CHAPTER 1

Introduction to VHDL

The VHSIC Hardware Description Language is an industry standard language used to describe hardware from the abstract to the concrete level. VHDL resulted from work done in the ’70s and early ’80s by the U.S. Department of Defense. Its roots are in the ADA language, as will be seen by the overall structure of VHDL as well as other VHDL statements.

VHDL usage has risen rapidly since its inception and is used by literally tens of thousands of engineers around the globe to create sophisticated electronic products. This chapter will start the process of easing the reader into the complexities of VHDL. VHDL is a powerful language with numerous language constructs that are capable of describing very complex behavior. Learning all the features of VHDL is not a simple task. Complex features will be introduced in a simple form and then more complex usage will be described.

In 1986, VHDL was proposed as an IEEE standard. It went through a number of revisions and changes until it was adopted as the IEEE 1076 standard in December 1987. The IEEE 1076-1987 standard VHDL is the VHDL used in this book. (Appendix D contains a brief description of VHDL 1076-1993.) All the examples have been described in IEEE 1076 VHDL, and compiled and simulated with the VHDL simulation environment from Model Technology Inc. The synthesis examples were synthesized with the Exemplar Logic Inc. synthesis tools.

VHDL Terms

Before we go any further, let's define some of the terms that we use throughout the book. These are the basic VHDL building blocks that are used in almost every description, along with some terms that are redefined in VHDL to mean something different to the average designer.

- *Entity.* All designs are expressed in terms of entities. An entity is the most basic building block in a design. The uppermost level of the design is the top-level entity. If the design is hierarchical, then the top-level description will have lower-level descriptions contained in it. These lower-level descriptions will be lower-level entities contained in the top-level entity description.
- *Architecture.* All entities that can be simulated have an architecture description. The architecture describes the behavior of the entity. A single entity can have multiple architectures. One architecture might be behavioral while another might be a structural description of the design.
- *Configuration.* A configuration statement is used to bind a component instance to an entity-architecture pair. A configuration can be considered like a parts list for a design. It describes which behavior to use for each entity, much like a parts list describes which part to use for each part in the design.
- *Package.* A package is a collection of commonly used data types and subprograms used in a design. Think of a package as a toolbox that contains tools used to build designs.
- *Driver.* This is a source on a signal. If a signal is driven by two sources, then when both sources are active, the signal will have two drivers.

- *Bus.* The term "bus" usually brings to mind a group of signals or a particular method of communication used in the design of hardware. In VHDL, a bus is a special kind of signal that may have its drivers turned off.
- *Attribute.* An attribute is data that are attached to VHDL objects or predefined data about VHDL objects. Examples are the current drive capability of a buffer or the maximum operating temperature of the device.
- *Generic.* A generic is VHDL's term for a parameter that passes information to an entity. For instance, if an entity is a gate level model with a rise and a fall delay, values for the rise and fall delays could be passed into the entity with generics.
- *Process.* A process is the basic unit of execution in VHDL. All operations that are performed in a simulation of a VHDL description are broken into single or multiple processes.

Describing Hardware in VHDL

VHDL Descriptions consist of primary design units and secondary design units. The primary design units are the Entity and the Package. The secondary design units are the Architecture and the Package Body. Secondary design units are always related to a primary design unit. Libraries are collections of primary and secondary design units. A typical design usually contains one or more libraries of design units.

Entity

A VHDL entity specifies the name of the entity, the ports of the entity, and entity-related information. All designs are created using one or more entities.

Let's take a look at a simple entity example:

```
ENTITY mux   IS
  PORT ( a, b, c, d : IN BIT;
         s0, s1 : IN BIT;
         x,  : OUT BIT);
END mux;
```

The keyword **ENTITY** signifies that this is the start of an entity statement. In the descriptions shown throughout the book, keywords of the language and types provided with the STANDARD package are shown in ALL CAPITAL letters. For instance, in the preceding example, the keywords are **ENTITY**, **IS**, **PORT**, **IN**, **INOUT**, and so on. The standard type provided is **BIT**. Names of user-created objects such as **mux**, in the example above, will be shown in lower case.

