 downto 0)
CONV_STD_LOGIC_VECTOR	Signed, <size>	std_logic_vector(size-1 downto 0)

The Synopsis packages have the ability to treat all std_logic_vectors in a design as either unsigned or signed by including an additional package. The **std_logic_unsigned** package, when included in conjunction with the std_logic_arith package, will treat all std_logic_vectors in the design as unsigned numbers. The syntax for using the Synopsis arithmetic packages on unsigned numbers is as follows. The *std_logic_1164* package is required to define types std_logic/std_logic_vector. The *std_logic_arith* package provides the computational operators for types std_logic/std_logic_vector. Finally, the *std_logic_unsigned* package treats all std_logic/std_logic_vector types as unsigned numbers when performing arithmetic operations:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
```

The **std_logic_signed** package works in a similar manner with the exception that it treats all std_logic/std_logic_vector types as signed numbers when performing arithmetic operations. The *std_logic_unsigned* and *std_logic_signed* packages are never used together since they will conflict with each other.

The syntax for using the std_logic_signed package is as follows:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_signed.all;
```

One of the more confusing aspects of the Synopsis packages is that they are included in the IEEE library. This means that both the *numeric_std* package (IEEE standard) and the *std_logic_arith* package (Synopsis, non-standard) are part of the same library, but one is recommended while the other is not. This is due to the fact that the Synopsis packages were developed first, and putting them into the IEEE library was the most natural location since this library was provided as part of the VHDL standard. When the numeric_std package was standardized by IEEE, it also was naturally inserted into the IEEE library. As a result, today's IEEE library contains both styles of packages.

CONCEPT CHECK

CC8.5(a) Why is standardization important for VHDL packages?

A) So that different CAD/CAE tools are interoperable.

B) To support IEEE.

C) So that synthesis is possible.

D) So that detailed manuals can be created.

CC8.5(b) Why doesn't the VHDL standard package simply include all of the functionality that has been created in all of the packages that were developed later?

A) There was not sufficient funding to keep the VHDL standard package updated.

B) If every package was included, compilation would take an excessive amount of time.

C) Explicitly defining packages helps remind the designer the proper way to create a VHDL model.

D) Because not all designs require all of the functionality in every package. Plus, some packages defined duplicate information. For example, both the numeric_bit and numeric_std have data types called *unsigned*.

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.
- A *test bench* is a way to simulate a device under test (DUT) by instantiating it as a component, driving in stimulus, and observing the outputs. Test benches do not have inputs or outputs and are unsynthesizable.
- The *report* and *assert* statements provide a way to perform automatic checking of the outputs of a DUT within a test bench.
- The IEEE STD_LOGIC_1164 package provides more realistic data types for modeling modern digital systems. This package provides the std_ulogic and std_logic data types. These data types can take on nine different values (U, X, 0, 1, Z, W, L, H, and -). The std_logic data type provides a resolution function that allows multiple outputs to be connected to the same signal. The resolution function will determine the value of the signal based on a predefined priority given in the function.
- The IEEE STD_LOGIC_1164 package provides logical operators and edge detection functions for the types std_ulogic and std_logic. It also provides conversion functions to and from the type *bit*.
- The IEEE NUMERIC_STD package provides the data types *unsigned* and *signed*. These types use the underlying data type std_logic. These types provide the ability to treat vectors as either unsigned or two's complement codes.
- The IEEE NUMERIC_STD package provides arithmetic operations for the types unsigned and signed. This package also provides conversion functions and type casts to and from the types integer and std_logic_vector.
- The TEXTIO and STD_LOGIC_TEXTIO packages provide the functionality to read and write to files from within a test bench. This allows more sophisticated vector patterns to be driven into a DUT. This also provides more sophisticated automatic checking of a DUT.

Exercise Problems

Section 8.1—The Process

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

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

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

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

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

8.1.6 When are signal assignments officially made in a process?

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

8.1.8 Can signals be declared in a process?

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

8.1.10 What happens to a variable when a process ends?

8.1.11 What is the assignment operator for variables?

Section 8.2—Conditional Programming Constructs

8.2.1 Design a VHDL model to implement the behavior described by the 4-input truth table in Fig. 8.4. 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 = 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

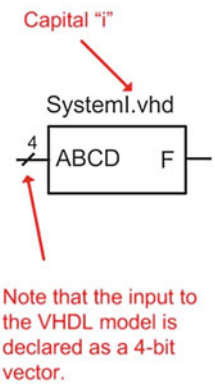


Fig. 8.4
System I Functionality

8.2.2 Design a VHDL model to implement the behavior described by the 4-input truth table in Fig. 8.4. 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.

