CHAPTER 9

Hardware Description Languages

9.1 INTRODUCTION[1]

Earlier chapters drew circuits that were designed. For example, Chapter 2 designed an automatic door opener circuit and drew the circuit shown in Figure 9.1. A drawing has more information than is necessary to describe the circuit, such as information about the location of the inputs and outputs. In Figure 9.1, the inputs are located on the left and the output is located on the right. The `c` input is located on the top, the `h` input in the middle, and the `p` input on the bottom. The drawing also gives information about the size and location of the components in the circuit: the inverter is at the top, the OR gate below the inverter, the AND gate on the right, and each component is about a half inch by a half inch. The drawing gives information about the wires too: the wire from the inverter goes to the right, then down, then to the right again, for example. However, all that information about the drawing is really irrelevant, and has nothing to do with how the design will be physically implemented. When a circuit is drawn, all that information must be defined, even if arbitrarily. A drawing of a circuit is commonly referred to as a circuit ***schematic***.

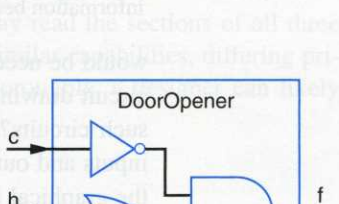


Figure 9.1 Drawn circuit

A problem with drawing circuits arises with larger circuits. Does the schematic in Figure 9.2 mean anything to you? That schematic has just a couple dozen components—what if there were a couple thousand components, as is quite common? Drawing a large circuit would require tremendous effort to figure out how to place each component in the drawing, and how to route wires among the components. And if a tool generated the circuit, the tool would have to spend compute time to figure out a visually appealing way to draw the circuit (rather than a spaghetti-like mess), and such computation is time-consuming and still may not result in a readable drawing. Furthermore, the files used to store schematics would be very large, as those files would contain all that extra information about the location and size of every component. All that extra effort, file size, and time,

1 Substantial content of this chapter was contributed by Roman Lysecky.

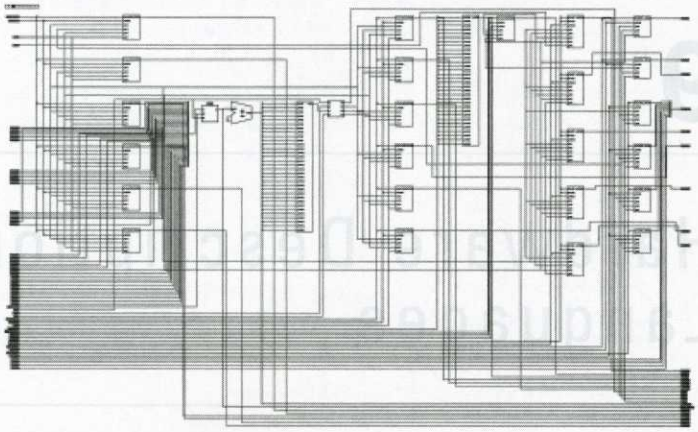


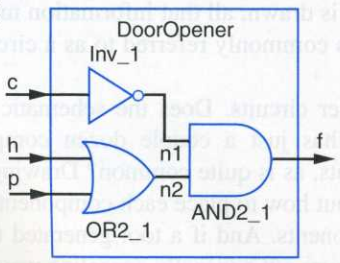
Figure 9.2 Schematics become hard to read beyond a dozen or so components—the graphical information becomes a nuisance rather than an aid.

would be needed for something that is really not very useful—humans can't comprehend circuit drawings of more than perhaps a few dozen gates, so what is the point of drawing such circuits? What is needed is a way to just describe the circuit itself—what are the inputs and outputs, what components exist, and what are the connections, without any of the graphical information like where each component is drawn or how big the component is drawn. Ideally, this description would be in a textual language so that people could type such descriptions with a computer keyboard just like they type email messages and C programs.

The circuit in Figure 9.3(a) could be described using the textual language of English as in Figure 9.3(b). We've given names to each gate in the circuit and to the internal wires in Figure 9.3(a).

Of course, English is not a good language if a computer tool will be used to read in the description—a computer tool requires a language with a precise syntax and precise meaning for every language construct. Computer-readable languages thus evolved in the

(a)



(b)

We'll now describe a circuit whose name is DoorOpener.
The external inputs are c, h, and p, which are bits.
The external output is f, which is a bit.

We assume you know the behavior of these components:
An inverter, which has a bit input x, and bit output F.
A 2-input OR gate, which has bit inputs x and y, and bit output F.
A 2-input AND gate, which has bit inputs x and y, and bit output F.

The circuit has internal wires n1 and n2, both bits.
The DoorOpener circuit internally consists of:
An inverter named Inv_1, whose input x connects to external input c, and whose output connects to n1.
A 2-input OR gate named OR2_1, whose inputs connect to external inputs h and p, and whose output connects to n2.
A 2-input AND gate named AND2_1, whose inputs connect to n1 and n2, and whose output connects to external output f.
That's all.

Figure 9.3 Describing a circuit using a textual language rather than a graphical drawing: (a) schematic, (b) textual description in the English language.

Structure

1970s and 1980s for describing hardware circuits. Such languages became known as ***hardware description languages***, or ***HDLs***. Hardware description languages not only enable describing the structural interconnections of components, but also include methods to describe the behavior of components themselves. Modern digital design relies heavily on the use of HDLs at all stages of design.

This chapter provides a brief introduction to the most popular hardware description languages—VHDL, Verilog, and SystemC—but to thoroughly learn each language, one may want to consult textbooks specifically dedicated to each language. Each section of this chapter can be covered immediately after corresponding earlier chapters (Section 9.2 after Chapter 2, Section 9.3 after Chapter 3, Section 9.4 after Chapter 4, and Section 9.5 after Chapter 5)—or these sections may be covered all at once after completing those earlier chapters. Furthermore, each section has three parts, one for VHDL, one for Verilog, and one for SystemC. Each of those parts is independent of the other parts of the section, so a reader interested only in one of the HDLs, say Verilog, can read only the Verilog parts of each section, skipping the VHDL or SystemC parts.

A reader interested in comparing the three HDLs may read the sections of all three HDLs. In doing so, you may notice that the HDLs have similar capabilities, differing primarily in their syntax. Thus, after learning one HDL thoroughly, a designer can likely learn other HDLs quickly.

9.2 COMBINATIONAL LOGIC DESCRIPTION USING HARDWARE DESCRIPTION LANGUAGES

Structure

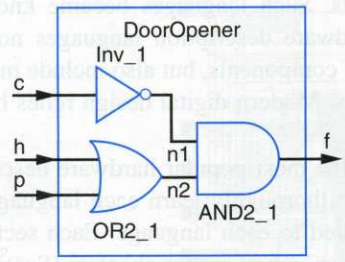
This chapter's introduction sought to describe a circuit using a textual language. This section shows how some different HDLs describe circuits. The term ***structure*** is sometimes used to refer to a circuit, with structure meaning an interconnection of component.

VHDL

Figure 9.4(c) shows a VHDL description of the *DoorOpener* circuit of Figure 9.4(a). For convenience, the English description appears in Figure 9.4(b), along with the correspondence between the English description and the VHDL description.

The description begins with an ***entity*** declaration, which defines the design's name and the design's inputs and outputs, known as ***ports***. An entity declaration says nothing about the internals of the design—just the design's name and interface. The description lists the port names and defines their type, which in this case is type ***std_logic***. That type means a bit, but isn't built into VHDL (the predefined *bit* type in VHDL is too limited, for reasons beyond this section's scope). Using *std_logic* requires including the statements: "*library ieee; use ieee.std_logic_1164.all;*" at the top of the file, which indicate that a ***library*** named *ieee* of predefined items will be used, in particular all the items within a package named *std_logic_1164*, which defines *std_logic* among other things.

The description continues with an ***architecture*** definition, which describes the internals of the design. The description names the architecture *Circuit*, but other names are possible like *DoorOpenerCircuit*, *DoorOpenerStructure*, *Structure*, or even *Fred*. Descriptive names that help in understanding the architecture are preferred. The architec-



(a)

We'll now describe a circuit whose name is DoorOpener.
The external inputs are c, h, and p, which are bits.
The external output is f, which is a bit.

We assume you know the behavior of these components:
An inverter, which has a bit input x, and bit output F.
A 2-input OR gate, which has bit inputs x and y, and bit output F.
A 2-input AND gate, which has bit inputs x and y, and bit output F.

The circuit has internal wires n1 and n2, both bits.

The DoorOpener circuit internally consists of:
An inverter named Inv_1, whose input x connects to external input c, and whose output connects to n1.
A 2-input OR gate named OR2_1, whose inputs connect to external inputs h and p, and whose output connects to n2.
A 2-input AND gate named AND2_1, whose inputs connect to n1 and n2, and whose output connects to external output f.
That's all.

(b)

```
library ieee;
use ieee.std_logic_1164.all;
entity DoorOpener is
  port ( c, h, p: in std_logic;
         f: out std_logic
  );
end DoorOpener;

architecture Circuit of DoorOpener is
  component Inv
    port (x: in std_logic;
          F: out std_logic);
  end component;
  component OR2
    port (x,y: in std_logic;
          F: out std_logic);
  end component;
  component AND2
    port (x,y: in std_logic;
          F: out std_logic);
  end component;
  signal n1,n2: std_logic; --internal wires

begin
  Inv_1: Inv port map (x=>c, F=>n1);
  OR2_1: OR2 port map (x=>h,y=>p,F=>n2);
  AND2_1: AND2 port map (x=>n1,y=>n2,F=>f);
end Circuit;
```

(c)

Figure 9.4 Describing a circuit using a textual language rather than a graphical drawing: (a) schematic, (b) textual description in the English language, (c) textual description in the VHDL language. Bolded words are reserved words in VHDL.

ture starts by declaring what *components* the design will be using—those components must be defined elsewhere in the description's file or in another file. Those components' definitions will be discussed later. Each component declaration must define the inputs and outputs of each component, and those inputs and outputs must match the component's entity declaration (found elsewhere) exactly.

The description then includes a declaration of the design's internal ***signals*** *n1* and *n2*, which are essentially internal wires. Next to that declaration, the description includes an example of a VHDL ***comment***: "*-- internal wires*". Comments start with "--" followed by any text on the rest of the line. That text is ignored by VHDL tools, but is useful to people who read the description.

Finally, the description instantiates the circuit's components and defines those components' connections using ***port maps***. For example, the statement "*Inv_1: Inv port map (x=>c, F=>n1);*" instantiates a component named *Inv_1*, which is a component of type

Inv (declared earlier in the VHDL description), and indicates that *Inv_1*'s input *x* connects to *c* and that *Inv_1*'s output *F* connects to *n1*, which is an internal signal. A more concise port map notation omits the port names: "*Inv_1: Inv port map (c, n1);*"—the order of the signals in the port map of *Inv* corresponds to the order of the ports in the component definition of *Inv*. Subsequent examples will use this more concise notation.

The bold words in the description represent ***reserved words***, also known as ***keywords***, in VHDL. Reserved words cannot be used for names of entities, architectures, signals, instantiated components, etc., as those words have special meaning that guide VHDL tools to understand our descriptions.

Summarizing, the VHDL structural description has an entity that describes the design's name, its inputs, and its outputs; a declaration of what components will be used; a declaration of internal signals; and finally, an instantiation of all components along with their interconnections.

The entity just defined could then be used as a component in another entity.

Verilog

Figure 9.5(c) shows a Verilog description of the *DoorOpener* circuit of Figure 9.5(a). For convenience, the English description also appears in Figure 9.5(b), along with the correspondence between the English description and the Verilog description.

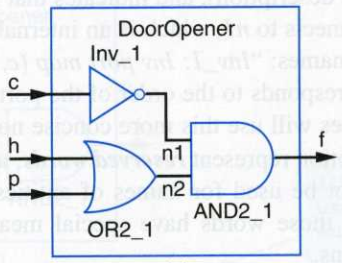
The description begins by defining modules for an inverter *Inv*, a 2-input OR gate *OR2*, and a 2-input AND gate *AND2*. We'll skip discussion of those modules, and begin our discussion with the definition of the fourth module *DoorOpener*.

The description declares a ***module*** named *DoorOpener*. The module declaration defines a design's name and the names of that design's inputs and outputs, known as ports. The module declaration says nothing about the internals of the design or the ports—just the design's name and interface.

The description then defines the type of each port, assigning the types ***input*** and ***output*** in this example.

The description then includes a declaration of the design's internal ***wires***, named *n1* and *n2*.

Finally, the description instantiates the circuit's components and defines those components' connections. For example, the statement "*Inv Inv_1(c, n1);*" instantiates a component named *Inv_1*, which is a component of type *Inv*. The connections to the inputs and outputs of the instantiated components are specified in the order in which the component's modules declare the inputs and outputs. In the instantiation of *Inv_1*, the input *c* is connected to the input *x* of the *Inv* component, and the wire *n1* is connected to output *F* of the component. In Verilog, the module does not need to specify the interface of a component within the module instantiating the component. For example, the *DoorOpener* module does not include a declaration of which components it will instantiate or any information regarding those components. The components, of course, must be defined elsewhere, perhaps earlier in the same file as shown in Figure 9.5(c), or perhaps in another file. For reference purposes, the example shown here provides incomplete specifications for the *Inv*, *AND2*, and *OR2* components in order to clearly show the ports and interface for these components. In place of specifying the internal behavior of these components, we simply included an example of a Verilog ***comment***. A comments starts with "//" and then can be followed by any text on the rest of the line.



(a)

We'll now describe a circuit whose name is DoorOpener.
The external inputs are c, h, and p, which are bits.
The external output is f, which is a bit.

We assume you know the behavior of these components:
An inverter, which has a bit input x, and bit output F.
A 2-input OR gate, which has bit inputs x and y, and bit output F.
A 2-input AND gate, which has bit inputs x and y, and bit output F.

The circuit has internal wires n1 and n2, both bits.

The DoorOpener circuit internally consists of:
An inverter named Inv_1, whose input x connects to external input c, and whose output connects to n1.
A 2-input OR gate named OR2_1, whose inputs connect to external inputs h and p, and whose output connects to n2.
A 2-input AND gate named AND2_1, whose inputs connect to n1 and n2, and whose output connects to external output f.
That's all.

(b)

```
module Inv(x, F);
   input x;
   output F;
   // details not shown
endmodule
module OR2(x, y, F);
   input x, y;
   output F;
   // details not shown
endmodule
module AND2(x, y, F);
   input x, y;
   output F;
   // details not shown
endmodule

module DoorOpener(c, h, p, f);
   input c, h, p;
   output f;
   wire n1, n2;
   Inv Inv_1(c, n1);
   OR2 OR2_1(h, p, n2);
   AND2 AND2_1(n1, n2, f);
endmodule
```

(c)

Figure 9.5 Describing a circuit using a textual language rather than a graphical drawing: (a) schematic, (b) textual description in the English language, (c) textual description in the Verilog language. Bold words are reserved words in Verilog.

The bold words in the description represent reserved words, also known as keywords, in Verilog. We cannot use reserved words for names of modules, ports, wires, instantiated components, etc., as those words have special meaning that guide Verilog tools to understand our descriptions.

Summarizing, the Verilog structural description has a module that describes the design name, lists the module's inputs and outputs, and specifies the type for each input and output; a declaration of internal wires; and finally, an instantiation of all components, along with their interconnections.

