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.

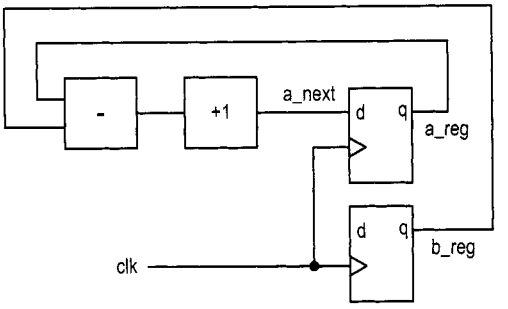
6.1.1 Single RT operation

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

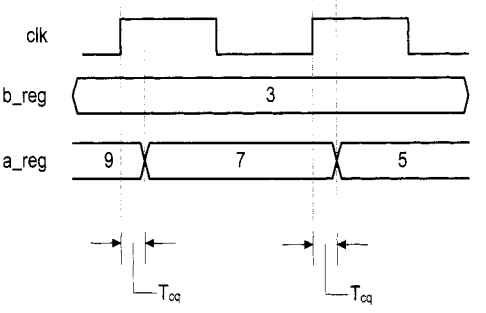
$$r_{dest} \leftarrow f(r_{src1}, r_{src2}, \ldots, r_{srcn})$$

where r_{dest} is the destination register, r_{src1}, r_{src2}, and 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

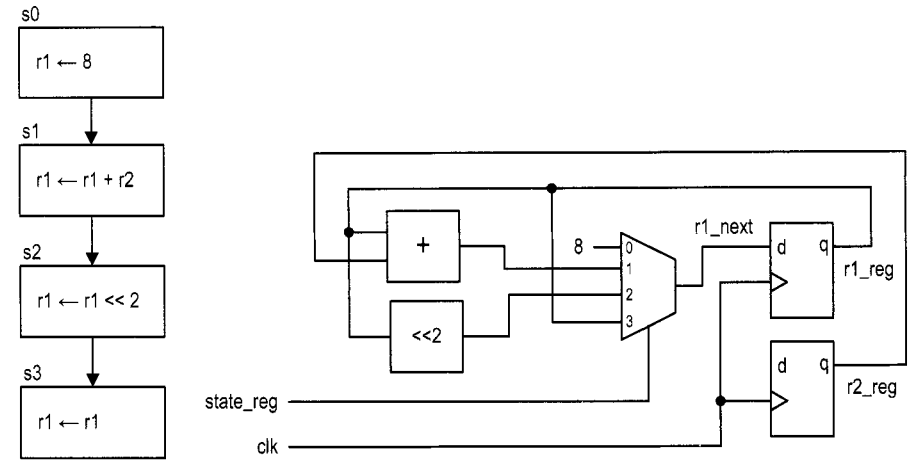
Figure 6.1 Block and timing diagrams of an RT operation.

- `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 in 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.

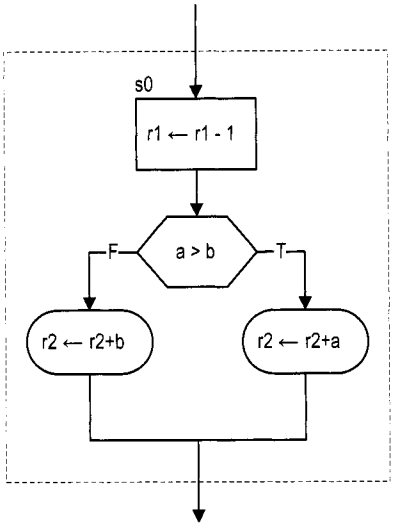
A segment of an ASMD chart is shown in Figure 6.2(a). It contains one destination register, `r1`, which is initialized with 8, added with content of the `r2` register, and then shifted left two positions. Note that the `r1` register must be specified in each state. When `r1` is not changed, the `r1` ← `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).

