CHAPTER 5

# FSM

## 5.1 INTRODUCTION

An FSM (finite state machine) is used to model a system that transits among a finite number of internal states. The transitions depend on the current state and external input. Unlike a regular sequential circuit, the state transitions of an FSM do not exhibit a simple, repetitive pattern. Its next-state logic is usually constructed from scratch and is sometimes known as "random" logic. This is different from the next-state logic of a regular sequential circuit, which is composed mostly of "structured" components, such as incrementors and shifters.

In this chapter, we provide an overview of the basic characteristics and representation of FSMs and discuss the derivation of HDL codes. In practice, the main application of an FSM is to act as the controller of a large digital system, which examines the external commands and status and activates proper control signals to control operation of a *data path*, which is usually composed of regular sequential components. This is known as an FSMD (finite state machine with data path) and is discussed in Chapter 6.

### 5.1.1 Mealy and Moore outputs

The basic block diagram of an FSM is the same as that of a regular sequential circuit and is repeated in Figure 5.1. It consists of a state register, next-state logic, and output logic. An FSM is known as a *Moore machine* if the output is only a function of state, and is known as a *Mealy machine* if the output is a function of state and external input. Both types of output may exist in a complex FSM, and we simply refer to it as containing a Moore

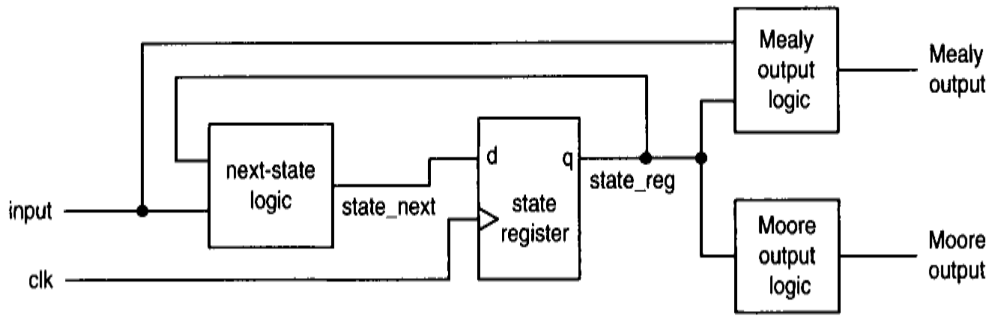


**Figure 5.1** Block diagram of a synchronous FSM.

output and a Mealy output. The Moore and Mealy outputs are similar but not identical. Understanding their subtle differences is the key for controller design. The example in Section 5.3.1 illustrates the behaviors and constructions of the two types of outputs.

### 5.1.2 FSM representation

An FSM is usually specified by an abstract *state diagram* or *ASM chart* (algorithmic state machine chart), both capturing the FSM's input, output, states, and transitions in a graphical representation. The two representations provide the same information. The FSM representation is more compact and better for simple applications. The ASM chart representation is somewhat like a flowchart and is more descriptive for applications with complex transition conditions and actions.

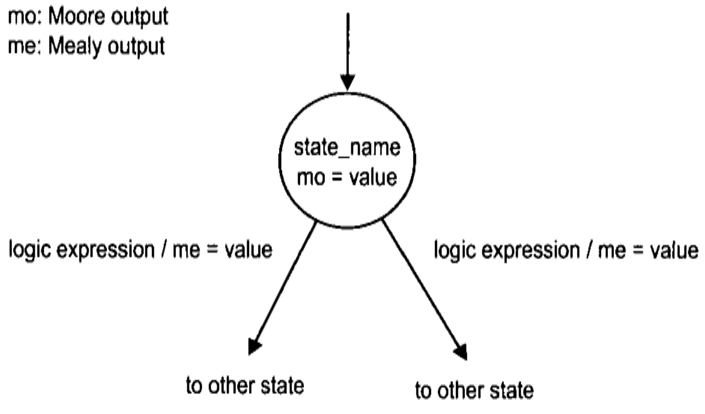
***State diagram*** A state diagram is composed of *nodes*, which represent states and are drawn as circles, and annotated *transitional arcs*. A single node and its transition arcs are shown in Figure 5.2(a). A logic expression expressed in terms of input signals is associated with each transition arc and represents a specific condition. The arc is taken when the corresponding expression is evaluated `true`.

The Moore output values are placed inside the circle since they depend only on the current state. The Mealy output values are associated with the conditions of transition arcs since they depend on the current state and external input. To reduce clutter in the diagram, only asserted output values are listed. The output signal takes the default (i.e., unasserted) value otherwise.

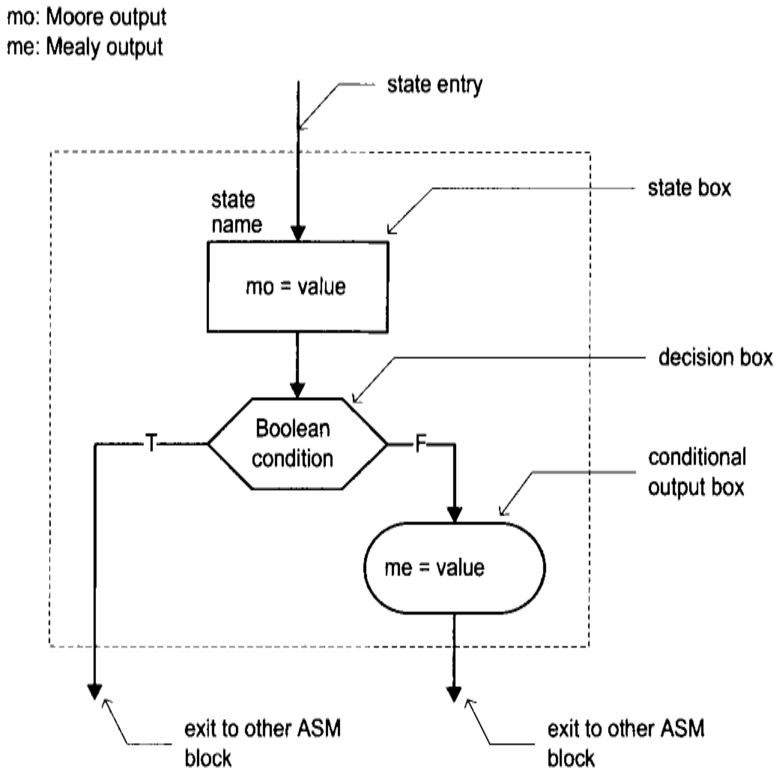
A representative state diagram is shown in Figure 5.3(a). The FSM has three states, two external input signals (i.e., `a` and `b`), one Moore output signal (i.e., `y1`), and one Mealy output signal (i.e., `y0`). The `y1` signal is asserted when the FSM is in the `s0` or `s1` state. The `y0` signal is asserted when the FSM is in the `s0` state and the `a` and `b` signals are "11".

***ASM chart*** An ASM chart is composed of a network of ASM blocks. An *ASM block* consists of one *state box* and an optional network of *decision boxes* and *conditional output boxes*. A representative ASM block is shown in Figure 5.2(b).

A state box represents a state in an FSM, and the asserted Moore output values are listed inside the box. Note that it has only one exit path. A decision box tests the input condition and determines which exit path to take. It has two exit paths, labeled `T` and `F`, which correspond to the true and false values of the condition. A conditional output box lists asserted Mealy output values and is usually placed after a decision box. It indicates that the listed output signal can be activated only when the corresponding condition in the decision box is met.



(a) Node



(b) ASM block

**Figure 5.2** Symbol of a state.

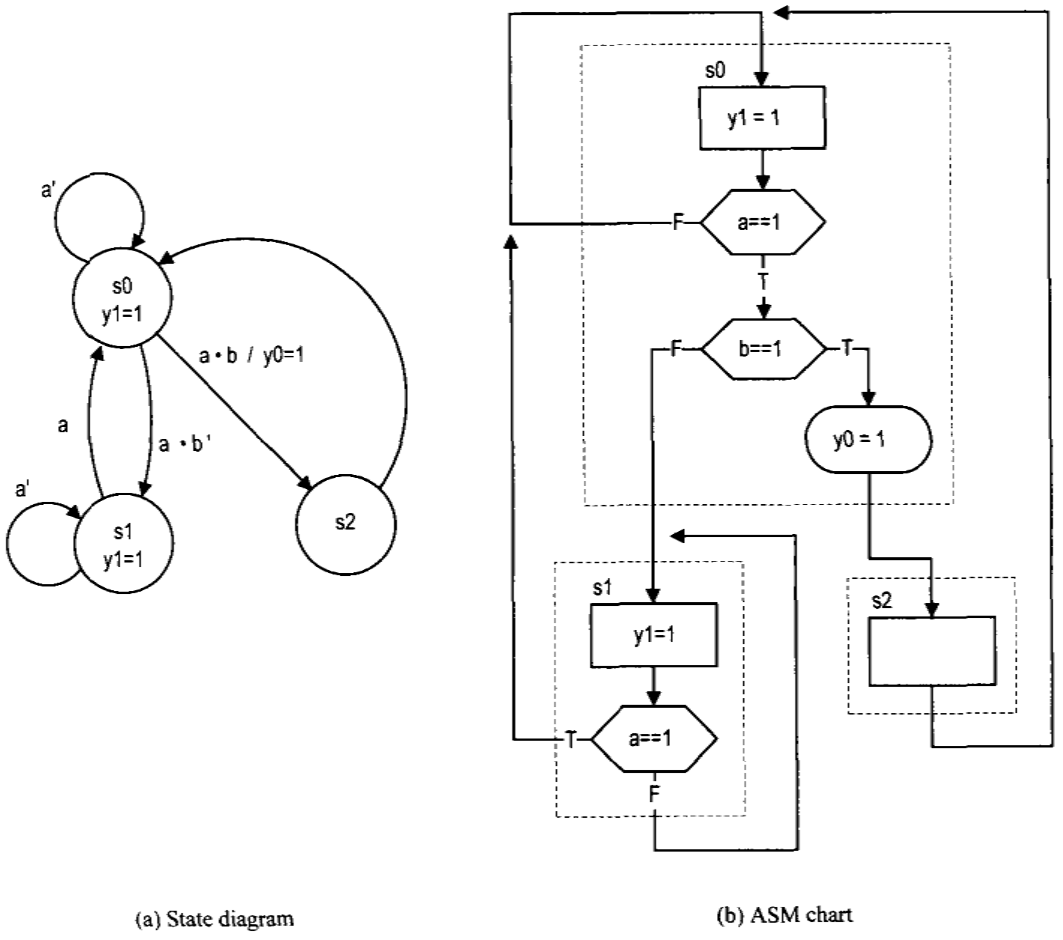


**Figure 5.3** Example of an FSM.

A state diagram can easily be converted to an ASM chart, and vice versa. The corresponding ASM chart of the previous FSM state diagram is shown in Figure 5.3(b).

## 5.2 FSM CODE DEVELOPMENT

The procedure of developing code for an FSM is similar to that of a regular sequential circuit. We first separate the state register and then derive the code for the combinational next-state logic and output logic. The main difference is the next-state logic. For an FSM, the code for the next-state logic follows the flow of a state diagram or ASM chart.

For clarity and flexibility, we use symbolic constants to represent the FSM's states. For examples, the three states in Figure 5.3 can be defined as

```
localparam [1:0] s0 = 2'b00,
                 s1 = 2'b01,
                 s2 = 2'b10;
```

During synthesis, software usually can recognize the FSM structure and may map these symbolic constants to different binary representations (e.g., one-hot codes), a process known as *state assignment*.

The complete code of the FSM is shown in Listing 5.1. It consists of segments for the state register, next-state logic, Moore output logic, and Mealy output logic.

**Listing 5.1** FSM example

```
module fsm_eg_mult_seg
   (
    input wire  clk, reset,
    input wire  a, b,
    output wire y0, y1
   );

   // symbolic state declaration
   localparam [1:0] s0 = 2'b00,
                    s1 = 2'b01,
                    s2 = 2'b10;

   // signal declaration
   reg [1:0] state_reg, state_next;

   // state register
    always @(posedge clk, posedge reset)
       if (reset)
          state_reg <= s0;
       else
          state_reg <= state_next;

   // next-state logic
   always @*
      case (state_reg)
         s0: if (a)
                if (b)
                   state_next = s2;
                else
                   state_next = s1;
             else
                state_next = s0;
         s1: if (a)
               state_next = s0;
             else
               state_next = s1;
         s2: state_next = s0;
         default: state_next = s0;
      endcase

   // Moore output logic
   assign y1 = (state_reg==s0) || (state_reg==s1);

   // Mealy output logic
   assign y0 = (state_reg==s0) & a & b;

endmodule
```

The key part is the next-state logic. It uses a case statement with the `state_reg` signal as the selection expression. The next state (i.e., `state_next` signal) is determined by the current state (i.e., `state_reg`) and external input. The code for each state basically follows the activities inside each ASM block of Figure 5.3(b).

An alternative code is to merge next-state logic and output logic into a single combinational block, as shown in Listing 5.2.

**Listing 5.2** FSM with merged combinational logic

```
module fsm_eg_2_seg
   (
    input wire  clk, reset,
    input wire  a, b,
    output reg y0, y1
   );

   // symbolic state declaration
   localparam [1:0] s0 = 2'b00,
                    s1 = 2'b01,
                    s2 = 2'b10;
   // signal declaration
   reg [1:0] state_reg, state_next;

   // state register
    always @(posedge clk, posedge reset)
       if (reset)
          state_reg <= s0;
       else
          state_reg <= state_next;

   // next-state logic and output logic
   always @*
   begin
      state_next = state_reg; // default next state: the same
      y1 = 1'b0;              // default output: 0
      y0 = 1'b0;              // default output: 0
      case (state_reg)
         s0: begin
                y1 = 1'b1;
                if (a)
                   if (b)
                      begin
                         state_next = s2;
                         y0 = 1'b1;
                      end
                   else
                      state_next = s1;
             end
         s1: begin
               y1 = 1'b1;
               if (a)
                  state_next = s0;
             end
         s2: state_next = s0;
         default: state_next = s0;
      endcase
   end
endmodule
```

Note that the default output values are listed at the beginning of the code.

The code for the next-state logic and output logic follows the ASM chart closely. Once a detailed state diagram or ASM chart is derived, converting an FSM to HDL code is almost a mechanical procedure. Listings 5.1 and 5.2 can serve as templates for this purpose.

**Xilinx specific**

Xilinx ISE includes a utility program called *StateCAD*, which allows a user to draw a state diagram in graphical format. The program then converts the state diagram to HDL code. It is a good idea to try it first with a few simple examples to see whether the generated code and its style are satisfactory, particularly for the output signals.