SystemC

Figure 9.6(c) shows a SystemC description of the *DoorOpener* circuit of Figure 9.6(a). For convenience, the English description also appears in Figure 9.6(b), along with the correspondence between the English description and the SystemC description. The

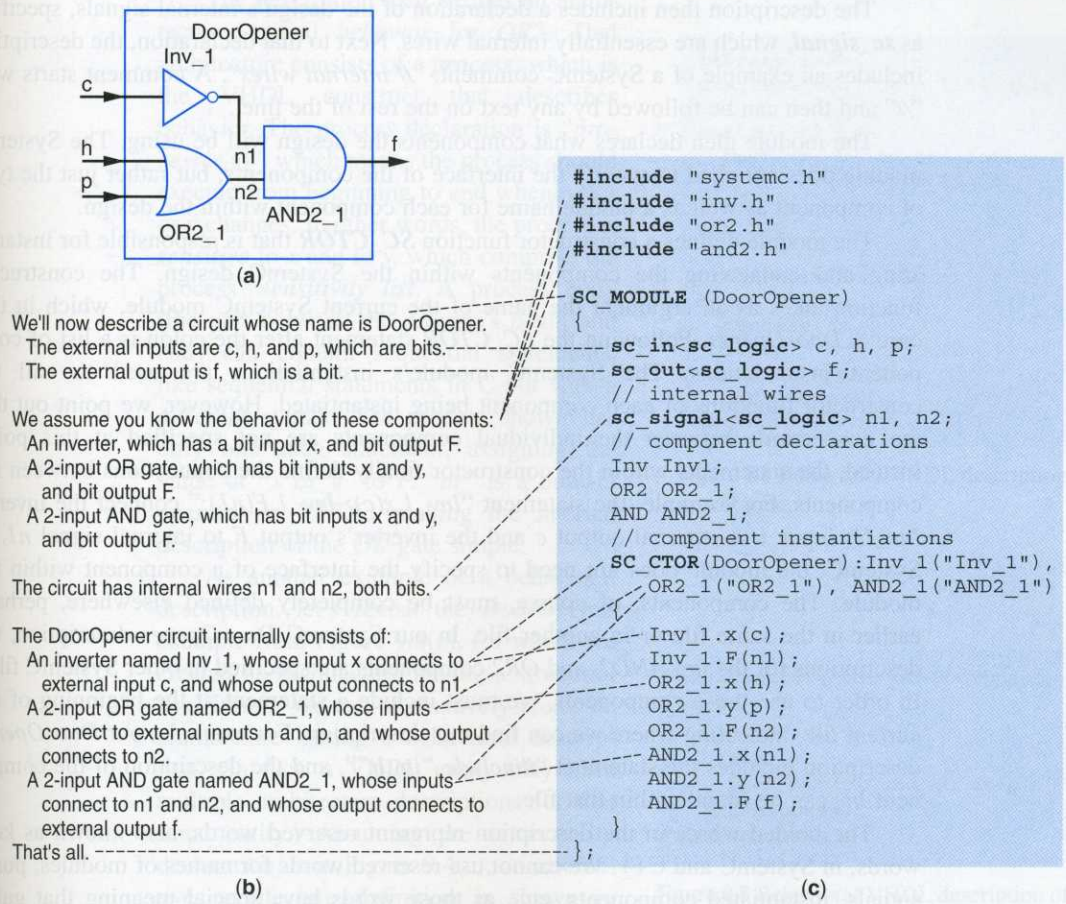


Figure 9.6 Describing a circuit using a textual language rather than a graphical drawing: (a) schematic, (b) textual description in the English language, (c) textual description in the SystemC language. Bold words are reserved words in SystemC.

SystemC language is built on top of the C++ programming language, but it is not necessary to be an expert C++ programmer to use SystemC. However, it is important to keep in mind that certain restrictions exist as a result, such as not using C++ keywords to name modules, ports, signals, etc.

Before defining the circuit behavior, we must include the statement "`#include "systemc.h"`" at the top of each SystemC file. The description begins with an ***SC_MODULE*** declaration, which defines the design's name, in this case *DoorOpener*. The module declaration says nothing about the internals of the design—just the design's name. Within the module description, the input and output ports of the design are specified, using the ***sc_in<>*** and ***sc_out<>*** statements respectively. The description lists the port names and defines their types, which in this case is type *sc_logic*, which specifies a single bit.

The description then includes a declaration of the design's internal signals, specified as ***sc_signal,*** which are essentially internal wires. Next to that declaration, the description includes an example of a SystemC comment: "*// internal wires*". A comment starts with "*//*" and then can be followed by any text on the rest of the line.

The module then declares what components the design will be using. The SystemC module does not need to specify the interface of the components, but rather just the type of component as well as a unique name for each component within the design.

The module defines a constructor function ***SC_CTOR*** that is responsible for instantiating and connecting the components within the SystemC design. The constructor function takes as an argument the name of the current SystemC module, which in this case is *DoorOpener*. Following the *SC_CTOR* statement after the colon is a list of component instantiations. The SystemC module's instantiations are used to call the constructor functions of each component being instantiated. However, we point out that the connections between the individual components are not specified at this point. Instead, the statements within the constructor finally define the connections between the components. For example, the statement "*Inv_1.x(c); Inv_1.F(n1);*" connect the inverter *Inv_1's* input *x* to external output *c* and the inverter's output *F* to internal signal *n1*. In SystemC, the module does not need to specify the interface of a component within the module. The components, of course, must be completely defined elsewhere, perhaps earlier in the same file or in another file. In our SystemC *DoorOpener* description, the descriptions for the *Inv, AND2,* and *OR2* components are specified in other SystemC files. In order to use those components, we must include a statement at the beginning of the current file, indicating where we can find this description. For example, our *DoorOpener* description includes the statement "*#include "inv.h"*", and the description of the component *Inv* can be found within that file.

The bolded words in the description represent reserved words, also known as keywords, in SystemC and C++. We cannot use reserved words for names of modules, ports, signals, instantiated components, etc., as those words have special meaning that guide SystemC and C++ tools to understand our descriptions.

Summarizing, the SystemC structural description has: a module that defines the design name; a list of inputs and outputs of the module specifying their types, a declaration of internal signals; a declaration of components providing the name for each component, a constructor function instantiating the module's components, and finally, the components' interconnections.

Combinational Behavior

HDLs typically support the ability to describe the internals of a design as behavior rather than as a circuit. This ability enables description of the bottom-level building-block components used in a design, such as the combinational behavior of an AND gate or OR gate.

VHDL

Figure 9.7 contains a behavioral description of a 2-input OR gate, which was used as a component in Figure 9.4(c). The description begins with the declarations necessary to use *std_logic*. It then declares the entity with the name *OR2* as having two input ports *x* and *y* and having output port *F*, all of type *std_logic* which means bit.

The description then defines an architecture named *behavior* for *OR2*. That architecture consists of a ***process***, which is the VHDL construct that describes behavior. The process declaration is "*process(x,y)*", which means the process should execute from beginning to end whenever *x* or *y* changes. In other words, the process is ***sensitive*** to *x* and to *y*, which comprise the process' ***sensitivity list***. A process body (the part between the process's *begin* and *end*) can contain sequential statements, like sequential statements in C but with a different syntax. The process shown has only one such statement, assigning the value of "*x or y*" to *F*. "*or*" is a built-in operator in VHDL, making the internal description of the OR gate simple.

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

entity OR2 is
  port (x, y: in std_logic;
        F: out std_logic
  );
end OR2;

architecture behavior of OR2 is
begin
  process (x, y)
  begin
    F <= x or y;
  end process;
end behavior;
```

Figure 9.7 Behavioral VHDL description of an OR gate.

As another example of a behavioral description, let's revisit the *DoorOpener* example from Figure 9.4(c), for which we created an architecture having a structural description. We can alternatively create an architecture having a behavioral description. (In fact, a VHDL entity may have multiple architecture descriptions for that same entity.) Assuming the same entity declaration as in Figure 9.4(c), an alternative architecture definition is shown in Figure 9.8. The behavior consists of a process that is sensitive to inputs *c*, *h*, and *p*. When the process executes (which is whenever *c*, *h*, or *p* changes), then the process executes its one statement, which updates the value of *f*.

```
architecture beh of DoorOpener is
begin
  process(c, h, p)
  begin
    f <= not(c) and (h or p);
  end process;
end beh;
```

Figure 9.8 Behavioral VHDL description of the *DoorOpener* design.

In designing the *DoorOpener* circuit, we might start with the behavioral description, and run a simulation to verify correct behavior. We might then create a structural description, and run simulation again to verify that the circuit has the same functionality as the behavior. In fact, tools exist that automatically convert such behavior to a circuit.

When describing a combinational circuit's behavior, care must be taken to include all the circuit's inputs in the process's sensitivity list. Omitting an input is not a VHDL error, but such omission is a mistake that results in different behavior than combinational behavior—with an input omitted, the output does not change when that input changes, meaning there must be some storage in the circuit.

Verilog

Figure 9.9 contains a behavioral description of a 2-input OR gate, which was used as a component in Figure 9.5. The description begins by declaring the module named *OR2* and specifying that the module has three ports named *x*, *y*, and *F*. The description then defines that the ports *x* and *y* are both inputs and the port *F* is an output. The description then defines the output *F* to be a *reg* output. In Verilog, a port is by default assumed to be a ***wire***, which does not store values. Instead, wires can only create connections between components. If we want to assign a value to an output port, we must define the port to be a ***reg***, which indicates the output port stores the values assigned to the port. The Verilog code for the design continues with an ***always*** procedure that defines a block of code that will be repeatedly executed whenever a change occurs on an input in the block's input list. The always procedure declaration is "*always @(x or y)*", which means the procedure should execute from beginning to end whenever *x* or *y* changes. In other words, the procedure is *sensitive* to *x* and *y*. *x* and *y* comprise the procedure's sensitivity list. The *always* procedure's statements (the part between the procedure's *begin* and *end*) can contain sequential statements, like sequential statements in C but with a different syntax. The block shown has only one such statement, assigning the value of "*x | y*" to *F*, where "|" is a built-in Verilog operation to compute an OR.

```
module OR2(x,y,F);
  input x, y;
  output F;
  reg F;

  always @(x or y)
  begin
    F <= x | y;
  end
endmodule
```

Figure 9.9 Behavioral Verilog description of an OR gate.

As another example of a behavioral description, let's revisit our *DoorOpener* example from Figure 9.5(c), for which we created a structural Verilog description. We can alternatively create a behavioral description. Figure 9.10 presents a behavioral Verilog description of the *DoorOpener* circuit. The module declaration is similar to the structural description of Figure 9.5(c), but in the behavioral description we need to declare the output *f* as a *reg*. The behavior consists of an always procedure sensitive to inputs *c*, *h*, and *p*. When the procedure executes (which is whenever *c*, *h*, or *p* changes), then the procedure executes a single statement that updates the value of *f*, by assigning the value "*(~c) & (h | p)*", where "~" , "&", and "|" perform the invert, AND, and OR operations, respectively.

```
module DoorOpener(c,h,p,f);
  input c, h, p;
  output f;
  reg f;

  always @(c or h or p)
  begin
    f <= (~c) & (h | p);
  end
endmodule
```

Figure 9.10 Behavioral Verilog description of the DoorOpener design.

In designing the *DoorOpener* circuit, we might start with the behavioral description and run a simulation to verify correct behavior. We might then create a structural description and run a simulation again to verify that the circuit has the same functionality as the behavior. In fact, tools exist that automatically convert such behavior to a circuit.

When describing a combinational circuit's behavior, care must be taken to include all the circuit's inputs in the procedure's sensitivity list. Omitting an input is not a Verilog error, but such omission is a mistake that results in different behavior than combinational behavior—with an input omitted, the output does not change when that input changes, meaning there must be some storage in the circuit. Verilog provides a means of avoiding such a mistake. The sensititivy list can be specified merely as "*always @(*)*", which means that the procedure is sensitive to all items that are read in the procedure.

SystemC

Figure 9.11 contains a SystemC behavioral description of a 2-input OR gate, which you'll recall we used as a component in Figure 9.6(c). The SystemC description declares the module with the name *OR2* and has two input ports *x* and *y* and one output port *F*, all of type *sc_logic*, indicating that each input and output is an individual bit. The module defines the constructor function *SC_CTOR* that consists of a single process named *comblogic* defined as a *SC_METHOD*. ***SC_METHOD*** is one SystemC construct that describes behavior. The process declaration here is "*SC_METHOD (comblogic); sensitive << x << y;*", which means the process will execute the circuit behavior described in the function *comblogic* whenever *x* or *y* changes. In other words, the process is ***sensitive*** to *x* and *y*. The process body is defined in the function *comblogic* and is declared as "*void comblogic()*". The process function (the part between the open brace "*{*" and close brace "*}*") can contain sequential statements, like sequential statements in C or C++ but sometimes with different syntax. The process shown has only one such statement, writing the value of "*x.read() | y.read()*" to *F*, where "|" executes an OR operation. In SystemC, the current value of an input port can be read using the ***read()*** function and an output port can be written using the ***write()*** function. While other methods of accessing the input and output ports are possible, the *read()* and *write()* functions are recommended.

```
#include "systemc.h"

SC_MODULE(OR2)
{
  sc_in<sc_logic> x, y;
  sc_out<sc_logic> F;

  SC_CTOR(OR2)
  {
    SC_METHOD(comblogic);
    sensitive << x << y;
  }

  void comblogic()
  {
    F.write(x.read() | y.read());
  }
};
```

Figure 9.11 Behavioral SystemC description of an OR gate.

As another example of a behavioral description, let's revisit our *DoorOpener* example from Figure 9.6(c), for which we created a structural SystemC description. We can alternatively create a behavioral description. Figure 9.12 presents a behavioral SystemC description of the *DoorOpener* circuit. The module declaration is the same as the structural description of Figure 9.6(c). The behavior consists of a single process, named *comblogic*, that is sensitive to inputs *c*, *h*, and *p*. When the process executes (which is whenever *c*, *h*, or *p* changes), then the process executes its one statement, which updates the value of *f* by assigning the value "*(~c.read()) & (h.read() | p.read())*", where "~" per-

```
#include "systemc.h"

SC_MODULE(DoorOpener)
{
  sc_in<sc_logic> c, h, p;
  sc_out<sc_logic> f;

  SC_CTOR(DoorOpener)
  {
    SC_METHOD(comblogic);
    sensitive << c << h << p;
  }

  void comblogic()
  {
    f.write((~c.read()) & (h.read() |
            p.read()));
  }
};
```

Figure 9.12 Behavioral SystemC description of the *DoorOpener* design.

forms an invert operation, "&" performs an AND operation, and "|" performs an OR operation.

In designing the *DoorOpener* circuit, we might start with the behavioral description, and run a simulation to verify correct behavior. We might then create a structural description, and run simulation again to verify that the circuit has the same functionality as the behavior. In fact, tools exist that automatically convert such behavior to a circuit.

Testbenches

One of the main uses of an HDL is that of simulating a new design to ensure that the design is correct. To simulate a design, we need to set the design's inputs to certain values, and then check that the design's output values are what we expect them to be. A system that sets input values and checks output values is known as a ***testbench***. We now show how to create an HDL testbench to test our *DoorOpener* circuit.

VHDL

Figure 9.13 shows a VHDL testbench for the *DoorOpener* design of Figure 9.4(c). Notice that the entity, named *Testbench*, has no ports—the entity is self-contained, requiring no inputs and generating no outputs. The architecture declares the component that we plan to test—namely, the *DoorOpener* component. The architecture instantiates one instance of the *DoorOpener* component, which is named *DoorOpener1*. A single process in the architecture sets the inputs of the component and checks for correct output. This testbench tries all possible cases of the three inputs, of which there are eight cases. Many components have too many inputs to try all possible cases—in that situation, we might try border cases (e.g., all 0s, all 1s) and then some random cases.

Each case sets the three inputs of the component to a particular input combination,

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

entity Testbench is
end Testbench;

architecture behavior of Testbench is
  component DoorOpener
    port ( c, h, p: in std_logic;
           f: out std_logic
    );
  end component;
  signal c, h, p, f: std_logic;
begin
  DoorOpener1: DoorOpener port map (c,h,p,f);

  process
  begin
    -- case 0
    c <= '0'; h <= '0'; p <= '0';
    wait for 1 ns;
    assert (f='0') report "Case 0 failed";

    -- case 1
    c <= '0'; h <= '0'; p <= '1';
    wait for 1 ns;
    assert (f='1') report "Case 1 failed";
    -- (cases 2-6 omitted from figure)
    -- case 7
    c <= '1'; h <= '1'; p <= '1';
    wait for 1 ns;
    assert (f='0') report "Case 7 failed";

    wait; -- process does not wake up again
  end process;
end behavior;
```

Figure 9.13 Behavioral VHDL description of *DoorOpener* testbench.

and waits for those values to propagate through the component—we arbitrarily wait for 1 ns of simulated time, but could have picked any time, since we didn't actually create a time delay within the component. But we do have to wait for some time, as VHDL simulation is defined such that no signal is updated instantaneously, but rather after an infinitely small period of simulated time. After waiting, each case checks for the correct value on the output *f*, using an ***assert*** statement. If the condition of the *assert* statement evaluates to true, simulation proceeds to the next statement. But if the condition evaluates to false, the corresponding error message will be reported and the simulation will terminate.