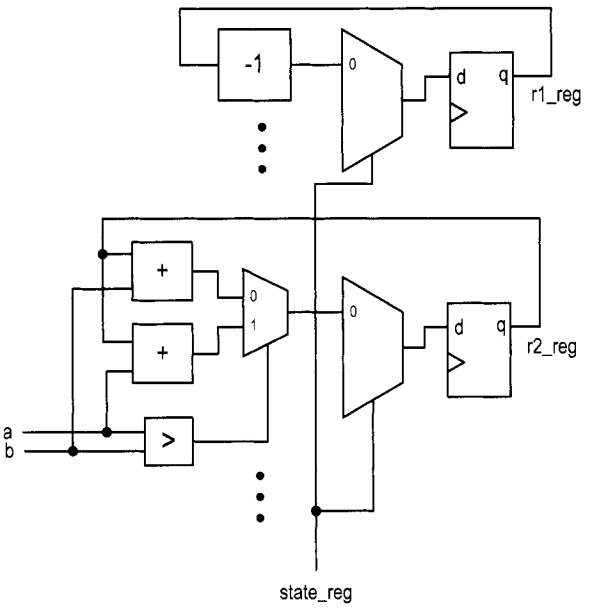
6.1.3 Decision box with a register

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).

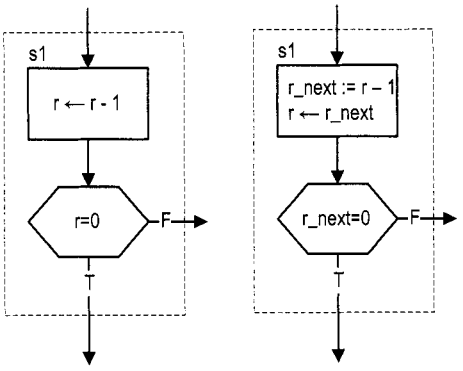


(a) ASM block



(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). Note that we use the `:=` notation, as in `r_next:=r-1`, to indicate the immediate assignment of `r_next`.

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

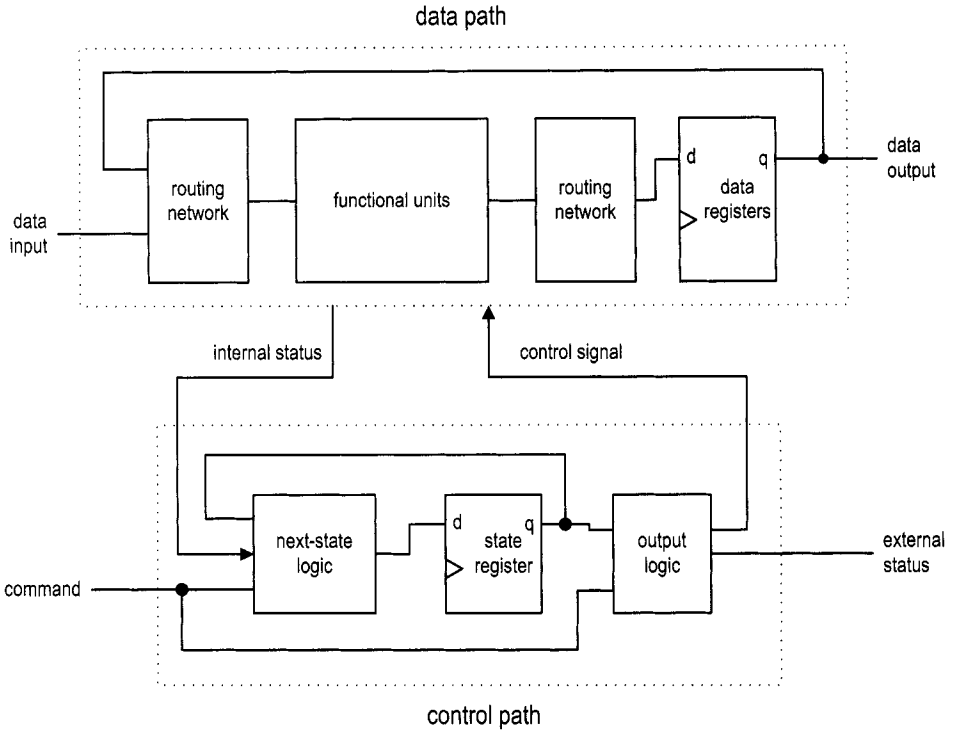


Figure 6.5 Block diagram of an FSMD.

