

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

$$r_{\text{dest}} \leftarrow f(r_{\text{src}1}, r_{\text{src}2}, \dots, r_{\text{src}n})$$

where r_{dest} is the destination register, $r_{\text{src}1}$, $r_{\text{src}2}$, and $r_{\text{src}n}$ 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.

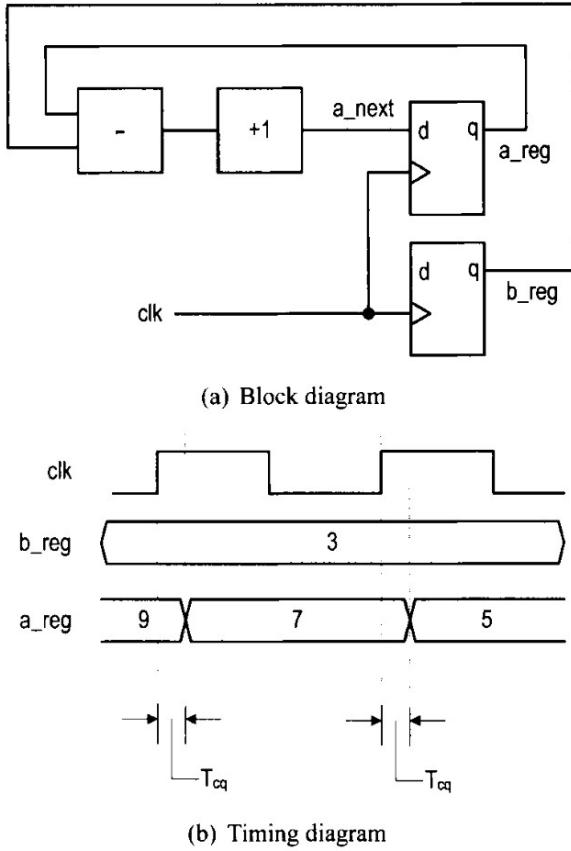


Figure 6.1 Block and timing diagrams of an RT operation.

- $r2 \leftarrow r2 \gg 3$. The $r2$ register is shifted right three positions and then written back to itself.
- $r2 \leftarrow r1$. The content of the $r1$ register is transferred to the $r2$ register.
- $i \leftarrow i + 1$. The content of the i register is incremented by 1 and the result is written back to itself.
- $d \leftarrow s1 + s2 + s3$. The summation of the $s1$, $s2$, and $s3$ registers is written to the d register.
- $y \leftarrow a^2$. 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 $a \leftarrow a - b + 1$ 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

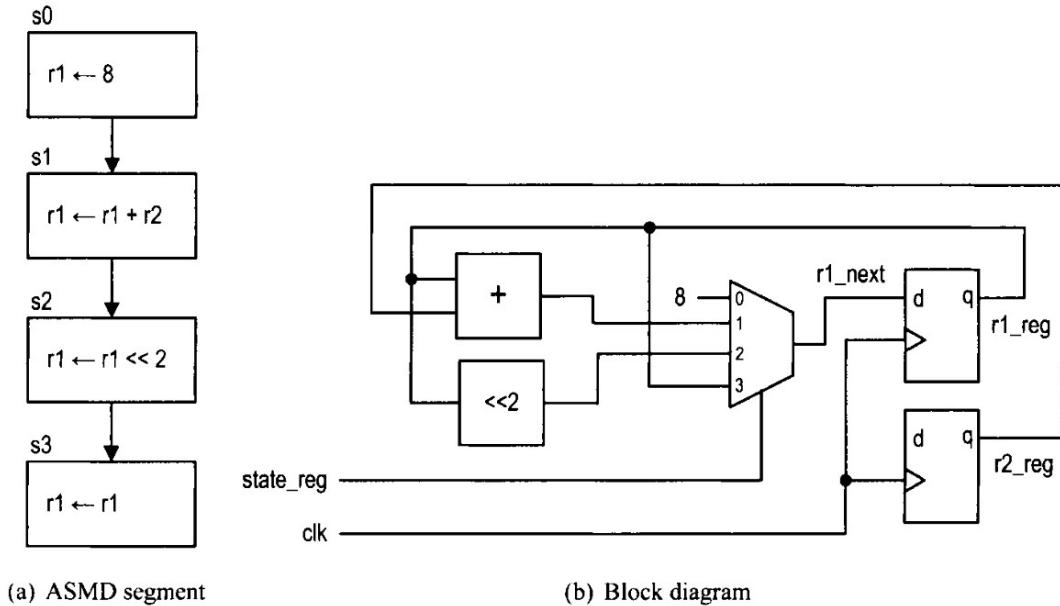


Figure 6.2 Realization of an ASMD segment.

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.

