Chapter 5

New and Changed Statements

VHDL provides various forms of statements for modeling the behavior of hardware and testbenches. Sequential statements are used to express algorithms within processes and subprograms, where there is just one thread of control. Concurrent statements, on the other hand, express multi-threaded control. They are also used to represent structural decomposition of a design into concurrently operating subsystems.

In this chapter, we look at the enhancements to the statement repertoire in VHDL-2008. We start with changes to assignment statements, which include new sequential forms that mirror conditional and selected concurrent assignments. Next, we look at changes to case statements that allow matching of standard-logic values with don't care elements. Finally, we look at extensions to if-generate statements that allow multiple conditions to be checked, and a new case-generate statement.

5.1 Conditional and Selected Assignments

In earlier versions of VHDL, sequential and concurrent signal assignment statements had different syntactic forms. Sequential signal assignments, appearing in processes and subprograms, could only take the simple form of a target signal on the left-hand side and a list of one or more values and delays on the right-hand side. Concurrent signal assignments, appearing in architectures, could take this simple form, but could also take conditional and selected forms. While we could embed a sequential assignment in an if statement or a case statement, the differences between the sequential and concurrent contexts was a cause for confusion among designers.

In this section, we describe the way VHDL-2008 extends assignments. This includes allowing conditional and selected forms of signal assignments in processes and subprograms, providing for a signal to be forced by a conditional or selected assignment, and providing selected and conditional variable assignments.

5.1.1 Sequential Signal Assignments

VHDL-2008 extends the allowed forms of signal assignments to be consistent between the sequential and concurrent contexts. Within a process or subprogram, we can write conditional and selected signal assignments in the same form as those in architecture bodies. The effect is equivalent to writing simple signal assignments within if statements or case statements, but the notation is more succinct.

EXAMPLE 5.1 *Register process using a conditional assignment*

A process representing a register with synchronous reset can be written using a conditional signal assignment as follows:

```
reg : process (clk) is
begin
  if rising_edge(clk) then
    q <= (others => '0') when reset else d;
  end if;
end process reg;
```

The conditional assignment in the process is equivalent to the if statement:

```
if reset then
  q <= (others => '0')
else
  q <= d;
end if;
```

EXAMPLE 5.2 *Next-state process for a finite-state machine*

Use of selected assignments simplifies description of the next-state logic of a finite-state machine, as is shown by the following process outline:

```
next_state_logic : process (all) is
begin
  with current_state select
    idle =>
      next_state <= pending1 when request and busy     else
                    active1  when request and not busy else
                    idle;
    pending1 =>
      ...
    ...
  end case;
end process next_state_logic;
```

EXAMPLE 5.3 *Combined multiplexer and register using a selected assignment*

We can model a register with a multiplexer at its input in a single process as follows:

```
mux_reg : process (clk) is
begin
  if rising_edge(clk) then
    with d_sel select
```

```
      q <= source0 when "00",
           source1 when "01",
           source2 when "10",
           source3 when "11";
  end if;
end process mux_reg;
```

The selected assignment in the process is equivalent to the case statement:

```
case d_sel is
  when "00" =>
    q <= source0;
  when "01"
    q <= source1;
  when "10"
    q <= source2;
  when "11"
    q <= source3;
end case;
```

When we write a conditional or selected signal assignment in a sequential context, we can include delays and multiple waveform values, just as we do in concurrent contexts. For example, in a stimulus-generator process, we could write the assignment:

```
req <= '1', '0' after T_fixed when fixed_delay_mode else
       '1', '0' after next_random_delay(ran_seed);
```

If we need to include an inertial or transport delay specification in a sequential assignment, we write it in the same way as in a concurrent assignment. For example, a sequential conditional assignment using transport delay could be written as:

```
wire_out <= transport
  wire_in after T_wire_delay when delay_mode = fixed else
  wire_in after delay_lookup("wire_out");
```