## 5.3 DESIGN EXAMPLES

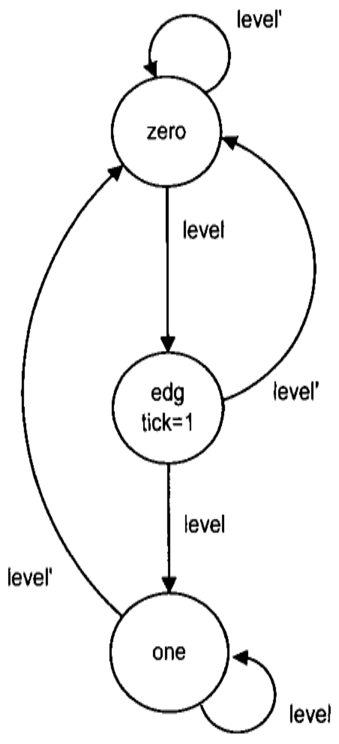
### 5.3.1 Rising-edge detector

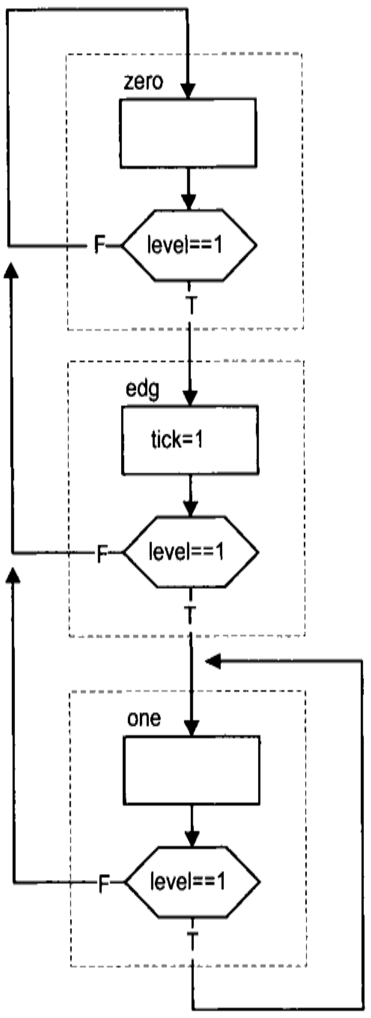
The rising-edge detector is a circuit that generates a short one-clock-cycle tick when the input signal changes from 0 to 1. It is usually used to indicate the onset of a slow time-varying input signal. We design the circuit using both Moore and Mealy machines, and compare their differences.

***Moore-based design*** The state diagram and ASM chart of a Moore machine–based edge detector are shown in Figure 5.4. The `zero` and `one` states indicate that the input signal has been 0 and 1 for a while. The rising edge occurs when the input changes to 1 in the `zero` state. The FSM moves to the `edg` state and the output, `tick`, is asserted in this state. A representative timing diagram is shown at the middle of Figure 5.5. The code is shown in Listing 5.3.

**Listing 5.3** Moore machine–based edge detector

```
module edge_detect_moore
   (
    input wire clk, reset,
    input wire level,
    output reg tick
   );

   // symbolic state declaration
   localparam  [1:0]
      zero  = 2'b00,
      edg  = 2'b01,
      one  = 2'b10;

   // signal declaration
   reg [1:0] state_reg, state_next;

   // state register
    always @(posedge clk, posedge reset)
       if (reset)
          state_reg <= zero;
       else
          state_reg <= state_next;

   // next-state logic and output logic
   always @*
```





(a) State diagram (b) ASM chart

**Figure 5.4** Edge detector based on a Moore machine.

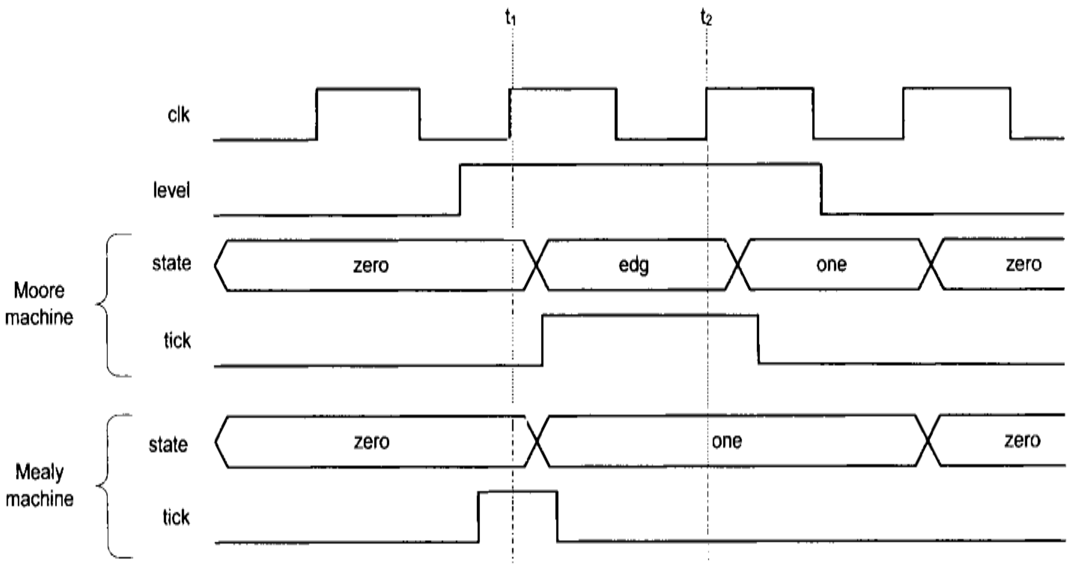


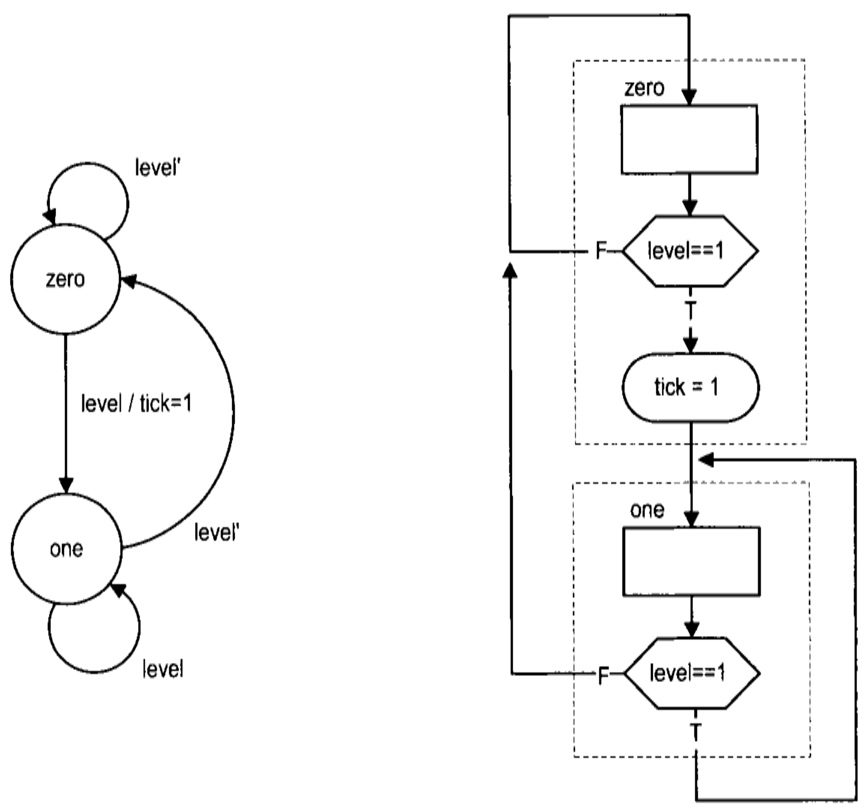
**Figure 5.5** Timing diagram of two edge detectors.

```
      begin
         state_next = state_reg;   // default state: the same
         tick = 1'b0;              // default output: 0
         case (state_reg)
            zero:
               if (level)
                  state_next = edg;
            edg:
               begin
                  tick = 1'b1;
                  if (level)
                     state_next = one;
                  else
                     state_next = zero;
               end
            one:
               if (~level)
                  state_next = zero;
            default: state_next = zero;
         endcase
      end
   endmodule
```

***Mealy-based design*** The state diagram and ASM chart of a Mealy machine–based edge detector are shown in Figure 5.6. The zero and one states have a similar meaning. When the FSM is in the zero state and the input changes to 1, the output is asserted immediately. The FSM moves to the one state at the rising edge of the next clock and the output is deasserted. A representative timing diagram is shown at the bottom of Figure 5.5. Note that due to the propagation delay, the output signal is still asserted at the rising edge of the next clock (i.e., at $t_1$). The code is shown in Listing 5.4.

**Listing 5.4** Mealy machine–based edge detector

```
module edge_detect_mealy
   (
    input wire  clk, reset,
    input wire  level,
    output reg tick
   );

   // symbolic state declaration
   localparam zero = 1'b0,
              one = 1'b1;

   // signal declaration
   reg state_reg, state_next;

   // state register
    always @(posedge clk, posedge reset)
       if (reset)
          state_reg <= zero;
       else
          state_reg <= state_next;
```



(a) State diagram (b) ASM chart

**Figure 5.6** Edge detector based on a Mealy machine.

```
   // next-state logic and output logic
   always @*
   begin
      state_next = state_reg;     // default state: the same
      tick = 1'b0;                // default output: 0
      case (state_reg)
         zero:
            if (level)
               begin
                  tick = 1'b1;
                  state_next = one;
               end
         one:
            if (~level)
               state_next = zero;
         default: state_next = zero;
      endcase
   end

endmodule
```

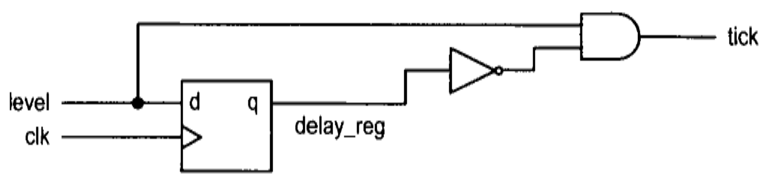


**Figure 5.7** Gate-level implementation of an edge detector.

***Direct implementation*** Since the transitions of the edge detector circuit are very simple, it can be implemented without using an FSM. We include this implementation for comparison purposes. The circuit diagram is shown in Figure 5.7. It can be interpreted that the output is asserted only when the current input is 1 and the previous input, which is stored in the register, is 0. The corresponding code is shown in Listing 5.5.

**Listing 5.5** Gate-level implementation of an edge detector

```
module edge_detect_gate
   (
    input wire  clk, reset,
    input wire  level,
    output wire tick
   );

   // signal declaration
   reg delay_reg;

   // delay register
    always @(posedge clk, posedge reset)
       if (reset)
          delay_reg <= 1'b0;
       else
          delay_reg <= level;

   // decoding logic
   assign tick = ~delay_reg & level;

endmodule
```

Although the descriptions in Listings 5.4 and 5.5 appear to be very different, they describe the same circuit. The circuit diagram can be derived from the FSM if we assign 0 and 1 to the `zero` and `one` states.

***Comparison*** Whereas both Moore machine– and Mealy machine–based designs can generate a short tick at the rising edge of the input signal, there are several subtle differences. The Mealy machine–based design requires fewer states and responds faster, but the width of its output may vary and input glitches may be passed to the output.

The choice between the two designs depends on the subsystem that uses the output signal. Most of the time the subsystem is a synchronous system that shares the same clock signal. Since the FSM's output is sampled only at the rising edge of the clock, the width and glitches do not matter as long as the output signal is stable around the edge. Note that the Mealy output signal is available for sampling at $t_1$, which is one clock cycle faster than

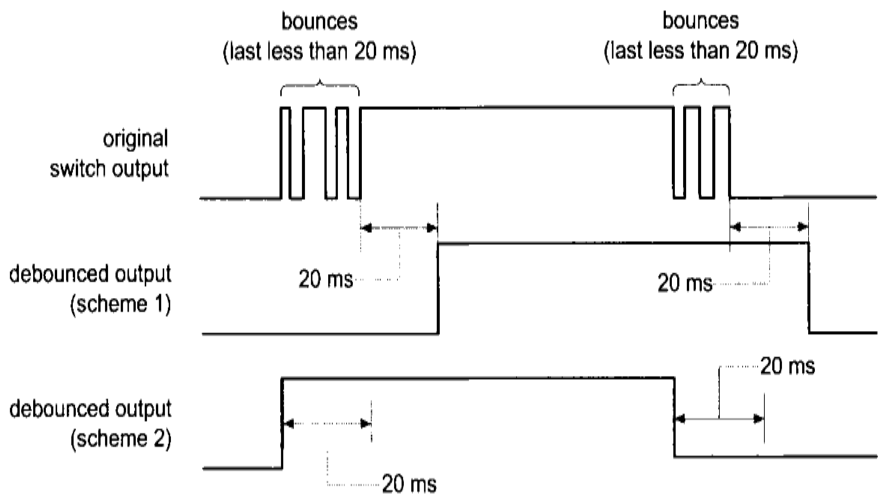


**Figure 5.8** Original and debounced waveforms.

the Moore output, which is available at $t_2$. Therefore, the Mealy machine–based circuit is preferred for this type of application.

### 5.3.2 Debouncing circuit

The slide and pushbutton switches on the prototyping board are mechanical devices. When pressed, the switch may bounce back and forth a few times before settling down. The bounces lead to glitches in the signal, as shown at the top of Figure 5.8. The bounces usually settle within 20 ms. The purpose of a debouncing circuit is to filter out the glitches associated with switch transitions. The debounced output signals from two FSM-based design schemes are shown in the two bottom parts of Figure 5.8. The first design scheme is discussed in this subsection and the second scheme is left as an exercise in Experiment 5.5.2. A better alternative FSMD-based scheme is discussed in Section 6.2.1.

An FSM-based design uses a free-running 10-ms timer and an FSM. The timer generates a one-clock-cycle enable tick (the `m_tick` signal) every 10 ms and the FSM uses this information to keep track of whether the input value is stabilized. In the first design scheme, the FSM ignores the short bounces and changes the value of the debounced output only after the input is stabilized for 20 ms. The output timing diagram is shown at the middle of Figure 5.8. The state diagram of this FSM is shown in Figure 5.9. The `zero` and `one` states indicate that the switch input signal, `sw`, has been stabilized with 0 and 1 values. Assume that the FSM is initially in the `zero` state. It moves to the `wait1_1` state when `sw` changes to 1. At the `wait1_1` state, the FSM waits for the assertion of `m_tick`. If `sw` becomes 0 in this state, it implies that the width of the 1 value does not last long enough and the FSM returns to the `zero` state. This action repeats two more times for the `wait1_2` and `wait1_3` states. The operation from the `one` state is similar except that the `sw` signal must be 0.

Since the 10-ms timer is free-running and the `m_tick` tick can be asserted at any time, the FSM checks the assertion three times to ensure that the `sw` signal is stabilized for at least 20 ms (it is actually between 20 and 30 ms). The code is shown in Listing 5.6. It includes a 10-ms timer and the FSM.