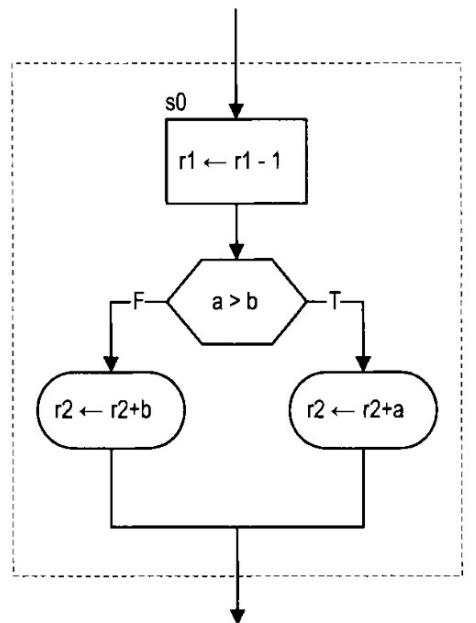
A segment of an ASMD chart is shown in Figure 6.2(a). It contains one destination register, r_1 , which is initialized with 8, added with content of the r_2 register, and then shifted left two positions. Note that the r_1 register must be specified in each state. When r_1 is not changed, the $r_1 \leftarrow r_1$ operation should be used to maintain its current content, as in the s_3 state. In future discussion, we assume that $r \leftarrow r$ 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 r_2 register shown in Figure 6.3(a). Depending on the $a > b$ condition, the FSMD performs either $r_2 \leftarrow r_2+a$ or $r_2 \leftarrow r_2+b$. Note that all operations are done in parallel inside an ASMD block. We need to realize the $a > b$, r_2+a , and r_2+b operations and use a multiplexer to route the desired value to r_2 . The block diagram is shown in Figure 6.3(b).

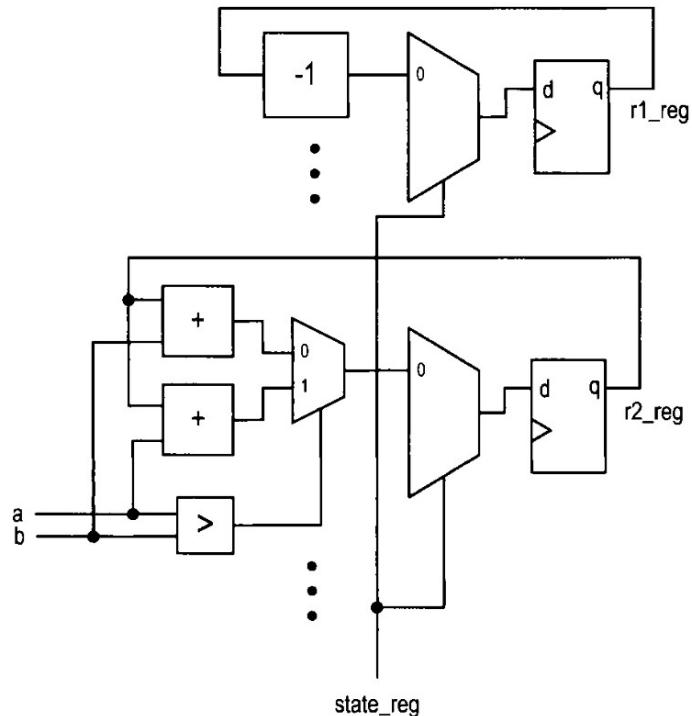
6.1.3 Decision box with a register

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 $r \leftarrow r-1$ operation actually means that:

- $r_{\text{next}} = r_{\text{reg}} - 1$;
- $r_{\text{reg}} \leftarrow r_{\text{next}}$ 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.

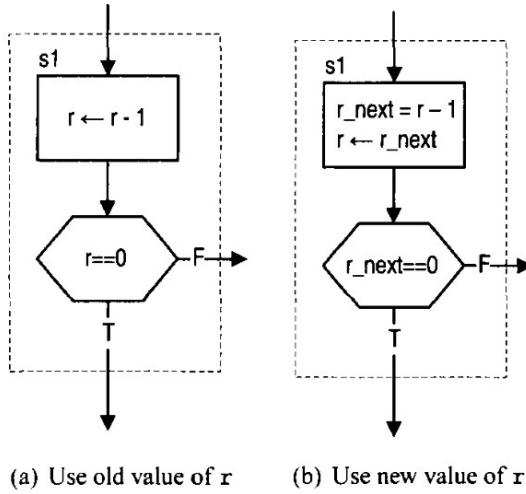


Figure 6.4 ASM block affected by a delayed store.

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 $r_{\text{next}}==0$), 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