Verilog

Figure 9.14 shows a Verilog testbench for the *DoorOpener* design of Figure 9.5(c). Notice that the module, named *Testbench*, has no ports—the module is self-contained, requiring no inputs and generating no outputs. The module first declares three registered signals *c*, *h*, and *p* and a single wire *f*. The signals *c*, *h*, and *p* are declared as *reg* because we must assign values to the signals that will be connected to the inputs of the design being tested. However, because we do not need to assign a value to the output being monitored, the signal *f* is declared as a *wire*. The testbench then instantiates one instance of the *DoorOpener* component, named *DoorOpener1*, and connects the inputs and outputs of the component to the internal wires. The testbench then contains an ***initial*** procedure that defines a block of code that will be executed only once when execution of the testbench begins. The *initial* procedure sets the inputs of the *DoorOpener* component and displays the resulting value of the component's output. This testbench tries all possible cases of the three inputs, of which there are eight cases. Many components have too many inputs to try all possible cases—in that situation, we might try border cases (e.g., all 0s, all 1s) and then some random cases.

```
module Testbench;
  reg c, h, p;
  wire f;

  DoorOpener DoorOpener1(c, h, p, f);

  initial
  begin
    // case 0
    c <= 0; h <= 0; p <= 0;
    #1 $display("f = %b", f);
    // case 1
    c <= 0; h <= 0; p <= 1;
    #1 $display("f = %b", f);
    // (cases 2-6 omitted from figure)
    // case 7
    c <= 1; h <= 1; p <= 1;
    #1 $display("f = %b", f);
  end
endmodule
```

Figure 9.14 Behavioral Verilog description of *DoorOpener* testbench.

Each case sets the three inputs of the component to a particular input combination, and waits for those values to propagate through the component—we arbitrarily wait for 1 unit of simulated time using the delay control statement "*#1*", but we could have picked any length of time, since we didn't actually create a time delay within the component. The Verilog language does not define standard time units, such as nanoseconds, but instead simply defines time in terms of time units, which a designer can use within a simulation environment. We do have to wait for some time, as the assignments within the testbench are nonblocking statements that are not updated until the current simulation

time completes. After waiting, each case outputs the value of the output *f* using a ***$display*** statement. The statement "*$display("f = %b", f)*" outputs the value of *f* in binary. For example, if the value of *f* is 1, then the display statement will output "*f = 1*". The display statement consists of a format string followed by a comma-separated list of wires, registers, or ports. Within the format string of our display statement, the *%b* indicates that the value of the signal specified after the format string will be displayed in binary. After simulation has finished, we can compare the values output during simulation to the expected values, to determine if our circuit is working correctly.

SystemC

Figure 9.15 shows a SystemC testbench for the *DoorOpener* design of Figure 9.6(c). Notice that the module, named *Testbench*, has three output ports, *c_t*, *h_t*, and *p_t*, and one input port *f_t*. In SystemC, we design the testbench circuit as a separate module that connects to the design we are testing. Therefore, for every input port on the circuit being tested, our testbench will have a corresponding output port. Likewise, for every output port on the circuit being tested, our testbench will have a corresponding input port. The testbench module defines a single process named *testbench_proc*. The testbench process is defined as an ***SC_THREAD***, which is similar to an *SC_METHOD* process except that the *SC_THREAD* allows us to use the ***wait()*** function within the process body to control the timing behavior of the process. In contrast, SystemC does not allow use of the *wait()* function within an *SC_METHOD* process. The testbench process controls the inputs of the circuit being tested and checks for correct output. This testbench tries all possible cases of the *DoorOpener's* three inputs, of which there are eight cases. Many components have too many inputs

```
#include "systemc.h"

SC_MODULE(Testbench)
{
  sc_out<sc_logic> c_t, h_t, p_t;
  sc_in<sc_logic> f_t;

  SC_CTOR(Testbench)
  {
    SC_THREAD(testbench_proc);
  }

  void testbench_proc()
  {
    // case 0
    c_t.write(SC_LOGIC_0);
    h_t.write(SC_LOGIC_0);
    p_t.write(SC_LOGIC_0);
    wait(1, SC_NS);
    assert( f_t.read() == SC_LOGIC_0 );

    // case 1
    c_t.write(SC_LOGIC_0);
    h_t.write(SC_LOGIC_0);
    p_t.write(SC_LOGIC_1);
    wait(1, SC_NS);
    assert( f_t.read() == SC_LOGIC_1 );

    // (cases 2-6 omitted from figure)
    // case 7
    c_t.write(SC_LOGIC_1);
    h_t.write(SC_LOGIC_1);
    p_t.write(SC_LOGIC_1);
    wait(1, SC_NS);
    assert( f_t.read() == SC_LOGIC_0 );

    sc_stop();
  }
};
```

Figure 9.15 Behavioral SystemC description of *DoorOpener* testbench.

to try all possible cases—in that situation, we might try border cases (e.g., all 0s, all 1s) and then some random cases.

Each case sets the three inputs of the *DoorOpener* circuit to a particular input combination, and waits for those values to propagate through the component—we arbitrarily wait for 1 ns of simulated time, but could have picked any time, since we didn't actually create a time delay within the component. But we do have to wait for some time, as SystemC simulation is defined such that no signal or port is updated instantaneously, but rather after an infinitely small period of simulated time. After waiting, each case checks for the correct output by reading the port *f_t* using an ***assert*** statement. If the condition of the assert statement evaluates to true, simulation proceeds to the next statement. But if the condition evaluates to false, simulation will stop and the corresponding error message will be reported.

In SystemC, values such as 0 and 1 are integer values and not logic values. Instead, SystemC defines the values *SC_LOGIC_0* and *SC_LOGIC_1* that correspond to the logic values of 0 and 1, respectively, which we used in the description.

▶ 9.3 SEQUENTIAL LOGIC DESCRIPTION USING HARDWARE DESCRIPTION LANGUAGES

Register

The most basic component in sequential logic is a register. We now show how to model a basic register in HDLs.

VHDL

Figure 9.16 shows a basic 4-bit register in VHDL. The register is identical to that described in Figure 3.36. The entity defines the data input *I* and the data output *Q*, as well as the clock input *clk*. The input *I* and output *Q* of this design correspond to 4-bit values. Instead of using 4 individual inputs and 4 individual outputs of types *std_logic*, the entity's *I* and *Q* ports are defined as *std_logic_vector*. A ***std_logic_vector*** is a vector, or array, of multiple *std_logic* elements. For example, the type declaration "*std_logic_vector(3 downto 0)*" defines a 4-bit vector of *std_logic* elements, where the bit positions within the vector are numbered from 3 to 0. The ***downto*** keyword defines the ordering of the elements within the vector, indicating that element 3 is located in the leftmost position. The statement "*I<="1000"*"

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

entity Reg4 is
    port ( I: in std_logic_vector(3 downto 0);
           Q: out std_logic_vector(3 downto 0);
           clk: in std_logic
    );
end Reg4;

architecture behavior of Reg4 is
begin
    process(clk)
    begin
       if (clk='1' and clk'event) then
        Q <= I;
       end if;
    end process;
end behavior;
```

Figure 9.16 Behavioral VHDL description of a 4-bit register.

would thus assign the value `1` to position 3 of the vector *I* and the value `0` to the remaining three positions. When assigning a value to a *std_logic_vector*, the vector's value must be specified within double quotations. For example, the decimal value 5 would be specified as a 4-bit *std_logic_vector* as "`0101`".

The architecture describes the register behaviorally, using a process statement. The process is sensitive to its *clk* input only. Because the process should only update its output during a rising clock edge, the process need not execute if input *I* changes. If *clk* changes, the process begins executing its statements. The first statement checks whether the process began executing due to a rising clock edge (`0` to `1`), as opposed to a falling clock edge (`1` to `0`). The statement checks for a rising edge by checking whether the *clk* input just changed (*clk'event*) and that change was to a `1` (*clk='1'*). If the process began executing due to a rising clock edge, then the process updates the register's contents using the statement "*Q <= I*". For a falling clock edge, the process will begin executing, check the *if* statement condition, and then reach the end of the process and hence stop executing without updating *Q*. Ideally, VHDL would have a way to begin executing a process only on a rising clock edge, but VHDL has no such feature.

In VHDL, output ports are a type of signal, and signals have memory in simulation. Thus, assigning *I* to *Q* causes *Q* to retain the new value, even when the process stops executing, thus implementing the storage part of the register.

Verilog

Figure 9.17 shows a basic 4-bit register in Verilog. The register is identical to that described in Figure 3.36. The module defines the data input *I* and the data output *Q*, as well as the clock input *clk*. The input *I* and output *Q* of this design correspond to a 4-bit value. Instead of using 4 individual inputs and 4 individual outputs, the module's *I* and *Q* ports are defined as vectors. For example, the type declaration "*input [3:0] I*" defines a 4-bit input vector where the bit positions within the vector are numbered from 3 to 0. The "*[3:0]*" defines the ordering of the elements within the vector, indicating that element 3 is located in the leftmost position. The statement "*I<=4'b1000*" would thus assign the value `1` to position 3 of the vector *I* and the value `0` to the remaining three positions. When assigning a value to a vector, we must specify the number of bits within the value, the base of the value, and the value itself. For example, the decimal value 5 would be specified as the 4-bit binary value *4'b0101*.

```
module Reg4(I, Q, clk);
  input [3:0] I;
  input clk;
  output [3:0] Q;
  reg [3:0] Q;

  always @(posedge clk)
  begin
    Q <= I;
  end
endmodule
```

Figure 9.17 Behavioral Verilog description of a 4-bit register.

The module describes the register behaviorally, using an *always* procedure. The procedure block is sensitive to the positive edge of the *clk* input, specified using the ***posedge*** keyword. Because the module should only update its output during a rising clock edge, the *always* procedure need not execute if *I* changes. On the positive edge of the clock, the procedure updates the register's contents using the statement "*Q <= I*". Because we defined the output *Q* as a *reg*, assigning *I* to *Q* causes *Q* to retain the new value, even when the procedure is done executing, thus implementing the storage part of the register.

SystemC

Figure 9.18 shows a basic 4-bit register in SystemC. The register is identical to that described in Figure 3.36. The module defines the data input *I* and the data output *Q*, as well as the clock input *clk*. The input *I* and output *Q* of this design correspond to a 4-bit value. Instead of using 4 individual inputs and 4 individual outputs of type *sc_logic*, the module's *I* and *Q* ports are defined as ***sc_lv***, which stands for logic vector. An *sc_lv* is a vector of multiple *sc_logic* elements. For example, the type declaration "*sc_lv<4>*" defines a 4-bit vector of *sc_logic* elements where the bit positions within the vector are numbered from 3 to 0. In SystemC, the ordering of the elements within the vector is defined such that the leftmost position is the most significant bit. For example, the statement *"I<="1000""* would thus assign the value `1` to position 3 of the vector *I* and the value `0` to the remaining three positions. When assigning a value to an *sc_lv*, the vector's value must be specified within double quotations. For example, the decimal value 5 would be specified as a 4-bit *sc_lv* as *"0101"*. Notice that in defining the input port for *I*, we included a space between the two closing angle brackets, "> >"—that space is required in SystemC.

```
#include "systemc.h"

SC_MODULE(Reg4)
{
  sc_in<sc_lv<4> > I;
  sc_out<sc_lv<4> > Q;
  sc_in<sc_logic> clk;

  SC_CTOR(Reg4)
  {
    SC_METHOD(seq_logic);
    sensitive_pos << clk;
  }

  void seq_logic()
  {
     Q.write(I.read());
  }
};
```

Figure 9.18 Behavioral SystemC description of a 4-bit register.

The module consists of a single process named *seq_logic* that is sensitive to the positive edge of the *clk* input. The positive edge is specified using the ***sensitive_pos*** statement for defining the sensitivity list. Bcause the module should only update its output during a rising clock edge, the *seq_logic* process need not wake up if *I* changes. On the positive edge of the clock, the register updates the register's contents using the statement "*Q.write(I.read())*".

In SystemC, output ports are a type of signal, and signals have memory. Thus, assigning *I* to *Q* causes *Q* to retain the new value, even when the process is done executing, thus implementing the storage part of the register.

Oscillator

VHDL

The register presented in Figure 9.16 has a clock input. We thus need to define an oscillator component that generates a clock signal. Figure 9.19 illustrates an oscillator described in VHDL. The entity defines one output, *clk*. The architecture consists of a process, but notice that the process does not have a sensitivity list. By default, such a process executes its statements as if they were enclosed in an infinite loop. So the process sets the clock to `0`, waits until 10 ns of simulated time passes, sets the clock to `1`, waits another 10 ns of simulated time, goes back to the first statement in the process that sets the clock to `0`, and so on. The output waveform for such an oscillator will be identical to the waveform shown in Figure 3.29.

The ***wait for*** statement in VHDL tells the simulator the amount of simulated time that the process should wait. A process *without* a sensitivity list *must* have at least one wait statement, otherwise the simulator will never finish simulating that process (because the process is in an implicit infinite loop), and thus the simulator will never get a chance to update outputs or to simulate other processes. On the other hand, a process *with* a sensitivity list *cannot* include wait statements, because by definition the sensitivity list defines when the process should execute.

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

entity Osc is
  port ( clk: out std_logic );
end Osc;

architecture behavior of Osc is
begin
  process
  begin
    clk <= '0';
    wait for 10 ns;
    clk <= '1';
    wait for 10 ns;
  end process;
end behavior;
```

Figure 9.19 VHDL oscillator description.

