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

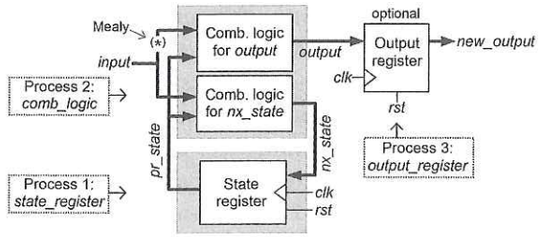

Figure 6.2
State machine architecture depicting how the VHDL code was broken (three processes).

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
```



```
35       --Default values:
36       output1 <= <value>;
37       output2 <= <value>;
38       ...
39       --Code:
40       case pr_state is
41          when A =>;
42             ...
43          when B =>
44             ...
45       end case;
46    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.

```
1   --Dangerous template (particular case of the general template)
2   library ieee;
3   use ieee.std_logic_1164.all;
4   ------------------------------------------------------------------
5   entity circuit is
6      generic (...);
7      port (
8         clk, rst: in std_logic;
9         input, ...: in std_logic_vector(...);
10        output, ...: out std_logic_vector(...);
11  end entity;
12  ------------------------------------------------------------------
13  architecture moore_fsm of circuit is
14     type state is (A, B, C, ...);
15     signal fsm_state: state;
16  begin
17     process (clk, rst)
18     begin
19        if rst then
20           fsm_state <= A;
21        elsif rising_edge(clk) then
22           case fsm_state is
23              when A =>
24                 output <= <value>;
25                 if <condition> then
26                    fsm_state <= B;
27                 elsif <condition> then
28                    fsm_state <= ...;
29                 else
30                    fsm_state <= A;
31                 end if;
32              when B =>
33                 output <= <value>;
34                 if <condition> then
35                 ...
36                 else
37                    fsm_state <= B;
38                 end if;
39              when C =>
40              ...
41           end case;
42        end if;
43     end process;
44  ------------------------------------------------------------------
```

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
```

```
44                output1 <= <value>;
45                output2 <= <value>;
46                ...
47                nx_state <= ...;
48             else
49                output1 <= <value>;
50                output2 <= <value>;
51                ...
52                nx_state <= B;
53             end if;
54          when C =>
55             ...
56       end case;
57    end process;
58
59    --Optional output register:
60    (same as for category 1 Moore, section 6.3)
61
62 end architecture;
63 ---------------------------------------------------------------
```

6.6 Design of a Small Counter