Pearson New International Edition

VHDL for Engineers

Kenneth L. Short

ALWAYS LEARNING™

PEARSON®

Pearson New International Edition

VHDL for Engineers

Kenneth L. Short

PEARSON

LISTING 5.6.1
A 4-bit magnitude comparator described using a for loop.

```
library ieee;
use ieee.std_logic_1164.all;

entity mag_comp is
   port ( p, q : in std_logic_vector (3 downto 0);
      p_gt_q_bar, p_eq_q_bar, p_lt_q_bar : out std_logic);
end mag_comp;

architecture behavior of mag_comp is
begin
   comp: process (p, q)
   begin

      p_gt_q_bar <= '1';  --defaults
      p_eq_q_bar <= '1';
      p_lt_q_bar <= '1';

      for i in 3 downto 0 loop
         if ((p(i) = '1') and (q(i) = '0')) then
            p_gt_q_bar <= '0';
            exit;
         elsif((p(i) = '0') and (q(i) = '1')) then
            p_lt_q_bar <= '0';
            exit;
         elsif i = 0 then
            p_eq_q_bar <= '0';
         end if;
      end loop;

   end process comp;
end behavior;
```

The process first sets all the outputs to `'1'`s, then uses a for loop to determine which output to make a `'0'`.

The range of the loop parameter `i` is from 3 down to 0. During each loop iteration, the if statement compares two corresponding elements of `p` and `q` to determine which vector is greater. As soon as the determination can be made, the loop is exited.

In each pass through the loop, the if clause determines whether `p(i)` is a `'1'` and `q(i)` is a `'0'`. If this condition is true, output `p_gt_q_bar` is assigned `'0'`, and the exit statement terminates the loop.

If the previous condition does not evaluate true, the first elsif clause determines whether `p(i)` is a `'0'` and `q(i)` is a `'1'`. If this condition is true, output `p_lt_q_bar` is assigned `'0'` and the exit statement terminates the loop.

The last elsif determines whether loop parameter `i` equals `0`. This condition can be true only on the fourth pass through the loop. If this condition is true, output `p_eq_q_bar` is assigned `'0'`, because all of the corresponding elements of the two vectors must have been equal to reach this point.

5.7 VARIABLES

The four kinds of VHDL objects were introduced in Chapter 3: signals, constants, variables, and files. Signals and constants have been used extensively in previous examples. We now examine variables and use them in descriptions.

Normal Variables and Shared Variables

There are two kinds of variables in VHDL, normal variables and shared variables. *Shared variables* can be accessed from multiple processes and are not synthesizable. Normal variables can only be accessed from a single process or subprogram and are synthesizable. The subsequent discussions of variables in this book are limited to normal variables.

Storing Intermediate Values in a Process or Subprogram

Variables are used to store intermediate values in a process or subprogram. Variables must be declared in the declarative part of the process or subprogram in which they are used. A variable is local to the process or subprogram in which it is declared (visible only within that process or subprogram).

Variables in a process preserve their values between executions of the process. That is, between its suspension and reactivation. In contrast, variables in a subprogram do not preserve their values between executions of the subprogram. Variables in subprograms are discussed further in Chapter 12.

The syntax for a variable declaration is given in Figure 5.7.1.

Variable Initial Value

A variable can optionally be assigned an initial value when it is declared. An initial value is assigned by using the variable assignment symbol := . For example:

```
variable count : std_logic_vector (3 downto 0)
          := "0000";
```

This statement declares variable `count` and gives it an initial value of `"0000"`. ***Importantly, IEEE 1076.6 compliant synthesizers ignore initial values assigned to***

variable_declaration ::=
[**~~shared~~**] **variable** identifier_list : subtype_indication [<u>:= expression</u>] ;

FIGURE 5.7.1
Syntax for a variable declaration.

variables. Accordingly, in a description, if a variable requires an initial value, it should be assigned by a separate variable assignment statement in the process or subprogram that uses the variable.