Likewise, a sequential conditional assignment using inertial delay could be written as:

```
with speed_grade select
  z <= reject Tpr inertial
    result after Tpd_std when std_grade,
    result after Tpd_fast when fast_grade,
    result after Tpd_redhot when redhot_grade;
```

We can also use the reserved word **unaffected** in a sequential signal assignment to represent no assignment to the target signal, for example:

```
with dut_state select
  dut_req <= '1' when ready,
             '0' when ack,
             unaffected when others;
```

A related change is that we can use the reserved word **unaffected** in a simple sequential signal assignment. This was previously illegal. In VHDL-2008, we can write the following statements within a process or subprogram:

```
if dut_busy then
  collision_count := collision_count + 1;
  dut_req <= unaffected;
else
  accepted_count := accepted_count + 1;
  dut_req <= '1';
end if;
```

The assignment using `unaffected` is the same as doing nothing (using a null statement), but the design intent is explicitly documented. It is clear that we did not inadvertently omit an assignment.

One aspect of concurrent signal assignments that we cannot include in a sequential assignment is the reserved word **guarded**. The effect of including **guarded** in a concurrent assignment is to cause the target signal to be disconnected when a `guard` signal is true, and to reconnect when the `guard` signal becomes false. Reconnection is done by executing the concurrent assignment. Thus, the `guard` signal has some external control over when the concurrent assignment is executed. This would not be appropriate for a sequential assignment, which should only be executed when control reaches it within the enclosing process or subprogram.

5.1.2 Forcing Assignments

In Section 2.2, we described the new features in VHDL-2008 for forcing and releasing signals. Force and release assignments are both forms of sequential signal assignment statements. VHDL-2008 also allows us to write forcing assignments in the form of conditional and selected assignments within processes and subprograms. A conditional forcing assignment has the form

```
signal_name <= force mode
  value when condition else
  ...
```

and a selected forcing assignment has the form

```
with expression select
  signal_name <= force mode
    value when choices,
    ...
```

The *mode* is optional, and can be either **in** or **out** to specify forcing of the effective value or the driving value of the target signal, respectively, as described in Section 2.2. The effect of these statements is to allow us to choose the value to force onto the target, depending on a number of conditions or on the value of an expression. They provide a more succinct way of writing the choice than embedding a number of simple forcing assignments in an if statement or case statement.

EXAMPLE 5.4 *Conditional forcing assignment*

A conditional forcing assignment can be used to choose between a randomly generated stimulus value or a directed-test stimulus value in a loop that applies successive tests. The stimulus value is used to force the effective value of a bidirectional port of a design under test. The code in the testbench is:

```
alias dut_d_bus is
  <<signal dut.d_bus:std_logic_vector(15 downto 0)>>;
...

for test_count in 1 to num_tests loop
  dut_d_bus <= force in
    next_random_stim(dut_d_bus'length)
      when test_mode = random else
    directed_stim(test_count);
  wait for test_interval;
end loop;
```

5.1.3 Variable Assignments

One of the reasons for providing sequential forms of conditional and selected signal assignments in VHDL-2008 is to provide consistency with concurrent signal assignments. In the further interest of consistency, VHDL-2008 also provides conditional and selected forms of variable assignment statement for use in processes and subprograms. A conditional variable assignment has the form

```
variable_name := value when condition else
                 ...
```

and a selected variable assignment has the form

```
with expression select
  variable_name := value when choices,
                   ...
```

EXAMPLE 5.5 *Conditional assignment for an intermediate variable*

A variable can be used for an intermediate value in a synthesizable process. No actual storage is implied for the variable, provided it is updated on all execution paths through the process. A conditional variable assignment allows us to make assignments to variables in the same succinct form that we can use for signals. Thus, in the process:

```
arith_unit : process (all) is
  variable tmp : operand_type;
begin
  tmp := a - b when mode else a + b;
  new_result <= result + scale * tmp;
end process arith_unit;
```

the assignment to the variable `tmp` is equivalent to the statements:

```
if mode then
  tmp := a - b;
else
  tmp := a + b;
end if;
```