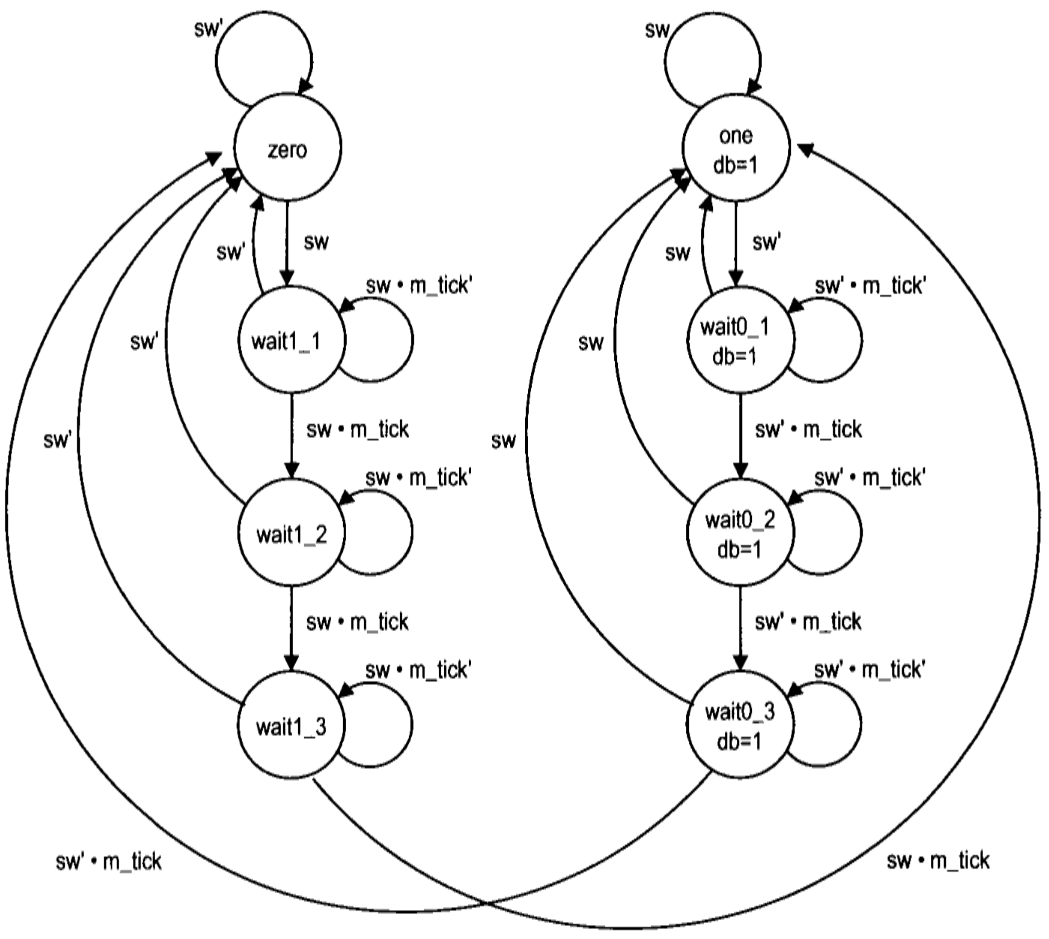


**Figure 5.9** State diagram of a debouncing circuit.

**Listing 5.6** FSM implementation of a debouncing circuit

```
module db_fsm
   (
    input wire clk, reset,
    input wire sw,
    output reg db
   );

   // symbolic state declaration
   localparam  [2:0]
               zero    = 3'b000,
               wait1_1 = 3'b001,
               wait1_2 = 3'b010,
               wait1_3 = 3'b011,
               one     = 3'b100,
               wait0_1 = 3'b101,
               wait0_2 = 3'b110,
               wait0_3 = 3'b111;

   // number of counter bits (2^N * 20ns = 10ms tick)
   localparam N =19;
```

```
// signal declaration
reg [N-1:0] q_reg;
wire [N-1:0] q_next;
wire m_tick;
reg [2:0] state_reg, state_next;

// body

//=============================================
// counter to generate 10 ms tick
//=============================================
always @(posedge clk)
   q_reg <= q_next;
// next-state logic
assign q_next = q_reg + 1;
// output tick
assign m_tick = (q_reg==0) ? 1'b1 : 1'b0;

//=============================================
// debouncing FSM
//=============================================
// state register
 always @(posedge clk, posedge reset)
    if (reset)
       state_reg <= zero;
    else
       state_reg <= state_next;

// next-state logic and output logic
always @*
begin
   state_next = state_reg;  // default state: the same
   db = 1'b0;               // default output: 0
   case (state_reg)
      zero:
         if (sw)
            state_next = wait1_1;
      wait1_1:
         if (~sw)
            state_next = zero;
         else
            if (m_tick)
               state_next = wait1_2;
      wait1_2:
         if (~sw)
            state_next = zero;
         else
            if (m_tick)
               state_next = wait1_3;
      wait1_3:
         if (~sw)
            state_next = zero;
```

```
               else
                  if (m_tick)
                     state_next = one;
            one:
               begin
                 db = 1'b1;
                 if (~sw)
                    state_next = wait0_1;
               end
            wait0_1:
               begin
                  db = 1'b1;
                  if (sw)
                     state_next = one;
                  else
                    if (m_tick)
                       state_next = wait0_2;
               end
            wait0_2:
               begin
                  db = 1'b1;
                  if (sw)
                     state_next = one;
                  else
                    if (m_tick)
                       state_next = wait0_3;
               end
            wait0_3:
               begin
                  db = 1'b1;
                  if (sw)
                     state_next = one;
                  else
                    if (m_tick)
                       state_next = zero;
               end
            default: state_next = zero;
         endcase
      end

   endmodule
```

### 5.3.3 Testing circuit

We use a bounce counting circuit to verify operation of the rising-edge detector and the debouncing circuit. The block diagram is shown in Figure 5.10. The input of the verification circuit is from a pushbutton switch. In the lower part, the signal is first fed to the debouncing circuit and then to the rising-edge detector. Therefore, a one-clock-cycle tick is generated each time the button is pressed and released. The tick in turn controls the enable input of an 8-bit counter, whose content is passed to the LED time-multiplexing circuit and shown on the left two digits of the prototyping board's seven-segment LED display. In the upper

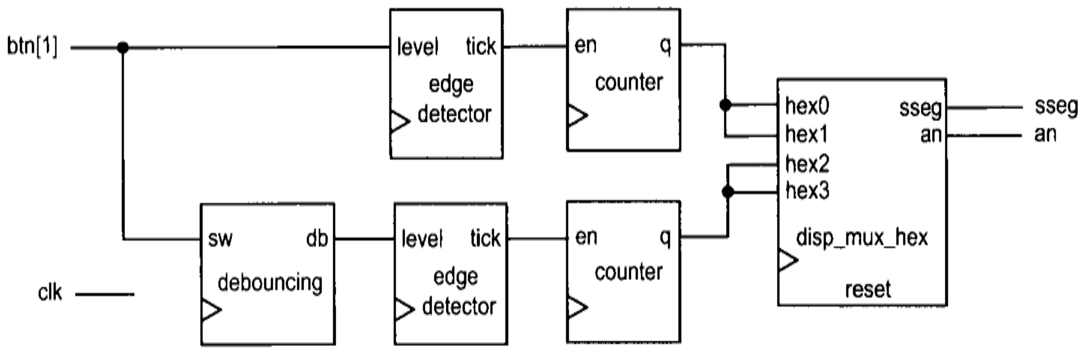


**Figure 5.10** Debouncing testing circuit.

part, the input signal is fed directly to the edge detector without the debouncing circuit, and the number is shown on the right two digits of the prototyping board's seven-segment LED display. The bottom counter thus counts one desired 0-to-1 transition as well as the bounces.

The code is shown in Listing 5.7. It basically uses component instantiation to realize the block diagram.

**Listing 5.7** Verification circuit for a debouncing circuit and rising-edge detector

```
module debounce_test
   (
    input wire clk, reset,
    input wire [1:0] btn,
    output wire [3:0] an,
    output wire [7:0] sseg
   );

   // signal declaration
   reg [7:0]  b_reg, d_reg;
   wire [7:0] b_next, d_next;
   reg  btn_reg, db_reg;
   wire db_level, db_tick, btn_tick, clr;

   // instantiate 7-seg LED display time-multiplexing module
   disp_hex_mux disp_unit
      (.clk(clk), .reset(reset),
       .hex3(b_reg[7:4]), .hex2(b_reg[3:0]),
       .hex1(d_reg[7:4]), .hex0(d_reg[3:0]),
       .dp_in(4'b1011), .an(an), .sseg(sseg));

   // instantiate debouncing circuit
   db_fsm db_unit
      (.clk(clk), .reset(reset), .sw(btn[1]), .db(db_level));

   // edge detection circuits
   always @(posedge clk)
      begin
         btn_reg <= btn[1];
         db_reg <= db_level;
      end
   assign btn_tick = ~btn_reg & btn[1];
   assign db_tick = ~db_reg & db_level;

   // two counters
   assign clr = btn[0];
   always @(posedge clk)
      begin
         b_reg <= b_next;
         d_reg <= d_next;
      end
   assign b_next = (clr)      ? 8'b0 :
                   (btn_tick) ? b_reg + 1 : b_reg;
   assign d_next = (clr)      ? 8'b0 :
                   (db_tick)  ? d_reg + 1 : d_reg;

endmodule
```

The seven-segment display shows the accumulated numbers of 0-to-1 edges of bounced and debounced switch input. After pressing and releasing the pushbutton switch several times, we can determine the average number of bounces for each transition.

## 5.4 BIBLIOGRAPHIC NOTES

The article "Coding and Scripting Techniques for FSM Designs with Synthesis-Optimized, Glitch-Free Outputs" by C. E. Cummings provides a detailed discussion on various coding styles of FSM.

## 5.5 SUGGESTED EXPERIMENTS

### 5.5.1 Dual-edge detector

A dual-edge detector is similar to a rising-edge detector except that the output is asserted for one clock cycle when the input changes from 0 to 1 (i.e., rising edge) and 1 to 0 (i.e., falling edge).

1. Design a circuit based on the Moore machine and draw the state diagram and ASM chart.
2. Derive the HDL code based on the state diagram of the ASM chart.
3. Derive a testbench and use simulation to verify operation of the code.
4. Replace the rising detectors in Section 5.3.3 with dual-edge detectors and verify their operations.
5. Repeat steps 1 to 4 for a Mealy machine–based design.

### 5.5.2 Alternative debouncing circuit

One problem with the debouncing design in Section 5.3.2 is the delayed response of the onset of a switch transition. An alternative is to react to the first edge in the transition and then wait for a small amount of time (at least 20 ms) to have the input signal settled. The output timing diagram is shown at the bottom of Figure 5.8. When the input changes from

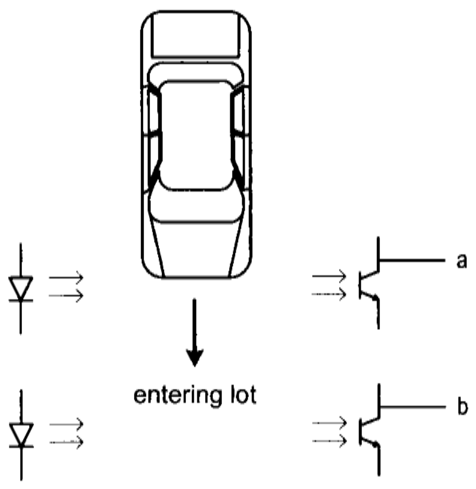


**Figure 5.11** Conceptual diagram of gate sensors.

0 to 1, the FSM responds immediately. The FSM then ignores the input for about 20 ms to avoid glitches. After this amount of time, the FSM starts to check the input for the falling edge. Follow the design procedure in Section 5.3.2 to design the alternative circuit.

1. Derive the state diagram and ASM chart for the circuit.
2. Derive the HDL code.
3. Derive the HDL code based on the state diagram and ASM chart.
4. Derive a testbench and use simulation to verify operation of the code.
5. Replace the debouncing circuit in Section 5.3.3 with the alternative design and verify its operation.

### 5.5.3 Parking lot occupancy counter

Consider a parking lot with a single entry and exit gate. Two pairs of photo sensors are used to monitor the activity of cars, as shown in Figure 5.11. When an object is between the photo transmitter and the photo receiver, the light is blocked and the corresponding output is asserted to 1. By monitoring the events of two sensors, we can determine whether a car is entering or exiting or a pedestrian is passing through. For example, the following sequence indicates that a car enters the lot:

- Initially, both sensors are unblocked (i.e., the a and b signals are "00").
- Sensor a is blocked (i.e., the a and b signals are "10").
- Both sensors are blocked (i.e., the a and b signals are "11").
- Sensor a is unblocked (i.e., the a and b signals are "01").
- Both sensors becomes unblocked (i.e., the a and b signals are "00").

Design a parking lot occupancy counter as follows:

1. Design an FSM with two input signals, a and b, and two output signals, `enter` and `exit`. The `enter` and `exit` signals assert one clock cycle when a car enters and one clock cycle when a car exits the lot, respectively.
2. Derive the HDL code for the FSM.
3. Design a counter with two control signals, `inc` and `dec`, which increment and decrement the counter when asserted. Derive the HDL code.

4. Combine the counter and the FSM and LED multiplexing circuit. Use two debounced pushbuttons to mimic operation of the two sensor outputs. Verify operation of the occupancy counter.

CHAPTER 6

# FSMD

## 6.1 INTRODUCTION

An FSMD (finite state machine with data path) combines an FSM and regular sequential circuits. The FSM, which is sometimes known as a *control path*, examines the external commands and status and generates control signals to specify operation of the regular sequential circuits, which are known collectively as a *data path*. The FSMD is used to implement systems described by *RT (register transfer) methodology*, in which the operations are specified as data manipulation and transfer among a collection of registers.

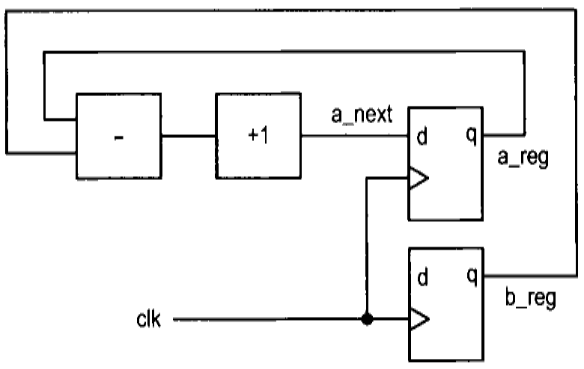
### 6.1.1 Single RT operation

An RT operation specifies data manipulation and transfer for a single destination register. It is represented by the notation

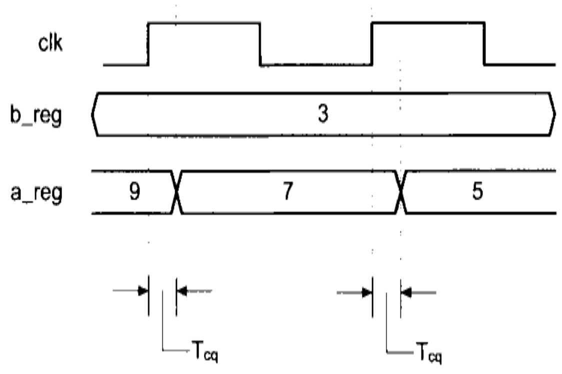
$$\mathrm{r_{dest}} \leftarrow f(\mathrm{r_{src1}}, \mathrm{r_{src2}}, \ldots, \mathrm{r_{srcn}})$$

where $\mathrm{r_{dest}}$ is the destination register, $\mathrm{r_{src1}}$, $\mathrm{r_{src2}}$, and $\mathrm{r_{srcn}}$ are the source registers, and $f(\cdot)$ specifies the operation to be performed. The notation indicates that the contents of the source registers are fed to the $f(\cdot)$ function, which is realized by a combinational circuit, and the result is passed to the input of the destination register and stored in the destination register at the next rising edge of the clock. Following are several representative RT operations:

- `r1` $\leftarrow$ 0. A constant 0 is stored in the `r1` register.
- `r1` $\leftarrow$ `r1`. The content of the `r1` register is written back to itself.