Variable Assignment Statement

Variables are assigned values by using a variable assignment statement. To distinguish assignments to variables from assignments to signals, two different assignment symbols are used. The symbol `:=` is the *variable assignment symbol*. For example:

```
count := "0000";
```

assigns the value `"0000"` to the variable `count`.

Variable Assignments Take Effect Immediately

When a variable is assigned a value, it takes that value immediately. This action is the same as for a variable in a conventional programming language. However, it is different from the situation for a signal. When a signal is assigned a value, the assigned value does not take effect immediately. When a signal is assigned a value in a process, the signal does not take the new value until after the process has suspended. This characteristic of a signal has not been previously emphasized. It is considered in detail in Chapter 6.

It will also become more clear in Chapter 6 when use of a variable in a process is preferable to use of a signal. However, as a general guideline, a variable is used to store data within a process when the new value assigned to the variable is to be used (read) in the same execution of the process.

Variable and Signal Differences

At this point, some of the obvious differences between a variable and a signal in a process can be stated.

- A variable used in a process must be declared inside the process. A signal used in a process must be declared outside of the process (in the declarative part of the architecture or as a port in the associated entity declaration).
- A variable is only visible in the process in which it is declared. A signal is visible to all processes in the architecture in which it is declared.

As an example of the use of variables, the magnitude comparator is described using variables in Listing 5.7.1.

LISTING 5.7.1
A 4-bit magnitude comparator description using variables.

```
library ieee;
use ieee.std_logic_1164.all;

entity mag_comp is
   port ( p, q : in std_logic_vector (3 downto 0);
      p_gt_q_bar, p_eq_q_bar, p_lt_q_bar : out std_logic);
end mag_comp;
```

```
architecture vari of mag_comp is
begin
   comp: process (p, q)

   variable  p_gt_q_bar_v, p_lt_q_bar_v, p_eq_q_bar_v : std_logic;

   begin

      p_gt_q_bar_v  := '1';  -- default values
      p_lt_q_bar_v  := '1';
      p_eq_q_bar_v  := '1';

      for i in 3 downto 0 loop
         if ((p(i) = '1') and (q(i) = '0')) then
            p_gt_q_bar_v := '0';
            exit;
         elsif ((p(i) = '0') and (q(i) = '1')) then
            p_lt_q_bar_v := '0';
            exit;
         end if;
      end loop;

      if ((p_gt_q_bar_v = '1') and (p_lt_q_bar_v = '1')) then
         p_eq_q_bar_v := '0';
      end if;

      p_gt_q_bar <= p_gt_q_bar_v;  -- assign variable values to ports
      p_lt_q_bar <= p_lt_q_bar_v;
      p_eq_q_bar <= p_eq_q_bar_v;

   end process comp;
end vari;
```

In this description, the determination as to whether to assert `p_eq_q_bar` is made based on the values of `p_gt_q_bar` and `p_lt_q_bar` determined during the same execution of the process. Variables are used to store intermediate values.

Variables `p_gt_q_bar_v`, `p_lt_q_bar_v`, and `p_eq_q_bar_v` are declared in the declarative part of the process. In the statement part of the process these variables are all given a default value of `'1'`. A loop is used to determine if either `p_gt_q_bar_v` or `p_lt_q_bar_v` should be asserted. If either of these variables is to be asserted, it is assigned a `'0'`. This assignment takes effect immediately.

After the loop is terminated, an if statement is used to determine whether variable `p_eq_q_bar_v` should be asserted. If either `p_gt_q_bar_v` or `p_lt_q_bar_v` were assigned a value of `'0'` in an execution of the loop statement, that `'0'` is detected in the execution of the if statement.

Before the process ends, the values of the variables are assigned to the corresponding ports. This must be done inside the process because variables are not visible outside of a process.

5.8 PARITY DETECTOR EXAMPLE

A few examples of designs for a parity detector are given in this section. These examples clarify some important concepts concerning variables and signals. Design entity `parity` takes a four-element std_logic_vector named `din` as its input and has a single std_logic output `oddp`. Output `oddp` is asserted when the number of 1s in the input vector is odd, otherwise it is unasserted.