Verilog

The register presented in Figure 9.17 has a clock input. We thus need to define an oscillator component that generates a clock signal. Figure 9.20 illustrates an oscillator described in Verilog. The module defines one output, *clk*. The module consists of an *always* procedure, but notice that the always procedure does not have a sensitivity list. By default, such a procedure executes its statements as if they were enclosed in an infinite loop. Assuming we are using a time scale of nanoseconds, the always procedure sets the clock to `0`, delays for 10 ns of simulated time, sets the clock to `1`, delays for another 10 ns of simulated time, goes back to the first statement in the procedure that sets the clock to `0`, and so on. The output waveform for such an oscillator will be identical to the waveform shown in Figure 3.29.

```
module Osc(clk);
  output clk;
  reg clk;

  always
  begin
    clk <= 0;
    #10;
    clk <= 1;
    #10;
  end
endmodule
```

Figure 9.20 Verilog oscillator description.

The ***delay control statement***, specified with the **#** character, tells the simulator the amount of simulated time that the procedure should delay. A procedure *without* a sensitivity list *must* have at least one delay control statement, otherwise the simulator will never finish simulating that procedure (because the procedure is in an implicit infinite loop), and thus the simulator will never get the chance to update outputs or to simulate other procedures. On the other hand, a procedure *with* a sensitivity list *cannot* include delay control statements, because by definition the sensitivity list defines when the procedure should awake.

SystemC

The register presented in Figure 9.18 has a clock input. We thus need to define an oscillator component that generates a clock signal. Figure 9.21 illustrates an oscillator described in SystemC. The module defines one output, *clk*. The module consists of a single process, named *seq_logic*, implemented as an *SC_THREAD*. By default, an *SC_THREAD* process is only executed once. In order to ensure the process executes continuously, we enclose the statements of the process in an infinite loop, implemented using the statement "*while(true)*". Thus, the loop will execute the statement included within the braces forever. During execution, the process sets the clock to `0`, suspends execution for 10 ns of simulated time, sets the clock to `1`, sleeps another 10 ns of simulated time, sets the clock to `0`, and so on. The output waveform for such an oscillator will be identical to the waveform in Figure 3.29.

```
#include "systemc.h"

SC_MODULE(Osc)
{
  sc_out<sc_logic> clk;

  SC_CTOR(Osc)
  {
    SC_THREAD(seq_logic);
  }

  void seq_logic()
  {
    while(true) {
      clk.write(SC_LOGIC_0);
      wait(10, SC_NS);
      clk.write(SC_LOGIC_1);
      wait(10, SC_NS);
    }
  }
};
```

Figure 9.21 SystemC oscillator description.

The ***wait()*** function in SystemC tells the simulator the amount of simulated time that the process should wait. For example, the statement "*wait(10, SC_NS);*" will suspend the execution of the process for 10 ns. An *SC_THREAD* process explicitly implementing an infinite loop *must* have at least one wait statement, otherwise the simulator would never finish simulating that process (because the process is in an infinite loop), and thus the simulator could not update outputs or simulate other processes.

Controllers

Recall that a common type of sequential circuit is a *controller*, which implements a finite-state machine. The controller consists of a state register and combinational logic.

VHDL

Figure 9.22 shows one way to model a controller in VHDL. The controller modeled is for the FSM shown in Figure 3.47. The VHDL entity, named *LaserTimer*, defines the controller's inputs and outputs.

The VHDL architecture describes the behavior of the entity. The architecture consists of two processes, one modeling the state register, the other modeling the combinational logic, that form the standard controller architecture from Figure 3.59.

The first process describes the controller's state register. That process, named *statereg*, is sensitive to inputs *clk* and *rst*. If the *rst* input is enabled, then the process asynchronously sets the *currentstate* signal to the FSM's initial state, *S_Off*. Otherwise, if the clock is rising, the process updates the state register with the next state.

The *currentstate* and *nextstate* signals are defined as a user-defined type named *statetype*. *statetype* is defined by the ***type*** statement and specifies the possible values a signal of that type can represent. The *type* declaration of *statetype* consists of the names of all the states in the controller, *S_Off*, *S_On1*, *S_On2*, and *S_On3*.

The second process describes the controller's combinational logic. That process, named *comblogic*, is sensitive to the inputs to the combinational logic of Figure 3.59, namely the external inputs (in this case, *b*) and the state register outputs (*currentstate*). When either of those items change, the process sets the FSM's outputs, in this case *x*, with the appropriate value for the current state. The process also determines what the next state should be, based on the current state and the values of inputs (i.e., the conditions on the FSM transitions). The next state will be loaded into the state register by the state register process on the next rising clock edge.

Notice that the architecture declares two signals, *currentstate* and *nextstate*. Signals are visible across all processes in an architecture. The *currentstate* signal represents the actual storage of the state register. The *nextstate* signal represents the value coming from the combinational logic and going to the state register. Notice also that the architecture declares those signals as type *statetype*, defined in the architecture as a type whose value can be either *S_Off*, *S_On1*, *S_On2*, or *S_On3*.

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

entity LaserTimer is
   port (b: in std_logic;
         x: out std_logic;
         clk, rst: in std logic
   );
end LaserTimer;

architecture behavior of LaserTimer is
   type statetype is
      (S_Off, S_On1, S_On2, S_On3);
   signal currentstate, nextstate:
      statetype;
begin
   statereg: process(clk, rst)
   begin
      if (rst='1') then -- intial state
         currentstate <= S_Off;
      elsif (clk='1' and clk'event) then
         currentstate <= nextstate;
      end if;
   end process;

   comblogic: process (currentstate, b)
   begin
      case currentstate is
         when S_Off =>
            x <= '0'; -- laser off
            if (b='0') then
               nextstate <= S_Off;
            else
               nextstate <= S_On1;
            end if;
         when S_On1 =>
            x <= '1'; -- laser on
            nextstate <= S_On2;
         when S_On2 =>
            x <= '1'; -- laser still on
            nextstate <= S_On3;
         when S_On3 =>
            x <= '1'; -- laser still on
            nextstate <= S_Off;
      end case;
   end process;
end behavior;
```

Figure 9.22 Behavioral VHDL description of the *LaserTimer* controller.

Verilog

Figure 9.23 shows one way to model a controller in Verilog. The controller modeled is for the FSM shown in Figure 3.47. The Verilog module, named *LaserTimer*, defines the controller's inputs and outputs.

The module consists of two procedures, one modeling the state register, the other modeling the combinational logic, that together form the standard controller architecture from Figure 3.59.

The state register procedure is sensitive to the positive edge of the *rst* input and the positive edge of the *clk* input. The state register has an asynchronous reset signal and in order to model the asynchronous reset, the state register procedure must be sensitive to the positive edge of the *rst* input. On the positive edge of the *rst* input, the procedure will wake asynchronously and sets the *currentstate* signal to the FSM's initial state, *S_Off*. On the rising edge of the clock input, *clk*, if the reset input is not enabled, the procedure updates the state register with the *nextstate* value determined by the combinational logic procedure.

In Verilog, we explicitly specify the size of the state registers as well as define the FSM's state encodings. Within the *LaserTimer* module, we declare four parameters *S_Off*, *S_On1*, *S_On2*, and *S_On3*, which specify the encodings. For example, "*S_Off = 2'b00*" defines the state name *S_Off* and assigns the 2-bit value "*00*" as the encoding of this state. We can then refer to this state throughout the module using *S_Off* instead of using specific bit values. While not required to define a state machine, using parameters increases the readability of a design and makes revisions to the FSM easier. Because the *LaserTimer*'s FSM has four states, we need a 2-bit state register and thus declare the *currentstate* and *nextstate* signals as 2-bit registers.

```
module LaserTimer(b, x, clk, rst);
  input b, clk, rst;
  output x;
  reg x;

  parameter S_Off = 2'b00,
            S_On1 = 2'b01,
            S_On2 = 2'b10,
            S_On3 = 2'b11;

  reg [1:0] currentstate;
  reg [1:0] nextstate;
  // state register procedure
  always @(posedge rst or posedge clk)
  begin
    if (rst==1) // initial state
      currentstate <= S_Off;
    else
      currentstate <= nextstate;
  end
  // combinational logic procedure
  always @(*)
  begin
    case (currentstate)
      S_Off: begin
        x <= 0; // laser off
        if (b==0)
          nextstate <= S_Off;
        else
          nextstate <= S_On1;
      end
      S_On1: begin
        x <= 1; // laser on
        nextstate <= S_On2;
      end
      S_On2: begin
        x <= 1; // laser still on
        nextstate <= S_On3;
      end
      S_On3: begin
        x <= 1; // laser still on
        nextstate <= S_Off;
      end
    endcase
  end
endmodule
```

Figure 9.23 Behavioral Verilog description of the *LaserTimer* controller.

The second procedure is the combinational procedure implementing the control logic of the FSM. That procedure is sensitive to all the items it reads (specified by the "*"), which are the inputs to the combinational logic of Figure 3.59, namely the external inputs (in this case, *b*) and the state register outputs (*currentstate*). When either of those items change, the

procedure sets the FSM's outputs, in this case *x*, with the appropriate value for the current state. The procedure also determines the next state based on the current state and the values of inputs (i.e., the conditions on the FSM transitions). The next state will be loaded into the state register by the state register procedure on the next positive clock edge.

Notice that the module declares two signals *currentstate* and *nextstate*. Signals are visible across all procedures in a module. The *currentstate* signal represents the actual storage of the state register. *nextstate* represents the value coming from the combinational logic going to the state register.

SystemC

Figure 9.24 shows one way to model a controller in SystemC, for the FSM in Figure 3.47. The module, named *LaserTimer*, defines the controller's inputs and outputs.

The module consists of two processes, one modeling the state register named *statereg*, the other process modeling the combinational logic named *comblogic*, that form the standard controller architecture of Figure 3.59.

The state register process is sensitive to the positive edge of the *rst* input and the positive edge of the *clk* input. The state register has an asynchronous reset signal. In order to model the asynchronous reset, the state register process is sensitive to the positive edge of the *rst* input. On the positive edge of the *rst* input, the process will wake asynchronously and sets the *currentstate* signal to the FSM's initial state, *S_Off*. On the rising edge of the clock input, *clk*, if the reset input is not enabled, the process updates the state register with the *nextstate* value determined by the combinational logic process.

```
#include "systemc.h"

enum statetype { S_Off, S_On1, S_On2, S_On3 };

SC_MODULE(LaserTimer)
{
  sc_in<sc_logic> b, clk, rst;
  sc_out<sc_logic> x;
  sc_signal<statetype> currentstate, nextstate;

  SC_CTOR(LaserTimer) {
    SC_METHOD(statereg);
    sensitive_pos << rst << clk;
    SC_METHOD(comblogic);
    sensitive << currentstate << b;
  }

  void statereg() {
    if( rst.read() == SC_LOGIC_1 )
      currentstate = S_Off; // initial state
    else
      currentstate = nextstate;
  }
  void comblogic() {
    switch (currentstate) {
      case S_Off:
        x.write(SC_LOGIC_0); // laser off
        if( b.read() == SC_LOGIC_0 )
          nextstate = S_Off;
        else
          nextstate = S_On1;
        break;
      case S_On1:
        x.write(SC_LOGIC_1); // laser on
        nextstate = S_On2;
        break;
      case S_On2:
        x.write(SC_LOGIC_1); // laser still on
        nextstate = S_On3;
        break;
      case S_On3:
        x.write(SC_LOGIC_1); // laser still on
        nextstate = S_Off;
        break;
    }
  }
};
```

Figure 9.24 Behavioral SystemC description of the *LaserTimer* controller.

▶ 9.4 DATA
USIN

Full-Adders

The *currentstate* and *nextstate* signals are defined as a user-defined type, named *statetype*. *statetype* is defined by the ***enum*** statement and specifies the possible values a signal of that type can represent. The *enum* declaration for *statetype* consists of the names of all the states in the controller, *S_Off, S_On1*, *S_On2*, and *S_On3*.

The second process, named *comblogic*, is sensitive to the inputs to the combinational logic of Figure 3.59, namely the external inputs (in this case, *b*) and the state register outputs (*currentstate*). When either of those items change, the process sets the FSM's outputs, in this case *x*, with the appropriate value for the current state. The process also determines what the next state should be, based on the current state and the values of inputs (i.e., the conditions on the FSM transitions). The next state will be loaded into the state register by the state register process on the next rising clock edge. Within the first state, we determine the next state depending on the value of input *b* by performing the comparison "*b.read() == SC_ LOGIC_0*". Note that the comparison for equality uses the syntax "==". A common mistake is to use "=" instead, which means assignment rather than equality.

Notice that the module declares two *sc_signals*: *currentstate* and *nextstate*. Signals are visible across all processes in a module. The *currentstate* signal represents the actual storage of the state register. The *nextstate* signal represents the value coming from the combinational logic and going to the state register. Notice also that the architecture declares those signals as a type *statetype*, defined in the architecture as a type whose value can be either *S_Off*, *S_On1*, *S_On2*, or *S_On3*.

▶ 9.4 DATAPATH COMPONENT DESCRIPTION USING HARDWARE DESCRIPTION LANGUAGES

Full-Adders

This section shows how to describe a full-adder behaviorally in an HDL. Recall from Figure 4.31 that a full-adder is a combinational circuit that adds three bits (`a`, `b`, and `ci`) and outputs a sum (`s`) and a carry-out (`co`) bit.

VHDL

Figure 9.25 shows a full-adder described behaviorally in VHDL. The VHDL entity, named *FullAdder*, defines the full-adder's three inputs *a*, *b*, and *ci* and two outputs *s* and *co*.

The architecture describes the behavior of the full-adder. The architecture consists of one process describing the combinational behavior of the full-adder. The

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

entity FullAdder is
   port ( a, b, ci: in std_logic;
          s, co: out std_logic
   );
end FullAdder;

architecture behavior of FullAdder is
begin
   process (a, b, ci)
   begin
      s <= a xor b xor ci;
      co <= (b and ci) or (a and ci) or (a and b);
   end process;
end behavior;
```

Figure 9.25 Behavioral VHDL description of a full-adder.

process is sensitive to all three inputs (*a*, *b*, and *ci*) of the full-adder. When any of the inputs change, the process executes its two statements updating the values for the sum (*s*) and carry-out (*co*).

Verilog

Figure 9.26 shows a full-adder described behaviorally in Verilog. The Verilog module, named *FullAdder*, defines the full-adder's three inputs *a*, *b*, and *ci* and two outputs *s* and *co*.

```
module FullAdder(a, b, ci, s, co);
  input a, b, ci;
  output s, co;
  reg s, co;

  always @(*)
  begin
    s <= a ^ b ^ ci;
    co <= (b & ci) | (a & ci) | (a & b);
  end
endmodule
```

Figure 9.26 Behavioral Verilog description of a full-adder.

The module describes the behavior of the full-adder and consists of a single *always* procedure describing the combinational behavior of the full-adder. The procedure is sensitive to all items read, which in this case are the three inputs (*a*, *b*, or *ci*) of the full-adder. When any of the inputs change, the procedure executes its two statements updating the values for the sum (*s*) and carry-out (*co*).

SystemC

Figure 9.27 shows a full-adder described behaviorally in SystemC. The SystemC module, named *FullAdder*, defines the full-adder's three inputs *a*, *b*, and *ci* and two outputs *s* and *co*.

```
#include "systemc.h"

SC_MODULE(FullAdder)
{
  sc_in<sc_logic> a, b, ci;
  sc_out<sc_logic> s, co;

  SC_CTOR(FullAdder)
  {
    SC_METHOD(comblogic);
    sensitive << a << b << ci;
  }

  void comblogic()
  {
    s.write(a.read() ^ b.read() ^ ci.read());
    co.write((b.read() & ci.read()) |
             (a.read() & ci.read()) |
             (a.read() & b.read()));
  }
};
```

Figure 9.27 Behavioral SystemC description of a full-adder.