The name of the entity is **mux**. The entity has seven ports in the **PORT** clause. Six ports are of mode **IN** and one port is of mode **OUT**. The four data input ports (**a**, **b**, **c**, **d**) are of type **BIT**. The two multiplexer select inputs, **s0** and **s1**, are also of type **BIT**. The output port is of type **BIT**.

The entity describes the interface to the outside world. It specifies the number of ports, the direction of the ports, and the type of the ports. A lot more information can be put into the entity than is shown here, but this gives us a foundation upon which we can build more complex examples.

Architectures

The entity describes the interface to the VHDL model. The architecture describes the underlying functionality of the entity and contains the statements that model the behavior of the entity. An architecture is always related to an entity and describes the behavior of that entity. An architecture for the counter device described earlier would look like this:

```
ARCHITECTURE dataflow OF mux IS
  SIGNAL select : INTEGER;
BEGIN
  select <= 0 WHEN s0 = '0' AND s1 = '0' ELSE
            1 WHEN s0 = '1' AND s1 = '0' ELSE
            2 WHEN s0 = '0' AND s1 = '1' ELSE
            3;

  x <= a AFTER 0.5 NS WHEN select = 0 ELSE
       b AFTER 0.5 NS WHEN select = 1 ELSE
       c AFTER 0.5 NS WHEN select = 2 ELSE
       d AFTER 0.5 NS;

END dataflow;
```

The keyword **ARCHITECTURE** signifies that this statement describes an architecture for an entity. The architecture name is **dataflow**. The entity the architecture is describing is called **mux**.

The reason for the connection between the architecture and the entity is that an entity can have multiple architectures describing the behavior of the entity. For instance, one architecture could be a behavioral description, and another could be a structural description.

The textual area between the keyword **ARCHITECTURE** and the keyword **BEGIN** is where local signals and components are declared for later use. In this example signal select is declared to be a local signal.

The statement area of the architecture starts with the keyword **BEGIN**. All statements between the **BEGIN** and the **END** netlist statement are called concurrent statements, because all the statements execute concurrently.

Concurrent Signal Assignment

In a typical programming language such as C or C++, each assignment statement executes one after the other and in a specified order. The order of execution is determined by the order of the statements in the source file. Inside a VHDL architecture, there is no specified ordering of the assignment statements. The order of execution is solely specified by events occurring on signals that the assignment statements are sensitive to.

Examine the first assignment statement from architecture **behave**, as shown here:

```
select <= 0 WHEN s0 = '0' AND s1 = '0' ELSE
           1 WHEN s0 = '1' AND s1 = '0' ELSE
           2 WHEN s0 = '0' AND s1 = '1' ELSE
           3;
```

A signal assignment is identified by the symbol **<=**. Signal **select** will get a numeric value assigned to it based on the values of **s0** and **s1**. This statement is executed whenever either signal **s0** or signal **s1** has an event occur on it. An event on a signal is a change in the value of that signal. A signal assignment statement is said to be sensitive to changes on any signals that are to the right of the **<=** symbol. This signal assignment statement is sensitive to **s0** and **s1**. The other signal assignment statement in architecture **dataflow** is sensitive to signal select.

Let's take a look at how these statements actually work. Suppose that we have a steady-state condition where both **s0** and **s1** have a value of 0, and signals **a, b, c,** and **d** currently have a value of 0. Signal **x** will have a 0 value because it is assigned the value of signal **a** whenever signals **s0** and **s1** are both 0. Now assume that we cause an event on signal **a** that changes its value to 1. When this happens, the first signal assignment

statement will not execute because this statement is not sensitive to changes to signal **a**. This happens because signal **a** is not on the right side of the operator. The second signal assignment statement will execute because it is sensitive to events on signal **a**. When the second signal assignment statement executes the new value of **a** will be assigned to signal **x**. Output port **x** will now change to 1.