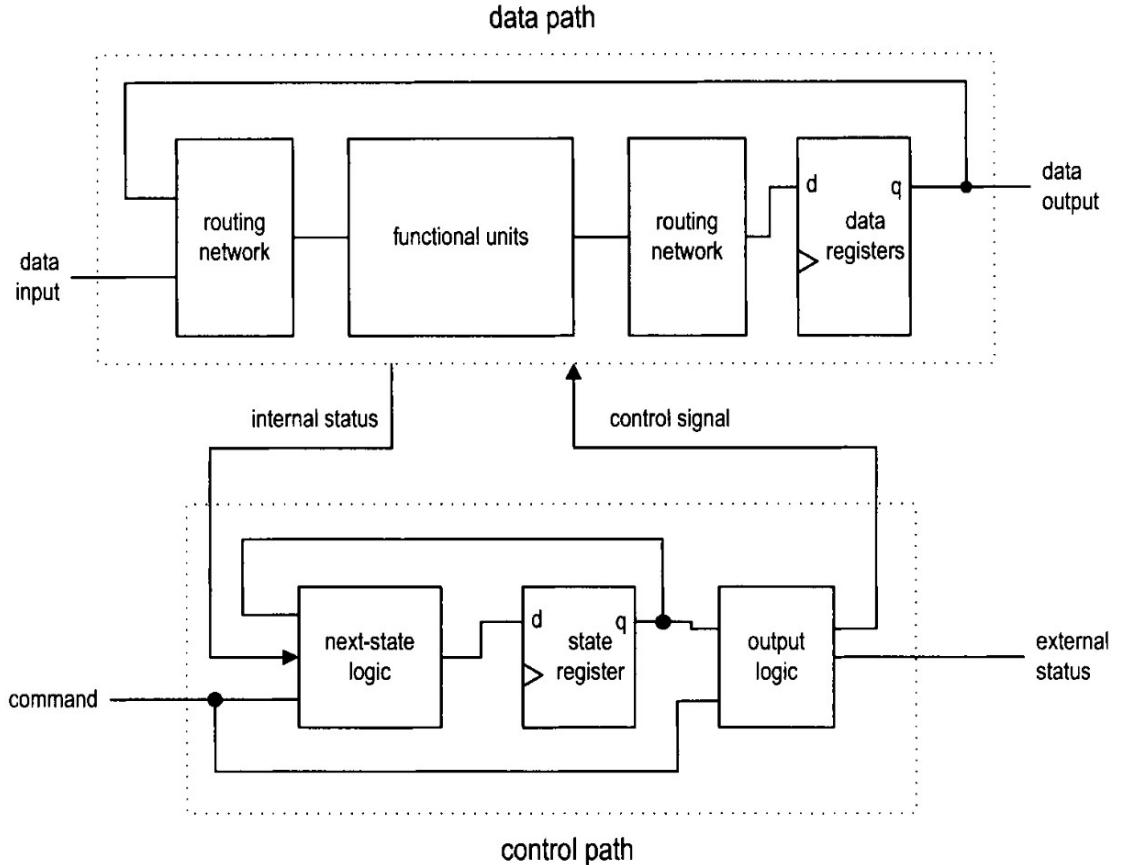


Figure 6.5 Block diagram of an FSMD.