The module describes the behavior of the full-adder and consists of a single process, named *comblogic*, describing the combinational behavior of the full-adder. The process is sensitive to all three inputs (*a*, *b*, or *ci*) of the full-adder. When any of the inputs change, the process executes its two statements updating the values for the sum (*s*) and carry-out (*co*).

Carry-Ripple Adders

This section shows how to structurally describe the 4-bit carry-ripple adder of Figure 4.32 using the full-adder designed in the previous section.

VHDL

Figure 9.28 is a VHDL description of a 4-bit carry-ripple adder with a carry-in. The VHDL entity, named *CarryRippleAdder4*, has two 4-bit inputs *a* and *b*, and a carry-in input *ci*. The carry-ripple adder outputs a 4-bit sum *s* and a final carry-out *co*.

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

entity CarryRippleAdder4 is
   port ( a:  in std_logic_vector(3 downto 0);
          b:  in std_logic_vector(3 downto 0);
          ci: in std_logic;
          s:  out std_logic_vector(3 downto 0);
          co: out std_logic
   );
end CarryRippleAdder4;

architecture structure of CarryRippleAdder4 is
   component FullAdder
      port ( a, b, ci: in std_logic;
             s, co: out std_logic
      );
   end component;
   signal co1, co2, co3: std_logic;
begin
   FullAdder1: FullAdder
      port map (a(0), b(0), ci, s(0), co1);
   FullAdder2: FullAdder
      port map (a(1), b(1), co1, s(1), co2);
   FullAdder3: FullAdder
      port map (a(2), b(2), co2, s(2), co3);
   FullAdder4: FullAdder
      port map (a(3), b(3), co3, s(3), co);
end structure;
```

Figure 9.28 Structural VHDL description of a 4-bit carry-ripple adder.

The architecture structurally describes the carry-ripple adder composed of four full-adders. The architecture begins by declaring the component *FullAdder*, which was described in the previous section. The design has three internal signals, *co1*, *co2*, and *co3*, that are used for internal connection between the full-adders. The architecture then instantiates four *FullAdder* components. In VHDL, each instantiated component must have a unique name. The four *FullAdder* components in this design are uniquely identified by the names *FullAdder1*, *FullAdder2*, *FullAdder3*, and *FullAdder4*.

In VHDL, the *std_logic_vector* type provides a convenient method of specifying ports or signals consisting of multiple bits. However, a design may need to access the individual bits of these vectors. The individual bits of a *std_logic_vector* can be accessed by specifying the desired bit position within parentheses after the vector's name. For example, to access bit 0 of the 4-bit input *a* of this design, one would use the syntax "*a(0)*". In defining the connections to the instantiated components in the carry-ripple adder, individual bits of the inputs *a* and *b* and output *s* are accessed using this syntax. The first full-adder, *FullAdder1*, connects bit 0 of the inputs *a* and *b* as well as the carry-ripple adder's carry-in *ci* to the full-adder's three inputs. The *s* output of *FullAdder1* is connected to bit 0 of the 4-bit adder's sum output *s*, represented as *s(0)*. The design then connects the carry-out bit of *FullAdder1* to the internal signal *co1*, which is subsequently connected to the carry-in input of the next full-adder *FullAdder2*. The component connections of the remaining three full-adders are connected in a similar manner, with the

exception of the last full-adder in the carry-ripple chain. The carry-out from that last full-adder *FullAdder4* is connected to the carry-out output *co* of the carry-ripple adder.

Verilog

Figure 9.29 is a Verilog description of a 4-bit carry-ripple adder with a carry-in. The Verilog module, which is named *CarryRippleAdder4*, has two 4-bit inputs *a* and *b*, and a carry-in input *ci*. The carry-ripple adder outputs a 4-bit sum *s* and a final carry-out *co*.

```
module CarryRippleAdder4(a, b, ci, s, co);
  input [3:0] a;
  input [3:0] b;
  input ci;
  output [3:0] s;
  output co;

  wire co1, co2, co3;

  FullAdder FullAdder1(a[0], b[0], ci,
                       s[0], co1);
  FullAdder FullAdder2(a[1], b[1], co1,
                       s[1], co2);
  FullAdder FullAdder3(a[2], b[2], co2,
                       s[2], co3);
  FullAdder FullAdder4(a[3], b[3], co3,
                       s[3], co);
endmodule
```

Figure 9.29 Structural Verilog description of a 4-bit carry-ripple adder.

The module structurally describes the carry-ripple adder composed of four full-adders. The design has three internal wires, *co1*, *co2*, and *co3*, that are used for internal connection between the full-adders. The module instantiates four *FullAdder* components. In Verilog, each instantiated component must have a unique name. The four *FullAdder* components in this design are uniquely identified by the names *FullAdder1*, *FullAdder2*, *FullAdder3*, and *FullAdder4*.

In Verilog, vectors provide a convenient method of specifying ports or signals consisting of multiple bits. However, a design may need to access the individual bits of these vectors. The individual bits of a vector can be accessed by specifying the desired bit position within brackets after the vector's name. For example, to access bit 0 of the 4-bit input *a* of this design, one would use the syntax "*a[0]*". In defining the connections to the instantiated components in the carry-ripple adder, individual bits of the inputs *a* and *b* and output *s* are accessed using this syntax. The first full-adder, *FullAdder1*, connects bit 0 of the inputs *a* and *b* as well as the carry-ripple adder's carry-in *ci* to the full-adder's three inputs. The *s* output of *FullAdder1* is connected to bit 0 of the 4-bit adder's sum output *s*, represented as *s[0]*. The design then connects the carry-out bit of *FullAdder1* to the internal signal *co1*, which is subsequently connected to the carry-in input of the next full-adder *FullAdder2*. The component connections of the remaining three full-adders are connected in a similar manner, with the exception of the last full-adder in the carry-ripple chain. The carry-out from the last full-adder *FullAdder4* is connected to the carry-out output *co* of the carry-ripple adder.

SystemC

Figure 9.30 is a SystemC description of a 4-bit carry-ripple adder with a carry-in. The SystemC module, named *CarryRippleAdder4*, has two 4-bit inputs *a* and *b*, and a carry-in input *ci*. The carry-ripple adder outputs a 4-bit sum *s* and a final carry-out *co*.

The module structurally describes the carry-ripple adder composed of four full-adders. The design has three internal signals, *co1*, *co2*, and *co3*, that are used for internal connection between the full-adders. The module first instantiates four *FullAdder* components. In SystemC, each instantiated component must have a unique name. The four *FullAdder* components in this design are uniquely identified by the names *FullAdder_1*, *FullAdder_2*, *FullAdder_3*, and *FullAdder_4*.

Previously, we defined multiple-bit inputs as an input vector using the *sc_lv* type. However, SystemC does not support connecting individual bits within a signal or port of type *sc_lv* in a structural description. In our *CarryRippleAdder4* design, we instead defined the inputs and outputs *a*, *b*, and *s* as arrays of *sc_logic* with four elements each, rather than using type *sc_lv*. The individual bits of the array can be accessed by specifying the desired bit position within brackets after the array's name. For example, to access bit 0 of the 4-element input array *a* of this design, one would use the syntax "*a[0]*". In defining the connections to the instantiated components in the carry-ripple adder, individual bits of the inputs *a* and *b* and output *s* are accessed using this syntax. The first full-adder *FullAdder_1* connects bit 0 of the inputs *a* and *b* as well as the carry-ripple adder's carry-in *ci* to the full-adder's three inputs. The *s* output of *FullAdder_1* is connected to bit 0 of the 4-bit adder's sum output *s*, represented as *s[0]*. The design then connects the carry-out bit of *FullAdder_1* to the internal signal *co1* that is subsequently connected to the carry-in input of the next full-adder *FullAdder_2*. The component connections of the remaining three full-adders are connected in a similar manner, with the exception of the last full-adder in the carry-ripple chain. The carry-out from the last full-adder *FullAdder_4* is connected to the carry-out output (*co*) of the carry-ripple adder.

```
#include "systemc.h"
#include "fulladder.h"

SC_MODULE(CarryRippleAdder4)
{
  sc_in<sc_logic> a[4];
  sc_in<sc_logic> b[4];
  sc_in<sc_logic> ci;
  sc_out<sc_logic> s[4];
  sc_out<sc_logic> co;

  sc_signal<sc_logic> co1, co2, co3;

  FullAdder FullAdder_1;
  FullAdder FullAdder_2;
  FullAdder FullAdder_3;
  FullAdder FullAdder_4;

  SC_CTOR(CarryRipple4):
    FullAdder_1("FullAdder_1"),
    FullAdder_2("FullAdder_2"),
    FullAdder_3("FullAdder_3"),
    FullAdder_4("FullAdder_4")
  {
    FullAdder_1.a(a[0]); FullAdder_1.b(b[0]);
    FullAdder_1.ci(ci); FullAdder_1.s(s[0]);
    FullAdder_1.co(co1);

    FullAdder_2.a(a[1]); FullAdder_2.b(b[1]);
    FullAdder_2.ci(co1); FullAdder_2.s(s[1]);
    FullAdder_2.co(co2);

    FullAdder_3.a(a[2]); FullAdder_3.b(b[2]);
    FullAdder_3.ci(co2); FullAdder_3.s(s[2]);
    FullAdder_3.co(co3);

    FullAdder_4.a(a[3]); FullAdder_4.b(b[3]);
    FullAdder_4.ci(co3); FullAdder_4.s(s[3]);
    FullAdder_4.co(co);
  }
};
```

Figure 9.30 Structural SystemC description of a 4-bit carry-ripple adder.

Up-Counter

This section shows how to structurally describe the 4-bit up-counter of Figure 4.66.

VHDL

Figure 9.31 is a VHDL description of a 4-bit up-counter. The entity, named *UpCounter*, defines the counter's inputs and outputs, consisting of a clock input *clk*, a count enable control input *cnt*, the 4-bit count value *C*, and a terminal count output *tc*.

The *UpCounter's* architecture structurally describes the design consisting of three components, namely *Reg4*, *Inc4*, and *AND4*. *Reg4* is a 4-bit parallel-load register with a load control input *ld*. *Inc4* is a 4-bit incrementer. *AND4* is a four-input AND gate that will output `1` when all four inputs are `1`. The architecture further specifies two signals, *tempC* and *incC*, used as internal wires within the structural description.

The architecture instantiates each of the three components and specifies the connections among them. *Reg4* is the only sequential component within the up-counter and thus the *clk* input only needs to be connected to the clock input of the register. The up-counter's counting is controlled by connecting the count enable input *cnt* to the load enable *ld* of the register. The output *Q* of *Reg4_1* is connected to the internal signal *tempC*, which connects the register's output to both the *Inc4_1* and *AND4_1* components. *Inc4_1* receives the current count from the *tempC* connection and outputs the incremented count on its output *s*, which is connected to the other internal signal *incC*. The *incC* signal connects the incremented count from *Inc4_1* to the parallel load input *I* of *Reg4_1*. The

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

entity UpCounter is
   port ( clk: in std_logic;
          cnt: in std_logic;
          C: out std_logic_vector(3 downto 0);
          tc: out std_logic
   );
end UpCounter;

architecture structure of UpCounter is
   component Reg4
      port ( I: in std_logic_vector(3 downto 0);
             Q: out std_logic_vector(3 downto 0);
             clk, ld: in std_logic
      );
   end component;
   component Inc4
      port ( a: in std_logic_vector(3 downto 0);
             s: out std_logic_vector(3 downto 0)
      );
   end component;
   component AND4
      port ( w,x,y,z: in std_logic;
             F: out std_logic
      );
   end component;
   signal tempC: std_logic_vector(3 downto 0);
   signal incC: std_logic_vector(3 downto 0);
begin
   Reg4_1: Reg4 port map(incC, tempC, clk, cnt);
   Inc4_1: Inc4 port map(tempC, incC);
   AND4_1: AND4 port map(tempC(3), tempC(2),
                         tempC(1), tempC(0), tc);

   outputC: process(tempC)
   begin
      C <= tempC;
   end process;
end structure;
```

Figure 9.31 Structural VHDL description of 4-bit up-counter.

current count is also connected to the four inputs of the *AND4_1* component. *AND4_1*'s output *F* is connected to the counter's terminal count output *tc*.

The *UpCounter* design must connect the output of the 4-bit register to the incrementer, the AND gate, and the counter's output port *C*. VHDL does not allow connecting multiple signals or ports within the *port map* of an instantiated component. Therefore, the architecture uses the *tempC* signal to connect *Reg4_1*'s output to both the *AND4_1* and *Inc4_1* components. We still need to connect the register's output to the output port *C*. The architecture makes this connection by specifying a process, named *outputC*, that is used to connect the output of the register to the output port *C*. The *outputC* process is sensitive to the signal *tempC*, previously used as an internal wire between the three components. Whenever *tempC* changes, which corresponds to a change in the up-counter's stored count, the *outputC* process assigns the new count to the output port *C*.

Verilog

Figure 9.32 is a Verilog description of a 4-bit up-counter. The module *UpCounter* defines the counter's inputs and outputs, consisting of a clock input *clk*, a count enable control input *cnt*, the 4-bit count value *C*, and a terminal count output *tc*.

The *UpCounter*'s module structurally describes the design using three components *Reg4*, *Inc4*, and *AND4*. *Reg4* is a 4-bit parallel load register with a load control input *ld*. *Inc4* is a 4-bit incrementer. *AND4* is a four-input AND gate that will output `1` if and only if all four inputs are `1`. The module further specifies two 4-bit wires, *tempC* and *incC*, used as internal wires within the structural description.

The module instantiates each of the three components and specifies the connections between them. *Reg4* is the only sequential component within the up-counter, and thus the *clk* input only needs to be connected to the clock input of the register. We control the up-counter's counting by connecting the count enable input, *cnt*, to the load enable, *ld*, of the register. The output *Q* of *Reg4_1* is connected to the internal signal *tempC*, which connects the register's output to both the *Inc4_1* and *AND4_1* components. *Inc4_1* receives the current count from the *tempC* connection and outputs the incremented count on its output *s*, which is

```
module Reg4(I, Q, clk, ld);
  input [3:0] I;
  input clk, ld;
  output [3:0] Q;
  // details not shown
endmodule

module Inc4(a, s);
  input [3:0] a;
  output [3:0] s;
  // details not shown
endmodule

module AND4(w,x,y,z,F);
  input w, x, y, z;
  output F;
  // details not shown
endmodule

module UpCounter(clk, cnt, C, tc);
  input clk, cnt;
  output [3:0] C;
  reg [3:0] C;
  output tc;

  wire [3:0] tempC;
  wire [3:0] incC;

  Reg4 Reg4_1(incC, tempC, clk, cnt);
  Inc4 Inc4_1(tempC, incC);
  AND4 AND4_1(tempC[3], tempC[2],
              tempC[1], tempC[0], tc);

  always @(tempC)
  begin
    C <= tempC;
  end
endmodule
```

Figure 9.32 Structural Verilog description of 4-bit up-counter.

connected to the other internal signal *incC*. The *incC* signal connects the incremented count from *Inc4_1* to the parallel load input *I* of *Reg4_1*. The current count is also connected to the four inputs of the *AND4_1* component. The *AND4_1*'s output *F* is then connected to the counter's terminal count output *tc*.

