6 VHDL Design of Regular (Category 1) State Machines

6.1 Introduction

This chapter presents several VHDL designs of category 1 state machines. It starts by presenting two VHDL templates, for Moore- and Mealy-based implementations, which are used subsequently to develop a series of designs related to the examples introduced in chapter 5.

The codes are always complete (not only partial sketches) and are accompanied by comments and simulation results, illustrating the design's main features. All circuits were synthesized using Quartus II (from Altera) or ISE (from Xilinx). The simulations were performed with Quartus II or ModelSim (from Mentor Graphics). The default encoding scheme for the states of the FSMs was regular sequential encoding (see encoding options in section 3.7; see ways of selecting the encoding scheme at the end of section 6.3).

The same designs will be presented in chapter 7 using SystemVerilog, so the reader can make a direct comparison between the codes.

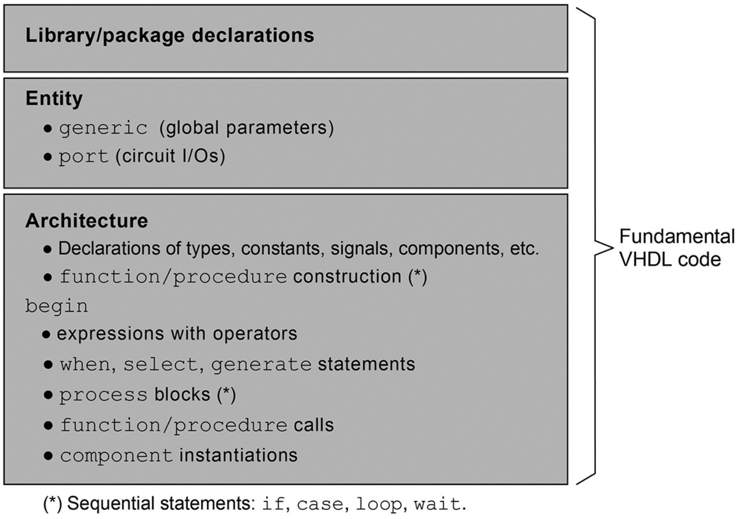
Note: See suggestions of VHDL books in the bibliography.

6.2 General Structure of VHDL Code

A typical structure of VHDL code for synthesis, with all elements that are needed in this and in coming chapters, is depicted in figure 6.1. It is composed of three fundamental sections, briefly described below.

Library/Package Declarations

As the name says, it contains the libraries and corresponding packages needed in the design. The most common package is *std_logic_1164*, from the IEEE library, which defines the types *std_logic* (for single bit) and *std_logic_vector* (for multiple bits), which are the industry standard.



Library/package declarations
Entity
• generic (global parameters)
• port (circuit I/Os)
Architecture
• Declarations of types, constants, signals, components, etc.
• function/procedure construction (*)
begin
• expressions with operators
• when, select, generate statements
• process blocks (*)
• function/procedure calls
• component instantiations
(*) Sequential statements: if, case, loop, wait.
Fundamental VHDL code

Figure 6.1
Typical VHDL code structure for synthesis.

Entity
The entity is divided into two main parts, called **generic** and **port**.

Generic: This portion is optional. It is used for the declaration of global parameters, which can be easily modified to fulfill different system specifications or, more importantly, can be overridden during instantiations (using the **component** construct) into other designs.
Port: This part of the code is mandatory for synthesis. It is just a list with specifications of all circuit ports (I/Os), including their name, mode (**in**, **out**, **inout**, or **buffer**), and type (plus range).

Architecture
The architecture too is divided into two parts, called *declarative part* and *statements part*.

Declarative part: This section precedes the keyword **begin** and is optional. It is used for all sorts of local declarations, including **type**, **signal**, and **component**. It also allows the construction of **function** and **procedure**. These declarations and functions/procedures can also be placed outside the main code, in a **package**.
Statements part: This portion, which starts at the keyword **begin**, constitutes the code proper. As shown in figure 6.1, its main elements (in no particular order) are the following: basic expressions using operators (for simple combinational circuits); expressions using concurrent statements (**when**, **select**, **generate**), generally for simple

to midcomplexity combinational circuits; sequential code using **process**, which is constructed using sequential statements (**if**, **case**, **loop**, **wait**), for sequential as well as (complex) combinational circuits; **function**/**procedure** calls; and, finally, **component** (that is, other design) instantiations, resulting in structural designs.

6.3 VHDL Template for Regular (Category 1) Moore Machines

The template is based on figure 6.2 (derived from figure 5.2), which shows three processes: 1) for the FSM state register; 2) for the FSM combinational logic; and 3) for the optional output register. Note the asterisk on one of the input connections; as we know, if that connection exists it is a Mealy machine, else it is a Moore machine.

There obviously are other ways of breaking the code instead of using the three processes indicated in figure 6.2. For example, the combinational logic section, being not sequential, could be implemented without a process (using purely concurrent code). At the other extreme the combinational logic section could be implemented with two processes, one with the logic for *output*, the other with the logic for *nx_state*.