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

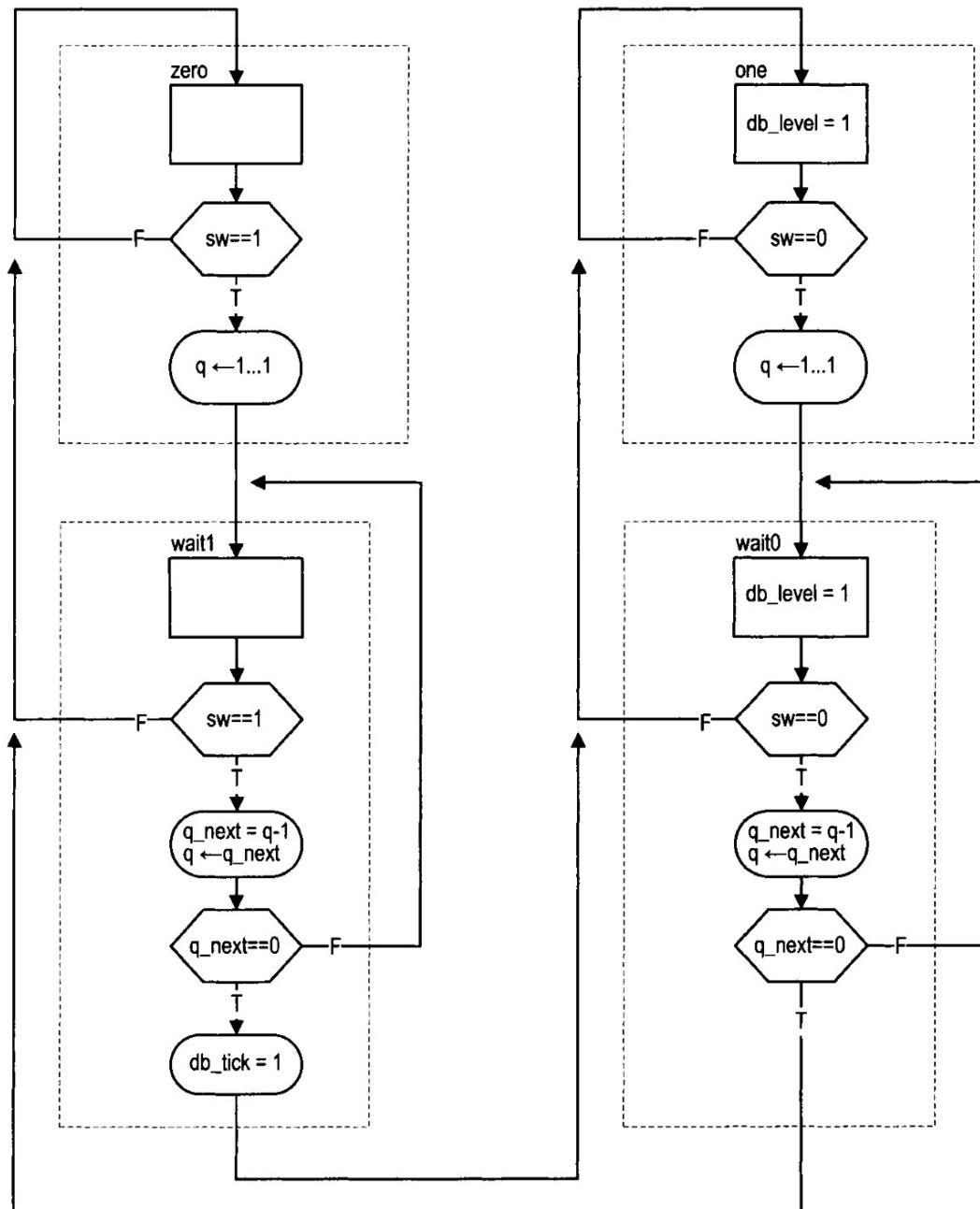


Figure 6.6 ASMD chart of a debouncing circuit.

(i.e., $2^{21} * 20 \text{ ns}$). 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
40 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);

45 // 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
                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;

```

```

20      reg [1:0] state_reg, state_next;

// body
// fsmd state & data registers
25      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
30
35      // next-state logic & data path functional units/routing
always @*
begin
    state_next = state_reg;      // default state: the same
40    q_next = q_reg;           // default q: unchanged
    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}};
                    end
45        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
60        end
        else // sw==0
            state_next = zero;
65    end
    one:
        begin
            db_level = 1'b1;
70        if (~sw)
            begin
                state_next = wait0;
            end
        end
    end

```

```

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

            end
        default: state_next = zero;
90    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

```

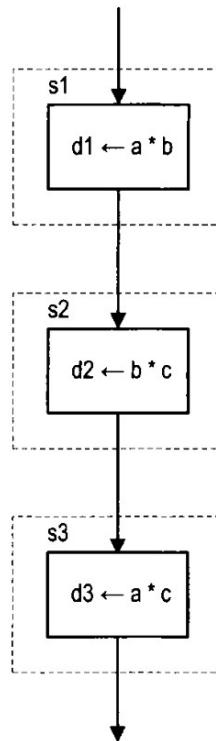


Figure 6.7 ASMD segment with sharing opportunity.

.....
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;
            ...
        end