The *UpCounter* design must connect the output of the 4-bit register to the incrementer, the AND gate, and the counter's output port *C*. Therefore, the module uses the *tempC* signal to connect *Reg4_1*'s output to both the *AND4_1* and *Inc4_1* components.

We still need to connect the register's output to the output port *C*. The module makes this connection by specifying a procedure that is used to connect the output of the register to the output port *C*. The procedure is sensitive to the signal *tempC*, previously used as an internal wire between the three components. Whenever *tempC* changes, which corresponds to a change in the up-counter's stored count, the procedure assigns the new count to the output port *C*.

```
#include "systemc.h"
#include "reg4.h"
#include "inc4.h"
#include "and4.h"

SC_MODULE(UpCounter)
{
  sc_in<sc_logic> clk, cnt;
  sc_out<sc_lv<4> > C;
  sc_out<sc_logic> tc;

  sc_signal<sc_lv<4> > tempC, incC;
  sc_signal<sc_logic> tempC_b[4];

  Reg4 Reg4_1;
  Inc4 Inc4_1;
  AND4 AND4_1;

  SC_CTOR(UpCounter) : Reg4_1("Reg4_1"),
                       Inc4_1("Inc4_1"),
                       AND4_1("AND4_1")
  {
   Reg4_1.I(incC); Reg4_1.Q(tempC);
   Reg4_1.clk(clk); Reg4_1.ld(cnt);

   Inc4_1.a(tempC); Inc4_1.s(incC);

   AND4_1.w(tempC_b[0]); AND4_1.x(tempC_b[1]);
   AND4_1.y(tempC_b[2]); AND4_1.z(tempC_b[3]);
   AND4_1.F(tc);

   SC_METHOD(comblogic);
   sensitive << tempC;
  }

  void comblogic()
  {
    tempC_b[0] = tempC.read()[0];
    tempC_b[1] = tempC.read()[1];
    tempC_b[2] = tempC.read()[2];
    tempC_b[3] = tempC.read()[3];
    C.write(tempC);
  }
};
```

Figure 9.33 Structural SystemC description of 4-bit up-counter.

SystemC

Figure 9.33 is a SystemC description of a 4-bit up-counter. The SystemC module, named *UpCounter*, defines the counter's inputs and outputs, consisting of a clock input *clk*, a count enable control input *cnt*, the 4-bit count value *C*, and a terminal count output *tc*.

The *UpCounter's* module structurally describes the design consisting of three components *Reg4*, *Inc4*, and *AND4*. *Reg4* is a 4-bit parallel load register with a load control input *ld*. *Inc4* is a 4-bit incrementer. *AND4* is a four-input AND gate that will output `1` if and only if all four inputs are `1`. The module further specifies two 4-bit signals, *tempC*

▶ 9.5 RT

High-Lev

and *incC*, used as internal wires within the structural description. Additionally, the module defines a four-element array of *sc_logic* signals named *tempC_b* used to access the individual bits within the 4-bit vector *tempC*.

The module first instantiates each of the three components and then specifies the connections between them. *Reg4* is the only sequential component within the up-counter, and thus the *clk* input only needs to be connected to the clock input of the register. We control the up-counter's counting by connecting the count enable input, *cnt*, to the load enable, *ld*, of the register. The output *Q* of *Reg4_1* is connected to the internal signal *tempC*, which connects the register's output to *Inc4_1*. *Inc4_1* receives the current count from the *tempC* connection and outputs the incremented count on its output *s*, which is connected to the internal signal *incC*. The *incC* signal connects the incremented count from *Inc4_1* to the parallel load input *I* of *Reg4_1*. The current count is also connected to the four inputs of the *AND4_1* component using the *tempC_b* array to access the individual bits. The *AND4_1*'s output *F* is then connected to the counter's terminal count output *tc*.

The *UpCounter* design must connect the output of the 4-bit register to the incrementer, the AND gate, and the counter's output port *C*. Therefore, the module uses the *tempC* signal to connect *Reg4_1*'s output to the *Inc4_1* component and uses the *tempC_b* array to connect *Reg4_1*'s output to the *AND4_1* component. Thus, we still need to connect the register's output to the output port *C* and assign the individual bits of the register's output to the *tempC_b* array. The module makes these connections by defining a process, named *comblogic*, that is sensitive to the signal *tempC*. Whenever *tempC* changes, which corresponds to a change in the up-counter's stored count, the *comblogic* process assigns the new count to the output port *C*. Additionally, the process assigns the bits within vector *tempC* to the individual *sc_logic* signals within the *tempC_b* array. In order to access the individual bits of the vector signal *tempC*, we use the syntax, "`tempC.read()[0]`".

9.5 RTL DESIGN USING HARDWARE DESCRIPTION LANGUAGES

This section demonstrates how to create RTL descriptions using HDLs. HDLs will describe the starting point of RTL design, namely high-level state machines (HLSMs), and the ending point of RTL design, namely connected controllers and datapaths (processors). RTL designers will commonly create a testbench to test the HLSM description, and then use that same testbench for the controller/datapath description, thus helping to verify that the designer created a correct controller/datapath implementation.

High-Level State Machine of the Laser-Based Distance Measurer

This section shows how to create an HDL description for the laser-based distance measurer HLSM from Figure 5.12.

VHDL

Figure 9.34 presents a VHDL description of an HLSM for the laser-based distance measurer. Two new *ieee* library packages are used, *std_logic_arith* and *std_logic_unsigned*, which support arithmetic operations (like addition) on *std_logic_vector* items representing unsigned binary numbers. The entity, named *LaserDistMeasurer*, defines the

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity LaserDistMeasurer is
   port (
      clk, rst : in  std_logic;
      B, S     : in  std_logic;
      L        : out std_logic;
      D        : out std_logic_vector
                     (15 downto 0)
);
end LaserDistMeasurer;

architecture behavior of
      LaserDistMeasurer is

   type statetype is (S0,S1,S2,S3,S4);
   signal State, StateNext : statetype;

   signal Dctr, DctrNext:
        std_logic_vector(15 downto 0);
   signal Dreg, DregNext:
        std_logic_vector(15 downto 0);

   constant U_ZERO :
        std_logic_vector(15 downto 0)
              := "0000000000000000";
   constant U_ONE  :
        std_logic_vector(15 downto 0)
              := "0000000000000001";

begin

Regs: process(clk, rst)
begin
   if(rst = '1') then
      State <= S0;
      Dctr <= U_ZERO;
      Dreg <= U_ZERO;
   elsif(clk'event and clk='1') then
      State <= StateNext;
      Dctr  <= DctrNext;
      Dreg  <= DregNext;
   end if;
end process;
```

```
CombLogic: process(State, Dctr, B, S)
   begin
      case State is
         when S0 =>
            L <= '0'; -- laser off
            DregNext <= U_ZERO; --clr D
            DctrNext <= U_ZERO; --clr Dctr
            StateNext <= S1;
         when S1 =>
            DctrNext <= U_ZERO; --clr Dctr
            L <= '0'; -- laser off

            if(B = '1') then
               StateNext <= S2;
            else
               StateNext <= S1;
            end if;
         when S2 =>
            L <= '1';  --laser on
            DctrNext <= U_ZERO;
            StateNext <= S3;
         when S3 =>
            L <= '0';  -- laser off
            DctrNext <= Dctr + 1;

            if( S = '1') then
               StateNext <= S4;
            else
               StateNext <= S3;
            end if;
         when S4 =>
            DctrNext <= Dctr;
            DregNext <= SHR(Dctr, U_ONE);
            L <= '0';
            StateNext <= S1;
         when others =>
            DregNext <= U_ZERO;
            DctrNext <= U_ZERO;
            L <= '0';
            StateNext <= S0;
      end case;
   end process;

--assign Dreg output to D output
D <= Dreg;

end behavior;
```

Figure 9.34 Behavioral VHDL description of an HLSM of the laser-based distance measurer.

inputs and outputs, including a user-pressed button input *B*, a laser sensor input *S*, a laser control output *L*, and a 16-bit output *D* for the distance measured.

The ***use*** statement specifies which packages will be used within a design. The package *ieee.std_logic_arith* defines the *unsigned* type as well as a set of operations and functions that can be performed on *unsigned* values.

The entity also defines a clock input *clk* and reset input *rst*. We assume that the clock input is 300 MHz, as was assumed in the laser-based distance measurer design from

Chapter 5. We omit details of generating the 300 MHz oscillator (see Section 9.3 for an example of describing an oscillator).

The VHDL architecture describes the behavior of the entity. The description defines constants *U_ZERO* and *U_ONE* for 16-bit numbers 0 and 1, respectively. As was the case for an FSM, the description for the HLSM defines two signals for the state register: a current state signal *State*, and a next state signal *StateNext*. Furthermore, the description declares current and next signals for every other register too, thus declaring signals *Dctr* and *DctrNext*, and *Dreg* and *DregNext*.

As was the case for describing an FSM, the description for the HLSM uses two processes, one for all registers, and one for all combinational logic. The register process is sensitive to the reset and clock inputs, and has an asynchronous reset that clears all the current register signals. On a rising clock, the register process updates the current register signals with the next register signals.

The combinational logic process is sensitive to all items that it reads. The process consists of a case statement carrying out the HLSM's actions and transitions. The process reads the current register signals and writes the next register signals; current register signals are never written and next register signals are never read by the process.

The HLSM for the laser-based distance measurer performs two arithmetic operations, addition and shifting. Incrementing the counter signal *Dctr* in state *S3* is done using the syntax "*DctrNext <= Dctr + 1;*". State *S4* calculates the distance *D* by dividing the value of *Dctr* by 2. However, this division is achieved using a right-shift-by-one operation. Performing the shift and assigning the value to the output *D* is done using the statement "*DregNext <= SHR(Dctr, U_ONE);*". The function ***SHR()***, defined within the *ieee.std_logic_arith* package, shifts the first parameter *Dctr* by the amount specified by the second parameter *U_ONE*, where *U_ONE* is a constant defined earlier in the architecture.

Finally, the output of *Dreg* is permanently assigned to the output *D* using the statement "*D <= Dreg;*". Note that the statement is not contained inside a process. The statement is known as a ***concurrent signal assignment***. The statement executes whenever a signal on its right side changes.

The two-process approach to describing an HLSM can be thought of as follows. The combinational logic process computes the inputs of all registers by reading current register values, performing combinational operations like addition, and writing to the next register signals. When a rising clock arrives, the register process updates the current contents of all registers with the next register values.

Verilog

Figure 9.35 presents a Verilog description of an HLSM for the laser-based distance measurer. The module, named *LaserDistMeasurer*, defines the inputs and outputs, including a user-pressed button input *B*, a laser sensor input *S*, a laser control output *L*, and a 16-bit output *D* for the distance measured.

The module also defines a clock input *clk* and reset input *rst*. We assume that the clock input is 300 MHz, as was assumed in the laser-based distance measurer design in

```
module LaserDistMeasurer(clk,rst,B,S,L,D);
   input clk, rst, B, S;
   output L;
   output [15:0] D;
   reg L;
   reg [15:0] D;

   parameter    S0 = 3'b000,
                S1 = 3'b001,
                S2 = 3'b010,
                S3 = 3'b011,
                S4 = 3'b100;

   reg [2:0] State, StateNext;
   reg [15:0] Dctr, DctrNext;
   reg [15:0] Dreg, DregNext;

   //Registers
   always@(posedge clk, posedge rst) begin
      if(rst == 1) begin //asynchr. reset
         State <= S0;
         Dctr  <= 0;
         Dreg  <= 0;
      end
      else begin
         State <= StateNext;
         Dctr  <= DctrNext;
         Dreg  <= DregNext;
      end
   end

   always @(Dreg) begin
      D <= Dreg;
   end

   //Combinational logic
   always@(State, Dctr, B, S) begin
      case (State)
         S0: begin
            L <= 0; //Laser off
            DregNext <= 0; //clr D
            StateNext <= S1;
            DctrNext  <= 0;
         end
         S1: begin
            DctrNext <= 0;
            L <= 0;

            if(B == 1)
               StateNext <= S2;
            else
               StateNext <= S1;
         end
         S2: begin
            L <= 1;      //Laser on
            DctrNext <= 0;
            StateNext <= S3;
         end
         S3: begin
            L <= 0;      //Laser off
            DctrNext <= Dctr + 1;

            if(S == 1)
               StateNext <= S4;
            else
               StateNext <= S3;
         end
         S4: begin
            DregNext <= Dctr >> 1;
            StateNext <= S1;
         end
      endcase
   end
endmodule
```

Figure 9.35 Behavioral Verilog description of an HLSM of the laser-based distance measurer.

Chapter 5. We omit details of generating the 300 MHz clock (see Section 9.3 for an example of describing an oscillator).

The Verilog module behaviorally describes the *LaserDistMeasurer's* HLSM. As was the case for describing an FSM, the module declares two signals for the state register: *State* for the current state, and *StateNext* for the next state. Furthermore, the description declares current and next signals for every other register too, thus declaring *Dctr* and *DctrNext*, and *Dreg* and *DregNext*.

As was the case for describing an FSM, the description for the HLSM uses two procedures, one for all registers, and one for all combinational logic. The register procedure is sensitive to the reset and clock inputs, and has an asynchronous reset that clears all the current register signals. On a rising clock edge, the register procedure updates the current register signals with the next register signals.

The combinational logic procedure is sensitive to all items that it reads. The procedure consists of a case statement carrying out the HLSM's actions and transitions. The procedure reads the current register signals and writes the next register signals; current register signals are never written and next register signals are never read by the procedure.

The HLSM for the laser-based distance measurer performs two arithmetic operations, addition and shifting. Incrementing the counter signal *Dctr* in state *S3* is done using the syntax "*DctrNext <= Dctr + 1;*". State *S4* calculates the distance *D* by dividing the value of *Dctr* by 2. However, this division is achieved using a right-shift-by-one operation. Performing the shift and assigning the value to the output *D* is done using the statement "*DregNext <= Dctr >> 1*". The operator ">>" shifts the value of left parameter *Dctr* by the amount specified by the right parameter 1.

Finally, the output of *Dreg* is permanently assigned to the output *D* using an *always* procedure sensitive to *Dreg* and executing the statement "*D <= Dreg;*". Thus *D* is updated whenever *Dreg* changes, so *D* will always equal *Dreg*.

The two-procedure approach to describing an HLSM can be thought of as follows. The combinational logic procedure computes the inputs of all registers by reading current register values, performing combinational operations like addition, and writing to the next register signals. When a rising clock arrives, the register procedure updates the current contents of all registers with the next register values.

SystemC

Figure 9.36 presents a SystemC description of an HLSM for the laser-based distance measurer. The module, named *LaserDistMeasurer*, defines the inputs and outputs, including a user-pressed button input *B*, a laser sensor input *S*, a laser control output *L*, and a 16-bit output *D* for the distance measured.

The module also defines a clock input *clk* and reset input *rst*. We assume that the clock input is 300 MHz, as was assumed in the laser-based distance measurer design in Chapter 5. We omit details of generating the 300 MHz clock (see Section 9.3 for an example of describing an oscillator).

The SystemC module behaviorally describes the *LaserDistMeasurer's* HLSM. As was the case for describing an FSM, the module declares two signals for the state register, *State* for the current state, and *StateNext* for the next state. Furthermore, the description declares current and next signals for every other register too, thus declaring *Dctr* and *DctrNext*, and *Dreg* and *DregNext*.

As was the case for describing an FSM, the description for the HLSM uses two processes, one for all registers, and one for all combinational logic. The register process is sensitive to the reset and clock inputs, and has an asynchronous reset that clears all the current register signals. On a rising clock, the register process updates the current register signals with the next register signals.

The combinational logic process is sensitive to all items that it reads. The process consists of a case statement carrying out the HLSM's actions and transitions. The procedure reads the current register signals and writes the next register signals; current register signals are never written and next register signals are never read by the procedure.

The HLSM for the laser-based distance measurer performs two arithmetic operations, addition and shifting. Incrementing the counter signal *Dctr* in state *S3* is done using the syntax "*DctrNext.write(Dctr.read() + 1);*". State *S4* calculates the distance *D* by dividing