The VHDL template for the design of category 1 Moore machines, based on figures 6.1 and 6.2, is presented below. Observe the following:

1) To improve readability, the three fundamental code sections (library/package declarations, entity, and architecture) are separated by dashed lines (lines 1, 4, 14, 76).
2) The library/package declarations (lines 2–3) show the package *std_logic_1164*, needed because the types used in the ports of all designs will be *std_logic* and/or *std_logic_vector* (industry standard).
3) The entity, called *circuit*, is in lines 5–13. As seen in figure 6.1, it usually contains two parts: **generic** (optional) and **port** (mandatory for synthesis). The former is employed for the declaration of generic parameters (if they exist), as illustrated in lines 6–8. The latter is a list of all circuit ports, with respective specifications, as illustrated

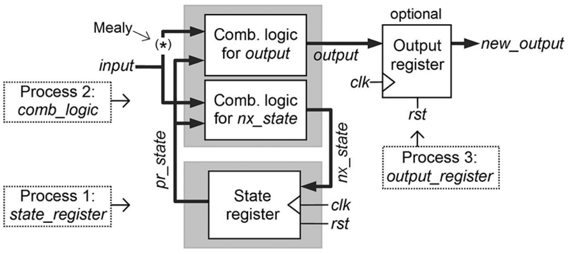


Figure 6.2
State machine architecture depicting how the VHDL code was broken (three processes).

in lines 9–12. Note that the type used for all ports (lines 10–12) is indeed *std_logic* or *std_logic_vector*.

4) The architecture, called *moore_fsm*, is in lines 15–75. It too is divided into two parts: declarative part (optional) and statements part (code proper, so mandatory).

5) The declarative part of the architecture is in lines 16–19. In lines 16–17 a special enumerated type, called *state*, is created, and then the signals *pr_state* and *nx_state* are declared using that type. In lines 18–19 an optional attribute called *enum_encoding* is shown, which defines the type of encoding desired for the machine's states (e.g., "sequential", "one-hot"). Another related attribute is *fsm_encoding*. See a description for both attributes after the template below. The encoding scheme can also be chosen using the compiler's setup, in which case lines 18–19 can be removed.

6) The statements part (code proper) of the architecture is in lines 20–75 (from **begin** on). In this template it is composed of three **process** blocks, described below.

7) The first process (lines 23–30) implements the state register (process 1 of figure 6.2). Because all of the machine's DFFs are in this section, clock and reset are only connected to this block (plus to the optional output register, of course, but that is not part of the FSM proper). Note that the code for this process is essentially standard, simply copying *nx_state* to *pr_state* at every positive clock transition (thus inferring the DFFs that store the machine's state).

8) The second process (lines 33–61) implements the entire combinational logic section of the FSM (process 2 of figure 6.2). This part must contain all states (A, B, C, . . .), and for each state two things must be declared: the output values/expressions and the next state. Observe, for example, in lines 36–46, relative to state A, the output declarations in lines 37–39 and the next-state declarations in lines 40–46. A very important point to note here is that there is no **if** statement associated with the outputs because in a Moore machine the outputs depend solely on the state in which the machine is, so for a given state each output value/expression is unique.

9) The third and final process (lines 64–73) implements the optional output register (process 3 of figure 6.2). Note that it simply copies each original output to a new output at every positive clock edge (it could also be at the negative edge), thus inferring the extra register. If this register is used, then the names of the new outputs must obviously be the names used in the corresponding port declarations (line 12). If the initial output values do not matter, reset is not required in this register.

10) To conclude, observe the completeness of the code and the correct use of registers (as requested in sections 4.2.8 and 4.2.9, respectively), summarized below.

a) Regarding the use of registers: The circuit is not overregistered. This can be observed in the **elsif rising_edge(clk)** statement of line 27 (responsible for the inference of flip-flops), which is closed in line 29, guaranteeing that only the machine state (line 28) gets registered. The circuit outputs are in the next process, which is purely combinational.

b) Regarding the outputs: The list of outputs (*output1*, *output2*, . . .) is the same in all states (see lines 37–39, 48–50, . . .), and the output values (or expressions) are always declared.

c) Regarding the next state: Again, the coverage is complete because all states (A, B, C, . . .) are included, and in each state the declarations are finalized with an **else** statement (lines 44, 55, . . .), guaranteeing that no condition is left unchecked.

Note 1: See also the comments in sections 6.4, which show some template variations.

Note 2: The VHDL 2008 review of the VHDL standard added the keyword **all** as a replacement for a process' sensitivity list, so **process (all)** is now valid. It also added boolean tests for *std_logic* signals and variables, so **if x='1' then . . .** can be replaced with **if x then. . . .** Both are supported by the current version (12.1) of Altera's Quartus II compiler but not yet by the current version (14.2) of Xilinx's ISE suite (XST compiler).

Note 3: Another implementation approach, for simple FSMs, will be seen in chapter 15.

```
1    -------------------------------------------------------------
2    library ieee;
3    use ieee.std_logic_1164.all;
4    -------------------------------------------------------------
5    entity circuit is
6       generic (
7          param1: std_logic_vector(...) := <value>;
8          param2: std_logic_vector(...) := <value>);
9       port (
10         clk, rst: in std_logic;
11         input1, input2, ...: in std_logic_vector(...);
12         output1, output2, ...: out std_logic_vector(...);
13   end entity;
14   -------------------------------------------------------------
15   architecture moore_fsm of circuit is
16      type state is (A, B, C, ...);
17      signal pr_state, nx_state: state;
18      attribute enum_encoding: string; --optional, see comments
19      attribute enum_encoding of state: type is "sequential";
20   begin
21
22      --FSM state register:
23      process (clk, rst)
24      begin
25         if rst='1' then  --see Note 2 above on boolean tests
26            pr_state <= A;
27         elsif rising_edge(clk) then
28            pr_state <= nx_state;
29         end if;
30      end process;
31
32      --FSM combinational logic:
33      process (all) --see Note 2 above on "all" keyword
34      begin
35         case pr_state is
```

```
36            when A =>
37               output1 <= <value>;
38               output2 <= <value>;
39               ...
40               if <condition> then
41                  nx_state <= B;
42               elsif <condition> then
43                  nx_state <= ...;
44               else
45                  nx_state <= A;
46               end if;
47            when B =>
48               output1 <= <value>;
49               output2 <= <value>;
50               ...
51               if <condition> then
52                  nx_state <= C;
53               elsif <condition> then
54                  nx_state <= ...;
55               else
56                  nx_state <= B;
57               end if;
58            when C =>
59               ...
60         end case;
61      end process;
62
63      --Optional output register:
64      process (clk, rst)
65      begin
66         if rst='1' then  --rst generally optional here
67            new_output1 <= ...;
68            ...
69         elsif rising_edge(clk) then
70            new_output1 <= output1;
71            ...
72         end if;
73      end process;
74
75   end architecture;
76   --------------------------------------------------------------
```