```

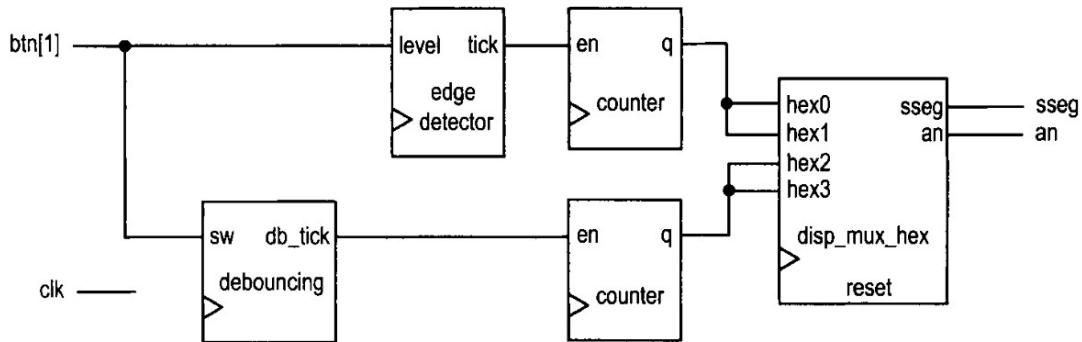


Figure 6.8 Debouncing testing circuit.

```

        end
        .
        .
        .
// 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]),
20    .dp_in(4'b1011), .an(an), .sseg(sseg));

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

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

    // two counters
    assign clr = btn[0];
    always @(posedge clk)
        begin
35        d_reg <= d_next;
        b_reg <= b_next;
        end
    //next-state logic for the counter
    40    assign b_next = (clr) ? 0 :
                                (btn_tick) ? b_reg + 1 :
                                              b_reg;
    assign d_next = (clr) ? 0 :
                                (db_tick) ? d_reg + 1 :
                                              d_reg;
45

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 t_1 and t_0 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.

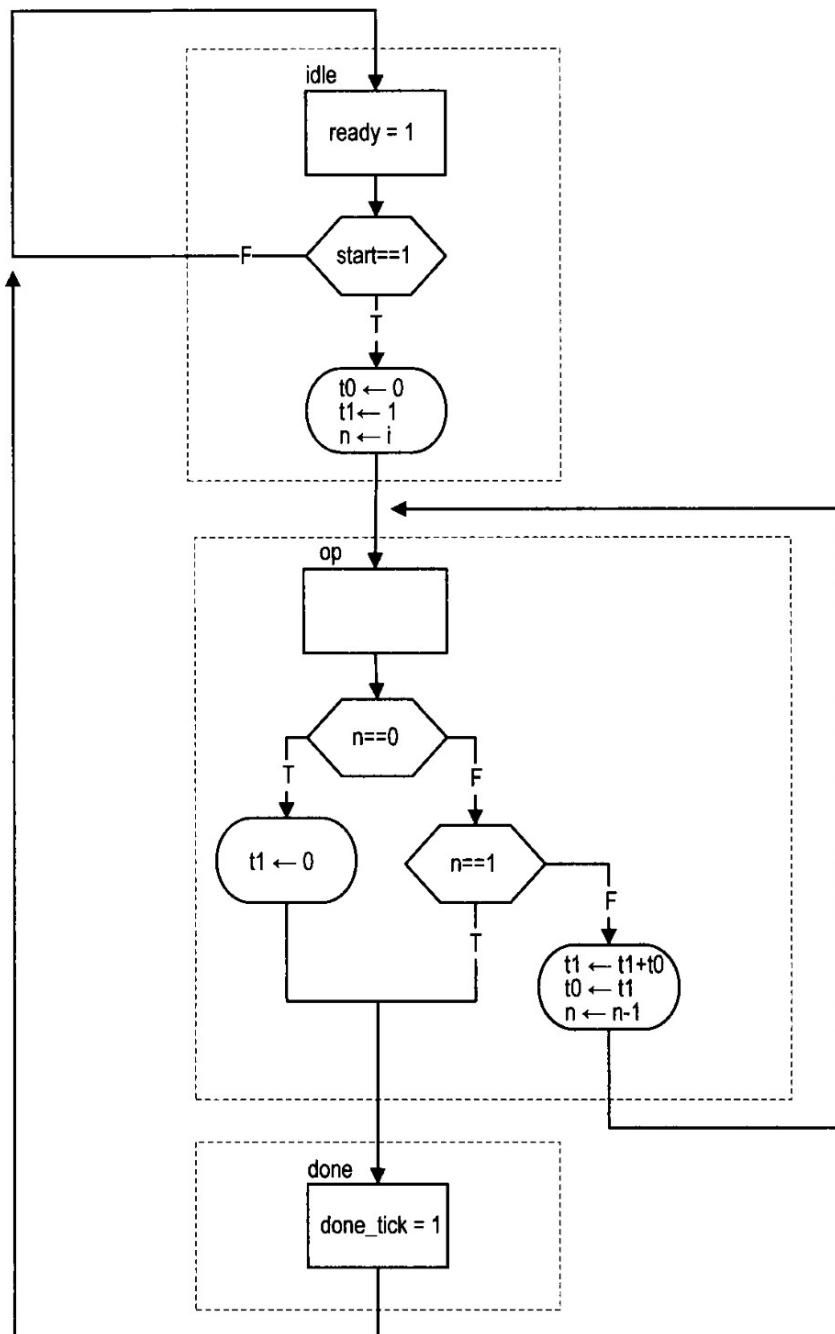


Figure 6.9 ASMD chart of a Fibonacci circuit.

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
);

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

15   // 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;

20   // 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;
30        end
        else
            begin
                state_reg <= state_next;
                t0_reg <= t0_next;
35                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;
45    t1_next = t1_reg;
    n_next = n_reg;
    case (state_reg)
        idle:
            begin
50            ready = 1'b1;
            if (start)
                begin
                    t0_next = 0;
                    t1_next = 20'd1;
55                    n_next = i;
                    state_next = op;
                end
            end
        op:
            if (n_reg==0)
                begin
                    t1_next = 0;
                    state_next = done;
                end
65            else if (n_reg==1)
                state_next = done;
            else
                begin
                    t1_next = t1_reg + t0_reg;
70                    t0_next = t1_reg;
                    n_next = n_reg - 1;
                end
            end
        done:
            begin
75            done_tick = 1'b1;
            state_next = idle;
        end
        default: state_next = idle;
    endcase
80    end
// output

```

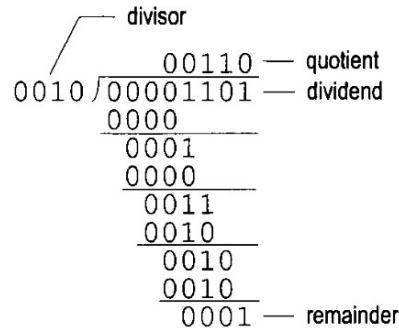


Figure 6.10 Long division of two 4-bit unsigned integers.

```
assign f = t1_reg;
endmodule
```

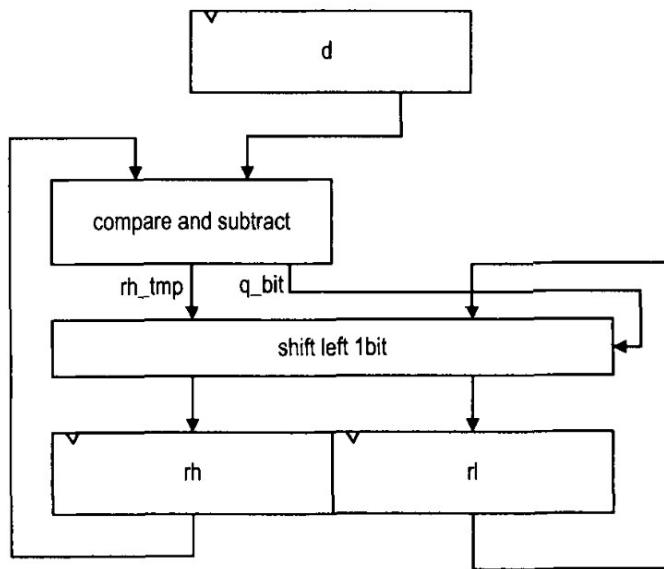
6.3.2 Division circuit

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 r1 registers. In each iteration, the rh and r1 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 r1 are shifted to the left, the rightmost bit of r1 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 r1.