Let's now look at the case where signal **s0** changes value. Assume that **s0** and **s1** are both 0, and ports **a**, **b**, **c**, and **d** have the values 0, 1, 0, and 1, respectively. Now let's change the value of port **s0** from 0 to 1. The first signal assignment statement is sensitive to signal **s0** and will therefore execute. When concurrent statements execute, the expression value calculation will use the current values for all signals contained in it.

When the first statement executes, it computes the new value to be assigned to **q** from the current value of the signal expression on the right side of the **<=** symbol. The expression value calculation uses the current values for all signals contained in it.

With the value of **s0** equal to 1 and **s1** equal to 0, signal **select** will receive a new value of 1. This new value of signal **select** will cause an event to occur on signal **select**, causing the second signal assignment statement to execute. This statement will use the new value of signal select to assign the value of port **b** to port **x.** The new assignment will cause port **x** to change from a 0 to a 1.

Event Scheduling

The assignment to signal **x** does not happen instantly. Each of the values assigned to signal **x** contain an AFTER clause. The mechanism for delaying the new value is called scheduling an event. By assigning port **x** a new value, an event was scheduled 0.5 nanoseconds in the future that contains the new value for signal **x.** When the event matures (0.5 nanoseconds in the future), signal **x** receives the new value.

Statement Concurrency

The first assignment is the only statement to execute when events occur on ports **s0** or **s1**. The second signal assignment statement does not execute unless an event on signal **select** occurs or an event occurs on ports **a**, **b**, **c**, **d**.

The two signal assignment statements in architecture **behave** form a behavioral model, or architecture, for the **mux** entity. The **dataflow** architecture contains no structure. There are no components instantiated in the architecture. There is no further hierarchy, and this architecture can be considered a leaf node in the hierarchy of the design.

Structural Designs

Another way to write the **mux** design is to instantiate subcomponents that perform smaller operations of the complete model. With a model as simple as the 4-input multiplexer that we have been using, a simple gate level description can be generated to show how components are described and instantiated. The architecture shown below is a structural description of the **mux** entity.

```
ARCHITECTURE netlist OF mux IS
  COMPONENT andgate
    PORT(a, b, c : IN bit; c : OUT BIT);
  END COMPONENT;
  COMPONENT inverter
    PORT(in1 : IN BIT; x : OUT BIT);
  END COMPONENT;
  COMPONENT orgate
    PORT(a, b, c, d : IN bit; x : OUT BIT);
  END COMPONENT;

SIGNAL s0_inv, s1_inv, x1, x2, x3, x4 : BIT;

BEGIN
  U1 : inverter(s0, s0_inv);
  U2 : inverter(s1, s1_inv);
  U3 : andgate(a, s0_inv, s1_inv, x1);
  U4 : andgate(b, s0, s1_inv, x2);
  U5 : andgate(c, s0_inv, s1, x3);
  U6 : andgate(d, s0, s1, x4);
  U7 : orgate(x2 => b, x1 => a, x4 => d, x3 => c, x => x);
END netlist;
```

This description uses a number of lower-level components to model the behavior of the **mux** device. There is an inverter component, an **andgate** component and an **orgate** component. Each of these components is declared in the architecture declaration section, which is between the architecture statement and the **BEGIN** keyword.

A number of local signals are used to connect each of the components to form the architecture description. These local signals are declared using the SIGNAL declaration.

The architecture statement area is located after the **BEGIN** keyword. In this example are a number of component instantiation statements. These statements are labeled **U1-U7**. Statement **U1** is a component instantiation statement that instantiates the inverter component. This statement connects port **s0** to the first port of the inverter component and signal **s0_inv** to the second port of the inverter component. The effect is that port **in1** of the inverter is connected to port **s0** of the **mux** entity, and port **x** of the inverter is connected to local signal **s0_inv**. In this statement the ports are connected by the order they appear in the statement.

Notice component instantiation statement **U7**. This statement uses the following notation:

```
U7 : orgate(x2 => b, x1 => a, x4 => d, x3 => c, x => x);
```

This statement uses named association to match the ports and signals to each other. For instance port **x2** of the **orgate** is connected to port **b** of the entity with the first association clause. The last instantiation clause connects port **x** of the **orgate** component to port **x** of the entity. The order of the clauses is not important. Named and ordered association can be mixed, but it is not recommended.