Final Comments

1) On the need for a reset signal: Note in the template above that the sequential portion of the FSM (process of lines 23–30) has a reset signal. As seen in sections 3.8 and 3.9, that is the usual situation. However, as also seen in those sections, if the circuit is implemented in an FPGA (so the flip-flops are automatically reset on power-up) and the codeword assigned to the intended initial (reset) state is the all-zero codeword, then reset will occur automatically.

2) On the *enum_encoding* and *fsm_encoding* attributes: As mentioned earlier, these attributes can be used to select the desired encoding scheme ("sequential", "one-hot", "001 011 010", and others—see options in section 3.7), overriding the compiler's

setup. It is important to mention, however, that support for these attributes varies among synthesis compilers. For example, Altera's Quartus II has full support for *enum_encoding*, so both examples below are fine (where "sequential" can also be "one-hot", "gray", and so on):

```
attribute enum_encoding: string;
attribute enum_encoding of state: type is "sequential";
attribute enum_encoding: string;
attribute enum_encoding of state: type is "001 100 101"; --user defined
```

Xilinx's XST (from the ISE suite), on the other hand, only supports *enum_encoding* for user-defined encoding; for the others ("sequential", "one-hot", etc.), *fsm_encoding* can be used. Two valid examples are shown below:

```
attribute enum_encoding: string;
attribute enum_encoding of state: type is "001 100 101";
attribute fsm_encoding: string;
attribute fsm_encoding of pr_state: signal is "sequential";
```

6.4 Template Variations

The template of section 6.3 can be modified in several ways with little or no effect on the final result. Some options are described below. These modifications are extensible to the Mealy template treated in the next section.

6.4.1 Combinational Logic Separated into Two Processes

A variation sometimes helpful from a didactic point of view is to separate the FSM combinational logic process into two processes: one for the output, another for the next state. Below, the process for the output logic is in lines 33–47, and that for the next state logic is in lines 50–69.

```
32   --FSM combinational logic for output:
33   process (all)
34   begin
35      case pr_state is
36         when A =>
37            output1 <= <value>;
38            output2 <= <value>;
39            ...
40         when B =>
41            output1 <= <value>;
42            output2 <= <value>;
43            ...
44         when C =>
```

```
45             ...
46       end case;
47    end process;
48
49    --FSM combinational logic for next state:
50    process (all)
51    begin
52       case pr_state is
53          when A =>
54             if <condition> then
55                nx_state <= B;
56             elsif <condition> then
57                nx_state <= ...;
58             else
59                nx_state <= A;
60             end if;
61          when B =>
62             if <condition> then
63                nx_state <= C;
64             ...
65             end if;
66          when C =>
67             ...
68       end case;
69    end process;
```

6.4.2 State Register Plus Output Register in a Single Process

A variation in the other direction (reducing the number of processes from three to two instead of increasing it to four) consists of joining the process for the state register with that for the output register. This is not recommended for three reasons. First, in most projects the optional output register is not needed. Second, having the output register in a separate process helps remind the designer that the need or not for such a register is an important case-by-case decision. Third, one might want to have the output register operating at the other (negative) clock edge, which is better emphasized by using separate processes.

6.4.3 Using Default Values

When the same signal or variable value appears several times inside the *same* process, a default value can be entered at the beginning of the process. An example is shown below for the process of the combinational logic section, with default values for the outputs included in lines 36–38. In lines 40–45 only the values that disagree with these must then be typed in. An example in which default values are used is seen in section 12.4.

```
32    --FSM combinational logic:
33    process (all)
34    begin
```

```
      --Default values:
      output1 <= <value>;
      output2 <= <value>;
      ...
      --Code:
      case pr_state is
         when A =>;
            ...
         when B =>
            ...
      end case;
   end process;
```

6.4.4 A Dangerous Template

A tempting template is shown next. Note that the entire FSM is in a single process (lines 17–43). Its essential point is that the **elsif rising_edge(clk)** statement encloses the whole circuit (it opens in line 21 and only closes in line 42), thus registering it completely (that is, not only the state is stored in flip-flops—this has to be done anyway—but also all the outputs).

This template has several *apparent* advantages. One is that a shorter code results (for instance, we can replace *pr_state* and *nx_state* with a single name—*fsm_state*, for example; also, only one process is needed). Another apparent advantage is that the code will work (no latches inferred) when the list of outputs is not exactly the same in all states. Such features, however, might hide serious problems.