EXAMPLE 5.6 *Selected assignment for combined multiplexer and register*

We can use a selected variable assignment for an intermediate variable representing a multiplexer at the input to a register. The process is:

```
mux_reg : process (clk) is
  variable mux : data_type;
begin
  if rising_edge(clk) then
    with mux_sel select
      mux := in0 when "00",
             in1 when "01",
             in2 when "10",
             in3 when "11",
             (others => 'X') when others;
    reg_out <= (others => '0') when reset else mux;
  end if;
end process mux_reg;
```

5.2 Matching Case Statements

A case statement in VHDL allows us to perform alternative actions depending on the value of an expression. We write choice values in each alternative, immediately preceding the "=>" symbol. The choices are compared for exact equality with the expression value to select an alternative. If the type of the case expression and choices is a vector of std_logic values, the exact comparison is not always what we want. In particular, we would like to be able to include don't care elements ('–') in the choices to indicate that we don't care about some elements of the case expression when selecting an alternative.

VHDL-2008 provides a new form of case statement, called a *matching case statement*, that uses the predefined "?=" operator described in Section 4.5 to compare choice values with the expression value. We include a question mark symbol after the keyword **case**, as follows:

```
case? expression is
  ...
end case?;
```

The most common use of a matching case statement is with an expression of a vector type whose elements are std_ulogic or std_logic values. That includes the standard types std_ulogic_vector, std_logic_vector, unsigned, signed, and so on. It also includes vector types that we might define. With a case expression of such a type, we can write choice values that include '–' elements to specify don't care matching.

EXAMPLE 5.7 *Priority arbiter using don't care matching*

Suppose we have vectors of request and grant signals, declared as follows:

```
signal request, grant : std_logic_vector(0 to 3);
```

We can use a matching case statement in a priority arbiter, with request 0 having highest priority:

```
case? request is
  when "1---" => grant <= "1000";
  when "01--" => grant <= "0100";
  when "001-" => grant <= "0010";
  when "0001" => grant <= "0001";
  when others => grant <= "0000";
end case?;
```

Each choice is compared with the case expression using the predefined "?=" operator. Thus, the first choice matches values "1000", "1001", "100X", "H000", and so on, and similarly for the remaining choices. This is a much more succinct way of describing the arbiter than using an ordinary case statement. Moreover, unlike a sequence of tests in an if statement, it does not imply chained decision logic.

When we use a matching case statement with a vector-type expression, the value of the expression must not include any '–' elements. (This is different from the choice values, which can include '–' elements.) The reason is that an expression value with a '–' element would match multiple choice values, making selection of an alternative ambiguous. Normally, this rule is not a problem, since we don't usually assign '–' values to signals or variables. They usually just occur in literal values for comparison and in testbench assertions.

In an ordinary case statement, we need to include choices for all possible values of the case expression. A related rule applies in a matching case statement. Each possible value of the case expression, except those that include any '–' elements, must be represented by exactly one choice. By "represented," we mean that comparison of the choice and the expression value using the "?=" operator yields '1'. Hence, our choice values would generally just include '0', '1', and '–' elements, matching with '0', 'L', '1', 'H' elements in the case expression value. We could also include 'L' and 'H' elements in a choice. However, we would not include 'U', 'X', 'W', or 'Z' choice elements, since they only ever produce 'U' or 'X' results, and so never match. As with an ordinary case statement, we can include an others choice to represent expression values not otherwise represented. Unlike an ordinary case statement, a choice can represent multiple expression values if it contains a '–' element.

We mentioned that a vector type including std_ulogic or std_logic values is the most common type for a matching case statement. Less commonly, we can write an expression of type std_ulogic, std_logic, bit, or a vector of bit elements. These are the other types for which the "?=" operator is predefined. For std_ulogic or std_ulogic expressions, the choice values would typically be either '0' (matching an expression value of '0' or 'L') or '1' (matching an expression value of '1' or 'H'). We would not write a choice of '–', since that would match all expression values, preventing us from selecting distinct alternatives. For case expressions of type bit or a vector of bit elements, a matching case statement has exactly the same behavior as an ordinary case statement. VHDL-2008 allows matching case statements of this form to allow synthesizable models to be written uniformly regardless of whether bit or std_logic data types are used.