```
#include "systemc.h"

enum statetype { S0, S1, S2, S3, S4 };

SC_MODULE(LaserDistMeasurer)
{
   sc_in<sc_logic> clk, rst;
   sc_in<sc_logic> B, S;
   sc_out<sc_logic> L;
   sc_out<sc_uint<16> > D;

   sc_signal<statetype> State, StateNext;
   sc_signal<sc_uint<16> > Dctr, DctrNext;
   sc_signal<sc_uint<16> > Dreg, DregNext;

   SC_CTOR(LaserDistMeasurer)
   {
      SC_METHOD(Regs);
      sensitive << clk.pos();

      SC_METHOD(CombLogic);
      sensitive << State << B << S << Dctr;

      SC_METHOD(Output);
      sensitive << Dreg;
   }

   void Regs(){
      if (rst.read() == SC_LOGIC_1){
         State.write(S0);
         Dctr.write(0);
         Dreg.write(0);
      }
      else{
         State.write(StateNext.read());
         Dctr.write(DctrNext.read());
         Dreg.write(DregNext.read());
      }
   }
```

```
void CombLogic(){
   switch (State) {
      case S0:
         L.write(SC_LOGIC_0);//laser off
         StateNext.write(S1);
         break;
      case S1:
         DctrNext.write(0); // clr count
         if (B.read() == SC_LOGIC_1){
            StateNext.write(S2);
         }
         break;
      case S2:
         L.write(SC_LOGIC_1);//laser on
         StateNext.write(S3);
         break;
      case S3:
         L.write(SC_LOGIC_0);//laser off
         DctrNext.write(Dctr.read()+1);
         if (S.read() == SC_LOGIC_1)
            StateNext.write(S4);
         else
            StateNext.write(S3);
         break;
      case S4:
         DregNext.write
            ((Dctr.read()>>1));
         StateNext.write(S1);
         break;
   }
}

void Output(){
   D.write(Dreg.read());
}

};
```

Figure 9.36 Behavioral SystemC description of an HLSM of the laser-based distance measurer.

the value of *Dctr* by 2. However, this division is achieved using a right-shift-by-one operation. Performing the shift and assigning the value to the output *D* is done using the statement "*DregNext.write(Dctr.read()>>1);*". The operator ">>" shifts the value of the left parameter *Dctr.read()* by the amount specified by the right parameter 1.

Finally, the output of *Dreg* is permanently assigned to the output *D* using a process sensitive to *Dreg* and executing the statement "*D.write(Dreg.read());*". Thus *D* is updated whenever *Dreg* changes, so *D* will always equal *Dreg*.

The two-process approach to describing an HLSM can be thought of as follows. The combinational logic process computes the inputs of all registers by reading current register values, performing combinational operations like addition, and writing to the next register signals. When a rising clock arrives, the register process updates the current contents of all registers with the next register values.

Controller and Datapath of the Laser-Based Distance Measurer

VHDL

Figure 9.37 is a VHDL description of the laser-based distance measurer controller/datapath from Figure 5.24. The entity, named *LaserDistMeasure* defines the inputs and outputs, including a user-pressed button input *B*, a laser sensor input *S*, a laser control output *L*, and a 16-bit output *D* for the distance measured. The entity also defines a 300 MHz clock input *clk* and reset input *rst* for the design's controller.

The *LaserDistMeasurer*'s architecture structurally describes the connections of the controller and datapath components. The architecture instantiates two components. *LDM_Controller_1* is the controller for the laser-based distance measurer, and *LDM_Datapath_1* is the datapath for this design. The architecture connects the entity's *clk*, *rst*, *B*, and *S* inputs to the inputs of *LDM_Controller_1* and connects the controller's laser control output to the corresponding output port *L*. Additionally, the four signals *Dreg_clr*, *Dreg_ld*, *Dctr_clr*, and *Dctr_ld* connect the controller's four control signals to the four inputs of *LDM_Datapath_1*. The *LaserDistMeasurer* datapath has a single output *D*, providing the distance measured, that is connected to the output port *D* of the entity.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity LaserDistMeasurer is
    port (
        clk, rst : in  std_logic;
        B, S     : in  std_logic;
        L        : out  std_logic;
        D        : out
           std_logic_vector(15 downto 0)
        );
end LaserDistMeasurer;

architecture structure of LaserDistMeasurer is
   component LDM_Controller
   port ( clk, rst : in std_logic;
          B, S     : in std_logic;
          L        : out std_logic;
          Dreg_clr, Dreg_ld : out std_logic;
          Dctr_clr, Dctr_ld : out std_logic
        );
   end component;

   component LDM_Datapath
   port ( clk : in std_logic;
          Dreg_clr, Dreg_ld : in std_logic;
          Dctr_clr, Dctr_ld : in std_logic;
          D : out std_logic_vector(15 downto 0)
        );
   end component;

   signal Dreg_clr, Dreg_ld : std_logic;
   signal Dctr_clr, Dctr_ld : std_logic;

  begin
    LDM_Controller_1 : LDM_Controller
      port map ( clk, rst, B, S, L,
                 Dreg_clr, Dreg_ld, Dctr_clr,
                    Dctr_ld);

    LDM_Datapath_1 : LDM_Datapath
      port map ( clk, Dreg_clr, Dreg_ld,
                 Dctr_clr, Dctr_ld, D);
end structure;
```

Figure 9.37 Structural description of top-level VHDL description of laser-based distance measurer.

Figure 9.38 is a VHDL description of the *LaserDistMeasurer's* datapath component shown in Figure 5.23. The entity, named *LDM_Datapath*, defines a clock input *clk*, four control inputs *Dreg_clr*, *Dreg_ld*, *Dctr_clr*, and *Dctr_ld*, and a 16-bit distance output *D*.

The architecture defines three components, a 16-bit register, a 16-bit right shifter that shifts right by one position, and a 16-bit adder. The architecture instantiates a register component named *Dctr*, an adder component named *Add1*, a shifter component *ShiftRight*, and another register *Dreg*. *Dctr*'s instantiation connects the datapath's *Dctr_clr* and *Dctr_ld* inputs to *Dctr*'s clear and load control inputs. *Dctr*'s output *Q* is

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity LDM_Datapath is
   port ( clk : in std_logic;
          Dreg_clr, Dreg_ld : in std_logic;
          Dctr_clr, Dctr_ld : in std_logic;
          D : out std_logic_vector(15 downto 0)
        );
end LDM_Datapath;

architecture structure of LDM_Datapath is

   component Reg16
      port ( I : in std_logic_vector(15 downto 0);
             Q : out std_logic_vector(15 downto 0);
             clk, clr, ld : in std_logic
           );
   end component;
   component ShiftR1_16
      port ( I: in std_logic_vector(15 downto 0);
             S: out std_logic_vector(15 downto 0)
           );
   end component;
   component Add16
      port ( A, B: in std_logic_vector(15 downto 0);
             S: out std_logic_vector(15 downto 0)
           );
   end component;

   signal tempC:  std_logic_vector(15 downto 0);
   signal addC :  std_logic_vector(15 downto 0);
   signal shiftC : std_logic_vector(15 downto 0);

   constant U_ONE: std_logic_vector(15 downto 0)
      := "0000000000000001";

   begin
   Dctr: Reg16
      port map(addC, tempC, clk, Dctr_clr, Dctr_ld);
   Add1 : Add16
      port map(U_ONE, tempC, addC);
   ShiftRight: ShiftR1_16
      port map(tempC, shiftC);
   Dreg: Reg16
      port map(shiftC, D, clk, Dreg_clr, Dreg_ld);

end structure;
```

Figure 9.38 Structural VHDL description of the laser-based distance measurer's datapath.

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

entity LDM_Controller is
   port ( clk, rst: in std_logic;
          B, S: in std_logic;
          L: out std_logic;
          Dreg_clr, Dreg_ld: out std_logic;
          Dctr_clr, Dctr_ld: out std_logic
        );
end LDM_Controller;

architecture behavior of LDM_Controller is

type statetype is (S0, S1, S2, S3, S4);
signal currentstate, nextstate: statetype;

begin
statereg: process(clk, rst)
begin
   if (rst='1') then
      currentstate <= S0; -- initial state
   elsif (clk='1' and clk'event) then
      currentstate <= nextstate;
    end if;
end process;
```

```
comblogic: process(currentstate, B, S)
begin
   L <= '0';
   Dreg_clr <= '0';
   Dreg_ld <= '0';
   Dctr_clr <= '0';
   Dctr_ld <= '0';
   case currentstate is
      when S0 =>
         L <= '0'; -- laser off
         Dreg_clr <= '1'; -- clr Dreg
         nextstate <= S1;
      when S1 =>
         Dctr_clr <= '1'; -- clr count
         if (B='1') then
            nextstate <= S2;
         else
            nextstate <= S1;
         end if;
      when S2 =>
         L <= '1'; -- laser on
         nextstate <= S3;
      when S3 =>
         L <= '0'; -- laser off
         Dctr_ld <= '1'; -- count up
         if (S='1') then
            nextstate <= S4;
         else
            nextstate <= S3;
         end if;
      when S4 =>
         Dreg_ld <= '1'; -- load Dreg
         Dctr_ld <= '0'; -- stop count
         nextstate <= S1;
   end case;
end process;

end behavior;
```

Figure 9.39 Behavioral VHDL description of laser-based distance measurer's controller.

then connected to the architecture's internal signal *tempC* that connects the count value to the *ShiftRight* shifter's input. The shifted count is then connected to the input of the *Dreg* register using the internal signal *shiftC*. The instantiation of the *Dreg* register connects the register's clear and load control inputs to the datapath's *Dreg_clr* and *Dreg_ld* input ports. Finally, the register's data output *Q* is connected to *LDM_datapath*'s measured distance output *D*.

Figure 9.39 is the VHDL description of the laser-based distance measurer's FSM controller described in Figure 5.26. The entity, named *LDM_Controller*, defines a clock input *clk*, a reset signal *rst*, a user-pressed button input *B*, a laser sensor input *S*, and five output control signals, *L, Dreg_clr, Dreg_ld, Dctr_clr,* and *Dctr_ld*. The output *L* is used to turn the laser on and off, where if *L* is `1`, the laser is on. The four other output signals are used to control the RTL design's datapath components.

The VHDL architecture describes the behavior of the entity. Similar to the controller design shown in Figure 9.22, the architecture consists of two processes, one modeling the

state register, the other modeling the combinational logic. The state register process, named *statereg*, is sensitive to inputs *clk* and *rst*. If *rst* is `1`, then the process asynchronously sets the *currentstate* signal to the FSM's initial state, *S0*. Otherwise, if the clock is rising, the process updates the state register with the next state.

The second process, named *comblogic*, is sensitive to the inputs to the combinational logic, namely, the external inputs *B* and *S*, and the state register output *currentstate*. When either of those items change, the process sets the FSM's outputs—in this case *L, Dreg_clr, Dreg_ld, Dctr_clr,* and *Dctr_ld*—with the appropriate value for the current state. In the controller example of Figure 9.22, the FSM's output *x* was defined within the case statement for all possible states. With five outputs that must be defined in the *LDM_Controller* and five possible states, assigning the values to all outputs in each state would be cumbersome. Furthermore, finding a mistake and making corrections or modifications to the controller would become difficult in a larger FSM consisting of more states and having many more outputs. The *comblogic* process uses a different approach, in which a default value for the outputs is first assigned and only the deviations from the defaults are assigned later. The *comblogic* process first assigns a default value of `0` to all five outputs. The process then evaluates the current state and assigns the values to the outputs only when the output should be `1`. The process also assigns the value `0` to several signals when these assignments are needed to clearly indicate the behavior of the controller (they are redundant but help make the description easier to understand).

The process also determines what the next state should be, based on the current state and the values of inputs *B* and *S*. The next state will be loaded into the state register by the state register process on the next rising clock edge.

Verilog

Figure 9.40 is a Verilog description of the laser-based distance measurer controller/datapath from Figure 5.24. The module, named *LaserDistMeasurer*, defines the inputs and outputs, including a user-pressed button input *B*, a laser sensor input *S*, a laser control output *L*, and a 16-bit output *D* for the distance measured. The module also defines a 300 MHz clock input *clk* and reset input *rst* for the design's controller.

```
module LaserDistMeasurer(clk,rst,B,S,L,D);
   input clk, rst, B, S;
   output L;
   output [15:0] D;

   wire Dreg_clr, Dreg_ld;
   wire Dctr_clr, Dctr_ld;

   LDM_Controller
      LDM_Controller_1(clk, rst, B, S, L,
                       Dreg_clr, Dreg_ld,
                       Dctr_clr, Dctr_ld);
   LDM_Datapath
      LDM_Datapath_1(clk, Dreg_clr, Dreg_ld,
                     Dctr_clr, Dctr_ld, D);
endmodule
```

Figure 9.40 Structural description of top-level Verilog description of laser-bsed distance measurer.

The *LaserDistMeasurer* structurally describes the connections of the controller and datapath components. The module instantiates two components. *LDM_Controller_1* is the controller for the laser-based distance measurer, and *LDM_Datapath_1* is the datapath for this design. The architecture connects the module's *clk*, *rst*, *B*, and *S* inputs to the inputs of *LDM_Controller_1* and connects the controller's laser control output to the corresponding output port *L*. Additionally, the four internal wires, *Dreg_clr*, *Dreg_ld*,

Dctr_clr, and *Dctr_ld*, connect the controller's four control signals to the four inputs of *LDM_Datapath_1*. The *LaserDistMeasurer* datapath has a single output *D*, providing the distance measured, that is connected to the output port *D* of the module.

Figure 9.41 is a Verilog description of the *LaserDistMeasurer's* datapath component shown in Figure 5.23. The module, named *LDM_Datapath*, defines a clock input *clk*, four control inputs *Dreg_clr*, *Dreg_ld*, *Dctr_clr*, and *Dctr_ld*, and a 16-bit distance output *D*.

```
module Add16(A, B, S);
   input [15:0] A, B;
   output [15:0] S;
   //details not shown
endmodule

module Reg16(I, Q, clk, clr, ld);
   input [15:0] I;
   input clk, clr, ld;
   output [15:0] Q;
   // details not shown
endmodule

module ShiftR1_16(I, S);
   input [15:0] I;
   output [15:0] S;
   // details not shown
endmodule

module LDM_Datapath(clk, Dreg_clr, Dreg_ld,
                    Dctr_clr, Dctr_ld, D);
   input clk;
   input Dreg_clr, Dreg_ld;
   input Dctr_clr, Dctr_ld;
   output [15:0] D;

   wire [15:0] addC, tempC, shiftC;

   Reg16 Dctr(addC,tempC,clk,Dctr_clr,Dctr_ld);
   Add16 Add1(1, tempC, addC);
   ShiftR1_16 ShiftRight(tempC, shiftC);
   Reg16 Dreg(shiftC,D,clk,Dreg_clr,Dreg_ld);
endmodule
```

Figure 9.41 Structural Verilog description of the laser-based distance measurer's datapath.