(a) Block diagram



(b) Timing diagram

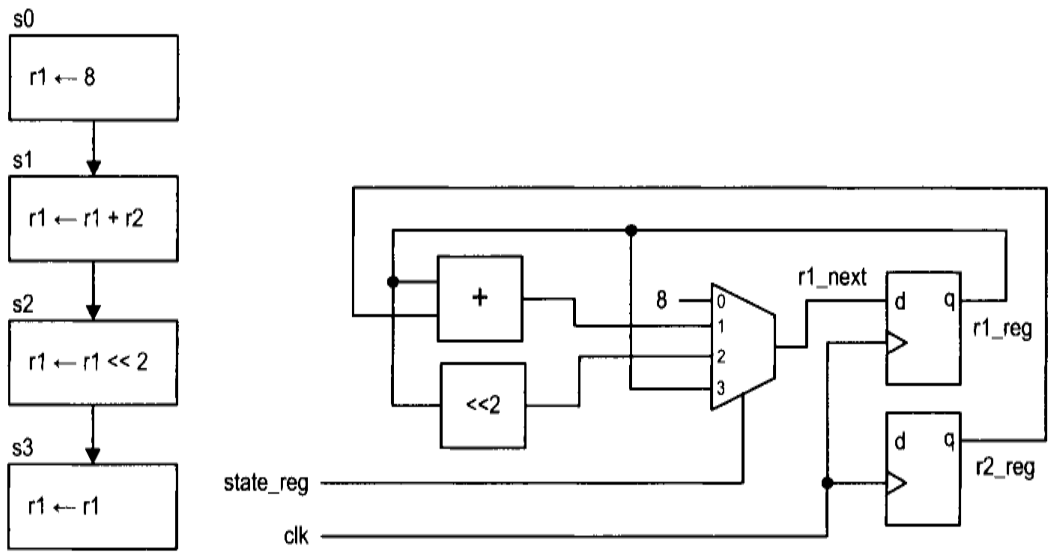
**Figure 6.1** Block and timing diagrams of an RT operation.

- r2 ← r2 >> 3. The r2 register is shifted right three positions and then written back to itself.
- r2 ← r1. The content of the r1 register is transferred to the r2 register.
- i ← i + 1. The content of the i register is incremented by 1 and the result is written back to itself.
- d ← s1 + s2 + s3. The summation of the s1, s2, and s3 registers is written to the d register.
- y ← a*a. The a squared is written to the y register.

A single RT operation can be implemented by constructing a combinational circuit for the $f(\cdot)$ function and connecting the input and output of the registers. For example, consider the a ← a-b+1 operation. The $f(\cdot)$ function involves a subtractor and an incrementor. The block diagram is shown in Figure 6.1(a). For clarity, we use the _reg and _next suffixes to represent the input and output of a register. Note that an RT operation is synchronized by an embedded clock. The result from the $f(\cdot)$ function is not stored to the destination register until the next rising edge of the clock. The timing diagram of the previous RT operation is shown in Figure 6.1(b).

### 6.1.2 ASMD chart

A circuit based on the RT methodology specifies which RT operations should be executed in each step. Since an RT operation is done on a clock-by-clock basis, its timing is similar to a state transition of an FSM. Thus, an FSM is a natural choice to specify the sequencing



(a) ASMD segment (b) Block diagram

**Figure 6.2** Realization of an ASMD segment.

of an RT algorithm. We extend the ASM chart to incorporate RT operations and call it an *ASMD* (ASM with data path) chart. The RT operations are treated as another type of activity and can be placed where the output signals are used.

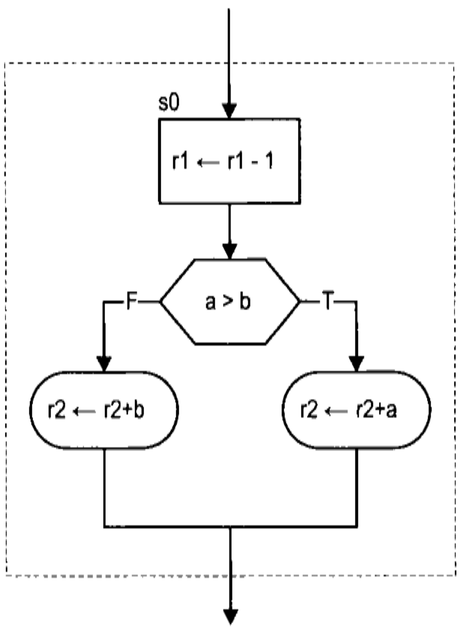
A segment of an ASMD chart is shown in Figure 6.2(a). It contains one destination register, `r1`, which is initialized with 8, added with content of the `r2` register, and then shifted left two positions. Note that the `r1` register must be specified in each state. When `r1` is not changed, the `r1` ← `r1` operation should be used to maintain its current content, as in the `s3` state. In future discussion, we assume that `r` ← `r` is the default RT operation for the `r` register and do not include it in the ASMD chart. Implementing the RT operations of an ASMD chart involves a multiplexing circuit to route the desired next value to the destination register. For example, the previous segment can be implemented by a 4-to-1 multiplexer, as shown in Figure 6.2(b). The current state (i.e., the output of the state register) of the FSM controls the selection signal of the multiplexer and thus chooses the result of the desired RT operation.

An RT operation can also be specified in a conditional output box, as the `r2` register shown in Figure 6.3(a). Depending on the `a>b` condition, the FSMD performs either `r2` ← `r2+a` or `r2` ← `r2+b`. Note that all operations are done in parallel inside an ASMD block. We need to realize the `a>b`, `r2+a`, and `r2+b` operations and use a multiplexer to route the desired value to `r2`. The block diagram is shown in Figure 6.3(b).

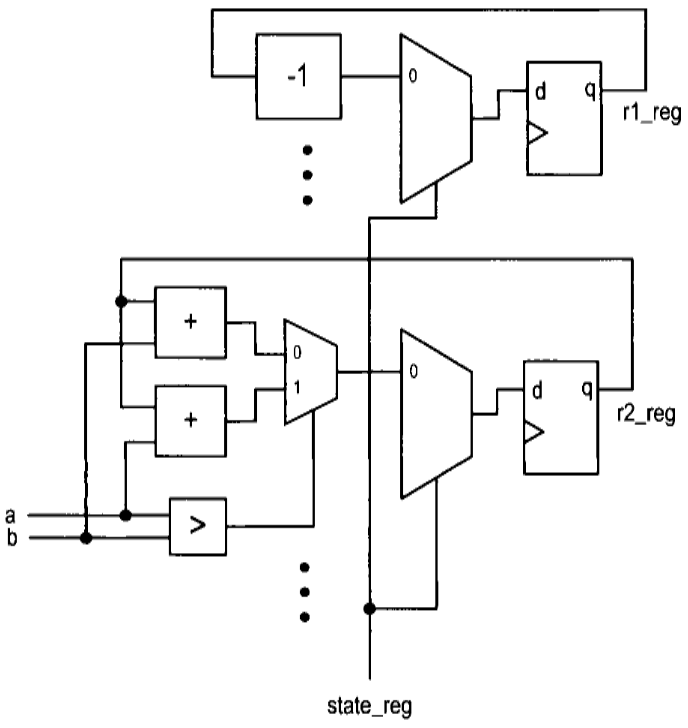
### 6.1.3 Decision box with a register

The appearance of an ASMD chart is similar to that of a normal flowchart. The main difference is that the RT operation in an ASMD chart is controlled by an embedded clock signal and the destination register is updated *when the FSMD exits the current ASMD block*, but not within the block. The `r` ← `r-1` operation actually means that:

- `r_next = r_reg - 1;`
- `r_reg <= r_next` at the rising edge of the clock (i.e., when the FSMD exits the current block).

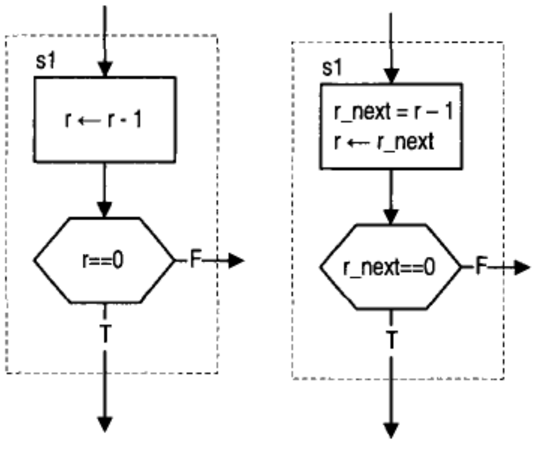


(a) ASM block



(b) Block diagram

**Figure 6.3** Realization of an RT operation in a conditional output box.



(a) Use old value of r (b) Use new value of r

**Figure 6.4** ASM block affected by a delayed store.

This "delayed store" may introduce subtle errors when a register is used in a decision box. Consider the FSMD segment in Figure 6.4(a). The `r` register is decremented in the state box and used in the decision box. Since the `r` register is not updated until the FSMD exits the block, the old content of `r` is used for comparison in the decision box. If the new value of `r` is desired, we should use the output of the combinational logic (i.e., `r_next`) in the decision box (i.e., replace the `r==0` expression with `r_next==0`), as shown in Figure 6.4(b).

***Block diagram of an FSMD*** The conceptual block diagram of an FSMD is divided into a data path and a control path, as shown in Figure 6.5. The data path performs the required RT operations. It consists of:

- *Data registers*: store the intermediate computation results
- *Functional units*: perform the functions specified by the RT operations
- *Routing network*: routes data between the storage registers and the functional units

The data path follows the `control` signal to perform the desired RT operations and generates the `internal status` signal.

The control path is an FSM. As a regular FSM, it contains a state register, next-state logic, and output logic. It uses the external `command` signal and the data path's `status` signal as the input and generates the `control` signal to control the data path operation. The FSM also generates the `external status` signal to indicate the status of the FSMD operation.

Note that although an FSMD consists of two types of sequential circuits, both circuits are controlled by the same clock, and thus the FSMD is still a synchronous system.

## 6.2 CODE DEVELOPMENT OF AN FSMD

We use an improved debouncing circuit to demonstrate derivation of the FSMD code. Although the debouncing circuit in Section 5.3.2 uses an FSM and a timer (which is a regular sequential circuit), it is not based on the RT methodology because the two units are running independently and the FSM has no control over the timer. Since the 10-ms enable tick can be asserted at any time, the FSM does not know how much time has elapsed when the first tick is detected in the `wait1_1` or `wait0_1` state. Thus, the waiting period in this

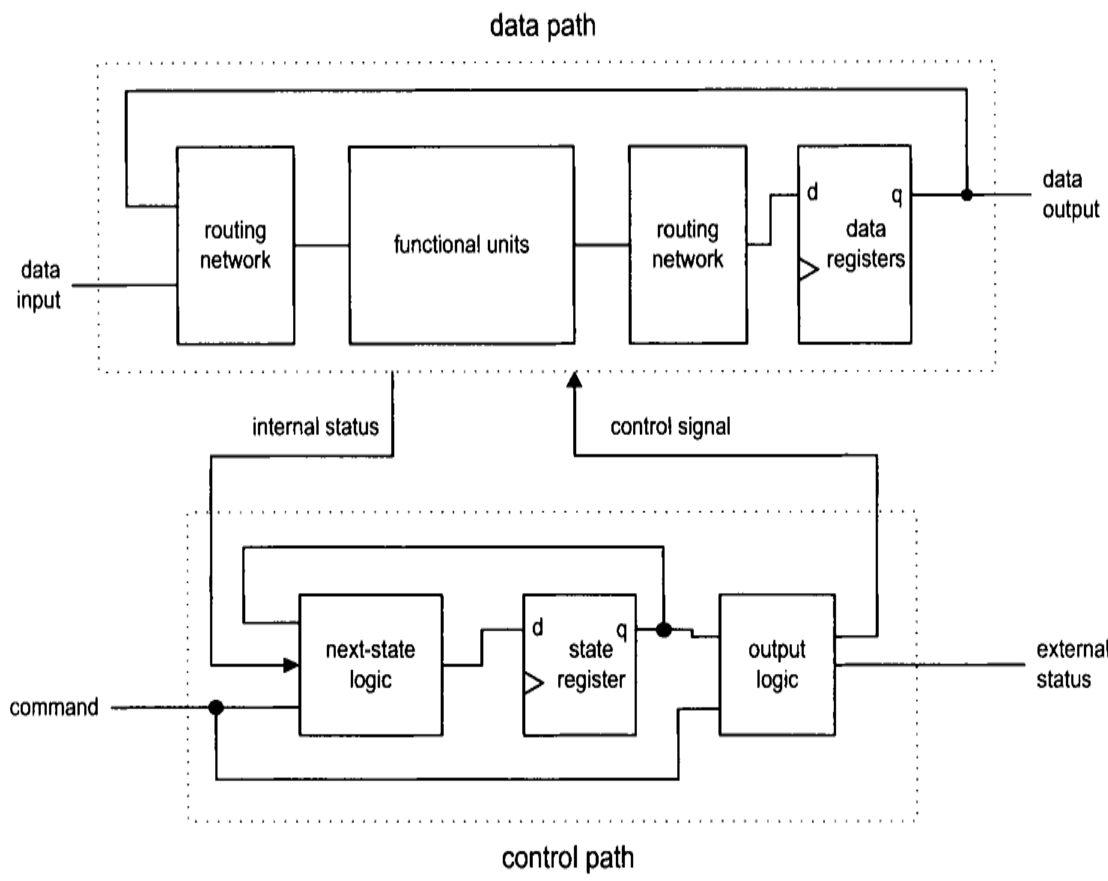


**Figure 6.5** Block diagram of an FSMD.

design is between 20 and 30 ms but is not an exact interval. This deficiency can be overcome by applying the RT methodology. In this section, we use this improved debouncing circuit to illustrate FSMD code development.

### 6.2.1 Debouncing circuit based on RT methodology

With the RT methodology, we can use an FSM to control the initiation of the timer to obtain the exact interval. The ASMD chart is shown in Figure 6.6. The circuit is expanded to include two output signals: `db_level`, which is the debounced output, and `db_tick`, which is a one-clock-cycle enable pulse asserted at the zero-to-one transition. The `zero` and `one` states mean that the `sw` input has been stabilized for 0 and 1, respectively. The `wait1` and `wait0` states are used to filter out short glitches. The `sw` signal must be stable for a certain amount of time or the transition will be treated as a glitch. The data path contains one register, q, which is 21 bits wide. Assume that the FSMD is originally in the `zero` state. When the `sw` input signal becomes 1, the FSMD moves to the `wait1` state and initializes q to "1...1". In the `wait1` state, the q decrements in each clock cycle. If `sw` remains as 1, the FSMD returns to this state repeatedly until q reaches "0...0" and then moves to the `one` state.