One of the problems is precisely the fact that the outputs are always registered, so the resulting circuit is never the FSM alone but the FSM plus the optional output register of figure 5.2c, which many times is unwanted.

Another problem is that, even if the optional output register were needed, we do not have the freedom to choose in which of the clock edges to operate it because the same edge is used for the FSM and for the output register in this template, reducing the design flexibility.

A third problem is the fact that, because the list of outputs does not need to be the same in all states (because they are registered, latches will not be inferred when an output value is not specified), the designer is prone to overlook the project specifications.

Finally, it is important to remember that VHDL (and SystemVerilog) is not a program but a code, and a shorter code *does not mean* a smaller or better circuit. In fact, longer, better-organized codes tend to ease the compiler's work, helping to optimize the final circuit.

In summary, the template below is a *particular case* of the general template introduced in section 6.3. The general template gets reduced to this one only when all outputs must be registered and the same clock edge must operate both the state register and the output register.

```
--Dangerous template (particular case of the general template)
library ieee;
use ieee.std_logic_1164.all;
-----------------------------------------------------------------
entity circuit is
   generic (...);
   port (
      clk, rst: in std_logic;
      input, ...: in std_logic_vector(...);
      output, ...: out std_logic_vector(...);
end entity;
-----------------------------------------------------------------
architecture moore_fsm of circuit is
   type state is (A, B, C, ...);
   signal fsm_state: state;
begin
   process (clk, rst)
   begin
      if rst then
         fsm_state <= A;
      elsif rising_edge(clk) then
         case fsm_state is
            when A =>
               output <= <value>;
               if <condition> then
                  fsm_state <= B;
               elsif <condition> then
                  fsm_state <= ...;
               else
                  fsm_state <= A;
               end if;
            when B =>
               output <= <value>;
               if <condition> then
               ...
               else
                  fsm_state <= B;
               end if;
            when C =>
            ...
         end case;
      end if;
   end process;
-----------------------------------------------------------------
```

6.5 VHDL Template for Regular (Category 1) Mealy Machines

This template, also based on figures 6.1 and 6.2, is presented below. The only difference with respect to the Moore template just presented is in the process for the combinational logic because the output is specified differently now. Recall that in a Mealy machine the output depends not only on the FSM's state but also on its input, so **if** statements are expected for the output in one or more states because the output values might not be unique. This is achieved by including the output *within* the conditional statements for *nx_state*. For example, observe in lines 20–36, relative to state A, that the output values are now conditional. Compare these lines against lines 36–46 in the template of section 6.3.

Please review the following comments, which can easily be adapted from the Moore case to the Mealy case:

—On the Moore template for category 1, in section 6.3, especially comment 10.
—On the *enum_encoding* and *fsm_encoding* attributes, also in section 6.3.
—On possible code variations, in section 6.4.

```
1    ----------------------------------------------------------------
2    library ieee;
3    use ieee.std_logic_1164.all;
4    ----------------------------------------------------------------
5    entity circuit is
6       (same as for category 1 Moore, section 6.3)
7    end entity;
8    ----------------------------------------------------------------
9    architecture mealy_fsm of circuit IS
10      (same as for category 1 Moore, Section 6.3)
11   begin
12
13      --FSM state register:
14      (same as for category 1 Moore, section 6.3)
15
16      --FSM combinational logic:
17      process (all) --list proc. inputs if “all” not supported
18      begin
19         case pr_state is
20            when A =>
21               if <condition> then
22                  output1 <= <value>;
23                  output2 <= <value>;
24                  ...
25                  nx_state <= B;
26               elsif <condition> then
27                  output1 <= <value>;
28                  output2 <= <value>;
29                  ...
30                  nx_state <= ...;
31               else
32                  output1 <= <value>;
33                  output2 <= <value>;
34                  ...
35                  nx_state <= A;
36               end if;
37            when B =>
38               if <condition> then
39                  output1 <= <value>;
40                  output2 <= <value>;
41                  ...
42                  nx_state <= C;
43               elsif <condition> then
```

```
               output1 <= <value>;
               output2 <= <value>;
               ...
               nx_state <= ...;
            else
               output1 <= <value>;
               output2 <= <value>;
               ...
               nx_state <= B;
            end if;
         when C =>
            ...
      end case;
   end process;

   --Optional output register:
   (same as for category 1 Moore, section 6.3)

end architecture;
--------------------------------------------------------------
```

6.6 Design of a Small Counter

This section presents a VHDL-based design for the 1-to-5 counter with enable and up-down controls introduced in section 5.4.1 (figure 5.3).

Because counters are inherently synchronous, the Moore approach is the natural choice for their implementation, so the VHDL template of section 6.3 is used. Because possible glitches during (positive) clock transitions are generally not a problem in counters, the optional output register shown in the last process of the template is not employed.

The entity, called *counter*, is in lines 5–9. All ports are of type *std_logic* or *std_logic_vector* (industry standard).

The architecture, called *moore_fsm*, is in lines 11–88. As usual, it contains a declarative part (before the keyword **begin**) and a statements part (from **begin** on).

In the declarative part of the architecture (lines 12–13), the enumerated type *state* is created to represent the machine's present and next states. Recall that when neither the *enum_encoding* nor the *fsm_encoding* attribute is used, the encoding scheme must be selected in the compiler's setup.

The first process (lines 17–24) in the statements part implements the state register. As in the template, this is a standard code with clock and reset present only in this process.