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 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 the 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 $\cdots$ 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 $\cdots$ 0" and then moves to the `one` state.

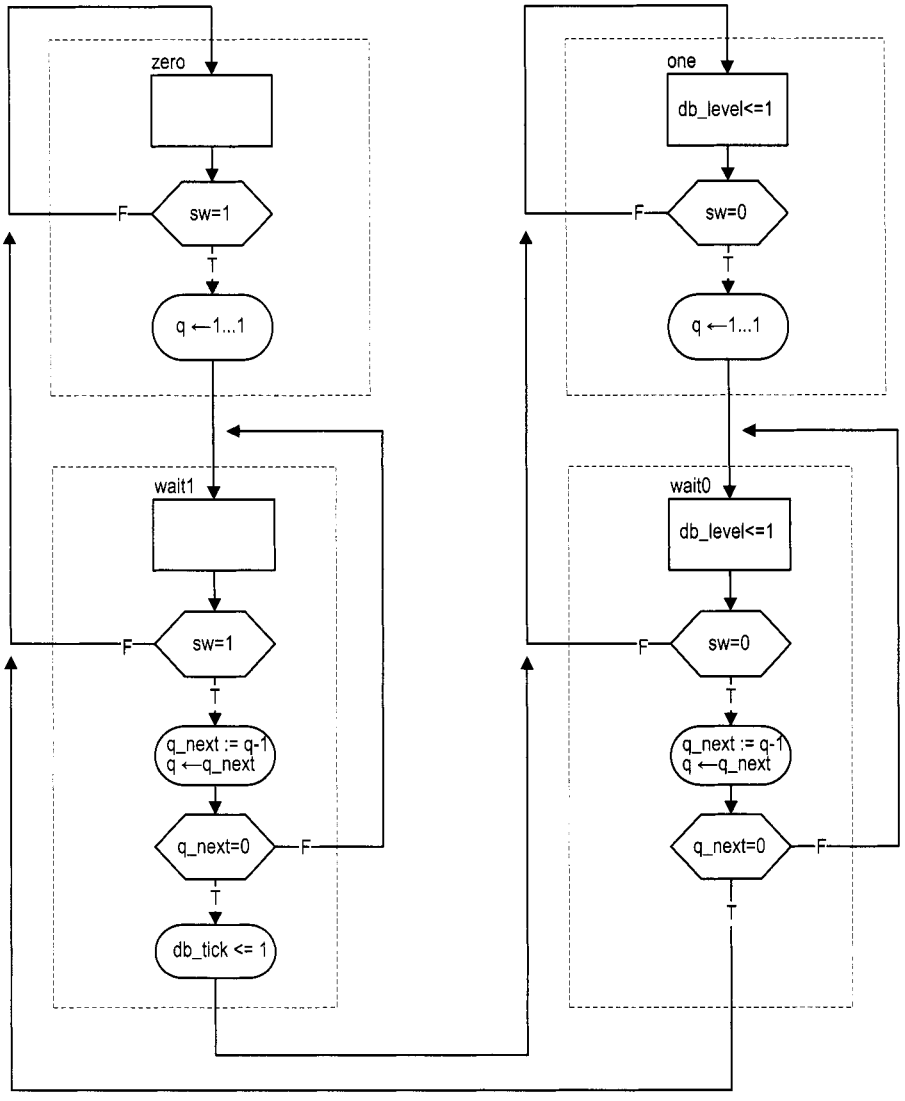


Figure 6.6 ASMD chart of a debouncing circuit.