Recall that the 50-MHz (i.e., 20-ns period) system clock is used on the prototyping board. Since the FSMD stays in the `wait1` state for $2^{21}$ clock cycles, it is about 40 ms

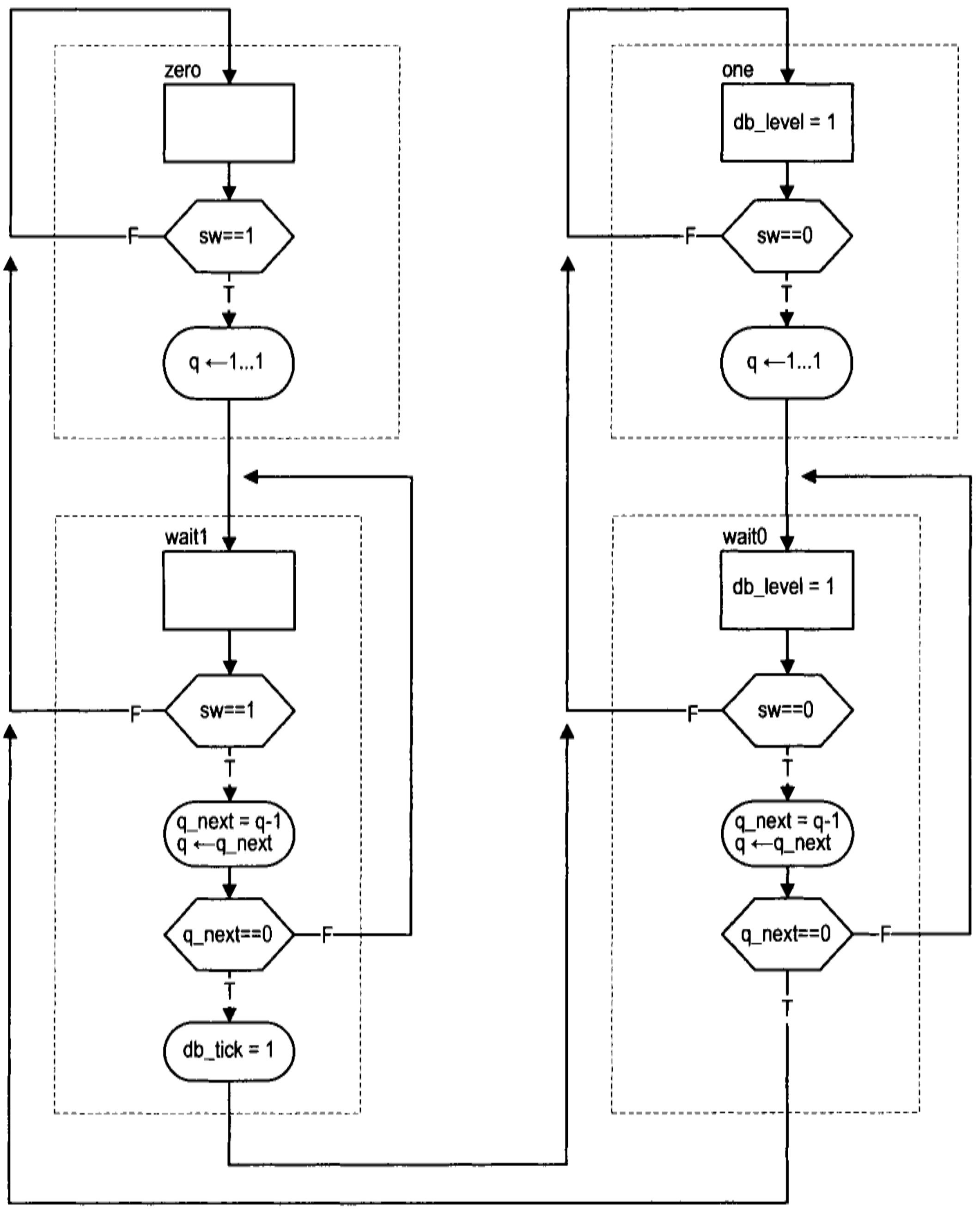


**Figure 6.6** ASMD chart of a debouncing circuit.

(i.e., $2^{21}*20$ ns). We can modify the initial value of the q register to obtain the desired wait interval.

There are two ways to derive the HDL code: one with *explicit description* of the data path components and the other with *implicit description* of the data path components.

### 6.2.2 Code with explicit data path components

The first approach to FSMD code development is to separate the control FSM and the key data path components. From an ASMD chart, we first identify the key components in the data path and the associated control signals and then describe these components in individual code segments.

The key data path component of the debouncing circuit ASMD chart is a custom 21-bit decrement counter that can:

- Be initialized with a specific value
- Count downward or pause
- Assert a status signal when the counter reaches 0

We can create a binary counter with a q_load signal to load the initial value and a q_dec signal to enable the counting. The counter also generates a q_zero status signal, which is asserted when the counter reaches zero. The complete data path is composed of the q register and the next-state logic of the custom decrement counter. A comparison circuit is included to generate the q_zero status signal. The control path consists of an FSM, which takes the sw input and the q_zero status and asserts the control signals, q_load and q_dec, according to the desired action in the ASMD chart. The HDL code follows the data path specification and the ASMD chart, and is shown in Listing 6.1.

**Listing 6.1** Debouncing circuit with an explicit data path component

```
module debounce_explicit
   (
    input wire clk, reset,
    input wire sw,
    output reg db_level, db_tick
   );

   // symbolic state declaration
   localparam  [1:0]
               zero  = 2'b00,
               wait0 = 2'b01,
               one   = 2'b10,
               wait1 = 2'b11;

   // number of counter bits (2^N * 20ns = 40ms)
   localparam N=21;

   // signal declaration
   reg [1:0] state_reg, state_next;
   reg [N-1:0] q_reg;
   wire [N-1:0] q_next;
   wire q_zero;
   reg q_load, q_dec;

   // body
// fsmd state & data registers
 always @(posedge clk, posedge reset)
    if (reset)
       begin
          state_reg <= zero;
          q_reg <= 0;
       end
    else
       begin
          state_reg <= state_next;
          q_reg <= q_next;
       end

// FSMD data path (counter) next-state logic
assign q_next = (q_load) ? {N{1'b1}} :   // load 1..1
                (q_dec)  ? q_reg - 1 :   // decrement
                           q_reg;
// status signal
assign q_zero = (q_next==0);

// FSMD control path next-state logic
always @*
begin
   state_next = state_reg;  // default state: the same
   q_load = 1'b0;           // default output: 0
   q_dec = 1'b0;            // default output: 0
   db_tick = 1'b0;          // default output: 0
   case (state_reg)
      zero:
         begin
            db_level = 1'b0;
            if (sw)
               begin
                  state_next = wait1;
                  q_load = 1'b1;
               end
         end
      wait1:
         begin
            db_level = 1'b0;
            if (sw)
               begin
                  q_dec = 1'b1;
                  if (q_zero)
                     begin
                        state_next = one;
                        db_tick = 1'b1;
                     end
               end
            else // sw==0
               state_next = zero;
         end
      one:
            begin
               db_level = 1'b1;
               if (~sw)
                  begin
                     state_next = wait0;
                     q_load = 1'b1;
                  end
            end
         wait0:
            begin
               db_level = 1'b1;
               if (~sw)
                  begin
                     q_dec = 1'b1;
                     if (q_zero)
                        state_next = zero;
                  end
               else // sw==1
                  state_next = one;
            end
         default: state_next = zero;
      endcase
   end

endmodule
```

### 6.2.3 Code with implicit data path components

An alternative coding style is to embed the RT operations within the FSM control path. Instead of explicitly defining the data path components, we just list RT operations with the corresponding FSM state. The code of the debouncing circuit is shown in Listing 6.2.

**Listing 6.2** Debouncing circuit with an implicit data path component

```
module debounce
   (
    input wire clk, reset,
    input wire sw,
    output reg db_level, db_tick
   );

   // symbolic state declaration
   localparam  [1:0]
               zero  = 2'b00,
               wait0 = 2'b01,
               one   = 2'b10,
               wait1 = 2'b11;

   // number of counter bits (2^N * 20ns = 40ms)
   localparam N=21;

   // signal declaration
   reg [N-1:0] q_reg, q_next;
reg [1:0] state_reg, state_next;

// body
// fsmd state & data registers
 always @(posedge clk, posedge reset)
    if (reset)
       begin
          state_reg <= zero;
          q_reg <= 0;
       end
    else
       begin
          state_reg <= state_next;
          q_reg <= q_next;
       end

// next-state logic & data path functional units/routing
always @*
begin
   state_next = state_reg;   // default state: the same
   q_next = q_reg;           // default q: unchnaged
   db_tick = 1'b0;           // default output: 0
   case (state_reg)
      zero:
         begin
            db_level = 1'b0;
            if (sw)
               begin
                  state_next = wait1;
                  q_next = {N{1'b1}}; // load 1..1
               end
         end
      wait1:
         begin
            db_level = 1'b0;
            if (sw)
               begin
                  q_next = q_reg - 1;
                  if (q_next==0)
                     begin
                        state_next = one;
                        db_tick = 1'b1;
                     end
               end
            else // sw==0
               state_next = zero;
         end
      one:
         begin
            db_level = 1'b1;
            if (~sw)
               begin
                  state_next = wait0;
```

```
                        q_next = {N{1'b1}}; // load 1..1
                     end
               end
            wait0:
               begin
                  db_level = 1'b1;
                  if (~sw)
                     begin
                        q_next = q_reg - 1;
                        if (q_next==0)
                           state_next = zero;
                     end
                  else // sw==1
                     state_next = one;

               end
            default: state_next = zero;
         endcase
      end

   endmodule
```

The code consists of a memory segment and a combinational logic segment. The former contains the state register of the FSM and the data register of the data path. The latter basically specifies the next-state logic of the control path FSM. Instead of generating control signals, the next data register values are specified in individual states. The next-state logic of the data path, which consists of functional units and a routing network, is created accordingly.

### 6.2.4 Comparison

Code with implicit data path components essentially follows the ASMD chart. We just convert the chart to an HDL description. Although this approach is simpler and more descriptive, we rely on synthesis software for data path construction and have less control. This can best be explained by an example. Consider the ASMD segment in Figure 6.7. The implicit description becomes

```
case (state_reg)
   s1:
     begin
       d1_next = a * b;
       ...
     end
   s2:
     begin
       d2_next = b * c;
       ...
     end
   s3:
     begin
       d3_next = a * c;
       ...
     end
```

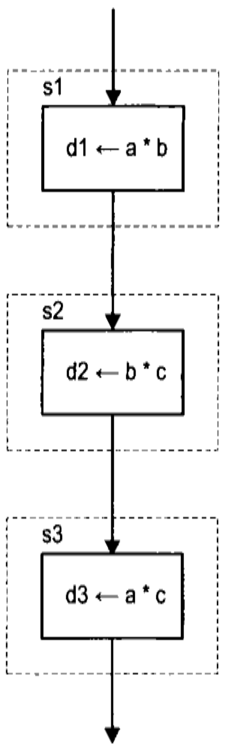


**Figure 6.7** ASMD segment with sharing opportunity.

```
   . . .
endcase
```

The synthesis software may infer three multipliers. Since a combinational multiplier is a complex circuit, it is more efficient to share the circuit. We can use explicit description to isolate the multiplier:

```
case (state_reg)
   s1:
     begin
       in1 = a;
       in2 = b;
       d1_next = m_out;
       . . .
     end
   s2:
     begin
       in1 = b;
       in2 = c;
       d2_next = m_out;
       . . .
     end
   s3:
     begin
       in1 = a;
       in2 = c;
       d3_next = m_out;
       . . .
```

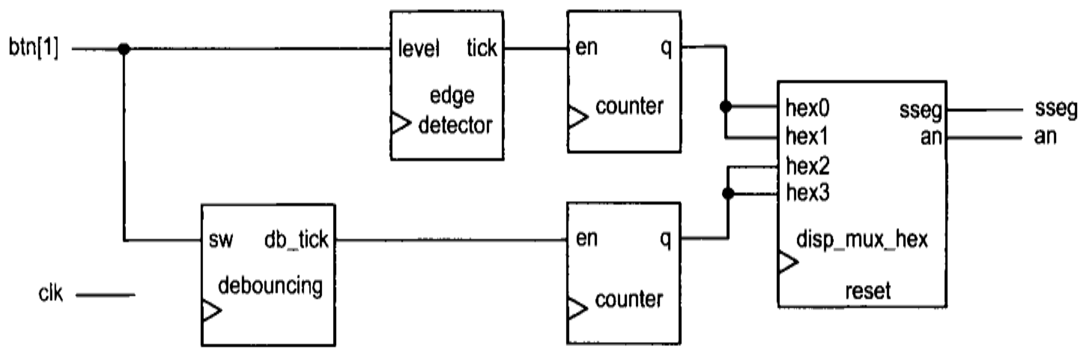


**Figure 6.8** Debouncing testing circuit.

```
        end
      . . .
   endcase

   . . .

   // explicit description of a single multiplier
   // outside the always block
   assign m_out = in1 * in2;
```

The code ensures that only one multiplier is inferred during synthesis. The implicit and explicit descriptions can be mixed for a complex FSMD design. We frequently isolate and extract complex data path components for code clarity and efficiency.

### 6.2.5 Testing circuit

The debouncing testing circuit discussed in Section 5.3.3 can be used to verify operation of the new design. Since the revised debouncing circuit's outputs include a one-clock-cycle tick signal, no edge detector is needed after the debouncing circuit. The revised block diagram is shown in Figure 6.8, and the corresponding code is shown in Listing 6.3.

**Listing 6.3** Verification circuit for a debouncing circuit

```
module debounce_fsmd_test
   (
    input wire clk, reset,
    input wire [1:0] btn,
    output wire [3:0] an,
    output wire [7:0] sseg
   );

   // signal declaration
   reg [7:0]  b_reg, d_reg;
   wire [7:0] b_next, d_next;
   reg  btn_reg;
   wire db_tick, btn_tick, clr;

   // instantiate 7-seg LED display time-multiplexing module
   disp_hex_mux disp_unit
      (.clk(clk), .reset(reset),
         .hex3(b_reg[7:4]), .hex2(b_reg[3:0]),
         .hex1(d_reg[7:4]), .hex0(d_reg[3:0]),
         .dp_in(4'b1011), .an(an), .sseg(sseg));

   // instantiate debouncing circuit
   debounce db_unit
      (.clk(clk), .reset(reset), .sw(btn[1]),
       .db_level(), .db_tick(db_tick));

   // edge detection circuit for un-debounced input
   always @(posedge clk)
       btn_reg <= btn[1];
   assign btn_tick = ~btn_reg & btn[1];

   // two counters
   assign clr = btn[0];
   always @(posedge clk)
      begin
         d_reg <= d_next;
         b_reg <= b_next;
      end
   //next-state logic for the counter
   assign b_next = (clr )      ? 0 :
                   (btn_tick) ? b_reg + 1 :
                                b_reg;
   assign d_next = (clr )      ? 0 :
                   (db_tick)  ? d_reg + 1 :
                                d_reg;

endmodule
```

## 6.3 DESIGN EXAMPLES

### 6.3.1 Fibonacci number circuit

The Fibonacci numbers constitute a sequence defined as

$$fib(i) = \begin{cases} 0 & \text{if } i = 0 \\ 1 & \text{if } i = 1 \\ fib(i-1) + fib(i-2) & \text{if } i > 1 \end{cases}$$

One way to calculate $fib(i)$ is to construct the function iteratively, from 0 to the desired $i$. This approach requires two temporary registers to store the two most recently calculated values [i.e., $fib(i-1)$ and $fib(i-2)$] and one index register to keep track of the number of iterations. The ASMD chart is shown in Figure 6.9, in which `t1` and `t0` are temporary storage registers and `n` is the index register. In addition to the regular data input and output signals, `i` and `f`, we include a command signal, `start`, which signals the beginning of operation, and two status signals: `ready`, which indicates that the circuit is idle and ready to take new input, and `done_tick`, which is asserted for one clock cycle when the operation is completed. Since this circuit, like many other FSMD designs, is probably a part of a larger system, these signals are needed to interface with other subsystems.