Sequential Behavior

There is yet another way to describe the functionality of a **mux** device in VHDL. The fact that VHDL has so many possible representations for similar functionality is what makes learning the entire language a big task. The third way to describe the functionality of the **mux** is to use a process statement to describe the functionality in an algorithmic representation. This is shown in architecture sequential, as shown in the following:

```
ARCHITECTURE sequential OF mux IS
    (a, b, c, d, s0, s1 )
     VARIABLE sel : INTEGER;
  BEGIN
     IF s0 = '0' and s1 = '0' THEN
       sel := 0;
     ELSIF s0 = '1' and s1 = '0' THEN
       sel := 1;
     ELSIF s0 = '0' and s1 = '0' THEN
       sel := 2;
     ELSE
       sel := 3;
     END IF;
     CASE sel IS
```

```
        WHEN 0 =>
          x <= a;
        WHEN 1 =>
          x <= b;
        WHEN 2 =>
          x <= c;
        WHEN OTHERS =>
          x <= d;
      END CASE;
  END PROCESS;
  END sequential;
```

The architecture contains only one statement, called a process statement. It starts at the line beginning with the keyword **PROCESS** and ends with the line that contains **END PROCESS**. All the statements between these two lines are considered part of the process statement.

Process Statements

The process statement consists of a number of parts. The first part is called the sensitivity list; the second part is called the process declarative part; and the third is the statement part. In the preceding example, the list of signals in parentheses after the keyword **PROCESS** is called the sensitivity list. This list enumerates exactly which signals cause the process statement to be executed. In this example, the list consists of **a**, **b**, **c**, **d**, **s0**, and **s1**. Only events on these signals cause the process statement to be executed.

Process Declarative Region

The process declarative part consists of the area between the end of the sensitivity list and the keyword **BEGIN**. In this example, the declarative part contains a variable declaration that declares local variable **sel**. This variable is used locally to contain the value computed based on ports **s0** and **s1**.

Process Statement Part

The statement part of the process starts at the keyword **BEGIN** and ends at the **END PROCESS** line. All the statements enclosed by the process are

sequential statements. This means that any statements enclosed by the process are executed one after the other in a sequential order just like a typical programming language. Remember that the order of the statements in the architecture did not make any difference; however, this is not true inside the process. The order of execution is the order of the statements in the process statement.

Process Execution

Let's see how this works by walking through the execution of the example in architecture `sequential`, line by line. To be consistent, let's assume that `s0` changes to 0. Because `s0` is in the sensitivity list for the process statement, the process is invoked. Each statement in the process is then executed sequentially. In this example the `IF` statement is executed first followed by the `CASE` statment. Each check that the `IF` statement performs is done sequentially starting with the first in the model.

The first check is to see if `s0` is equal to a 0. This statement fails because `s0` is equal to a 1 and `s1t` is equal to a 0. The signal assignment statement that follows the first check will not be executed. Instead, the next check is performed. This check succeeds and the signal assignment statements following the check for `s0 = 1` and `s1 = 0` are executed. This statement is shown below.

```
sel := 1;
```

Sequential Statements

This statement will execute sequentially. Once it is executed, the next check of the `IF` statement is not performed. Whenever a check succeeds, no other checks are done. The `IF` statement has completed and now the `CASE` statement will execute. The `CASE` statement will evaluate the value of `sel` computed earlier by the `IF` statement and then execute the appropriate statement that matches the value of `sel`. In this example the value of `sel` is 1 therefore the following statement will be executed:

```
x <= b;
```

The value of port `b` will be assigned to port `x` and process execution will terminate because there are no more statements in the architecture.