The second and final process (lines 27–86) implements the entire combinational logic section. It is just a list of all states, each containing the output value and the next state. Note that in each state the output value is unique because in a Moore machine the output depends only on the state in which the machine is.

Observe the correct use of registers and the completeness of the code, as described in comment 10 of section 6.3. Note in particular the following:

1) Regarding the use of registers: The circuit is not overregistered. This can be observed in the **elsif rising_edge(clk)** statement of line 21 (responsible for the inference of flip-flops), which is closed in line 23, guaranteeing that only the machine state (line 22) gets stored. The output (*outp*) is in the next process, which is purely combinational (thus not registered).
2) Regarding the outputs: The list of outputs (just *outp* in this example) is exactly the same in all states (see lines 31, 42, 53, 64, 75), and the corresponding output values are always properly declared.
3) Regarding the next state: Again, the coverage is complete because all states are included (see lines 30, 41, 52, 63, 74), and in each state the conditional declarations for the next state are always finalized with an **else** statement (lines 38, 49, 60, 71, 82), guaranteeing that no condition is left unchecked.

```
1    ---------------------------------------------------------
2    library ieee;
3    use ieee.std_logic_1164.all;
4    ---------------------------------------------------------
5    entity counter is
6       port (
7          ena, up, clk, rst: in std_logic;
8          outp: out std_logic_vector(2 downto 0));
9    end entity;
10   ---------------------------------------------------------
11   architecture moore_fsm of counter is
12      type state is (one, two, three, four, five);
13      signal pr_state, nx_state: state;
14   begin
15
16      --FSM state register:
17      process (clk, rst)
18      begin
19         if rst='1' then
20            pr_state <= one;
21         elsif rising_edge(clk) then
22            pr_state <= nx_state;
23         end if;
24      end process;
25
26      --FSM combinational logic:
27      process (all) --list proc. inputs if "all" not supported
28      begin
29         case pr_state is
30            when one =>
31               outp <= "001";
32               if ena='1' then
33                  if up='1' then
34                     nx_state <= two;
35                  else
```

```
                  nx_state <= five;
               end if;
            else
               nx_state <= one;
            end if;
         when two =>
            outp <= "010";
            if ena='1' then
               if up='1' then
                  nx_state <= three;
               else
                  nx_state <= one;
               end if;
            else
               nx_state <= two;
            end if;
         when three =>
            outp <= "011";
            if ena='1' then
               if up='1' then
                  nx_state <= four;
               else
                  nx_state <= two;
               end if;
            else
               nx_state <= three;
            end if;
         when four =>
            outp <= "100";
            if ena='1' then
               if up='1' then
                  nx_state <= five;
               else
                  nx_state <= three;
               end if;
            else
               nx_state <= four;
            end if;
         when five =>
            outp <= "101";
            if ena='1' then
               if up='1' then
                  nx_state <= one;
               else
                  nx_state <= four;
               end if;
            else
               nx_state <= five;
            end if;
      end case;
   end process;

end architecture;
----------------------------------------------------------
```

Synthesis results using the VHDL code above are presented in figure 6.3. The circuit's structure can be seen in the RTL view of figure 6.3a, while the FSM can be seen

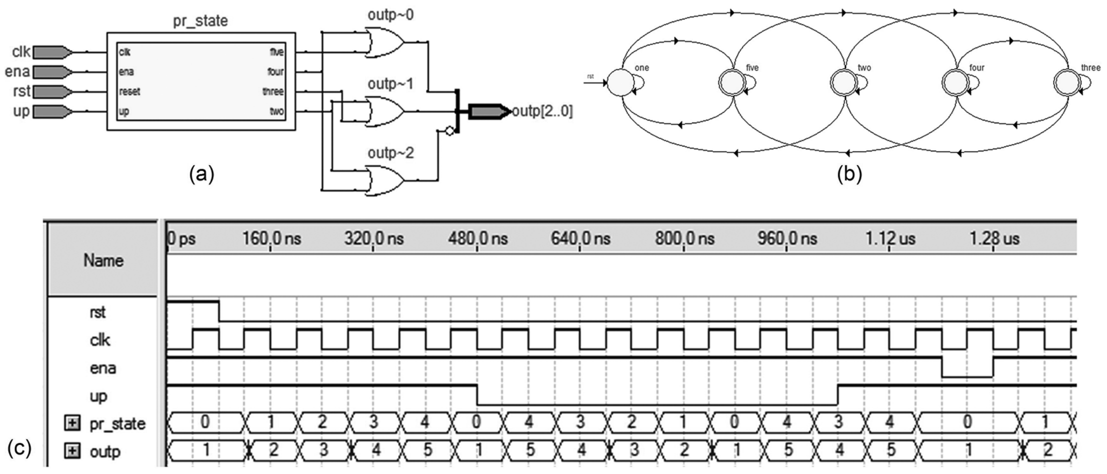


Figure 6.3
Results from the VHDL code for the 1-to-5 counter with enable and up-down controls of figure 5.3. (a) RTL view. (b) State machine view. (c) Simulation results.

in figure 6.3b. As expected, the latter coincides with the intended state transition diagram (figure 5.3). Simulation results are exhibited in figure 6.3c. Note that the output changes only at positive clock transitions, counting up when *up* = ‘1’, down when *up* = ‘0’, and stopping if *ena* = ‘0’.

The number of flip-flops inferred by the compiler after synthesizing the code above was three for sequential, Gray, or Johnson encoding and five for one-hot, matching the predictions made in section 5.4.1.