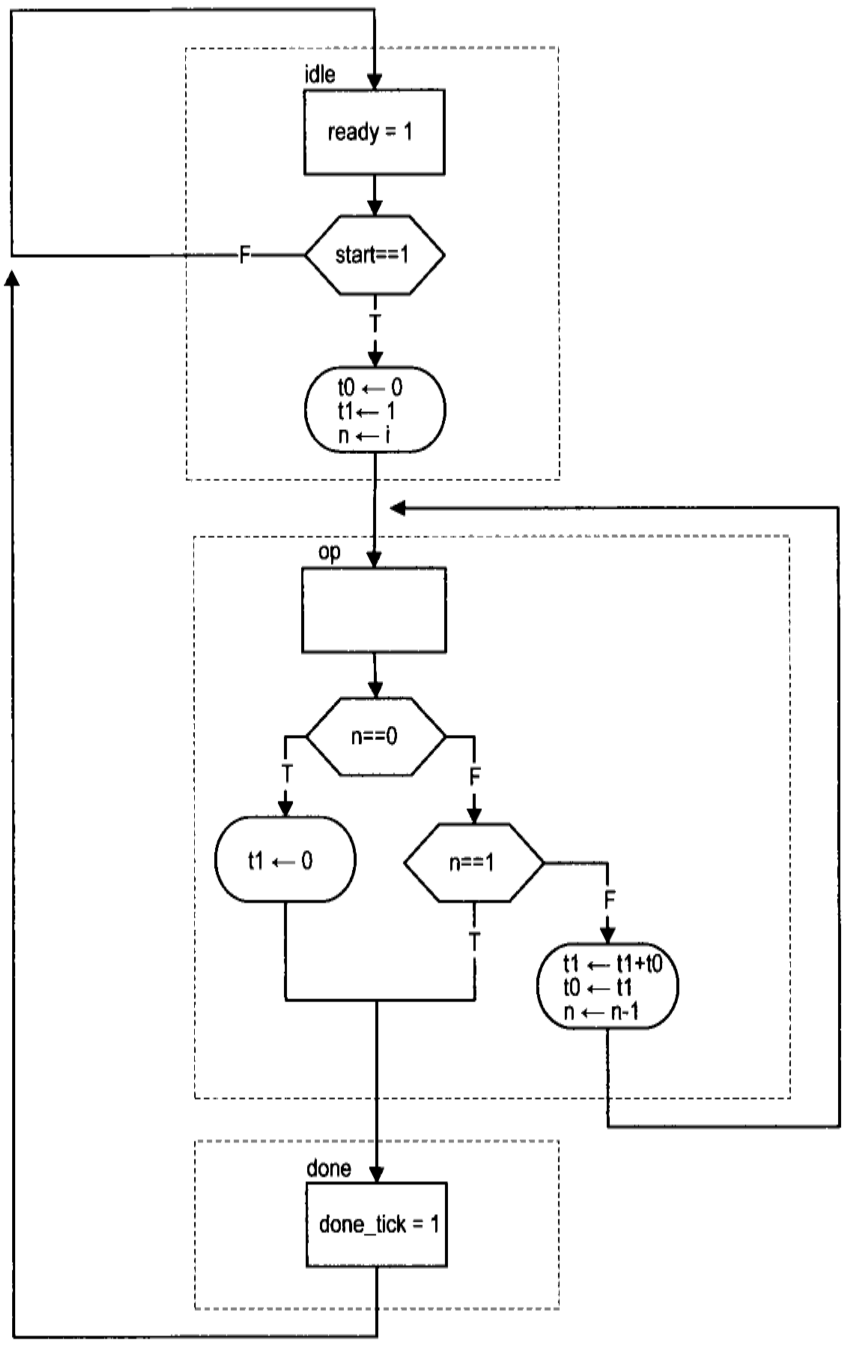


**Figure 6.9** ASMD chart of a Fibonacci circuit.

The ASMD chart has three states. The `idle` state indicates that the circuit is currently idle. When `start` is asserted, the FSMD moves to the op state and loads initial values to three registers. The `t0` and `t1` registers are loaded with 0 and 1, which represent $fib(0)$ and $fib(1)$, respectively. The `n` register is loaded with `i`, the desired number of iterations.

The main computation is iterated through the op state by three RT operations:

- `t1` $\leftarrow$ `t1 + t0`
- `t0` $\leftarrow$ `t1`
- `n` $\leftarrow$ `n - 1`

The first two RT operations obtain a new value and store the two most recently calculated values in `t1` and `t0`. The third RT operation decrements the iteration index. The iteration ended when `n` reaches 1 or its initial value is 0 [i.e., $fib(0)$]. Unlike a regular flowchart, the operations in an ASMD block can be performed concurrently in the same clock cycle. We put all comparison and RT operations in the op state to reduce the computation time. Note that the new values of the `t1` and `t0` registers are loaded at the same time when the FSMD exits the op state (i.e., at the next rising edge of the clock). Thus, the original value of `t1`, not `t1+t0`, is stored to `t0`. The purpose of the done state is to generate the one-clock-cycle `done_tick` signal to indicate completion of the computation. This state can be omitted if this status signal is not needed.

The code follows the ASMD chart and is shown in Listing 6.4. Note that the Fibonacci function grows rapidly and the output signal should be wide enough to accommodate the desired result.

**Listing 6.4** Fibonacci number circuit

```
module fib
   (
    input wire clk, reset,
    input wire start,
    input wire [4:0] i,
    output reg ready, done_tick,
    output wire [19:0] f
   );

   // symbolic state declaration
   localparam [1:0]
      idle = 2'b00,
      op   = 2'b01,
      done = 2'b10;

   // signal declaration
   reg [1:0] state_reg, state_next;
   reg [19:0] t0_reg, t0_next, t1_reg, t1_next;
   reg [4:0] n_reg, n_next;

   // body
   // FSMD state & data registers
   always @(posedge clk, posedge reset)
      if (reset)
         begin
            state_reg <= idle;
            t0_reg <= 0;
            t1_reg <= 0;
            n_reg <= 0;
         end
      else
         begin
            state_reg <= state_next;
            t0_reg <= t0_next;
            t1_reg <= t1_next;
            n_reg <= n_next;
         end
   // FSMD next-state logic
   always @*
   begin
      state_next = state_reg;
      ready = 1'b0;
      done_tick = 1'b0;
      t0_next = t0_reg;
      t1_next = t1_reg;
      n_next = n_reg;
      case (state_reg)
         idle:
            begin
               ready = 1'b1;
               if (start)
                  begin
                     t0_next = 0;
                     t1_next = 20'd1;
                     n_next = i;
                     state_next = op;
                  end
            end
         op:
            if (n_reg==0)
               begin
                  t1_next = 0;
                  state_next = done;
               end
            else if (n_reg==1)
               state_next = done;
            else
               begin
                  t1_next = t1_reg + t0_reg;
                  t0_next = t1_reg;
                  n_next = n_reg - 1;
              end
         done:
            begin
               done_tick = 1'b1;
               state_next = idle;
            end
         default: state_next = idle;
      endcase
   end
   // output
```

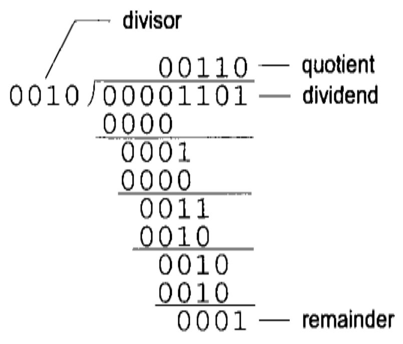


**Figure 6.10** Long division of two 4-bit unsigned integers.

```
    assign f = t1_reg;

endmodule
```

### 6.3.2 Division circuit

Because of complexity, the division operator cannot be synthesized automatically. We use an FSMD to implement the long-division algorithm in this subsection. The algorithm is illustrated by the division of two 4-bit unsigned integers in Figure 6.10. The algorithm can be summarized as follows:

1. Double the dividend width by appending 0's in front and align the divisor to the leftmost bit of the extended dividend.
2. If the corresponding dividend bits are greater than or equal to the divisor, subtract the divisor from the dividend bits and make the corresponding quotient bit 1. Otherwise, keep the original dividend bits and make the quotient bit 0.
3. Append one additional dividend bit to the previous result and shift the divisor to the right one position.
4. Repeat steps 2 and 3 until all dividend bits are used.

The sketch of the data path is shown in Figure 6.11. Initially, the divisor is stored in the `d` register and the extended dividend is stored in the `rh` and `rl` registers. In each iteration, the `rh` and `rl` registers are shifted to the left one position. This corresponds to shifting the divisor to the right of the preceding algorithm. We can then compare `rh` and `d` and perform subtraction if `rh` is greater than or equal to `d`. When `rh` and `rl` are shifted to the left, the rightmost bit of `rl` becomes available. It can be used to store the current quotient bit. After we iterate through all dividend bits, the result of the last subtraction is stored in `rh` and becomes the remainder of the division, and all quotients are shifted into `rl`.

The ASMD chart of the division circuit is somewhat similar to that of the previous Fibonacci circuit. The FSMD consists of four states: `idle`, `op`, `last`, and `done`. To make the code clear, we extract the *compare and subtract* circuit to separate code segments. The main computation is performed in the `op` state, in which the dividend bits and divisor are compared and subtracted and then shifted left 1 bit. Note that the remainder should not be shifted in the last iteration. We create a separate state, `last`, to accommodate this special requirement. As in the preceding example, the purpose of the `done` state is to generate a one-clock-cycle `done_tick` signal to indicate completion of the computation. The code is shown in Listing 6.5.

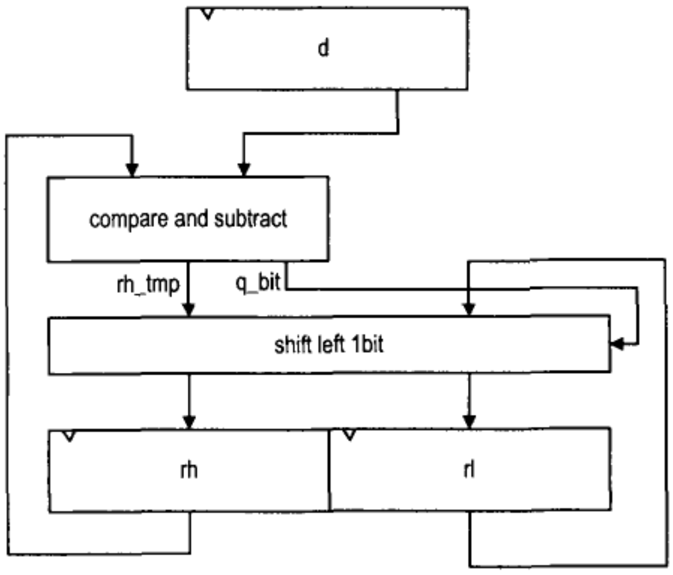


**Figure 6.11** Sketch of division circuit's data path.

**Listing 6.5** Division circuit

```
module div
   #(
     parameter W = 8,
               CBIT = 4    // CBIT=log2(W)+1
    )
   (
    input wire clk, reset,
    input wire start,
    input wire [W-1:0] dvsr, dvnd,
    output reg ready, done_tick,
    output wire [W-1:0] quo, rmd
   );

   // symbolic state declaration
   localparam [1:0]
      idle = 2'b00,
      op   = 2'b01,
      last = 2'b10,
      done = 2'b11;

   // signal declaration
   reg [1:0] state_reg, state_next;
   reg [W-1:0] rh_reg, rh_next, rl_reg, rl_next, rh_tmp;
   reg [W-1:0] d_reg, d_next;
   reg [CBIT-1:0] n_reg, n_next;
   reg q_bit;

   // body
   // FSMD state & data registers
```

```
always @(posedge clk, posedge reset)
   if (reset)
      begin
         state_reg <= idle;
         rh_reg <= 0;
         rl_reg <= 0;
         d_reg <= 0;
         n_reg <= 0;
      end
   else
      begin
         state_reg <= state_next;
         rh_reg <= rh_next;
         rl_reg <= rl_next;
         d_reg <= d_next;
         n_reg <= n_next;
      end

// FSMD next-state logic
always @*
begin
   state_next = state_reg;
   ready = 1'b0;
   done_tick = 1'b0;
   rh_next = rh_reg;
   rl_next = rl_reg;
   d_next = d_reg;
   n_next = n_reg;
   case (state_reg)
      idle:
         begin
            ready = 1'b1;
            if (start)
               begin
                  rh_next = 0;
                  rl_next = dvnd;  // dividend
                  d_next = dvsr;   // divisor
                  n_next = CBIT;   // index
                  state_next = op;
               end
         end
      op:
         begin
            // shift rh and rl left
            rl_next = {rl_reg[W-2:0], q_bit};
            rh_next = {rh_tmp[W-2:0], rl_reg[W-1]};
            // decrease index
            n_next = n_reg - 1;
            if (n_next==1)
               state_next = last;
         end
      last: // last iteration
         begin
```

```
                  rl_next = {rl_reg[W-2:0], q_bit};
                  rh_next = rh_tmp;
                  state_next = done;
               end
            done:
               begin
                  done_tick = 1'b1;
                  state_next = idle;
               end
            default: state_next = idle;
         endcase
   end

   // compare and subtract circuit
   always @*
      if (rh_reg >= d_reg)
         begin
            rh_tmp = rh_reg - d_reg;
            q_bit = 1'b1;
         end
      else
         begin
            rh_tmp = rh_reg;
            q_bit = 1'b0;
         end

   //output
   assign quo = rl_reg;
   assign rmd = rh_reg;

endmodule
```

### 6.3.3 Binary-to-BCD conversion circuit

We discussed the BCD format in Section 4.5.2. In this format, a decimal number is represented as a sequence of 4-bit BCD digits. A binary-to-BCD conversion circuit converts a binary number to the BCD format. For example, the binary number "0010 0000 0000" becomes "0101 0001 0010" (i.e., $512_{10}$) after conversion.

The binary-to-BCD conversion can be processed by a special BCD shift register, which is divided into 4-bit groups internally, each representing a BCD digit. Shifting a BCD sequence to the left requires adjustment if a BCD digit is greater than $9_{10}$ after shifting. For example, if a BCD sequence is "0001 0111" (i.e., $17_{10}$), it should become "0011 0100" (i.e., $34_{10}$) rather than "0010 1110". The adjustment requires subtracting $10_{10}$ (i.e., "1010") from the right BCD digit and adding 1 (which can be considered as a carry-out) to the next BCD digit. Note that subtracting $10_{10}$ is equivalent to adding $6_{10}$ for a 4-bit binary number. Thus, the foregoing adjustment can also be achieved by adding $6_{10}$ to the right BCD digit. The carry-out bit is generated automatically in this process.