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 (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

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity debounce is
   port(
      clk, reset: in std_logic;
      sw: in std_logic;
      db_level, db_tick: out std_logic
   );
end debounce ;

architecture exp_fsmd_arch of debounce is
   constant N: integer:=21;  -- filter of 2^N * 20ns = 40ms
   type state_type is (zero, wait0, one, wait1);
   signal state_reg, state_next: state_type;
   signal q_reg, q_next: unsigned(N-1 downto 0);
   signal q_load, q_dec, q_zero: std_logic;
begin
   -- FSMD state & data registers
   process(clk,reset)
   begin
      if reset='1' then
         state_reg <= zero;
```

```
      q_reg <= (others=>'0');
   elsif (clk'event and clk='1') then
      state_reg <= state_next;
      q_reg <= q_next;
   end if;
end process;

-- FSMD data path (counter) next-state logic
q_next <= (others=>'1') when q_load='1' else
          q_reg - 1 when q_dec='1' else
          q_reg;
q_zero <= '1' when q_next=0 else '0';

-- FSMD control path next-state logic
process(state_reg,sw,q_zero)
begin
   q_load <= '0';
   q_dec <= '0';
   db_tick <= '0';
   state_next <= state_reg;
   case state_reg is
      when zero =>
         db_level <= '0';
         if (sw='1') then
            state_next <= wait1;
            q_load <= '1';
         end if;
      when wait1=>
         db_level <= '0';
         if (sw='1') then
            q_dec <= '1';
            if (q_zero='1') then
               state_next <= one;
               db_tick <= '1';
            end if;
         else -- sw='0'
            state_next <= zero;
         end if;
      when one =>
         db_level <= '1';
         if (sw='0') then
            state_next <= wait0;
            q_load <= '1';
         end if;
      when wait0=>
         db_level <= '1';
         if (sw='0') then
            q_dec <= '1';
            if (q_zero='1') then
               state_next <= zero;
            end if;
         else -- sw='1'
            state_next <= one;
```

```
            end if;
         end case;
      end process;
80 end exp_fsmd_arch;
```

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

```
architecture imp_fsmd_arch of debounce is
   constant N: integer:=21;  -- filter of 2^N * 20ns = 40ms
   type state_type is (zero, wait0, one, wait1);
   signal state_reg, state_next: state_type;
5  signal q_reg, q_next: unsigned(N-1 downto 0);
begin
   -- FSMD state & data registers
   process(clk,reset)
   begin
10    if reset='1' then
         state_reg <= zero;
         q_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
15       q_reg <= q_next;
      end if;
   end process;
   -- next-state logic & data path functional units/routing
   process(state_reg,q_reg,sw,q_next)
20 begin
      state_next <= state_reg;
      q_next <= q_reg;
      db_tick <= '0';
      case state_reg is
25       when zero =>
            db_level <= '0';
            if (sw='1') then
               state_next <= wait1;
               q_next <= (others=>'1');
30          end if;
         when wait1=>
            db_level <= '0';
            if (sw='1') then
               q_next <= q_reg - 1;
35             if (q_next=0) then
                  state_next <= one;
                  db_tick <= '1';
               end if;
            else -- sw='0'
40             state_next <= zero;
```

```
            end if;
         when one =>
            db_level <= '1';
            if (sw='0') then
               state_next <= wait0;
               q_next <= (others=>'1');
            end if;
         when wait0=>
            db_level <= '1';
            if (sw='0') then
               q_next <= q_reg - 1;
               if (q_next=0) then
                  state_next <= zero;
               end if;
            else -- sw='1'
               state_next <= one;
            end if;
      end case;
   end process;
end imp_fsmd_arch;
```

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 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
   when s1
      d1_next <= a * b;
      . . .
   when s2
      d2_next <= b * c;
      . . .
   when s3
      d3_next <= a * c;
      . . .
end case;
```

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
   when s1
```

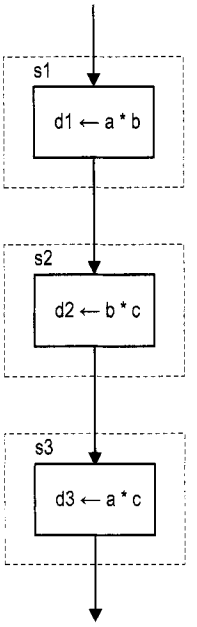


Figure 6.7 ASMD segment with sharing opportunity.