8.2.3 Design a VHDL model to implement the behavior described by the 4-input minterm list in Fig. 8.5. 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.

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

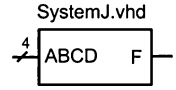


Fig. 8.5
System J Functionality

8.2.4 Design a VHDL model to implement the behavior described by the 4-input minterm list in Fig. 8.5. 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.

8.2.5 Design a VHDL model to implement the behavior described by the 4-input maxterm list in Fig. 8.6. 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.

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

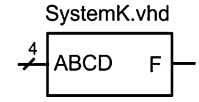


Fig. 8.6
System K Functionality

8.2.6 Design a VHDL model to implement the behavior described by the 4-input maxterm list in Fig. 8.6. 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.

8.2.7 Design a VHDL model to implement the behavior described by the 4-input truth table in Fig. 8.7. 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

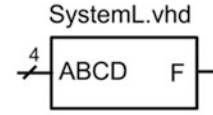


Fig. 8.7
System L Functionality

8.2.8 Design a VHDL model to implement the behavior described by the 4-input truth table in Fig. 8.7. 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.

8.2.9 Figure 8.8 shows the topology of a *4-bit shift register* when implemented structurally using D-flip-flops. Design a VHDL model to describe this functionality using a single process and sequential signal assignments instead of instantiating D-flip-flops. The figure also provides the block diagram for the entity definition. Use std_logic and std_logic_vector types for your signals.

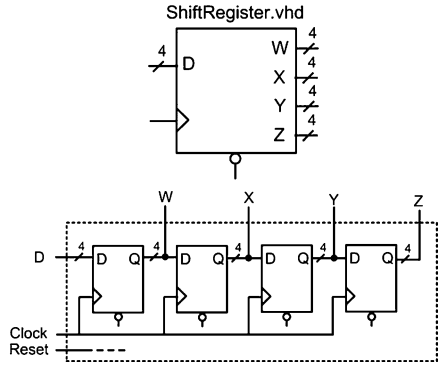


Fig. 8.8
4-Bit Shift Register Functionality

8.2.10 Design a VHDL model for a counter using a for loop with an output type of integer. Figure 8.9 shows the block diagram for the entity definition. The counter should increment from 0 to 31 and then start over. Use wait statements within your process to update the counter value every 10 ns. Consider using the loop variable of the for loop to generate your counter value. NOTE: This design is not synthesizable.

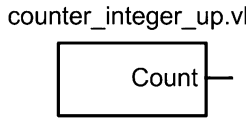


Fig. 8.9
Integer Counter Block Diagram

8.2.11 Design a VHDL model for a counter using a for loop with an output type of std_logic_vector (4 downto 0). Figure 8.10 shows the block diagram for the entity definition. The counter should increment from 00000_2 to 11111_2 and then start over. Use wait statements within your process to update the counter value every 10 ns. Consider using the loop variable of the for loop to generate an integer version of your count value, and then use a type conversion function to convert the integer to std_logic_vector. NOTE: This design is not synthesizable.

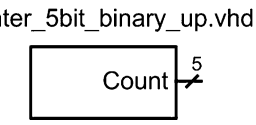


Fig. 8.10
5-Bit Binary Counter Block Diagram

Section 8.3—Signal Attributes

8.3.1 What is the purpose of a signal attribute?

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

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

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

Section 8.4—Test Benches

8.4.1 Design a VHDL test bench to verify the functional operation of the system in Fig. 8.4. Your test bench should drive in each input code for the vector ABCD in the order they appear in the truth table (i.e., "0000", "0001", "0010", ...). Have your test bench change the input pattern every 10 ns using the *wait for* statement within your stimulus process.

8.4.2 Design a VHDL test bench to verify the functional operation of the system in Fig. 8.4 using report and assert statements. Your test bench should drive in each input code for the vector ABCD in the order they appear in the truth table (i.e., "0000", "0001", "0010", ...). Have your test bench change the input pattern every 10 ns using the *wait for* statement within your stimulus process. Use the report and assert statements to output a message on the status of each test to the simulation transcript window. For each input vector, create a message that indicates the current input vector being tested, the resulting output of your DUT, and whether the DUT output is correct.

8.4.3 Design a VHDL test bench to verify the functional operation of the system in Fig. 8.5. Your test bench should drive in each input code for the vector ABCD in the order they appear in the truth table (i.e., "0000", "0001", "0010", ...). Have your test bench change the input pattern every 10 ns using the *wait for* statement within your stimulus process.

