4.7.5 Libraries and Packages

A VHDL library is a place where the VHDL compiler stores information about a particular design project, including intermediate files that are used in the analysis, simulation, and synthesis of the design. The location of the library within a host computer's file system is implementation dependent. For a given VHDL design, the compiler automatically creates and uses a library named "work".

A complete VHDL design usually has multiple files, each containing different design units including entities and architectures. When the VHDL compiler analyzes each file in the design, it places the results in the "work" library, and it also searches this library for needed definitions, such as other entities. Because of this feature, a large design can be broken up into multiple files, yet the compiler will find external references as needed.

Not all of the information needed in a design may be in the "work" library. For example, a designer may rely on common definitions or functional modules across a family of different projects. Each project has its own "work" library (typically a subdirectory within that project’s overall directory), but it must also refer to a common library containing the shared definitions. Even small projects may use a standard library such as the one containing IEEE standard definitions. The designer can specify the name of such a library using a library clause at the beginning of the design file. For example, we can specify the IEEE library:

```vhdl
library ieee;
```

The clause "library work;" is included implicitly at the beginning of every VHDL design file.

Specifying a library name in a design gives it access to any previously analyzed entities and architectures stored in the library, but it does not give access to type definitions and the like. This is the function of "packages" and "use clauses," described next.

A VHDL package is a file containing definitions of objects that can be used in other programs. The kind of objects that can be put into a package include signal, type, constant, function, procedure, and component declarations.

Signals that are defined in a package are "global" signals, available to any VHDL entity that uses the package. Types and constants defined in a package are known in any file that uses the package. Likewise, functions and procedures defined in a package can be called in files that use the package, and components (described in the next subsection) can be “instantiated” in architectures that use the package.

A design can “use” a package by including a use clause at the beginning of the design file. For example, to use all of the definitions in the IEEE standard 1164 package, we would write

```vhdl
use ieee.std_logic_1164.all;
```
Here, "ieee" is the name of a library which has been previously given in a library clause. Within this library, the file named "std_logic_1164" contains the desired definitions. The suffix "all" tells the compiler to use all of the definitions in this file. Instead of "all", you can write the name of a particular object to use just its definition, for example,

```vhdl
use ieee.std_logic_1164.std_ulogic
```

This clause would make available just the definition of the `std_ulogic` type in Table 4-32 on page 273, without all of the related types and functions. However, multiple "use" clauses can be written to use additional definitions.

Defining packages is not limited to standards bodies. Anyone can write a package, using the syntax shown in Table 4-40. All of the objects declared between "package" and the first "end" statement are visible in any design file that uses the package; objects following the "package body" keyword are local. In particular, notice that the first part includes "function declarations," not definitions. A function declaration lists only the function name, arguments, and type, up to but not including the "is" keyword in Table 4-35 on page 276. The complete function definition is given in the package body and is not visible to function users.

---

**IEEE VHDL STANDARDS**

VHDL has excellent capabilities for extending its data types and functions. This is important, because the language's built-in BIT and BIT_VECTOR actually are quite inadequate for modeling real circuits that also handle three-state, unknown, don't-care, and varying-strength signals.

As a result, soon after the language was formalized as IEEE standard 1076, commercial vendors began to introduce their own built-in data types to deal with logic values other than 0 and 1. Of course, each vendor had different definitions for these extended types, creating a potential "Tower of Babel."

To avoid this situation, the IEEE developed the 1164 standard logic package `std_logic_1164` with a nine-valued logic system that satisfies most designers' needs. This was later followed by standard 1076-3, discussed in Section 5.9.6, which includes several packages with standard types and operations for vectors of `STD_LOGIC` components that are interpreted as signed or unsigned integers. The packages include `std_logic_arith`, `std_logic_signed`, and `std_logic_unsigned`.

By using IEEE standards, designers can ensure a high degree of portability and interoperability among their designs. This is increasingly important, as the deployment of very large ASICs necessitates cooperation not only among multiple designers but also among multiple vendors who may each contribute different pieces of a "system-on-a-chip" design.
4.7.6 Structural Design Elements

We’re finally ready to look at the guts of a VHDL design, the “executable” portion of an architecture. Recall from Table 4-28 on page 271 that the body of an architecture is a series of concurrent statements. In VHDL, each concurrent statement executes simultaneously with the other concurrent statements in the same architecture body.