```
        in1 <= a;
        in2 <= b;
        d1_next <= m_out;
        . . .
    when s2
        in1 <= b;
        in2 <= c;
        d2_next <= m_out;
        . . .
    when s3
        in1 <= a;
        in2 <= c;
        d3_next <= m_out;
        . . .
end case;
-- explicit description of a single multiplier
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

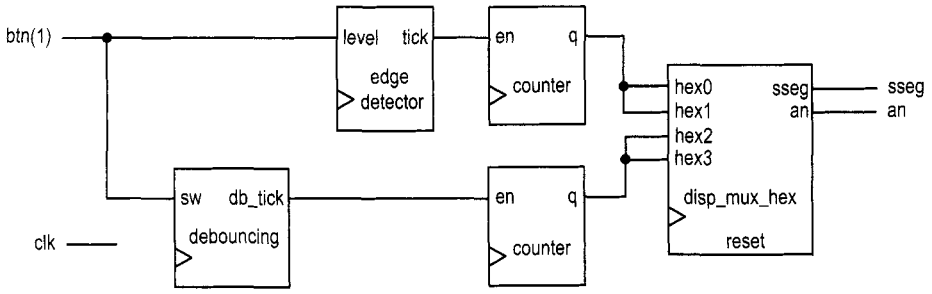


Figure 6.8 Debouncing testing circuit.

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

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity debounce_test is
   port(
      clk: in std_logic;
      btn: in std_logic_vector(3 downto 0);
      an: out std_logic_vector(3 downto 0);
      sseg: out std_logic_vector(7 downto 0)
   );
end debounce_test;

architecture arch of debounce_test is
   signal q1_reg, q1_next: unsigned(7 downto 0);
   signal q0_reg, q0_next: unsigned(7 downto 0);
   signal b_count, d_count: std_logic_vector(7 downto 0);
   signal btn_reg: std_logic;
   signal db_tick, btn_tick, clr: std_logic;
begin
   -- instantiate debouncing circuit
   db_unit: entity work.debounce(fsmd_arch)
      port map(
         clk=>clk, reset=>'0', sw=>btn(1),
         db_level=> open, db_tick=>db_tick
      );
   -- instantiate hex display time-multiplxing circuit
   disp_unit: entity work.disp_hex_mux
      port map(
         clk=>clk, reset=>'0',
         hex3=>b_count(7 downto 4), hex2=>b_count(3 downto 0),
         hex1=>d_count(7 downto 4), hex0=>d_count(3 downto 0),
         dp_in=>"1011", an=>an, sseg=>sseg
      );

   --=================================================
   -- edge detection circuit for un-debounced input
```