The datapath defines three components, a 16-bit register, a 16-bit adder, and a 16-bit right shifter that shifts right by one position. The datapath module instantiates a register named *Dctr*, an adder named *Add1*, a shifter named *ShiftRight*, and another register named *Dreg*. The module connects the datapath's *Dctr_clr* and *Dctr_ld* inputs to *Dctr*'s clear and load control inputs, respectively. The counter's count output *C* is then connected to the 16-bit internal wire *tempC* that connects the count value to the *ShiftRight* shifter's input. The shifted count is then connected to the input of the *Dreg* register using the internal 16-bit wire *shiftC*. The module connects the *Dreg* register's clear and load control inputs to the datapath's *Dreg_clr* and *Dreg_ld* input ports. Finally, the register's data output *Q* is connected to *LDM_datapath's* measured distance output *D*.

Figure 9.42 is the Verilog description of the laser-based distance measurer's FSM controller described in Figure 5.26. The module, named *LDM_Controller*, defines a clock input *clk*, a reset signal *rst*, a user-pressed button input *B*, a laser sensor input *S*, and five output control signals, *L, Dreg_clr, Dreg_ld, Dctr_clr,* and *Dctr_ld*. The output *L* is used to turn the laser on and off, where if *L* is `1`, the laser is on. The four other output signals are used to control the RTL design's datapath components.

The Verilog module behaviorally describes the *LaserDistMeasurer*'s FSM. Similar to the controller design shown in Figure 9.23, the module consists of two procedures, one

```
module LDM_Controller
   (clk, rst, B, S, L, Dreg_clk,
    Dreg_ld, Dctr_clr, Dctr_ld);
   input clk, rst, B, S;
   output L;
   output Dreg_clk, Dreg_ld;
   output Dctr_clr, Dctr_ld;
   reg L;
   reg Dreg_clr, Dreg_ld;
   reg Dctr_clr, Dctr_ld;

   parameter S0 = 3'b000,
             S1 = 3'b001,
             S2 = 3'b010,
             S3 = 3'b011,
             S4 = 3'b100;

   reg [2:0] State;
   reg [2:0] StateNext;

   always @(posedge rst or posedge clk)
   begin
      if (rst==1)
         State <= S0; // initial state
      else
         State <= StateNext;
   end

   always @(State or B or S)
     begin
         L <= 0;
         Dreg_clr <= 0;
         Dreg_ld <= 0;
         Dctr_clr <= 0;
         Dctr_ld <= 0;
         case (State)
            S0: begin
               L <= 0; // laser off
               Dreg_clr <= 1; // clr Dreg
               StateNext <= S1;
            end
            S1: begin
               Dctr_clr <= 1; // clr count
               if (B==1)
                  StateNext <= S2;
               else
                  StateNext <= S1;
            end
            S2: begin
               L <= 1; // laser on
               StateNext <= S3;
            end
            S3: begin
               L <= 0; // laser off
               Dctr_ld <= 1; // count up
               if (S==1)
                  StateNext <= S4;
               else
                  StateNext <= S3;
            end
            S4: begin
               Dreg_ld <= 1; // load Dreg
               Dctr_ld <= 0; // stop count
               StateNext <= S1;
            end
         endcase
   end
endmodule
```

Figure 9.42 Behavioral Verilog description of laser-based distance measurer's controller.

modeling the state register, the other modeling the FSM's control logic. The state register procedure is sensitive to the positive edge of the reset input, *rst*, and the positive edge of the clock input, *clk*. If the *rst* input is enabled, then the procedure asynchronously sets the *currentstate* signal to the FSM's initial state, *S0*. Otherwise, on the rising edge of the clock, the procedure updates the state register with the next state.

The second procedure is sensitive to the inputs to the combinational logic, namely, the external inputs *B* and *S*, and the state register output *currentstate*. When either of those items change, the procedure sets the FSM's outputs, in this case *L, Dreg_clr, Dreg_ld, Dctr_clr,* and *Dctr_ld*, with the appropriate value for the current state. In the controller example of Figure 9.22, the FSM's output *x* was defined within the case statement for all possible states. With five outputs that must be defined in the *LDM_Controller* and five possible states, assigning the values to all outputs in each state would be cumbersome. Furthermore, finding a mistake and making corrections or modifications to the controller would become very difficult in a larger FSM consisting of more states and having many more outputs. Instead, the procedure uses a different approach, in which a

default value for all the outputs is first assigned and only the deviations from the defaults are assigned later. The procedure first assigns a default value of `0` to all five outputs. The procedure then evaluates the current state and assigns the values to the outputs only when the output should be `1`. The procedure also assigns the value `0` to several signals within the *case* statements to clearly indicate the behavior of the controller (those assignments are redundant but help make the description easier to understand).

The procedure also determines what the next state should be, based on the current state and the values of inputs *B* and *S*. The next state will be loaded into the state register by the state register procedure on the next positive clock edge.

SystemC

Figure 9.43 is a SystemC description of the laser-based distance measurer shown in Figure 5.24. The module, named *LaserDistMeasurer*, defines the inputs and outputs, including a user-pressed button input *B*, a laser sensor input *S*, a laser control output *L*, and a 16-bit output *D* for the distance measured. The module also defines a 300 MHz clock input *clk* and reset input *rst* for the design's controller.

```
#include "systemc.h"
#include "LDM_Controller.h"
#include "LDM_Datapath.h"

SC_MODULE(LaserDistMeasurer)
{
   sc_in<sc_logic> clk, rst;
   sc_in<sc_logic> B, S;
   sc_out<sc_logic> L;
   sc_out<sc_uint<16> > D;

   sc_signal<sc_logic> Dreg_clr, Dreg_ld;
   sc_signal<sc_logic> Dctr_clr, Dctr_ld;

   LDM_Controller LDM_Controller_1;
   LDM_Datapath LDM_Datapath_1;

   SC_CTOR(LaserDistMeasurer) :
      LDM_Controller_1("LDM_Controller_1"),
      LDM_Datapath_1("LDM_Datapath_1")
   {
      LDM_Controller_1.clk(clk);
      LDM_Controller_1.rst(rst);
      LDM_Controller_1.B(B);
      LDM_Controller_1.S(S);
      LDM_Controller_1.Dreg_clr(Dreg_clr);
      LDM_Controller_1.Dreg_ld(Dreg_ld);
      LDM_Controller_1.Dctr_clr(Dctr_clr);
      LDM_Controller_1.Dctr_ld(Dctr_ld);
      LDM_Datapath_1.clk(clk);
      LDM_Datapath_1.Dreg_clr(Dreg_clr);
      LDM_Datapath_1.Dreg_ld(Dreg_ld);
      LDM_Datapath_1.Dctr_clr(Dctr_clr);
      LDM_Datapath_1.Dctr_ld(Dctr_ld);
      LDM_Datapath_1.D(D);
   }
};
```

Figure 9.43 Structural description of top-level SystemC description of laser-based distance measurer.

The *LaserDistMeasurer* structurally describes the connections of the controller and datapath components. The architecture instantiates two components. *LDM_Controller_1* is the controller for the laser-based distance measurer, and *LDM_Datapath_1* is the datapath for this design. The module connects the module's *clk*, *rst*, *B*, and *S* inputs to the inputs of *LDM_ Controller_1* and connects the controller's laser control output to the corresponding output port *L*. Additionally, the four internal wires, *Dreg_clr*, *Dreg_ld*, *Dctr_clr*, and *Dctr_ld*, connect the controller's four control signals to the four inputs of *LDM_Data-path_1*. The *LaserDist-*

Measurer datapath has a single output *D*, providing the distance measured, that is connected to the output port *D* of the module.

Figure 9.44 is a SystemC description of the *LaserDistMeasurer's* datapath component shown in Figure 5.23. The module, named *LDM_Datapath*, defines a clock input *clk*, four control inputs *Dreg_clr*, *Dreg_ld*, *Dctr_clr*, and *Dctr_ld*, and a 16-bit distance output *D*.

The datapath includes three components, a 16-bit adder, a 16-bit register, and a 16-bit right shifter that shifts right by one position. The datapath module instantiates an adder named *Add1*, a register named *Dctr*, a register named *Dreg*, and a shifter named *ShiftRight*. The module connects the datapath's *Dctr_clr* and *Dctr_ld* inputs to *Dctr*'s clear and load control inputs, respectively. The counter's count output *C* is then connected to the 16-bit internal signal *tempC* that connects the count value to the *ShiftRight* shifter's input. The shifted count value is then connected to the input of the *Dreg* register using the internal signal *shiftC*. The module connects the *Dreg* register's clear and load control inputs to the datapath's *Dreg_clr* and *Dreg_ld* input ports. Finally, the register's data output *Q* is connected to *LDM_datapath*'s measured distance output *D*.

```
#include "systemc.h"
#include "add16.h"
#include "reg16.h"
#include "shiftr1_16.h"

SC_MODULE(LDM_Datapath)
{
   sc_in<sc_logic> clk;
   sc_in<sc_logic> Dreg_clr, Dreg_ld;
   sc_in<sc_logic> Dctr_clr, Dctr_ld;
   sc_out<sc_uint<16> > D;

   sc_signal<sc_uint<16> > tempC;
   sc_signal<sc_uint<16> > addC;
   sc_signal<sc_uint<16> > shiftC;

   Add16 Add1;
   Reg16 Dctr;
   Reg16 Dreg;
   ShiftR1_16 ShiftRight;

   SC_CTOR(LDM_Datapath) :
      Dctr("Dctr"),
      Dreg("Dreg"),
      Add1("Add1"),
      ShiftRight("ShiftRight")
   {
      Add1.A(1);
      Add1.B(tempC);
      Add1.B(addC);

      Dctr.I(addC);
      Dctr.Q(tempC);
      Dctr.clk(clk);
      Dctr.clr(Dctr_clr);
      Dctr.ld(Dctr_ld);

      ShiftRight.I(tempC);
      ShiftRight.S(shiftC);

      Dreg.I(shiftC);
      Dreg.Q(D);
      Dreg.clk(clk);
      Dreg.clr(Dreg_clr);
      Dreg.ld(Dreg_ld);
   }
};
```

Figure 9.44 Structural SystemC description of the laser-based distance measurer's datapath.

```
#include "systemc.h"

enum statetype { S0, S1, S2, S3, S4 };
SC_MODULE(LDM_Controller)
{
   sc_in<sc_logic> clk, rst, B, S;
   sc_out<sc_logic> L;
   sc_out<sc_logic> Dreg_clr, Dreg_ld;
   sc_out<sc_logic> Dctr_clr, Dctr_ld;

   sc_signal<statetype> State, StateNext;

   SC_CTOR(LDM_Controller)
   {
      SC_METHOD(statereg);
      sensitive << clk.pos();

      SC_METHOD(comblogic);
      sensitive << State << B << S;
   }

   void statereg() {
      if ( rst.read() == SC_LOGIC_1 )
         State = S0; // initial state
      else
         State = StateNext;
   }
```

```
   void comblogic() {
      L.write(SC_LOGIC_0);
      Dreg_clr.write(SC_LOGIC_0);
      Dreg_ld.write(SC_LOGIC_0);
      Dctr_clr.write(SC_LOGIC_0);
      Dctr_ld.write(SC_LOGIC_0);

      switch (State) {
         case S0:
            L.write(SC_LOGIC_0); // laser off
            Dreg_clr.write(SC_LOGIC_0);
            StateNext.write(S1);
            break;
         case S1:
            Dctr_clr.write(SC_LOGIC_1);
            if (B.read() == SC_LOGIC_1)
               StateNext.write(S2);
            else
               StateNext.write(S1);
            break;
         case S2:
            L.write(SC_LOGIC_1); // laser on
            StateNext.write(S3);
            break;
         case S3:
            L.write(SC_LOGIC_0); // laser off
            Dctr_ld.write(SC_LOGIC_1);
            if (S.read() == SC_LOGIC_1)
               StateNext.write(S4);
            else
               StateNext.write(S3);
            break;
         case S4:
            Dreg_ld.write(SC_LOGIC_1);
            Dctr_ld.write(SC_LOGIC_0);
            StateNext.write(S1);
            break;
      }
   }
};
```

Figure 9.45 Behavioral SystemC description of laser-based distance measurer's controller.

Figure 9.45 is the SystemC description of the laser-based distance measurer's FSM controller described in Figure 5.26. The module, named *LDM_Controller*, has a clock input *clk*, a reset signal *rst*, a user-pressed button input *B*, a laser sensor input *S*, and five output control signals, *L, Dreg_clr, Dreg_ld, Dctr_clr,* and *Dctr_ld*. The output *L* is used to turn the laser on and off; where is *L* is 1, the laser is on. The four other output signals are used to control the RTL design's datapath components.

The SystemC module behaviorally describes the *LaserDistMeasurer's* FSM. Similar to the controller design shown in Figure 9.24, the module consists of two processes, one modeling the state register, the other modeling the FSM's control logic. The state register process, named *statereg*, is sensitive to the positive edge of the reset input *rst* and the positive edge of the clock input *clk*. If the *rst* is enabled, then the process asynchronously sets

the *currentstate* to the FSM's initial state, *S0*. Otherwise, on the rising edge of the clock, the process updates the state register with the *nextstate*.

The second process, named *comblogic*, is sensitive to the inputs to the combinational logic, namely, the external inputs *B* and *S*, and the state register output *currentstate*. When either of those signals changes, the process sets the FSM's outputs, in this case *L, Dreg_clr, Dreg_ld, Dctr_clr,* and *Dctr_ld*, with the appropriate value for the current state. In the controller example of Figure 9.24, the FSM's output *x* was defined within the case statement for all possible states. With five outputs that we must define in the *LDM_Controller* and five possible states, assigning the values to all outputs in each state would be cumbersome. Furthermore, finding a mistake and making corrections or modification to the controller would become very difficult in a larger FSM consisting of more states and having many more outputs. Instead, the process uses a different approach, in which a default value for the all outputs is first assigned and only the deviations from the defaults are assigned later. The process first assigns a default value of `0` to all five outputs. The process then evaluates the current state and assigns the values to the outputs only when the output should be `1`. The process also assigns the value `0` to several signals within the ***case*** statements; however, these assignments are included only to clearly indicate the behavior of the controller (they are redundant but help make the description easier to understand).

The process also determines what the next state should be, based on the current state and the values of inputs *B* and *S*. The next state will be loaded into the state register by the state register process on the next positive clock edge.

▶ 9.6 CHAPTER SUMMARY

This chapter stated that hardware description languages (HDLs) are widely used in modern digital design. The chapter introduced three popular HDLs: VHDL, Verilog, and SystemC. The chapter introduced those HDLs primarily through the use of examples, illustrating how each HDL might be used to describe combinational logic, sequential logic, datapath components, and RTL behavior and structure. To become proficient at the use of HDLs, a more thorough study of a particular HDL might be helpful. This chapter also illustrated the point that the three different HDLs have several aspects in common.

▶ 9.7 EXERCISES

The following exercises can be completed using any of the HDLs described in this chapter. The solution to Exercise 9.1 is needed in order to solve many of the remaining exercises.

SECTION 9.2: COMBINATIONAL LOGIC DESCRIPTION USING HARDWARE DESCRIPTION LANGUAGES

9.1 (a) Create combinational behavioral HDL descriptions of *Inv*, *AND2*, *AND3*, *OR2*, and *OR3* gates, such that those gates can then be used as components in another design. Use names *a*, *b*, and *c* for inputs (as needed) and *F* for the output. (b) Create a testbench for the *AND3* gate to test its correctness.