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= $\log_2(W)+1$ 
)
(
    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

```

```

30      always @(posedge clk, posedge reset)
31          if (reset)
32              begin
33                  state_reg <= idle;
34                  rh_reg <= 0;
35                  rl_reg <= 0;
36                  d_reg <= 0;
37                  n_reg <= 0;
38              end
39      else
40          begin
41              state_reg <= state_next;
42              rh_reg <= rh_next;
43              rl_reg <= rl_next;
44              d_reg <= d_next;
45              n_reg <= n_next;
46          end

// FSMD next-state logic
always @*
50      begin
51          state_next = state_reg;
52          ready = 1'b0;
53          done_tick = 1'b0;
54          rh_next = rh_reg;
55          rl_next = rl_reg;
56          d_next = d_reg;
57          n_next = n_reg;
58          case (state_reg)
59              idle:
60                  begin
61                      ready = 1'b1;
62                      if (start)
63                          begin
64                              rh_next = 0;
65                              rl_next = dvnd; // dividend
66                              d_next = dvsr; // divisor
67                              n_next = CBIT; // index
68                              state_next = op;
69                          end
70                  end
71              op:
72                  begin
73                      // shift rh and rl left
74                      rl_next = {rl_reg[W-2:0], q_bit};
75                      rh_next = {rh_tmp[W-2:0], rl_reg[W-1]};
76                      // decrease index
77                      n_next = n_reg - 1;
78                      if (n_next==1)
79                          state_next = last;
80                  end
81              last: // last iteration
82                  begin

```

```

rl_next = {rl_reg[W-2:0], q_bit};
rh_next = rh_tmp;
state_next = done;
85      end
done:
begin
done_tick = 1'b1;
90      state_next = idle;
end
default: state_next = idle;
endcase
end
95      // compare and subtract circuit
always @*
if (rh_reg >= d_reg)
begin
100     rh_tmp = rh_reg - d_reg;
q_bit = 1'b1;
end
else
begin
105     rh_tmp = rh_reg;
q_bit = 1'b0;
end

// output
110     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 shift left 1 bit			1 (1 ₁₀)	11 1111
Bit 5	no adjustment shift left 1 bit			11 (3 ₁₀)	1 1111
Bit 4	no adjustment shift left 1 bit			111 (7 ₁₀)	1111
Bit 3	BCD digit 0 adjustment shift left 1 bit		1 (1 ₁₀)	1010 0101 (5 ₁₀)	111
Bit 2	BCD digit 0 adjustment shift left 1 bit		1 11 (3 ₁₀)	1000 0001 (1 ₁₀)	11
Bit 1	no adjustment shift left 1 bit		110 (6 ₁₀)	0011 (3 ₁₀)	1
Bit 0	BCD digit 1 adjustment shift left 1 bit	1 (1 ₁₀)	1001 0010 (2 ₁₀)	0011 0111 (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
);

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

15   // signal declaration
reg [1:0] state_reg, state_next;
reg [12:0] p2s_reg, p2s_next;
reg [3:0] n_reg, n_next;
20   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;

25   // 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;
35           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;
45           bcd1_reg <= bcd1_next;
            bcd0_reg <= bcd0_next;
        end

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

```

```

state_next = state_reg;
ready = 1'b0;
55
done_tick = 1'b0;
p2s_next = p2s_reg;
bcd0_next = bcd0_reg;
bcd1_next = bcd1_reg;
bcd2_next = bcd2_reg;
bcd3_next = bcd3_reg;
60
n_next = n_reg;
case (state_reg)
idle:
begin
65
    ready = 1'b1;
    if (start)
        begin
            state_next = op;
            bcd3_next = 0;
70
            bcd2_next = 0;
            bcd1_next = 0;
            bcd0_next = 0;
            n_next = 4'b1101; // index
            p2s_next = bin; // shift register
            state_next = op;
75
        end
    end
op:
begin
80
    // 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,
    85
    // 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]};
    90
    n_next = n_reg - 1;
    if (n_next==0)
        state_next = done;
    end
done:
begin
95
    done_tick = 1'b1;
    state_next = idle;
    end
100
default: state_next = idle;
endcase
end

// data path function units
105
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;

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

115   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.