This behavior is markedly different from that of statements in conventional software programming languages, where statements execute sequentially. Concurrent statements are necessary to simulate the behavior of hardware, where connected elements affect each other continuously, not just at particular, ordered time steps. Thus, in a VHDL architecture body, if the last statement updates a signal that is used by the first statement, then the simulator will go back to that first statement and update its results according to the signal that just changed. In fact, the simulator will keep propagating changes and updating results until the simulated circuit stabilizes; we’ll discuss this in more detail in Section 4.7.9.

VHDL has several different concurrent statements, as well as a mechanism for bundling a set of sequential statements to operate as a single concurrent statement. Used in different ways, these statements give rise to three somewhat distinct styles of circuit design and description, which we cover in this and the next two subsections.

The most basic of VHDL’s concurrent statements is the component statement, whose basic syntax is shown in Table 4-41. Here, component-name is the name of a previously defined entity that is to be used, or instantiated, within the current architecture body. One instance of the named entity is created for
each component statement that invokes its name, and each instance must be
named by a unique label.

The port map keywords introduce a list that associates ports of the named
entity with signals in the current architecture. The list may be written in either of
two different styles. The first is a positional style; as in conventional program-
ning languages, the signals in the list are associated with the entity's ports in the
same order in which they appear in the entity's definition. The second is an
explicit style; each of the entity's ports is connected to a signal using the "=>
operator, and these associations may be listed in any order.

Before being instantiated in an architecture, a component must be declared
in a component declaration in the architecture's definition (see Table 4-28 on
page 271). As shown in Table 4-42, a component declaration is essentially the
same as the port-declaration part of the corresponding entity declaration—it lists
the name, mode, and type of each of its ports.

The components used in an architecture may be ones that were previously
declared as part of a design, or they may be part of a library. Table 4-43 is an
example of a VHDL entity and its architecture that uses components, a "prime-
number detector" that is structurally identical to the gate-level circuit in
Figure 4-30(c) on page 226. The entity declaration names the inputs and the
output of the circuit. The declarations section of the architecture defines all of
the signal names and the components that are used internally. The components,
INV, AND2, AND3, and OR4, are predefined in the design environment in which
this example was created and compiled (Xilinx Foundation 1.5, see References).

Note that component statements in Table 4-43 execute concurrently. Even
if the statements were listed in a different order, the same circuit would be
synthesized, and the simulated circuit operation would be the same.

A VHDL architecture that uses components is often called a structural
description or structural design, because it defines the precise interconnection
structure of signals and entities that realize the entity. In this regard, a pure
structural description is equivalent to a schematic or a net list for the circuit.

In some applications it is necessary to create multiple copies of a particular
structure within an architecture. For example, we'll see in Section 5.10.2 that an
n-bit "ripple adder" can be created by cascading n "full adders." VHDL includes
a generate statement that allows you to create such repetitive structures using a
kind of "for loop," without having to write out all of the component instantia-
tions individually.
library IEEE;
use IEEE.std_logic_1164.all;

entity prime is
  port ( N: in STD_LOGIC_VECTOR (3 downto 0);
         F: out STD_LOGIC );
end prime;

architecture prime1_arch of prime is
signal N3_L, N2_L, N1_L: STD_LOGIC;
signal N3L_NO, N3L_N2L_N1, N2L_N1_NO, N2_N1L_NO: STD_LOGIC;
component INV port ( I: in STD_LOGIC; O: out STD_LOGIC); end component;
component AND2 port (I0,I1: in STD_LOGIC; O: out STD_LOGIC); end component;
component AND3 port (I0,I1,I2: in STD_LOGIC; O: out STD_LOGIC); end component;
component OR4 port (I0,I1,I2,I3: in STD_LOGIC; O: out STD_LOGIC); end component;
begin
  U1: INV port map (N(3), N3_L);
  U2: INV port map (N(2), N2_L);
  U3: INV port map (N(1), N1_L);
  U4: AND2 port map (N3_L, N(0), N3L_NO);
  U5: AND3 port map (N3_L, N2_L, N(1), N3L_N2L_N1);
  U6: AND3 port map (N2_L, N(1), N(0), N2L_N1_NO);
  U7: AND3 port map (N(2), N1_L, N(0), N2_N1L_NO);
  U8: OR4 port map (N3L_NO, N3L_N2L_N1, N2L_N1_NO, N2_N1L_NO, F);
end prime1_arch;

The syntax of a simple iterative generate loop is shown in Table 4-44. The identifier is implicitly declared as a variable with type compatible with the range. The concurrent statement is executed once for each possible value of the identifier within the range, and identifier may be used within the concurrent statement. For example, Table 4-45 shows how an 8-bit inverter can be created.