```
   --=================================================
   process(clk)
   begin
40    if (clk'event and clk='1') then
         btn_reg <= btn(1);
      end if;
   end process;
   btn_tick <= (not btn_reg) and btn(1);
45
   --=================================================
   -- two counters
   --=================================================
   clr <= btn(0);
50 process(clk)
   begin
      if (clk'event and clk='1') then
         q1_reg <= q1_next;
         q0_reg <= q0_next;
55    end if;
   end process;
   -- next-state logic for the counter
   q1_next <= (others=>'0') when clr='1' else
              q1_reg + 1 when btn_tick='1' else
60            q1_reg;
   q0_next <= (others=>'0') when clr='1' else
              q0_reg + 1 when db_tick='1' else
              q0_reg;
   -- counter output
65 b_count <= std_logic_vector(q1_reg);
   d_count <= std_logic_vector(q0_reg);
end arch;
```

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

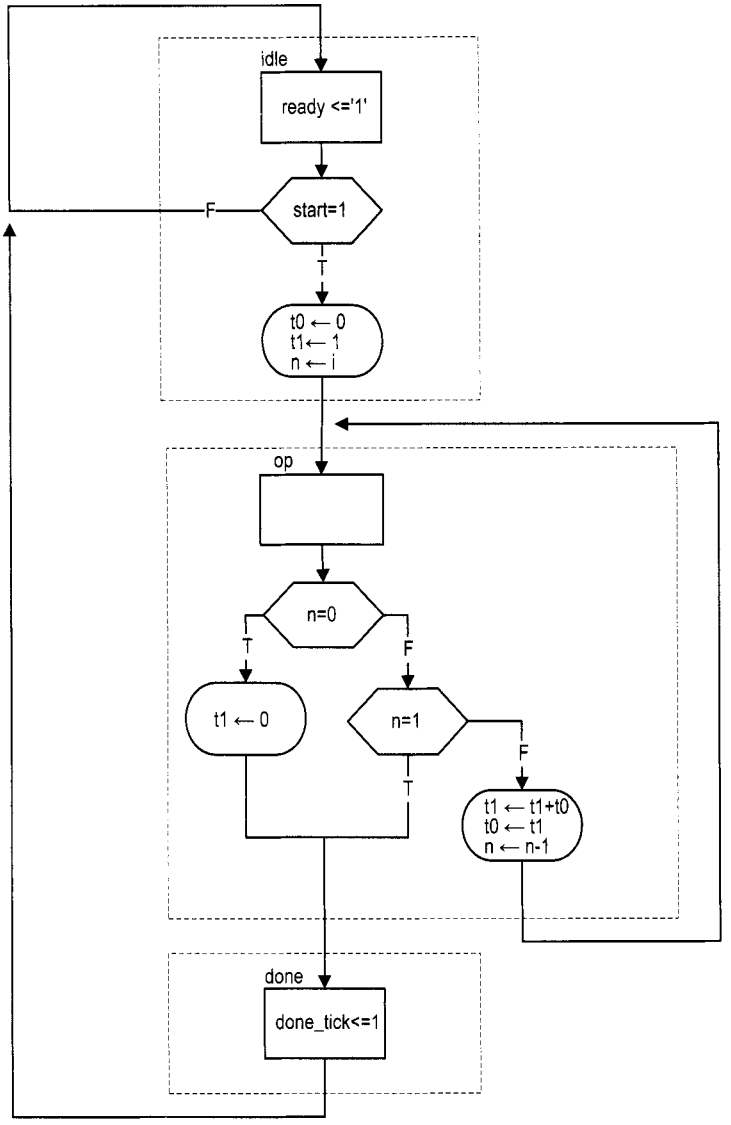


Figure 6.9 ASMD chart of a Fibonacci circuit.

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.

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

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity fib is
    port(
        clk, reset: in std_logic;
        start: in std_logic;
        i: in std_logic_vector(4 downto 0);
        ready, done_tick: out std_logic;
        f: out std_logic_vector(19 downto 0)
    );
end fib;

architecture arch of fib is
    type state_type is (idle,op,done);
    signal state_reg, state_next: state_type;
    signal t0_reg, t0_next: unsigned(19 downto 0);
    signal t1_reg, t1_next: unsigned(19 downto 0);
    signal n_reg, n_next: unsigned(4 downto 0);
begin
    -- fsmd state and data registers
    process(clk,reset)
    begin
       if reset='1' then
          state_reg <= idle;
          t0_reg <= (others=>'0');
```

```
         t1_reg <= (others=>'0');
         n_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
         t0_reg <= t0_next;
         t1_reg <= t1_next;
         n_reg <= n_next;
      end if;
   end process;
   -- fsmd next-state logic
   process(state_reg,n_reg,t0_reg,t1_reg,start,i,n_next)
   begin
      ready <='0';
      done_tick <= '0';
      state_next <= state_reg;
      t0_next <= t0_reg;
      t1_next <= t1_reg;
      n_next <= n_reg;
      case state_reg is
         when idle =>
            ready <= '1';
            if start='1' then
               t0_next <= (others=>'0');
               t1_next <= (0=>'1', others=>'0');
               n_next <= unsigned(i);
               state_next <= op;
            end if;
         when op =>
            if n_reg=0 then
               t1_next <= (others=>'0');
               state_next <= done;
            elsif n_reg=1 then
               state_next <= done;
            else
               t1_next <= t1_reg + t0_reg;
               t0_next <= t1_reg;
               n_next <= n_reg - 1;
            end if;
         when done =>
            done_tick <= '1';
            state_next <= idle;
      end case;
   end process;
   -- output
   f <= std_logic_vector(t1_reg);
end arch;
```

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

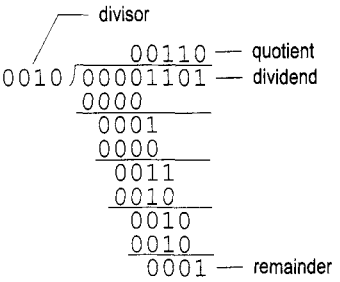


Figure 6.10 Long division of two 4-bit unsigned integers.