5.2.1 Matching Selected Assignments

Selected assignments in VHDL are shorthand notations for assignments within case statements. This applies to concurrent selected signal assignments in earlier versions of VHDL, as well as to sequential selected signal and variable assignments in VHDL-2008. In all of these forms of selected assignment, we can include a "?" symbol after the **select** keyword to indicate that the implied case statement is a matching case statement instead of an ordinary case statement. The rules covering the type of the case expression and the way in which choices are matched then apply to the selected assignment.

EXAMPLE 5.8 *Priority arbiter using matching selected assignment*

We can rewrite the priority arbiter from Example 5.7 using a matching selected assignment as follows:

```
with request select?
  grant <= "1000" when "1---",
           "0100" when "01--",
           "0010" when "001-",
           "0001" when "0001",
           "0000" when others;
```

5.3 If and Case Generate

Earlier versions of VHDL provide two forms of generate statements. A for-generate statement allows us to include multiple copies of component instances or other concurrent statements. An if-generate statement allows us to decide whether or not to include concurrent statements based on the value of a condition. If we want to decide between alternate sets of concurrent statements depending on whether a condition is true or false, we use two if-generate statements with complementary conditions, as follows:

```
L1: if condition generate
  -- first alternative
  ...
end generate L1;

L2: if not condition generate
  -- second alternative
  ...
end generate L2;
```

This is somewhat cumbersome, and inconsistent with sequential if statements, in which we can specify alternates using **elsif** and **else** clauses. VHDL-2008 extends the form of if generate statements to allow us to specify alternatives in a way similar to sequential if statements. We can rewrite the above pair of if-generate statements as follows:

```
L: if condition generate
  -- first alternative
  ...
else generate
  -- second alternative
  ...
end generate L;
```

We can also include further conditions to test, as follows:

```
L: if condition1 generate
  -- first alternative
  ...
elsif condition2 generate
  -- second alternative
```

```
    ...
  ...
  else generate
    -- last alternative
    ...
  end generate L;
```

Each of the alternatives can be just a set of concurrent statements, or it can include declarations as well as concurrent statements. In the latter case, we write **begin** and **end** keywords around the statements, as follows:

```
  L: if condition1 generate
      -- first alternative declarations
      ...
    begin
      -- first alternative statements
      ...
    end;
  elsif condition2 generate
    ...
  end generate L;
```

When the model is elaborated, the conditions in the if-generate statement are tested from first to last until one is found that is true. The corresponding declarations (if any) and concurrent statements are then included in the elaborated model. If no condition is true and there is an **else** generate alternative, the declarations and statements from that alternative are included. The **else generate** alternative is optional, allowing for the possibility of no declarations or statements being included if none of the conditions is true. Of course, if we omit the **else generate** alternative and there is only one condition to test, the if-generate statement collapses down to the pre-VHDL-2008 form.

EXAMPLE 5.9 *Boundary conditions in a replicated structure*

We often use for-generate statements to replicate cells in a regular structure, and include nested if-generate statements to deal with the differences between the end replications and those in the middle. For example, a ripple-carry adder has a half adder at the least-significant end and has different carry in and out connections for the cells at the ends and in the middle. We can use a nested if-generate with three alternatives to deal with the differences:

```
adder: for i in width-1 downto 0 generate
  signal carry_chain : unsigned(width-1 downto 1);
begin
  adder_cell: if i = width-1 generate -- most-significant cell
    add_bit: component full_adder
      port map (a => a(i), b => b(i), s => s(i),
                c_in => carry_chain(i), c_out => c_out);
      elsif i = 0 generate -- least-significant cell
        add_bit: component half_adder
          port map (a => a(i), b => b(i), s => s(i),
                    c_out => carry_chain(i+1));
      else generate -- middle cell
        add_bit: component full_adder
          port map (a => a(i), b => b(i), s => s(i),
                    c_in => carry_chain(i),
                    c_out => carry_chain(i+1));
      end generate adder_cell;
    end generate adder;
```