8.4.4 Design a VHDL test bench to verify the functional operation of the system in Fig. 8.5 using report and assert statements. Your test bench should drive in each input code for the vector ABCD in the order they appear in the truth table (i.e., "0000", "0001", "0010", ...). Have your test bench change the input pattern every 10 ns using the *wait for* statement within your stimulus process. Use the report and assert statements to output a message on the status of each test to the simulation transcript window.

For each input vector, create a message that indicates the current input vector being tested, the resulting output of your DUT, and whether the DUT output is correct.

8.4.5 Design a VHDL test bench to verify the functional operation of the system in Fig. 8.6. Your test bench should drive in each input code for the vector ABCD in the order they appear in the truth table (i.e., "0000", "0001", "0010", ...). Have your test bench change the input pattern every 10 ns using the *wait for* statement within your stimulus process.

8.4.6 Design a VHDL test bench to verify the functional operation of the system in Fig. 8.6 using report and assert statements. Your test bench should drive in each input code for the vector ABCD in the order they appear in the truth table (i.e., "0000", "0001", "0010", ...). Have your test bench change the input pattern every 10 ns using the *wait for* statement within your stimulus process. Use the report and assert statements to output a message on the status of each test to the simulation transcript window. For each input vector create a message that indicates the current input vector being tested, the resulting output of your DUT, and whether the DUT output is correct.

8.4.7 Design a VHDL test bench to verify the functional operation of the system in Fig. 8.7. Your test bench should drive in each input code for the vector ABCD in the order they appear in the truth table (i.e., "0000", "0001", "0010", ...). Have your test bench change the input pattern every 10 ns using the *wait for* statement within your stimulus process.

8.4.8 Design a VHDL test bench to verify the functional operation of the system in Fig. 8.7 using report and assert statements. Your test bench should drive in each input code for the vector ABCD in the order they appear in the truth table (i.e., "0000", "0001", "0010", ...). Have your test bench change the input pattern every 10 ns using the *wait for* statement within your stimulus process. Use the report and assert statements to output a message on the status of each test to the simulation transcript window. For each input vector, create a message that indicates the current input vector being tested, the resulting output of your DUT, and whether the DUT output is correct.

Section 8.5—Packages

8.5.1 What are all the possible values that a signal of type *std_logic* can take on?

8.5.2 What is the difference between the types *std_ulogic* and *std_logic*?

8.5.3 If a signal of type *std_logic* is assigned both a 0 and Z at the same time, what will the final signal value be?

8.5.4 If a signal of type *std_logic* is assigned both a 1 and X at the same time, what will the final signal value be?

8.5.5 If a signal of type *std_logic* is assigned both a 0 and L at the same time, what will the final signal value be?

8.5.6 Are any arithmetic operations provided for the type *std_logic_vector* in the STD_LOGIC_1164 package?

8.5.7 If you declare a signal of type *unsigned* from the NUMERIC_STD package, what are all the possible values that the signal can take on?

8.5.8 If you declare a signal of type *signed* from the NUMERIC_STD package, what are all the possible values that the signal can take on?

8.5.9 If two signals (A and B) are declared of type *signed* from the NUMERIC_STD package and hold the values A <= "1111" and B <= "0000", which signal has a greater value?

8.5.10 If two signals (A and B) are declared of type *unsigned* from the NUMERIC_STD package and hold the values A <= "1111" and B <= "0000", which signal has a greater value?

8.5.11 If you are using the NUMERIC_STD package, what is the syntax to convert a signal of type *unsigned* into *std_logic_vector*?

8.5.12 If you are using the NUMERIC_STD package, what is the syntax to convert a signal of type *integer* into *std_logic_vector*?

8.5.13 Design a self-checking VHDL test bench that reads in test vectors from an external file to verify the functional operation of the system in Fig. 8.7. Create an input text file called "input_vectors.txt" that contains each input code for the vector ABCD in the order they appear in the truth table (i.e., "0000", "0001", "0010", ...) on a separate line. The test bench should read in each line of the file individually and use the corresponding input vector to drive the DUT. Write the output results to an external file called "output_vectors.txt".

8.5.14 Design a self-checking VHDL test bench that reads in test vectors from an external file to verify the functional operation of the system in Fig. 8.7. Create an input text file called "input_vectors.txt" that contains each input code for the vector ABCD in the order they appear in the truth table (i.e., "0000", "0001", "0010", ...) on a separate line. The test bench should read in each line of the file individually and use the corresponding input vector to drive the DUT. Write the output results to the STD_OUTPUT of the simulator.