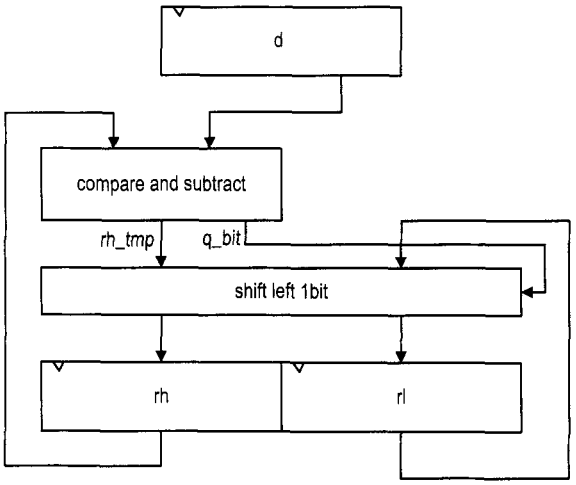


Figure 6.11 Sketch of division circuit's data path.

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 previous 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.

Listing 6.5 Division circuit

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity div is
   generic(
      W: integer:=8;
      CBIT: integer:=4    -- CBIT=log2(W)+1
   );
   port(
      clk, reset: in std_logic;
      start: in std_logic;
      dvsr, dvnd: in std_logic_vector(W-1 downto 0);
      ready, done_tick: out std_logic;
      quo, rmd: out std_logic_vector(W-1 downto 0)
   );
end div;

architecture arch of div is
   type state_type is (idle,op,last,done);
   signal state_reg, state_next: state_type;
   signal rh_reg, rh_next: unsigned(W-1 downto 0);
   signal rl_reg, rl_next: std_logic_vector(W-1 downto 0);
   signal rh_tmp: unsigned(W-1 downto 0);
   signal d_reg, d_next: unsigned(W-1 downto 0);
   signal n_reg, n_next: unsigned(CBIT-1 downto 0);
   signal q_bit: std_logic;
begin
   -- fsmd state and data registers
   process(clk,reset)
   begin
      if reset='1' then
         state_reg <= idle;
         rh_reg <= (others=>'0');
         rl_reg <= (others=>'0');
         d_reg <= (others=>'0');
         n_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
         rh_reg <= rh_next;
```

```
40          rl_reg <= rl_next;
            d_reg <= d_next;
            n_reg <= n_next;
         end if;
      end process;
45
      -- fsmd next-state logic and data path logic
      process(state_reg,n_reg,rh_reg,rl_reg,d_reg,
              start,dvsr,dvnd,q_bit,rh_tmp,n_next)
      begin
50       ready <='0';
         done_tick <= '0';
         state_next <= state_reg;
         rh_next <= rh_reg;
         rl_next <= rl_reg;
55       d_next <= d_reg;
         n_next <= n_reg;
         case state_reg is
            when idle =>
               ready <= '1';
60             if start='1' then
                  rh_next <= (others=>'0');
                  rl_next <= dvnd;                    -- dividend
                  d_next <= unsigned(dvsr);           -- divisor
                  n_next <= to_unsigned(W+1, CBIT); -- index
65                state_next <= op;
               end if;
            when op =>
               -- shift rh and rl left
               rl_next <= rl_reg(W-2 downto 0) & q_bit;
70             rh_next <= rh_tmp(W-2 downto 0) & rl_reg(W-1);
               --decrease index
               n_next <= n_reg - 1;
               if (n_next=1) then
                  state_next <= last;
75             end if;
            when last =>  -- last iteration
               rl_next <= rl_reg(W-2 downto 0) & q_bit;
               rh_next <= rh_tmp;
               state_next <= done;
80          when done =>
               state_next <= idle;
               done_tick <= '1';
         end case;
      end process;
85
      -- compare and subtract
      process(rh_reg, d_reg)
      begin
         if rh_reg >= d_reg then
90          rh_tmp <= rh_reg - d_reg;
            q_bit <= '1';
         else
```