Note: As smentioned in section 5.4.1, counters can be designed very easily without employing the FSM approach when using VHDL or SystemVerilog. The design above was included, nevertheless, because it illustrates well the construction of VHDL code for category 1 machines. A similar counter, but without the up-down control, results from the code below, where the FSM technique is not employed. Moreover, it is fine for any number of bits.

```
-----------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
-----------------------------------------------------
entity counter is
   generic (
      bits: natural := 3;
      xmin: natural := 1;
      xmax: natural := 5);
   port (
      clk, rst, ena: in std_logic;
      x_out: out std_logic_vector(bits-1 downto 0));
end entity;
----------------------------------------------------
architecture direct_counter of counter is
   signal x: natural range 0 to xmax;
begin
   process (clk, rst)
   begin
      if rst='1' then
         x <= xmin;
      elsif rising_edge(clk) and ena='1' then
         if x<xmax then
            x <= x + 1;
         else
            x <= xmin;
         end if;
      end if;
   end process;
   x_out <= conv_std_logic_vector(x, bits);
end architecture;
----------------------------------------------------
```

6.7 Design of a Garage Door Controller

This section presents a VHDL-based design for the garage door controller introduced in section 5.4.5. The Moore template of section 6.3 is employed to implement the FSM of figure 5.9c.

The entity, called *garage_door_controller*, is in lines 5–9. All ports are of type *std_logic* or *std_logic_vector* (industry standard).

The architecture, called *moore_fsm*, is in lines 11–94. As usual, it contains a declarative part (before the keyword **begin**) and a statements part (from **begin** on).

In the declarative part of the architecture (lines 12–14), the enumerated type *state* is created to represent the machine's present and next states.

The first process (lines 18–25) in the statements part implements the state register. As in the template, this is a standard code with clock and reset present only in this process.

The second and final process (lines 28–92) implements the entire combinational logic section. It is just a list of all states, each containing the output value and the next state. Note that in each state the output value is unique because in a Moore machine the output depends only on the state in which the machine is.

Observe the correct use of registers and the completeness of the code as described in comment number 10 of section 6.3. Note in particular the following:

1) Regarding the use of registers: The circuit is not overregistered. This can be observed in the **elsif rising_edge(clk)** statement of line 22 (responsible for the inference of flip-flops), which is closed in line 24, guaranteeing that only the machine state (line 23) gets stored. The output (*ctr*) is in the next process, which is purely combinational (thus not registered).

2) Regarding the outputs: The list of outputs (just *ctr* in this example) is exactly the same in all states (see lines 32, 39, 46, …), and the corresponding output value is always properly declared.

3) Regarding the next state: Again, the coverage is complete because all states are included (see lines 31, 38, 45, …), and in each state the conditional declarations for the next state are always finalized with an **else** statement (lines 35, 42, 51, …), guaranteeing that no condition is left unchecked.

```
1    ---------------------------------------------------------
2    library ieee;
3    use ieee.std_logic_1164.all;
4    ---------------------------------------------------------
5    entity garage_door_controller is
6       port (
7          remt, sen1, sen2, clk, rst: in std_logic;
8          ctr: out std_logic_vector(1 downto 0));
9    end entity;
10   ---------------------------------------------------------
11   architecture moore_fsm of garage_door_controller is
12      type state is (closed1, closed2, opening1, opening2,
13         open1, open2, closing1, closing2);
14      signal pr_state, nx_state: state;
15   begin
16
17      --FSM state register:
18      process (clk, rst)
19      begin
20         if rst='1' then
21            pr_state <= closed1;
22         elsif rising_edge(clk) then
23            pr_state <= nx_state;
24         end if;
25      end process;
26
27      --FSM combinational logic:
28      process (all) --or (pr_state, remt, sen1, sen2)
29      begin
30         case pr_state is
31            when closed1 =>
32               ctr <= "0-";
33               if remt='0' then
34                  nx_state <= closed2;
35               else
36                  nx_state <= closed1;
37               end if;
38            when closed2 =>
39               ctr <= "0-";
40               if remt='1' then
41                  nx_state <= opening1;
42               else
43                  nx_state <= closed2;
44               end if;
45            when opening1 =>
```

```
            ctr <= "10";
            if sen1='1' then
               nx_state <= open1;
            elsif remt='0' then
               nx_state <= opening2;
            else
               nx_state <= opening1;
            end if;
         when opening2 =>
            ctr <= "10";
            if remt='1' or sen1='1' then
               nx_state <= open1;
            else
               nx_state <= opening2;
            end if;
         when open1 =>
            ctr <= "0-";
            if remt='0' then
               nx_state <= open2;
            else
               nx_state <= open1;
            end if;
         when open2 =>
            ctr <= "0-";
            if remt='1' then
               nx_state <= closing1;
            else
               nx_state <= open2;
            end if;
         when closing1 =>
            ctr <= "11";
            if sen2='1' then
               nx_state <= closed1;
            elsif remt='0' then
               nx_state <= closing2;
            else
               nx_state <= closing1;
            end if;
         when closing2 =>
            ctr <= "11";
            if remt='1' or sen2='1' then
               nx_state <= closed1;
            else
               nx_state <= closing2;
            end if;
      end case;
   end process;

end architecture;
---------------------------------------------------------
```

The number of flip-flops inferred by the compiler after synthesizing the code above was three for sequential or Gray encoding, four for Johnson, and eight for one-hot, matching the predictions made in section 5.4.5.