The value of a constant must be known at the time that a VHDL program is compiled. In many applications it is useful to design and compile an entity and its architecture while leaving some of its parameters, such as bus width, unspecified. VHDL's "generic" facility lets you do this.

One or more generic constants can be defined in an entity declaration with a generic declaration before the port declaration, using the syntax shown in Table 4-46. Each of the named constants can be used within the architecture definition for the entity, and the value of the constant is deferred until the entity is instantiated using a component statement within another architecture. Within that component statement, values are assigned to the generic constants using a generic map clause in the same style as the port map clause. Table 4-47 is an example that combines generic and generate statements to define a "bus inverter" with a user-specifiable width. Multiple copies of this inverter, each with a different width, are instantiated in the program in Table 4-48.
library IEEE;
use IEEE.std_logic_1164.all;

table 4-44
Syntax of a VHDL for-generate loop.

entity inv8 is
  port ( X: in STD_LOGIC_VECTOR (1 to 8);
        Y: out STD_LOGIC_VECTOR (1 to 8) );
end inv8;

architecture inv8_arch of inv8 is
component INV port ( I: in STD_LOGIC; O: out STD_LOGIC); end component;
begin
  g1: for b in 1 to 8 generate
    U1: INV port map (X(b), Y(b));
    end generate;
end inv8_arch;

entity entity-name is
  generic ( constant-names : constant-type;
    constant-names : constant-type;
    ... constant-names : constant-type);
  port ( signal-names : mode signal-type;
    signal-names : mode signal-type;
    ... signal-names : mode signal-type);
end entity-name;

library IEEE;
use IEEE.std_logic_1164.all;

table 4-45
VHDL entity and architecture for an 8-bit inverter.

entity businv is
  generic ( WIDTH: positive);
  port ( X: in STD_LOGIC_VECTOR (WIDTH-1 downto 0);
        Y: out STD_LOGIC_VECTOR (WIDTH-1 downto 0) );
end businv;

architecture businv_arch of businv is
component INV port ( I: in STD_LOGIC; O: out STD_LOGIC); end component;
begin
  g1: for b in WIDTH-1 downto 0 generate
    U1: INV port map (X(b), Y(b));
    end generate;
end businv_arch;

Table 4-46
Syntax of a VHDL generic declaration within an entity declaration.

Table 4-47
VHDL entity and architecture for an arbitrary-width bus inverter.
library IEEE;
use IEEE.std_logic_1164.all;

entity businv_example is
  port ( IN8: in STD_LOGIC_VECTOR (7 downto 0);
        OUT8: out STD_LOGIC_VECTOR (7 downto 0);
        IN16: in STD_LOGIC_VECTOR (15 downto 0);
        OUT16: out STD_LOGIC_VECTOR (15 downto 0);
        IN32: in STD_LOGIC_VECTOR (31 downto 0);
        OUT32: out STD_LOGIC_VECTOR (31 downto 0) );
end businv_example;

architecture businv_ex_arch of businv_example is
  component businv
    generic (WIDTH: positive);
    port ( X: in STD_LOGIC_VECTOR (WIDTH-1 downto 0);
           Y: out STD_LOGIC_VECTOR (WIDTH-1 downto 0) );
  end component;
begin
  U1: businv generic map (WIDTH=>8) port map (IN8, OUT8);
  U2: businv generic map (WIDTH=>16) port map (IN16, OUT16);
  U3: businv generic map (WIDTH=>32) port map (IN32, OUT32);
end businv_ex_arch;

4.7.7 Dataflow Design Elements

If component statements were its only concurrent statements, then VHDL would be little more than a strongly typed, hierarchical net-list description language. Several additional concurrent statements allow VHDL to describe a circuit in terms of the flow of data and operations on it within the circuit. This style is called a dataflow description or dataflow design.

Two additional concurrent statements used in dataflow designs are shown in Table 4-49. The first of these is the most often used and is called a concurrent signal-assignment statement. You can read this as “signal-name gets expression.” Because of VHDL’s strong typing, the type of expression must be compatible with that of signal-name. In general, this means that either the types must be identical or expression’s type is a subtype of signal-name’s. In the case of arrays, both the element type and the length must match; however, the index range and direction need not match.