VHDL-2008 also provides a case-generate statement, in which we specify alternatives in a similar way to a case statement. We specify a static expression (one whose value can be computed during elaboration), and choice values for each alternative. The form of a case-generate statement is:

```
L: case expression generate
  when choice1 =>
    -- first alternative
    ...
  when choice2 =>
    -- second alternative
    ...
  ...
end generate L;
```

As in the if-generate statement, each alternative can be just a set of concurrent statements, or it can include declarations as well as concurrent statements, with **begin** and **end** keywords around the statements. The rules governing sequential case statement expressions and choices also apply to the expression and choices in a case-generate statement, with the further stipulation that the expression be static. When the model is elaborated, the expression is evaluated, and the alternative whose choice is the same as the expression value is selected. The declarations (if any) and the statements from that alternative are included in the elaborated model.

EXAMPLE 5.10 *Alternative structures for a complex multiplier*

Multiplication of complex numbers in Cartesian form involves four scalar multiplications, a subtraction, and an addition. Depending on the constraints that apply to a design, these operations can be implemented in one clock cycle using multiple function units, in multiple clock cycles using fewer function units, or in a pipeline. Suppose we have an enumeration type, defined as follows, for specifying the implementation to use:

```
type implementation_type is
      (single_cycle, multicycle, pipelined);
```

An entity declaration for a complex multiplier has a generic constant of this type controlling the implementation:

```
entity complex_multiplier is
  generic ( implementation : implementation_type; ... );
  port ( ... );
end entity complex_multiplier;
```

Within the architecture, we use the value of the generic constant in a case-generate statement to determine what components to instantiate and how to interconnect them:

```
architecture rtl of complex_multiplier is
  ...
begin

  mult_structure : case implementation generate
    when single_cycle =>
        signal real_pp1, real_pp2 : ...;
        ...
      begin
        real_mult1 : component multiplier
          port map ( ... );
        ...
      end;
    when multicycle =>
        signal real_pp1, real_pp2 : ...;
        ...
      begin
        mult : component multiplier
          port map ( ... );
        ...
      end;
    when pipelined =>
        signal real_pp1, real_pp2 : ...;
        ...
      begin
        mult1 : component multiplier
          port map ( ... );
        ...
      end;
  end generate mutl_structure;

end architecture rtl;
```

The case-generate statement includes three alternatives, one for each possible implementation style. Each alternative can have local declarations and concurrent statements with the same names and labels as those in other alternatives, as well as differently named declarations and differently labeled statements.

5.3.1 Configuration of If and Case Generate

One of the main difficulties that has prevented introduction of case-generate statements and if-generate statements with multiple alternatives in earlier versions of VHDL has been working out a way of configuring the alternatives. VHDL-2008 handles this by requiring each alternative to be labeled if it is to be referenced in a configuration declaration. The alternative labels are in addition to the overall statement label. In an if-generate statement, we include a label before each condition:

```
L: if A1: condition1 generate
    -- first alternative declarations
    ...
  begin
    -- first alternative statements
    ...
  end;
elsif A2: condition2 generate
  ...
else A3: generate
  ...
end generate L;
```

We can then use the labels in block configurations for the alternatives within a configuration declaration:

```
for L(A1)
  ...
end for;

for L(A2)
  ...
end for;

for L(A3)
  ...
end for;
```

This is similar to the way in which we write a value or a range in a configuration for a for-generate statement to identify a replication of the generate statement body to configure.