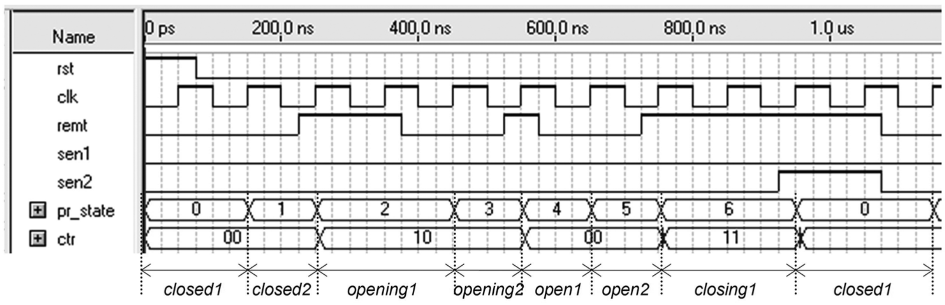


Figure 6.4
Simulation results from the VHDL code for the garage door controller of figure 5.9c.

Simulation results are depicted in figure 6.4. The encoding chosen for the states was *sequential* (section 3.7). The states are enumerated from 0 to 7 (there are eight states), in the order in which they were declared in lines 12–13. Be aware, however, that some compilers reserve the value zero for the reset state; because the reset (initial) state in the present example is *closed1* (see lines 20–21), which is the first state in the declaration list, that is not a concern here.

In this simulation the sequence *closed1–closed2–opening1–opening2–open1–open2–closing1–closed1* (see state names in the lower part of figure 6.4) was tested. Note that pulses of various widths were used to illustrate the fact that their width has no effect beyond the first positive clock edge.

6.8 Design of a Datapath Controller for a Greatest Common Divisor Calculator

This section presents a VHDL-based design for the control unit introduced in section 5.4.8, which controls a datapath to produce a greatest common divisor (GCD) calculator. The Moore template of section 6.3 is employed to implement the FSM of figure 5.13e.

The entity, called *control_unit_for_GCD*, is in lines 5–11. All ports are of the type *std_logic* or *std_logic_vector* (industry standard).

The architecture, called *moore_fsm*, is in lines 13–80. As usual, it contains a declarative part (before the keyword **begin**) and a statements part (from **begin** on).

In the declarative part of the architecture (lines 14–15), the enumerated type *state* is created to represent the machine's present and next states.

The first process (lines 19–26) in the statements part implements the state register. As in the template, this is a standard code with clock and reset present only in this process.

The second and final process (lines 29–78) implements the entire combinational logic section. It is just a list of all states, each containing the output values and the next state. Note that in each state the output values are unique because in a Moore machine the outputs depend only on the state in which the machine is.

Observe the correct use of registers and the completeness of the code, as described in comment 10 of section 6.3. Note in particular the following:

1) Regarding the use of registers: The circuit is not overregistered. This can be observed in the **elsif rising_edge(clk)** statement of line 23 (responsible for the inference of flip-flops), which is closed in line 25, guaranteeing that only the machine state (line 24) gets stored. The outputs are in the next process, which is purely combinational (thus not registered).
2) Regarding the outputs: The list of outputs (*selA*, *selB*, *wrA*, *wrB*, *ALUop*) is exactly the same in all states (see lines 33–37, 44–48, 51–55, . . .), and the corresponding output values are always properly declared.
3) Regarding the next state: Again, the coverage is complete because all states are included (see lines 32, 43, 50, . . .), and in each state any conditional declarations for the next state are finalized with an **else** statement (lines 40 and 60), guaranteeing that no condition is left unchecked.

```
1    -----------------------------------------------------
2    library ieee;
3    use ieee.std_logic_1164.all;
4    -----------------------------------------------------
5    entity control_unit_for_GCD is
6       port (
7          dv, clk, rst: in std_logic;
8          sign: in std_logic_vector(1 downto 0)
9          selA, selB, wrA, wrB: out std_logic;
10         ALUop: out std_logic_vector(1 downto 0));
11   end entity;
12   -----------------------------------------------------
13   architecture moore_fsm of control_unit_for_GCD is
14      type state is (idle, load, waitt, writeA, writeB);
15      signal pr_state, nx_state: state;
16   begin
17
18      --FSM state register:
19      process (clk, rst)
20      begin
21         if rst='1' then
22            pr_state <= idle;
23         elsif rising_edge(clk) then
24            pr_state <= nx_state;
25         end if;
26      end process;
27
28      --FSM combinational logic:
29      process (all)  --or (pr_state, dv, sign)
```

```
   begin
      case pr_state is
         when idle =>
            selA <= '-';
            selB <= '-';
            wrA <= '0';
            wrB <= '0';
            ALUop <= "00";
            if dv='1' then
               nx_state <= load;
            else
               nx_state <= idle;
            end if;
         when load =>
            selA <= '1';
            selB <= '1';
            wrA <= '1';
            wrB <= '1';
            ALUop <= "00";
            nx_state <= waitt;
         when waitt =>
            selA <= '-';
            selB <= '-';
            wrA <= '0';
            wrB <= '0';
            ALUop <= "10";
            if sign="01" then
               nx_state <= writeA;
            elsif sign="10" then
               nx_state <= writeB;
            else
               nx_state <= idle;
            end if;
         when writeA =>
            selA <= '0';
            selB <= '-';
            wrA <= '1';
            wrB <= '0';
            ALUop <= "10";
            nx_state <= waitt;
         when writeB =>
            selA <= '-';
            selB <= '0';
            wrA <= '0';
            wrB <= '1';
            ALUop <= "11";
            nx_state <= waitt;
      end case;
   end process;

end architecture;
------------------------------------------------------
```