Table 4-49
Syntax of VHDL concurrent signal-assignment statements.

signal-name <= expression;
signal-name <= expression when boolean-expression else expression when boolean-expression else ...
......
expression when boolean-expression else expression;
architecture prime2_arch of prime is
signal N3L_NO, N3L_N2L_N1, N2L_N1_NO, N2_N1L_NO: STD_LOGIC;
begin
  N3L_NO <= not N(3) and N(0);
  N3L_N2L_N1 <= not N(3) and not N(2) and N(1);
  N2L_N1_NO <= not N(2) and N(1) and N(0);
  N2_N1L_NO <= N(2) and not N(1) and N(0);
  F <= N3L_NO or N3L_N2L_N1 or N2L_N1_NO or N2_N1L_NO;
end prime2_arch;

Table 4-50
Dataflow VHDL architecture for the prime-number detector.

<table>
<thead>
<tr>
<th>Conditional signal-assignment statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>when</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>relational operators</td>
</tr>
<tr>
<td>=, /=, &gt;, &gt; =, &lt;, &lt;=</td>
</tr>
</tbody>
</table>

Table 4-51
Prime-number-detector architecture using conditional assignments.

architecture prime3_arch of prime is
signal N3L_NO, N3L_N2L_N1, N2L_N1_NO, N2_N1L_NO: STD_LOGIC;
begin
  N3L_NO <= '1' when N(3)'0' and N(0)'1' else '0';
  N3L_N2L_N1 <= '1' when N(3)'0' and N(2)'0' and N(1)'1' else '0';
  N2L_N1_NO <= '1' when N(2)'0' and N(1)'1' and N(0)'1' else '0';
  N2_N1L_NO <= '1' when N(2)'1' and N(1)'0' and N(0)'1' else '0';
  F <= N3L_NO or N3L_N2L_N1 or N2L_N1_NO or N2_N1L_NO;
end prime3_arch;
Table 4-52

Syntax of VHDL selected signal-assignment statement.

```
with expression select
  signal-name <= signal-value when choices,
  signal-value when choices,
  ...
  signal-value when choices;
```

may be a single value of *expression* or a list of values separated by vertical bars (|). The *choices* for the entire statement must be mutually exclusive and all inclusive. The keyword *others* can be used in the last when clause to denote all values of *expression* that have not yet been covered.

Table 4-53 is an architecture for the prime-number detector that uses a selected signal-assignment statement. All of the *choices* for which F is '1' could have been written in a single when clause, but multiple clauses are shown just for instructional purposes. In this example, the selected signal-assignment statement reads somewhat like a listing of the on-set of the function F.

We can modify the previous architecture slightly to take advantage of the numeric interpretation of N in the function definition. Using the `CONV_INTEGER` function that we defined previously, Table 4-54 writes the *choices* in terms of integers, which we can readily see are prime as required. We can think of this version of the architecture as a “behavioral” description, because it describes the desired function in such a way that its behavior is quite evident.

Table 4-53

Prime-number detector architecture using selected signal assignment.

```
architecture prime4_arch of prime is
begin
  with N select
    F <= '1' when "0001",
         '1' when "0010",
         '1' when "0011" | "0101" | "0111",
         '1' when "1011" | "1101",
         '0' when others;
end prime4_arch;
```

**COVERING ALL THE CASES**

Conditional and selected signal assignments require all possible conditions to be covered. In a conditional signal assignment, the final “*else expression*” covers missing conditions. In a selected signal assignment, “*others*” can be used in the final when clause to pick up the remaining conditions.

In Table 4-53, you might think that instead of writing “*others*” in the final when clause, we could have written the nine remaining 4-bit combinations, "0000", "0100", and so on. But that’s not true! Remember that STD_LOGIC is a nine-valued system, so a 4-bit STD_LOGIC_VECTOR actually has $9^4$ possible values. So “*others*” in this example is really covering 6,554 cases!
architecture prime5_arch of prime is
begin
  with CONV_INTEGER(N) select
  F <= '1' when 1 | 2 | 3 | 5 | 7 | 11 | 13,
       '0' when others;
end prime5_arch;

4.7.8 Behavioral Design Elements

As we saw in the last example, it is sometimes possible to directly describe a desired logic-circuit behavior using a concurrent statement. This is a good thing, as the ability to create a behavioral design or behavioral description is one of the key benefits of hardware-description languages in general and VHDL in particular. However, for most behavioral descriptions, we need to employ some additional language elements described in this subsection.