In the actual implementation, it is more efficient to first perform the necessary adjustment on a BCD digit and then shift. We can check whether a BCD digit is greater than $4_{10}$ and, if this is the case, add $3_{10}$ to the digit. After all the BCD digits are corrected, we can then shift the entire register to the left one position. A binary-to-BCD conversion circuit can

**Table 6.1** Binary-to-BCD conversion example

| Operation | | Special BCD shift register | | | Binary input |
|---|---|---|---|---|---|
| | | **BCD digit 2** | **BCD digit 1** | **BCD digit 0** | |
| Initial | | | | | 111 1111 |
| Bit 6 | no adjustment<br>shift left 1 bit | | | 1<br>$(1_{10})$ | 11 1111 |
| Bit 5 | no adjustment<br>shift left 1 bit | | | 11<br>$(3_{10})$ | 1 1111 |
| Bit 4 | no adjustment<br>shift left 1 bit | | | 111<br>$(7_{10})$ | 1111 |
| Bit 3 | BCD digit 0 adjustment<br>shift left 1 bit | | 1<br>$(1_{10})$ | 1010<br>0101<br>$(5_{10})$ | 111 |
| Bit 2 | BCD digit 0 adjustment<br>shift left 1 bit | | 1<br>11<br>$(3_{10})$ | 1000<br>0001<br>$(1_{10})$ | 11 |
| Bit 1 | no adjustment<br>shift left 1 bit | | 110<br>$(6_{10})$ | 0011<br>$(3_{10})$ | 1 |
| Bit 0 | BCD digit 1 adjustment<br>shift left 1 bit | 1<br>$(1_{10})$ | 1001<br>0010<br>$(2_{10})$ | 0011<br>0111<br>$(7_{10})$ | |

be constructed by shifting the binary input to a BCD shift register bit by bit, from MSB to LSB. Its operation can be summarized as follows:

1. For each 4-bit BCD digit in a BCD shift register, check whether the digit is greater than 4. If this is the case, add $3_{10}$ to the digit.
2. Shift the entire BCD register left one position and shift in the MSB of the input binary sequence to the LSB of the BCD register.
3. Repeat steps 1 and 2 until all input bits are used.

The conversion process of a 7-bit binary input, "111 1111" (i.e., $127_{10}$), is demonstrated in Table 6.1.

The code of a 13-bit conversion circuit is shown in Listing 6.6. It uses a simple FSMD to control the overall operation. When the `start` signal is asserted, the binary input is stored to the p2s register. The FSM then iterates through the 13 bits, similar to the process described in previous examples. Four adjustment circuits are used to correct the four BCD digits. For clarity, they are isolated from the next-state logic and described in a separate code segment.

**Listing 6.6** Binary-to-BCD conversion circuit

```
module bin2bcd
   (
    input wire clk, reset,
    input wire start,
    input wire [12:0] bin,
    output reg ready, done_tick,
    output wire [3:0] bcd3, bcd2, bcd1, bcd0
   );

   // symbolic state declaration
   localparam [1:0]
      idle = 2'b00,
      op   = 2'b01,
      done = 2'b10;

   // signal declaration
   reg [1:0] state_reg, state_next;
   reg [12:0] p2s_reg, p2s_next;
   reg [3:0] n_reg, n_next;
   reg [3:0] bcd3_reg, bcd2_reg, bcd1_reg, bcd0_reg;
   reg [3:0] bcd3_next, bcd2_next, bcd1_next, bcd0_next;
   wire [3:0] bcd3_tmp, bcd2_tmp, bcd1_tmp, bcd0_tmp;


   // body
   // FSMD state & data registers
   always @(posedge clk, posedge reset)
      if (reset)
         begin
            state_reg <= idle;
            p2s_reg <= 0;
            n_reg <= 0;
            bcd3_reg <= 0;
            bcd2_reg <= 0;
            bcd1_reg <= 0;
            bcd0_reg <= 0;
         end
      else
         begin
            state_reg <= state_next;
            p2s_reg <= p2s_next;
            n_reg <= n_next;
            bcd3_reg <= bcd3_next;
            bcd2_reg <= bcd2_next;
            bcd1_reg <= bcd1_next;
            bcd0_reg <= bcd0_next;
         end


   // FSMD next-state logic
   always @*
   begin
```

```
      state_next = state_reg;
      ready = 1'b0;
      done_tick = 1'b0;
      p2s_next = p2s_reg;
      bcd0_next = bcd0_reg;
      bcd1_next = bcd1_reg;
      bcd2_next = bcd2_reg;
      bcd3_next = bcd3_reg;
      n_next = n_reg;
      case (state_reg)
         idle:
            begin
               ready = 1'b1;
               if (start)
                  begin
                     state_next = op;
                     bcd3_next = 0;
                     bcd2_next = 0;
                     bcd1_next = 0;
                     bcd0_next = 0;
                     n_next = 4'b1101; // index
                     p2s_next = bin;   // shift register
                     state_next = op;
                  end
            end
         op:
            begin
               // shift in binary bit
               p2s_next = p2s_reg << 1;
               // shift 4 BCD digits
               //{bcd3_next, bcd2_next, bcd1_next, bcd0_next}=
               //{bcd3_tmp[2:0], bcd2_tmp, bcd1_tmp, bcd0_tmp,
               // p2s_reg[12]}

               bcd0_next = {bcd0_tmp[2:0], p2s_reg[12]};
               bcd1_next = {bcd1_tmp[2:0], bcd0_tmp[3]};
               bcd2_next = {bcd2_tmp[2:0], bcd1_tmp[3]};
               bcd3_next = {bcd3_tmp[2:0], bcd2_tmp[3]};
               n_next = n_reg - 1;
               if (n_next==0)
                   state_next = done;
            end
         done:
            begin
               done_tick = 1'b1;
               state_next = idle;
            end
         default: state_next = idle;
      endcase
   end

   // data path function units
   assign bcd0_tmp = (bcd0_reg > 4) ? bcd0_reg+3 : bcd0_reg;
   assign bcd1_tmp = (bcd1_reg > 4) ? bcd1_reg+3 : bcd1_reg;
   assign bcd2_tmp = (bcd2_reg > 4) ? bcd2_reg+3 : bcd2_reg;
   assign bcd3_tmp = (bcd3_reg > 4) ? bcd3_reg+3 : bcd3_reg;

   // output
   assign bcd0 = bcd0_reg;
   assign bcd1 = bcd1_reg;
   assign bcd2 = bcd2_reg;
   assign bcd3 = bcd3_reg;

endmodule
```

### 6.3.4 Period counter

A period counter measures the period of a periodic input waveform. One way to construct the circuit is to count the number of clock cycles between two rising edges of the input signal. Since the frequency of the system clock is known, the period of the input signal can be derived accordingly. For example, if the frequency of the system clock is $f$ and the number of clock cycles between two rising edges is $N$, the period of the input signal is $N * \frac{1}{f}$.

The design in this subsection measures the period in milliseconds. Its ASMD chart is shown in Figure 6.12. The period counter takes a measurement when the `start` signal is asserted. We use a rising-edge detection circuit to generate a one-clock-cycle tick, edge, to indicate the rising edge of the input waveform. After `start` is asserted, the FSMD moves to the `waite` state to wait for the first rising edge of the input. It then moves to the `count` state when the next rising edge of the input is detected. In the `count` state, we use two registers to keep track of the time. The `t` register counts for 50,000 clock cycles, from 0 to 49,999, and then wraps around. Since the period of the system clock is 20 ns, the `t` register takes 1 ms to circulate through 50,000 cycles. The p register counts in terms of milliseconds. It is incremented once when the `t` register reaches 49,999. When the FSMD exits the `count` state, the period of the input waveform is stored in the p register and its unit is milliseconds. The FSMD asserts the `done_tick` signal in the `done` state, as in previous examples.

The code follows the ASMD chart and is shown in Listing 6.7. We use a constant, `CLK_MS_COUNT`, for the boundary of the millisecond counter. It can be replaced if a different measurement unit is desired.

**Listing 6.7** Period counter

```
module period_counter
   (
    input wire clk, reset,
    input wire start, si,
    output reg ready, done_tick,
    output wire [9:0] prd
   );

   // symbolic state declaration
   localparam [1:0]
      idle  = 2'b00,
      waite = 2'b01,
      count = 2'b10,
```

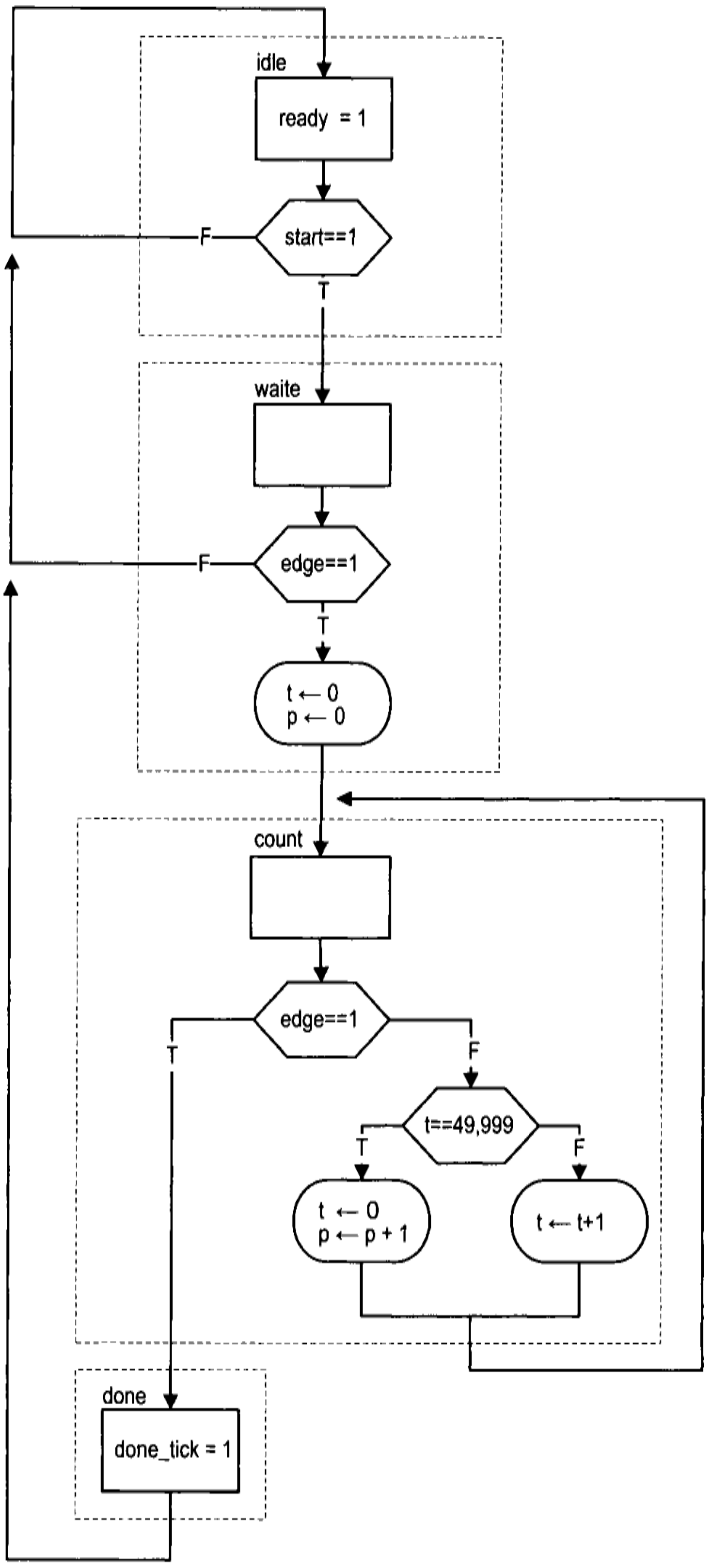


**Figure 6.12** ASMD chart of a period counter.

```
      done   = 2'b11;

   // constant declaration
   localparam CLK_MS_COUNT= 50000; // 1 ms tick

   // signal declaration
   reg [1:0] state_reg, state_next;
   reg [15:0] t_reg, t_next;  // up to 50000
   reg [9:0] p_reg, p_next;   // up to 1 sec
   reg delay_reg;
   wire edg;

   // body
   // FSMD state & data registers
   always @(posedge clk, posedge reset)
      if (reset)
         begin
            state_reg <= idle;
            t_reg <= 0;
            p_reg <= 0;
            delay_reg <= 0;
         end
      else
         begin
            state_reg <= state_next;
            t_reg <= t_next;
            p_reg <= p_next;
            delay_reg <= si;
         end

   // rising-edge tick
   assign edg = ~delay_reg & si;

   // FSMD next-state logic
   always @*
   begin
      state_next = state_reg;
      ready = 1'b0;
      done_tick = 1'b0;
      p_next = p_reg;
      t_next = t_reg;
      case (state_reg)
         idle:
             begin
                ready = 1'b1;
                if (start)
                   state_next = waite;
             end
         waite: // wait for the first edge
            if (edg)
               begin
                  state_next = count;
                  t_next = 0;
                  p_next = 0;
               end
         count:
            if (edg)   // 2nd edge arrived
               state_next = done;
            else        // otherwise count
               if (t_reg == CLK_MS_COUNT-1) // 1 ms tick
                  begin
                     t_next = 0;
                     p_next = p_reg + 1;
                  end
               else
                  t_next = t_reg + 1;
         done:
            begin
               done_tick = 1'b1;
               state_next = idle;
            end
         default: state_next = idle;
      endcase
   end

   //ouput
   assign prd = p_reg;

endmodule
```

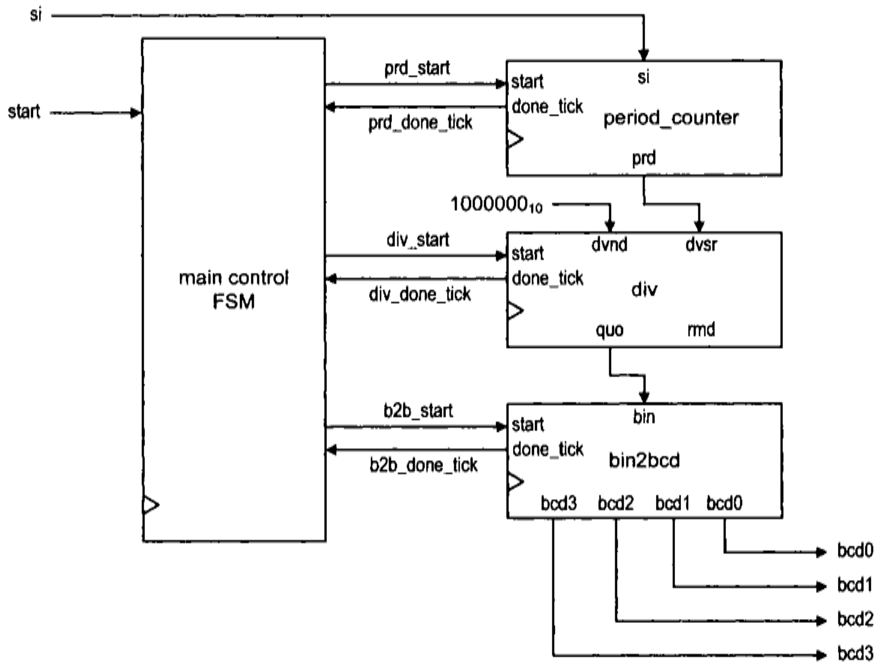
### 6.3.5 Accurate low-frequency counter

A frequency counter measures the frequency of a periodic input waveform. The common way to construct a frequency counter is to count the number of input pulses in a fixed amount of time, say, 1 second. Although this approach is fine for high-frequency input, it cannot measure a low-frequency signal accurately. For example, if the input is around 2 Hz, the measurement cannot tell whether it is 2.123 Hz or 2.567 Hz. Recall that the frequency is the reciprocal of the period (i.e., $frequency = \frac{1}{period}$). An alternative approach is to measure the period of the signal and then take the reciprocal to find the frequency. We use this approach to implement a low-frequency counter in this subsection.