EXAMPLE 5.11 *Configuring a replicated structure with boundary differences*

In Example 5.9 we showed a structure for a ripple carry adder, in which differences among bit positions were handled by alternatives of an if-generate statement. We can revise the statement to include labels in each alternative:

```
adder: for i in width-1 downto 0 generate
  signal carry_chain : unsigned(width-1 downto 1);
begin
  adder_cell: if most_significant: i = width-1 generate
    add_bit: component full_adder
      port map (a => a(i), b => b(i), s => s(i),
                c_in => carry_chain(i), c_out => c_out);
  elsif least_significant: i = 0 generate
    add_bit: component half_adder
      port map (a => a(i), b => b(i), s => s(i),
                c_out => carry_chain(i+1));
  else middle: generate
    add_bit: component full_adder
      port map (a => a(i), b => b(i), s => s(i),
                c_in => carry_chain(i),
                c_out => carry_chain(i+1));
  end generate adder_cell;
end generate adder;
```

We can now write a configuration declaration for the enclosing entity and architecture:

```
configuration widget_cfg of arith_unit is
  for ripple_adder
    for adder

      for adder_cell(most_significant)
        for add_bit: full_adder
          use entity widget_lib.full_adder(asic_cell);
      end for;

      for adder_cell(middle)
        for add_bit: full_adder
          use entity widget_lib.full_adder(asic_cell);
      end for;

      for adder_cell(least_significant)
        for add_bit: half_adder
          use entity widget_lib.half_adder(asic_cell);
      end for;
```

```
        end for; -- adder
      end for; -- ripple_adder
    end configuration widget_cfg;
```

The block configuration "**for adder** ... **end for**" configures the for-generate statement. Within it, we have three block configurations, one for each alternative of the if-generate statement. We identify each alternative with a combination of the if-generate statement label (adder_cell) and the alternative label (most_significant, least_significant, and middle, respectively). The configuration information for each alternative is only acted upon during elaboration if the corresponding condition is true and the alternative is included in the design hierarchy.

We handle configuration of alternatives in a case-generate statement in a similar way, by including a label before the choice value or values in each alternative. The form is:

```
L: case expression generate
  when A1: choice1 =>
    -- first alternative
    ...
  when A2: choice2 =>
    -- second alternative
    ...
  ...
end generate L;
```

We also write the configuration information in a similar way, including the label for the alternative in parentheses after the generate statement label.

EXAMPLE 5.12 *Configuring the alternative structures for the complex multiplier*

We can revise the case-generate statement in Example 5.10 to include alternative labels, allowing the alternatives to be configured:

```
mult_structure : case implementation generate
  when single_cycle_mult: single_cycle =>
     ...
  when multicycle_mult: multicycle =>
     ...
  when pipelined_mult: pipelined =>
     ...;
end generate mutl_structure;
```

We can now write a configuration declaration for the complex multiplier:

```
configuration wallace_tree of complex_multiplier is
  for rtl
```

```
      for mult_structure(single_cycle_mult)
        for real_mult1 : multiplier
          use entity work.multiplier(wallace_tree);
        ...
      end for;

      for mult_structure(multicycle_mult)
        for mult : multiplier
          use entity work.multiplier(wallace_tree);
        ...
      end for;

      for mult_structure(pipelined_mult)
        for mult1 : multiplier
          use entity work.multiplier(wallace_tree);
        ...
      end for;

    end for; -- rtl
  end for wallage_tree;
```

The alternative labels in an if-generate or case-generate statement allow us to configure the alternatives of the statement. If we do not need to write an explicit configuration for an alternative, we can leave the alternative unlabeled. In the examples in the first part of Section 5.3, we weren't concerned with configuration for any of the alternatives, so we omitted labels from all alternatives.

Earlier versions of VHDL did not allow for an alternative label in the single alternative of an if-generate statement and did not allow for specification of an alternative label in a corresponding block configuration. VHDL-2008 provides for backward compatibility by allowing a block configuration for an if-generate statement to omit the alternative label and surrounding parentheses. In that case, the block configuration applies to the first alternative of the if-generate statement, and the information in the block configuration is used only if the first condition in the if-generate statement is true.