VHDL's key behavioral element is the "process." A process is a collection of "sequential" statements (described shortly) that executes in parallel with other concurrent statements and other processes. Using a process, you can specify a complex interaction of signals and events in a way that executes in essentially zero simulated time during simulation and that gives rise to a synthesized combinational or sequential circuit that performs the modeled operation directly.

A VHDL process statement can be used anywhere that a concurrent statement can be used. A process statement is introduced by the keyword process and has the syntax shown in Table 4-55. Since a process statement is written within the scope of an enclosing architecture, it has visibility of the types, signals, constants, functions, and procedures that are declared or are otherwise visible in the enclosing architecture. However, you can also define types, variables, constants, functions, and procedures that are local to the process.

Note that a process may not declare signals, only "variables." A VHDL variable keeps track of the state within a process and is not visible outside of the process. Depending on its use, it may or may not give rise to a corresponding signal in a physical realization of the modeled circuit. The syntax for defining a

<table>
<thead>
<tr>
<th>process (signal-name, signal-name, ..., signal-name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>type declarations</td>
</tr>
<tr>
<td>variable declarations</td>
</tr>
<tr>
<td>constant declarations</td>
</tr>
<tr>
<td>function definitions</td>
</tr>
<tr>
<td>procedure definitions</td>
</tr>
<tr>
<td>begin</td>
</tr>
<tr>
<td>sequential-statement</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>sequential-statement</td>
</tr>
<tr>
<td>end process;</td>
</tr>
</tbody>
</table>

Table 4-54
A more behavioral description of the prime-number detector.

Table 4-55
Syntax of a VHDL process statement.
variable within a process is similar to the syntax for a signal declaration within an architecture, except that the keyword `variable` is used:

```vhdl
variable variable-names : variable-type;
```

A VHDL process is always either `running` or `suspended`. The list of signals in the process definition, called the `sensitivity list`, determines when the process runs. A process initially is suspended; when any signal in its sensitivity list changes value, the process resumes execution, starting with its first sequential statement and continuing until the end. If any signal in the sensitivity list changes value as a result of running the process, it runs again. This continues until the process runs without any of these signals changing value. In simulation, all of this happens in zero simulated time.

Upon resumption, a properly written process will suspend after one or a few runs. However, it is possible to write an incorrect process that never suspends. For example, consider a process with just one sequential statement, “`X <= not X`” and a sensitivity list of “`(X)`”. Since `X` changes on every pass, the process will run forever in zero simulated time—not very useful! In practice, simulators have safeguards that normally can detect such unwanted behavior, terminating the misbehaving process after a thousand or so passes.

The sensitivity list is optional; a process without a sensitivity list starts running at time zero in simulation. One application of such a process is to generate input waveforms in a test bench, as in Table 4-65 on page 296.

VHDL has several kinds of sequential statements. The first is a `sequential signal-assignment statement`; this has the same syntax as the concurrent version (`signal-name <= expression;`), but it occurs within the body of a process rather than an architecture. An analogous statement for variables is the `variable-assignment statement`, which has the syntax “`variable-name := expression;`”. Notice that a different assignment operator, `:=`, is used for variables.

For instruction purposes, the dataflow architecture of the prime-number detector in Table 4-50 is rewritten as a process in Table 4-56. Notice that we’re still working off the same original entity declaration of `prime` that appeared in

<table>
<thead>
<tr>
<th>Table 4-56</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process-based dataflow VHDL architecture for the prime-number detector.</td>
</tr>
</tbody>
</table>

```vhdl
architecture prime6_arch of prime is
begin
  process(N)
    variable N3L_N0, N3L_N2L_N1, N2L_N1_N0, N2_N1L_N0: STD_LOGIC;
  begin
    N3L_N0 := not N(3) and N(0);
    N3L_N2L_N1 := not N(3) and not N(2) and N(1);
    N2L_N1_N0 := not N(2) and N(1) and N(0);
    N2_N1L_N0 := N(2) and not N(1) and N(0);
    F <= N3L_N0 or N3L_N2L_N1 or N2L_N1_N0 or N2_N1L_N0;
  end process;
end prime6_arch;
```
WEIRD BEHAVIOR

Remember that the statements within a process are executed sequentially. Suppose that for some reason we wrote the last statement in Table 4-56 (the signal assignment to F) as the first. Then we would see rather weird behavior from this process.