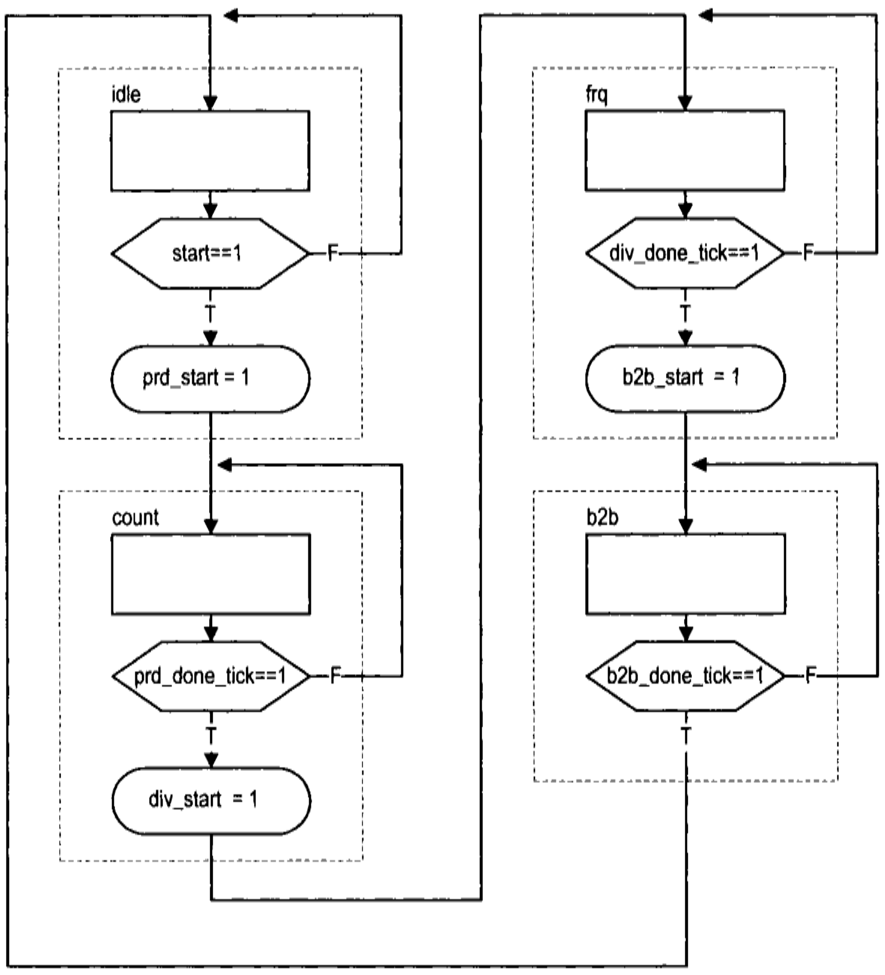
This design example demonstrates how to use the previously designed parts to construct a large system. For simplicity, we assume that the frequency of the input is between 1 and 10 Hz (i.e., the period is between 100 and 1000 ms). The operation of this circuit includes three tasks:

1. Measure the period.
2. Find the frequency by performing a division operation.
3. Convert the binary number to BCD format.

We can use the period counter, division circuit, and binary-to-BCD converter to perform the three tasks and create another FSM as the master control to sequence and coordinate the operation of the three circuits. The block diagram is shown in Figure 6.13(a), and the ASM chart of the master control is shown in Figure 6.13(b). The FSM uses the `start` and `done_tick` signals of these circuits to initialize each task and to detect completion of the task. The code is shown in Listing 6.8.



(a) Top-level block diagram



(b) ASM chart of main control

**Figure 6.13** Accurate low-frequency counter.

**Listing 6.8** Low-frequency counter

```
module low_freq_counter
   (
    input wire clk, reset,
    input wire start, si,
    output wire [3:0] bcd3, bcd2, bcd1, bcd0
   );

   // symbolic state declaration
   localparam  [1:0]
               idle  = 2'b00,
               count = 2'b01,
               frq   = 2'b10,
               b2b   = 2'b11;

   // signal declaration
   reg [1:0] state_reg, state_next;
   wire [9:0] prd;
   wire [19:0] dvsr, dvnd, quo;
   reg prd_start, div_start, b2b_start;
   wire prd_done_tick, div_done_tick, b2b_done_tick;

   //===============================================
   // component instantiation
   //===============================================
   // instantiate period counter
   period_counter prd_count_unit
      (.clk(clk), .reset(reset), .start(prd_start), .si(si),
       .ready(), .done_tick(prd_done_tick), .prd(prd));
   // instantiate division circuit
   div #(.W(20), .CBIT(5)) div_unit
      (.clk(clk), .reset(reset), .start(div_start),
       .dvsr(dvsr), .dvnd(dvnd), .quo(quo), .rmd(),
       .ready(), .done_tick(div_done_tick));
   // instantiate binary-to-BCD convertor
   bin2bcd b2b_unit
      (.clk(clk), .reset(reset), .start(b2b_start),
       .bin(quo[12:0]), .ready(), .done_tick(b2b_done_tick),
       .bcd3(bcd3), .bcd2(bcd2), .bcd1(bcd1), .bcd0(bcd0));
   // signal width extension
   assign dvnd = 20'd1000000;
   assign dvsr = {10'b0, prd};

   //===============================================
   // master FSM
   //===============================================
   always @(posedge clk, posedge reset)
     if (reset)
        state_reg <= idle;
     else
        state_reg <= state_next;

   always @*
   begin
      state_next = state_reg;
      prd_start = 1'b0;
      div_start = 1'b0;
      b2b_start = 1'b0;
      case (state_reg)
         idle:
            if (start)
               begin
                  prd_start = 1'b1;
                  state_next = count;
               end
         count:
            if (prd_done_tick)
               begin
                  div_start = 1'b1;
                  state_next = frq;
               end
         frq:
            if (div_done_tick)
               begin
                  b2b_start = 1'b1;
                  state_next = b2b;
               end
         b2b:
            if (b2b_done_tick)
               state_next = idle;
      endcase
   end

endmodule
```

## 6.4 BIBLIOGRAPHIC NOTES

FSMD is usually discussed in the context of *high-level synthesis*. *Principles of Digital Design* by D. D. Gajski contains a comprehensive chapter discussing relevant issues and algorithms of FSMD design and implementation.

## 6.5 SUGGESTED EXPERIMENTS

### 6.5.1 Alternative debouncing circuit

Consider the alternative debouncing circuit in Experiment 5.5.2. Redesign the circuit using the RT methodology:

1. Derive the ASMD chart for the circuit.
2. Derive the HDL code based on the ASMD chart.
3. Replace the debouncing circuit in Section 6.2.5 with the alternative design and verify its operation.

### 6.5.2 BCD-to-binary conversion circuit

A BCD-to-binary conversion converts a BCD number to the equivalent binary representation. Assume that the input is an 8-bit signal in BCD format (i.e., two BCD digits) and the output is a 7-bit signal in binary representation. Follow the procedure in Section 6.3.3 to design a BCD-to-binary conversion circuit:

1. Derive the conversion algorithm and ASMD chart.
2. Derive the HDL code based on the ASMD chart.
3. Derive a testbench and use simulation to verify operation of the code.
4. Synthesize the circuit, program the FPGA, and verify its operation.

### 6.5.3 Fibonacci circuit with BCD I/O: design approach 1

To make the Fibonacci circuit more user friendly, we can modify the circuit to use the BCD format for the input and output. Assume that the input is an 8-bit signal in BCD format (i.e., two BCD digits) and the output is displayed as four BCD digits on the seven-segment LED display. Furthermore, the LED will display "9999" if the resulting Fibonacci number is larger than 9999 (i.e., overflow). The operation can be done in three steps: convert input to the binary format, compute the Fibonacci number, and convert the result back to BCD format.

The first design approach is to follow the procedure outlined in Section 6.3.5. We first construct three smaller subsystems, which are the BCD-to-binary conversion circuit, Fibonacci circuit, and binary-to-BCD conversion circuit, and then use a master FSM to control the overall operation. Design the circuit as follows:

1. Implement the BCD-to-binary conversion circuit in Experiment 6.5.2.
2. Modify the Fibonacci number circuit in Section 6.3.1 to include an output signal to indicate the overflow condition.
3. Derive the top-level block diagram and the master control FSM state diagram.
4. Derive the HDL code.
5. Derive a testbench and use simulation to verify operation of the code.
6. Synthesize the circuit, program the FPGA, and verify its operation.

### 6.5.4 Fibonacci circuit with BCD I/O: design approach 2

An alternative to the "subsystem approach" in Experiment 6.5.3 is to integrate the three subsystems into a single system and derive a customized FSMD for this particular application. The approach eliminates the overhead of the control FSM and provides opportunities to share registers among the three tasks. Design the circuit as follows:

1. Redesign the circuit of Experiment 6.5.3 using one FSMD. The design should eliminate all unnecessary circuits and states, such as the various `done_tick` signals and the `done` states, and exploit the opportunity to share and reuse the registers in different steps.
2. Derive the ASMD chart.
3. Derive the HDL code based on the ASMD chart.
4. Derive a testbench and use simulation to verify operation of the code.
5. Synthesize the circuit, program the FPGA, and verify its operation.
6. Check the synthesis report and compare the number of LEs used in the two approaches.
7. Calculate the number of clock cycles required to complete the operation in the two approaches.

### 6.5.5 Auto-scaled low-frequency counter

The operation of the low-frequency counter in Section 6.3.5 is very restricted. The frequency range of the input signal is limited between 1 and 10 Hz. It loses accuracy when the frequency is beyond this range. Recall that the accuracy of this frequency counter depends on the accuracy of the period counter of Section 6.3.5, which counts in terms of millisecond ticks. We can modify the `t` counter to generate a microsecond tick (i.e., counting from 0 to 49) and increase the accuracy 1000-fold. This allows the range of the frequency counter to increase to 9999 Hz and still maintain at least four-digit accuracy.

Using a microsecond tick introduces more than four accuracy digits for low-frequency input, and the number must be shifted and truncated to be displayed on the seven-segment LED. An auto-scaled low-frequency counter performs the adjustment automatically, displays the four most significant digits, and places a decimal point in the proper place. For example, according to their range, the frequency measurements will be shown as "1.234", "12.34", "123.4", or "1234".

The auto-scaled low-frequency counter needs an additional BCD adjustment circuit. It first checks whether the most significant BCD digit (i.e., the four MSBs) of a BCD sequence is zero. If this is the case, the circuit shifts the BCD sequence to the left four positions and increments the decimal point counter. The operation is repeated until the most significant BCD digit is not "0000".

The complete auto-scaled low-frequency counter can be implemented as follows:

1. Modify the period counter to use the microsecond tick.
2. Extend the size of the binary-to-BCD conversion circuit.
3. Derive the ASMD chart for the BCD adjustment circuit and the HDL code.
4. Modify the control FSM to include the BCD adjustment in the last step.
5. Design a simple decoding circuit that uses the decimal-point counter's output to activate the desired decimal point of the seven-segment LED display.
6. Derive a testbench and use simulation to verify operation of the code.
7. Synthesize the circuit, program the FPGA, and verify its operation.

### 6.5.6 Reaction timer

Eye–hand coordination is the ability of the eyes and hands to work together to perform a task. A reaction timer circuit measures how fast a human hand can respond after a person sees a visual stimulus. This circuit operates as follows:

1. The circuit has three input pushbuttons, corresponding to the `clear`, `start`, and `stop` signals. It uses a single discrete LED as the visual stimulus and displays relevant information on the seven-segment LED display.
2. A user pushes the `clear` button to force the circuit to return to the initial state, in which the seven-segment LED shows a welcome message, "HI," and the stimulus LED is off.
3. When ready, the user pushes the `start` button to initiate the test. The seven-segment LED goes off.
4. After a random interval between 2 and 15 seconds, the stimulus LED goes on and the timer starts to count upward. The timer increases every millisecond and its value is displayed in the format of "0.000" second on the seven-segment LED.
5. After the stimulus LED goes on, the user should try to push the `stop` button as soon as possible. The timer pauses counting once the `stop` button is asserted. The seven-

segment LED shows the reaction time. It should be around 0.15 to 0.30 second for most people.

6. If the `stop` button is not pushed, the timer stops after 1 second and displays "1.000".
7. If the `stop` button is pushed before the stimulus LED goes on, the circuit displays "9.999" on the seven-segment LED and stops.

Design the circuit as follows:

1. Derive the ASMD chart.
2. Derive the HDL code based on the ASMD chart.
3. Synthesize the circuit, program the FPGA, and verify its operation.

### 6.5.7 Babbage difference engine emulation circuit

The Babbage difference engine is a mechanical digital computation device designed to tabulate a polynomial function. It was proposed by Charles Babbage, an English mathematician, in the nineteenth century. The engine is based on Newton's method of differences and avoids the need for multiplication. For example, consider a second-order polynomial $f(n) = 2n^2 + 3n + 5$. We can find the difference between $f(n)$ and $f(n-1)$:

$$f(n) - f(n-1) = 4n + 1$$

Assume that $n$ is an integer and $n \geq 0$. The $f(n)$ can be defined recursively as

$$f(n) = \begin{cases} 5 & \text{if } n = 0 \\ f(n-1) + 4n + 1 & \text{if } n > 0 \end{cases}$$

This process can be repeated for the $4n + 1$ expression. Let $g(n) = 4n + 1$. We can find the difference between $g(n)$ and $g(n-1)$:

$$g(n) - g(n-1) = 4$$

The $g(n)$ can be defined recursively as

$$g(n) = \begin{cases} 5 & \text{if } n = 1 \\ g(n-1) + 4 & \text{if } n > 1 \end{cases}$$

and $f(n)$ can be rewritten as

$$f(n) = \begin{cases} 5 & \text{if } n = 0 \\ f(n-1) + g(n) & \text{if } n > 0 \end{cases}$$

Note that only additions are involved in the recursive definitions of $f(n)$ and $g(n)$.

Based on the definition of the last two recursive equations, we can derive an algorithm to compute $f(n)$. Two temporary registers are needed to keep track of the most recently calculated $f(n)$ and $g(n)$, and two additions are needed to update $f(n)$ and $g(n)$. Assume that $n$ is a 6-bit input and interpreted as an unsigned integer. Design this circuit using the RT methodology:

1. Derive the ASMD chart.
2. Derive the HDL code based on the ASMD chart.
3. Derive a testbench and use simulation to verify operation of the code.
4. Synthesize the circuit, program the FPGA, and verify its operation.
5. Let $h(n) = n^3 + 2n^2 + 2n + 1$. Use the method above to find the recursive representation of $h(n)$ (note that three levels of recursive equations are needed for a three-order polynomial). Repeat steps 1 to 4.