Simulation results are presented in figure 6.5. The encoding chosen for the states was *sequential* (section 3.7). The states are enumerated from 0 to 4 (there are five states)

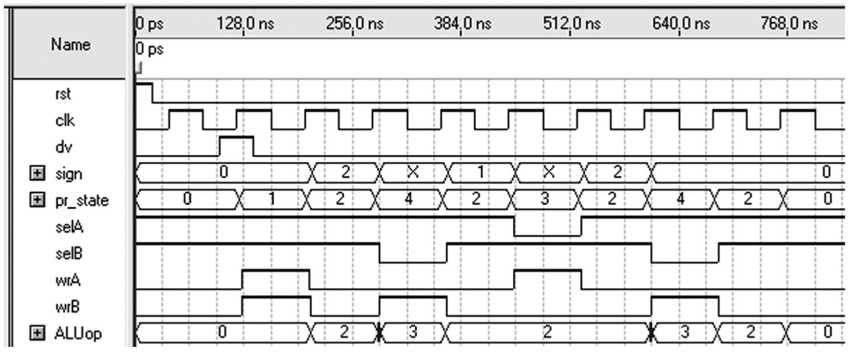


Figure 6.5
Simulation results from the VHDL code for the control unit of figure 5.13e, which controls a datapath for GCD calculation.

in the order in which they were declared in line 14 (be aware, however, that some compilers reserve the value zero for the reset state). The stimuli are exactly as in figure 5.13d (GCD for 9 and 15). The reader is invited to inspect these results and compare them against the waveforms in figure 5.13d.

6.9 Exercises

Exercise 6.1: Parity Detector

This exercise concerns the parity detector of figure 5.5c.

a) How many flip-flops are needed to implement it with sequential and one-hot encoding?
b) Implement it using VHDL. Check whether the number of DFFs inferred by the compiler matches each of your predictions.
c) Simulate it using the same stimuli of figure 5.5b and check if the same waveform results for *y*.

Exercise 6.2: One-Shot Circuits

This exercise concerns the one-shot circuits of figures 5.7c,d.

a) Solve exercise 5.5 if not done yet.
b) How many flip-flops are needed to implement each FSM with sequential encoding?
c) Implement both circuits using VHDL. Check whether the number of DFFs inferred by the compiler matches each of your predictions.
d) Simulate each circuit using the same stimuli of exercise 5.5 (figure 5.16) and check whether the same results are obtained here.

Exercise 6.3: Manchester Encoder

This exercise concerns the Manchester encoder treated in exercise 5.8.

a) Solve exercise 5.8 if not done yet.
b) Implement the Moore machine relative to part a of that exercise using VHDL. Simulate it using the same stimuli of part b, checking if the results match.
c) Implement the Mealy machine relative to part c of that exercise using VHDL. Simulate it using the same stimuli of part d, checking if the results match.

Exercise 6.4: Differential Manchester Encoder

This exercise concerns the differential Manchester encoder treated in exercise 5.9.

a) Solve exercise 5.9 if not done yet.
b) Implement the FSM relative to part a of that exercise using VHDL. Simulate it using the same waveforms of part b, checking if the results match.

Exercise 6.5: String Detector

This exercise concerns the string detector of figure 5.14a, which detects the sequence "*abc*".

a) Solve exercise 5.12 if not done yet.
b) Implement the FSM of figure 5.14a using VHDL. Simulate it using the same stimuli of exercise 5.12, checking if the same results are obtained here.

Exercise 6.6: Generic String Detector

This exercise concerns the generic string detector of figure 5.14b. Implement it using VHDL and simulate it for the following cases:

a) To detect the sequence "*abc*".
b) To detect the sequence "*aab*".
c) To detect the sequence "*aaa*".

Exercise 6.7: Keypad Encoder

This exercise concerns the keypad encoder treated in exercise 5.14. It is repeated in figure 6.6, with a seven-segment display (SSD—see figure 8.13) at the output, which must display the last key pressed (use the characters "A" and "b" for * and #, respectively). (A deboucer is generally needed in this kind of design; see exercise 8.9.)

a) Solve exercise 5.14 if not done yet.
b) Implement the FSM obtained above using VHDL. Instead of encoding *r*(3:0) according to the table in figure 5.23c, encode it as an SSD driver, using the table in figure 8.13d (so *key* is now a 7-bit signal).

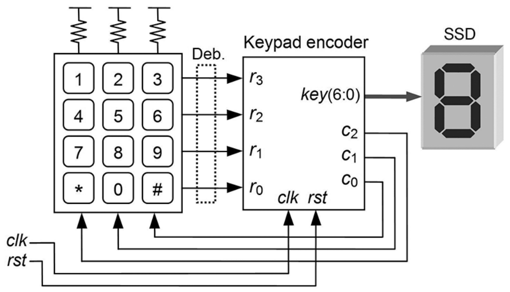


Figure 6.6

c) Physically test your design by connecting an actual keypad (or an arrangement of pushbuttons) to the FPGA in your development board, with *key* displayed by one of the board's SSDs.

Exercise 6.8: Datapath Controller for a Largest-Value Detector

This exercise concerns the control unit treated in exercise 5.15.

a) Solve exercise 5.15 if not done yet.

b) Implement the FSM obtained above using VHDL. Present meaningful simulation results.