Architecture Selection

So far, three architectures have been described for one entity. Which architecture should be used to model the **mux** device? It depends on the accuracy wanted and if structural information is required. If the model is going to be used to drive a layout tool, then the structural architecture netlist is probably most appropriate. If a structural model is not wanted for some other reason, then a more efficient model can be used. Either of the other two methods (architectures **dataflow** and **sequential**) are probably more efficient in memory space required and speed of execution. How to choose between these two methods may come down to a question of programming style. Would the modeler rather write concurrent or sequential VHDL code? If the modeler wants to write concurrent VHDL code, then the style of architecture **dataflow** is the way to go; otherwise, architecture **sequential** should be chosen. Typically, modelers are more familiar with sequential coding styles, but concurrent statements are very powerful tools for writing small efficient models.

We will also look at yet another architecture that can be written for an entity. This is the architecture that can be used to drive a synthesis tool. Synthesis tools convert a *Register Transfer Level* (RTL) VHDL description into an optimized gate-level description. Synthesis tools can offer greatly enhanced productivity compared to manual methods. The synthesis process is discussed in Chapters 9, “Synthesis” and 10, “VHDL Synthesis.”

Configuration Statements

An entity can have more than one architecture, but how does the modeler choose which architecture to use in a given simulation? The configuration statement maps component instantiations to entities. With this powerful statement, the modeler can pick and choose which architectures are used to model an entity at every level in the design.

Let’s look at a configuration statement using the netlist architecture of the **rsff** entity. The following is an example configuration:

```
CONFIGURATION muxcon1 OF mux IS
  FOR netlist
    FOR U1,U2 :
       inverter USE ENTITY WORK.myinv(version1);
    END FOR;
    FOR U3,U4,U5,U6 : andgate USE ENTITY WORK.myand(ver-
     sion1);
    END FOR;
```

```
    FOR U7 : orgate USE ENTITY WORK.myor(version1);
    END FOR;
  END FOR;
END muxcon1;
```

The function of the configuration statement is to spell out exactly which architecture to use for every component instance in the model. This occurs in a hierarchical fashion. The highest-level entity in the design needs to have the architecture to use specified, as well as any components instantiated in the design.

The preceding configuration statement reads as follows: This is a configuration named **muxcon1** for entity **mux**. Use architecture **netlist** as the architecture for the topmost entity, which is **mux**. For the two component instances **U1** and **U2** of type **inverter** instantiated in the **netlist** architecture, use entity **myinv**, architecture **version1** from the library called **WORK**. For the component instances **U3-U6** of type **andgate**, use entity **myand**, architecture **version1** from library **WORK**. For component instance **U7** of type **orgate** use entity **myor**, architecture **version1** from library **WORK.** All of the entities now have architectures specified for them. Entity **mux** has architecture **netlist**, and the other components have architectures named **version1** specified.

Power of Configurations

By compiling the entities, architectures, and the configuration specified earlier, you can create a simulatable model. But what if you did not want to simulate at the gate level? What if you really wanted to use architecture **BEHAVE** instead? The power of the configuration is that you do not need to recompile your complete design; you only need to recompile the new configuration. Following is an example configuration:

```
CONFIGURATION muxcon2 OF mux IS
  FOR dataflow
  END FOR;
  END muxcon2;
```

This is a configuration named **muxcon2** for entity **mux**. Use architecture **dataflow** for the topmost entity, which is **mux**. By compiling this configuration, the architecture **dataflow** is selected for entity **mux** in this simulation.

This configuration is not necessary in standard VHDL, but gives the designer the freedom to specify exactly which architecture will be used for the entity. The default architecture used for the entity is the last one compiled into the working library.

SUMMARY

In this chapter, we have had a basic introduction to VHDL and how it can be used to model the behavior of devices and designs. The first example showed how a simple dataflow model in VDHL is specified. The second example showed how a larger design can be made of smaller designs —in this case a 4-input multiplexer was modeled using **AND**, **OR** and **INVERTER** gates. The first example provided a structural view of VHDL.

The last example showed an algorithmic or behavioral view of the **mux**. All these views of the **mux** successfully model the functionality of a **mux** and all can be simulated with a VHDL simulator. Ultimately, however, a designer will want to use the model to facilitate building a piece of hardware. The most common use of VHDL in actually building hardware today is through synthesis tools. Therefore, the focus of the rest of the book is not only on the simulation of VHDL but also on the synthesis of VHDL.