The first time the process was run, the simulator would complain that the values of the variables were being read before any value was assigned to them. On subsequent resumptions, a value would be assigned to F based on the previous values of the variables, which are remembered while the process is suspended. New values would then be assigned to the variables and remembered until the next resumption. So the circuit's output value would always be one input-change behind.

Table 4-43. Within the new architecture (prime6_arch), we have just one concurrent statement, which is a process. The process sensitivity list contains just N, the primary inputs of the desired combinational logic function. The AND-gate outputs must be defined as variables rather than signals, since signal definitions are not allowed within a process. Otherwise, the body of the process is very similar to that of the original architecture. In fact, a typical synthesis tool would probably create the same circuit from either description.

Other sequential statements, beyond simple assignment, can give us more creative control in expressing circuit behavior. The if statement, with the syntax shown in Table 4-57, is probably the most familiar of these. In the first and simplest form of the statement, a boolean-expression is tested, and a sequential-statement is executed if the expression's value is true. In the second form,
architecture prime7_arch of prime is
begin
  process(N)
    variable NI: INTEGER;
  begin
    NI := CONV_INTEGER(N);
    if NI=1 or NI=2 then F <= '1';
    elsif NI=3 or NI=5 or NI=7 or NI=11 or NI=13 then F <= '1';
    else F <= '0';
    end if;
  end process;
end prime7_arch;

we've added an "else" clause with another sequential-statement that's executed if the expression's value is false.

To create nested if-then-else statements, VHDL uses a special keyword elsif, which introduces the "middle" clauses. An elsif clause's sequential-statement is executed if its boolean-expression is true and all of the preceding boolean-expressions were false. The optional final else-clause's sequential-statement is executed if all of the preceding boolean-expressions were false.

Table 4-58 is a version of the prime-number-detector architecture that uses an if statement. A local variable NI is used to hold a converted, integer version of the input N, so that the comparisons in the if statement can be written using integer values.

The boolean expressions in Table 4-58 are nonoverlapping; that is, only one of them is true at a time. For this application we really didn't need the full power of nested if statements. In fact, a synthesis engine might create a circuit that evaluates the boolean expressions in series, with slower operation than might otherwise be possible. When we need to select among multiple alternatives based on the value of just one signal or expression, a case statement is usually more readable and may yield a better synthesized circuit.

Table 4-59 shows the syntax of a case statement. This statement evaluates the given expression, finds a matching value in one of the choices, and executes the corresponding sequential-statements. Note that one or more sequential statements can be written for each set of choices. The choices may take the form of a single value or of multiple values separated by vertical bars (|). The choices must be mutually exclusive and include all possible values of expression's type; the keyword others can be used as the last choices to denote all values that have not yet been covered.

<table>
<thead>
<tr>
<th>Table 4-59</th>
<th>Syntax of a VHDL case statement.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>case expression is</td>
</tr>
<tr>
<td></td>
<td>when choices =&gt; sequential-statements</td>
</tr>
<tr>
<td></td>
<td>when choices =&gt; sequential-statements</td>
</tr>
<tr>
<td></td>
<td>end case;</td>
</tr>
</tbody>
</table>
architecture prime8_arch of prime is
begin
    process(N)
    begin
        case CONV_INTEGER(N) is
            when 1 => F <= '1';
            when 2 => F <= '1';
            when 3 | 5 | 7 | 11 | 13 => F <= '1';
            when others => F <= '0';
        end case;
    end process;
end prime8_arch;

Table 4-60 is yet another architecture for the prime-number detector, this time coded with a case statement. Like the concurrent version, the select statement in Table 4-54 on page 289, the case statement makes it very easy to see the desired functional behavior.

Another important class of sequential statements are the loop statements. The simplest of these has the syntax shown in Table 4-61 and creates an infinite loop. Although infinite loops are undesirable in conventional software programming languages, we’ll show in Section 7.12 how such a loop can be very useful in hardware modeling.

loop
    sequential-statement
    ...
    sequential-statement
end loop;

A more familiar loop, one that we’ve seen before, is the for loop, with the syntax shown in Table 4-62. Note that the loop variable, identifier, is declared implicitly by its appearance in the for loop and has the same type as range. This variable may be used within the loop’s sequential statements, and it steps through all of the values in range, from left to right, one per iteration.