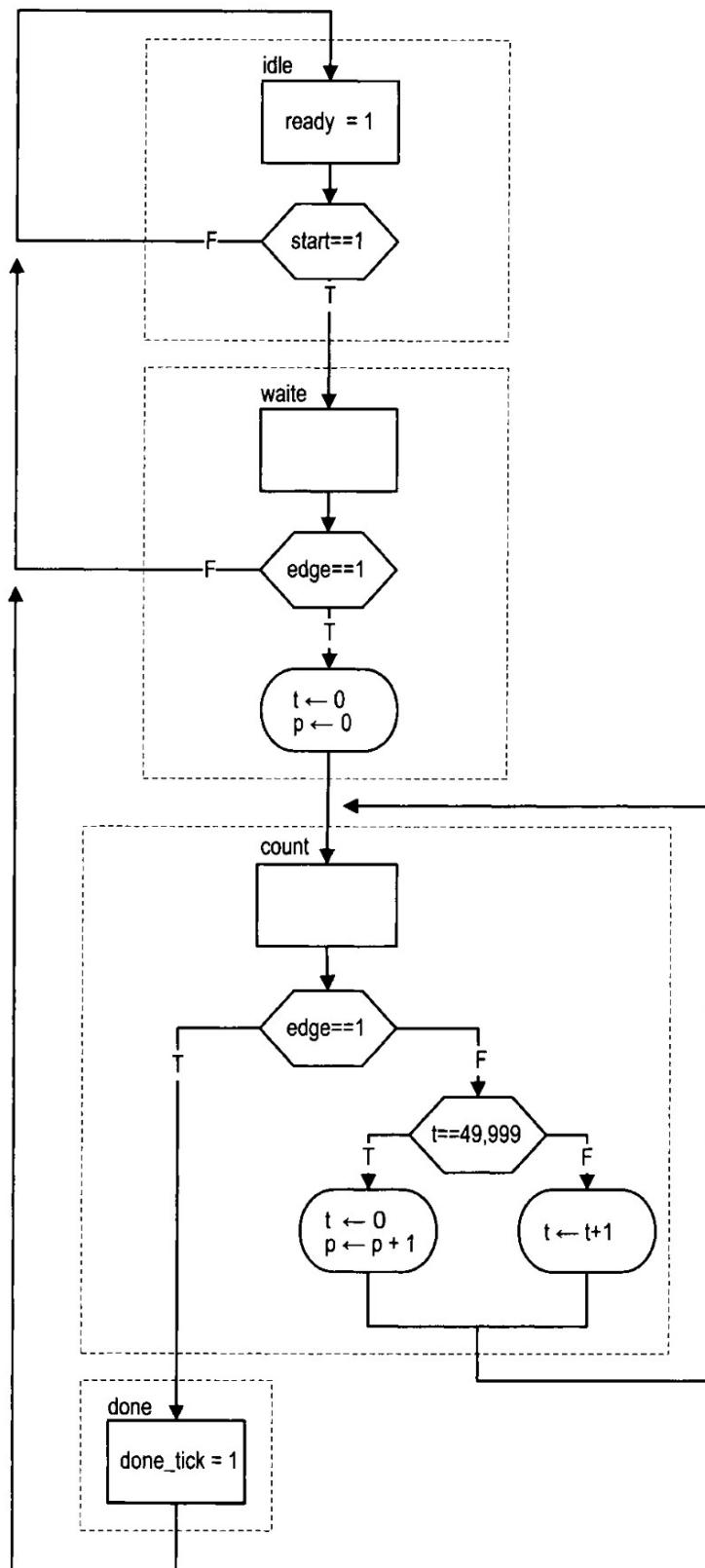
Listing 6.7 Period counter

```

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

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

```

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

```

        done   = 2'b11;
15
// constant declaration
localparam CLK_MS_COUNT= 50000; // 1 ms tick

// signal declaration
20 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;

25
// body
// FSMD state & data registers
always @ (posedge clk, posedge reset)
    if (reset)
30    begin
        state_reg <= idle;
        t_reg <= 0;
        p_reg <= 0;
        delay_reg <= 0;
    end
35 else
    begin
        state_reg <= state_next;
        t_reg <= t_next;
40        p_reg <= p_next;
        delay_reg <= si;
    end

45
// rising-edge tick
assign edg = ~delay_reg & si;

// FSMD next-state logic
50 always @*
begin
    state_next = state_reg;
    ready = 1'b0;
    done_tick = 1'b0;
    p_next = p_reg;
    t_next = t_reg;
55    case (state_reg)
        idle:
            begin
                ready = 1'b1;
                if (start)
                    state_next = waite;
60            end
        waite: // wait for the first edge
            if (edg)
                begin
65                    state_next = count;
                    t_next = 0;
                end
            end
    end
end

```

```

        p_next = 0;
    end
    count:
    70    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;
  80  done:
    begin
      done_tick = 1'b1;
      state_next = idle;
    end
  85  default: state_next = idle;
endcase
end

// output
90  assign prd = p_reg;

endmodule