Dataflow Description of Parity Detector

If we are aware that a four-input XOR gate detects odd parity for a 4-bit vector, the simplest way to describe the parity detector is using a single concurrent signal assignment statement. A dataflow style description using this approach is given in Listing 5.8.1.

A simulation of this design description results in the waveforms in Figure 5.8.1.

The testbench sequences through all possible binary input combinations. Input vector `din` is displayed as both a composite signal, whose value is expressed in hexadecimal, and in terms of its separate elements. The waveforms indicate that the description is functionally correct.

Synthesis of the design description in Listing 5.8.1 produces the hierarchical representation of the logic in Figure 5.8.2. The notation [3:0] represents a 4-bit bus. The notation [3] indicates bit 3 of the bus. As expected, the logic is a four-input XOR gate.

LISTING 5.8.1
Dataflow design description of parity detector.

```
library ieee;
use ieee.std_logic_1164.all;

entity parity is
   port (
      din: in std_logic_vector (3 downto 0); -- 4-bit input vector
      oddp: out std_logic                     -- asserted for odd parity
      );
end parity;

architecture dataflow of parity is
begin
   oddp <= din(3) xor din(2) xor din(1) xor din(0);
end dataflow;
```

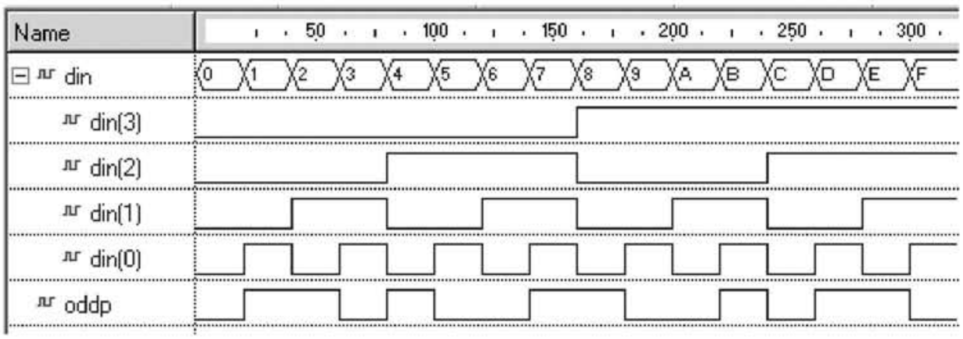

FIGURE 5.8.1
Simulation of dataflow style parity detector.

Loop and Signal Description of Parity Detector

In contrast to the dataflow approach, we might envision the function of the parity detector in terms of an algorithm that examines each bit of the input vector one at a time and keeps track of whether the parity of all bits previously examined is odd. After the last bit has been examined, the parity of the entire vector is known.

An architecture using this iterative approach is given in Listing 5.8.2. The entity declaration for the parity detector remains the same.

In Listing 5.8.2, a loop is used to examine each bit of the input vector in turn, starting with the leftmost bit. A signal named `odd` is used to keep track of the computed parity. This signal is declared in the declarative part of the architecture.

In the process, signal `odd` is first assigned the value `'0'`. The loop is then entered. Each time through the loop, `odd` is assigned the XOR of the current value of `odd` and the element of the input vector selected by the loop index. When the loop is complete, the value of signal `odd` is assigned to output port `oddp`.

When the description in Listing 5.8.2 is simulated using the same testbench as before, the waveforms in Figure 5.8.3 result.

It is obvious from the waveforms that the output is incorrect. The value of `oddp` remains `'U'` for the entire simulation. The value `'U'` is indicated in the waveform by the dotted line halfway between the `'0'` and `'1'` logic levels.

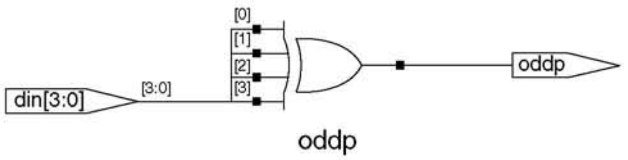

FIGURE 5.8.2
Logic synthesized from the dataflow style parity detector description in Listing 5.8.1.