Two more useful sequential statements that can be executed within a loop are “exit” and “next”. When executed, exit transfers control to the statement immediately following the loop end. On the other hand, next causes any remaining statements in the loop to be bypassed and begins the next iteration of the loop.

for identifier in range loop
    sequential-statement
    ...
    sequential-statement
end loop;

Table 4-62 Syntax of a VHDL for loop.
library IEEE;
use IEEE.std_logic_1164.all;

entity prime9 is
    port ( N: in STD_LOGIC_VECTOR (15 downto 0);
           F: out STD_LOGIC);
end prime9;

architecture prime9_arch of prime9 is
begin
    process(N)
        variable NI: INTEGER;
        variable prime: boolean;
    begin
        NI := CONV_INTEGER(N);
        prime := true;
        if NI=1 or NI=2 then null; -- take care of boundary cases
        else for i in 2 to 253 loop
                if NI mod i = 0 then
                    prime := false; exit;
                end if;
            end loop;
        end if;
        if prime then F <= '1'; else F <= '0'; end if;
    end process;
end prime9_arch;

Our good old prime-number detector is coded one more time in Table 4-63, this time using a for loop. The striking thing about this example is that it is truly a behavioral description—we have actually used VHDL to compute whether the input N is a prime number. We’ve also increased the size of N to 16 bits, just to emphasize the fact that we were able to create a compact model for the circuit without having to explicitly list hundreds of primes.

BAD DESIGN
Table 4-63 has a good example of a for loop, but is a bad example of how to design a circuit. Although VHDL is a powerful programming language, design descriptions that use its full power may be inefficient or unsynthesizable.

The culprit in Table 4-63 is the mod operator. This operation requires an integer division, and most VHDL tools are unable to synthesize division circuits except for special cases, such as division by a power of two (realized as a shift).

Even if the tools could synthesize a divider, we wouldn’t want to specify a prime number detector in this way. The description in Table 4-63 implies a combinational circuit, and the tools would have to create 252 combinational dividers, one for each value of i, to “unroll” the for loop and realize the circuit!
while boolean-expression loop
    sequential-statement
    ... sequential-statement
end loop;

Table 4-64
Syntax of a VHDL while loop.

The last kind of loop statement is the while loop, with the syntax shown in Table 4-64. In this form, boolean-expression is tested before each iteration of the loop, and the loop is executed only if the value of the expression is true.

We can use processes to write behavioral descriptions of both combinational and sequential circuits. Many more examples of combinational-circuit descriptions appear in the VHDL subsections of Chapter 5. A few additional language features are needed to describe sequential circuits; these are described in Section 7.12, and sequential examples appear in the VHDL subsections of Chapter 8.

4.7.9 The Time Dimension and Simulation

None of the examples that we’ve dealt with so far models the time dimension of circuit operation—everything happens in zero simulated time. However, VHDL has excellent facilities for modeling the time, and it is indeed another significant dimension of the language. In this book we won’t go into detail on this subject, but we’ll introduce just a few ideas here.

VHDL allows you to specify a time delay using the keyword after in any signal-assignment statement, including sequential, concurrent, conditional, and selected assignments. For example, in the inhibit-gate architecture of Table 4-26 on page 269 you could write

\[ Z <= '1' \] after 4 ns when \( X='1' \) and \( Y='0' \) else '0' after 3 ns;

This allows you to model an inhibit gate that has 4 ns of delay on a 0-to-1 output transition and only 3 ns on a 1-to-0 transition. In typical ASIC design environments, the VHDL models for all of the low-level components in the component library include such delay parameters. Using these estimates, a VHDL simulator can predict the approximate timing behavior of a larger circuit that uses these components.

Another way to invoke the time dimension is with wait, a sequential statement. This statement can be used to suspend a process for a specified time period. Table 4-65 is an example program that uses wait to create simulated input waveforms to test the operation of the inhibit gate for four different input combinations at 10-ns time steps.

Once you have a VHDL program whose syntax and semantics are correct, you can use a VHDL simulator to observe its operation. Although we won’t go into great detail, it’s useful to have a basic understanding of how such a simulator works.