```

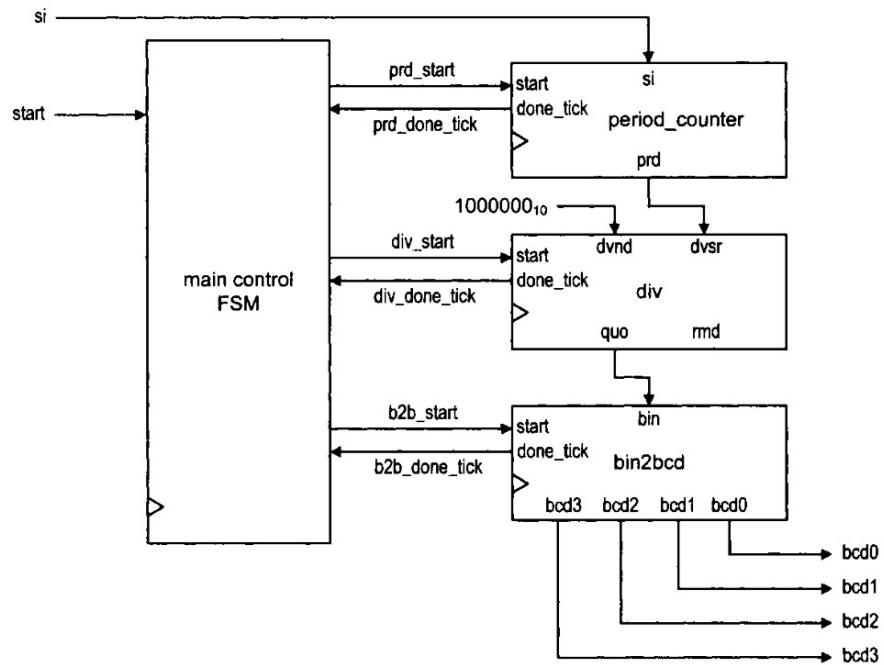
6.3.5 Accurate low-frequency counter

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.

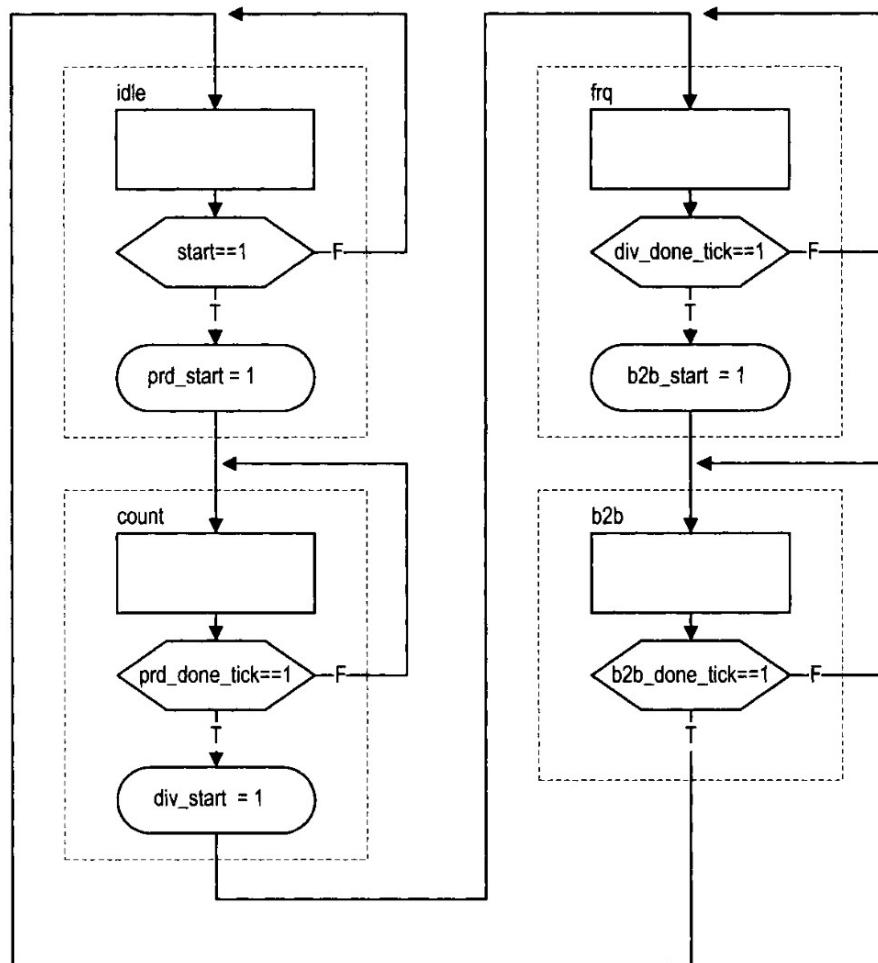
This design example demonstrates how to use the previously designed parts to construct a large system. For simplicity, we assume that the frequency of the input is between 1 and 10 Hz (i.e., the period is between 100 and 1000 ms). The operation of this circuit includes three tasks:

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


#(.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;
55    prd_start = 1'b0;
    div_start = 1'b0;
    b2b_start = 1'b0;
    case (state_reg)
        idle:
60        if (start)
            begin
                prd_start = 1'b1;
                state_next = count;
            end
        count:
65        if (prd_done_tick)
            begin
                div_start = 1'b1;
                state_next = frq;
            end
        frq:
70        if (div_done_tick)
            begin
                b2b_start = 1'b1;
                state_next = b2b;
            end
        b2b:
75        if (b2b_done_tick)
            state_next = idle;
80    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.