```
            rh_tmp <= rh_reg;
            q_bit <= '0';
        end if;
    end process;

    -- output
    quo <= rl_reg;
    rmd <= std_logic_vector(rh_reg);
end arch;
```

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 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 into 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

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

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})	

```
entity bin2bcd is
   port(
      clk: in std_logic;
      reset: in std_logic;
      start: in std_logic;
      bin: in std_logic_vector(12 downto 0);
      ready, done_tick: out std_logic;
      bcd3,bcd2,bcd1,bcd0: out std_logic_vector(3 downto 0)
   );
end bin2bcd ;

architecture arch of bin2bcd is
   type state_type is (idle, op, done);
   signal state_reg, state_next: state_type;
   signal p2s_reg, p2s_next: std_logic_vector(12 downto 0);
   signal n_reg, n_next: unsigned(3 downto 0);
   signal bcd3_reg, bcd2_reg, bcd1_reg, bcd0_reg:
          unsigned(3 downto 0);
   signal bcd3_next, bcd2_next, bcd1_next, bcd0_next:
          unsigned(3 downto 0);
   signal bcd3_tmp, bcd2_tmp, bcd1_tmp, bcd0_tmp:
```

```
          unsigned(3 downto 0);
begin
   -- state and data registers
   process (clk,reset)
   begin
      if reset='1' then
         state_reg <= idle;
         p2s_reg <= (others=>'0');
         n_reg <= (others=>'0');
         bcd3_reg <= (others=>'0');
         bcd2_reg <= (others=>'0');
         bcd1_reg <= (others=>'0');
         bcd0_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         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 if;
   end process;

   -- fsmd next-state logic / data path operations
   process(state_reg,start,p2s_reg,n_reg,n_next,bin,
           bcd0_reg,bcd1_reg,bcd2_reg,bcd3_reg,
           bcd0_tmp,bcd1_tmp,bcd2_tmp,bcd3_tmp)
   begin
      state_next <= state_reg;
      ready <= '0';
      done_tick <= '0';
      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 is
         when idle =>
            ready <= '1';
            if start='1' then
               state_next <= op;
               bcd3_next <= (others=>'0');
               bcd2_next <= (others=>'0');
               bcd1_next <= (others=>'0');
               bcd0_next <= (others=>'0');
               n_next <="1101";  -- index
               p2s_next <= bin;  -- input shift register
               state_next <= op;
            end if;
         when op =>
            -- shift in binary bit
```

```
            p2s_next <= p2s_reg(11 downto 0) & '0';
            -- shift 4 BCD digits
            bcd0_next <= bcd0_tmp(2 downto 0) & p2s_reg(12);
            bcd1_next <= bcd1_tmp(2 downto 0) & bcd0_tmp(3);
            bcd2_next <= bcd2_tmp(2 downto 0) & bcd1_tmp(3);
            bcd3_next <= bcd3_tmp(2 downto 0) & bcd2_tmp(3);
            n_next <= n_reg - 1;
            if (n_next=0) then
                state_next <= done;
            end if;
         when done =>
            state_next <= idle;
            done_tick <= '1';
      end case;
   end process;

   -- data path function units
   -- four BCD adjustment circuits
   bcd0_tmp <= bcd0_reg + 3 when bcd0_reg > 4 else
               bcd0_reg;
   bcd1_tmp <= bcd1_reg + 3 when bcd1_reg > 4 else
               bcd1_reg;
   bcd2_tmp <= bcd2_reg + 3 when bcd2_reg > 4 else
               bcd2_reg;
   bcd3_tmp <= bcd3_reg + 3 when bcd3_reg > 4 else
               bcd3_reg;

   -- output
   bcd0 <= std_logic_vector(bcd0_reg);
   bcd1 <= std_logic_vector(bcd1_reg);
   bcd2 <= std_logic_vector(bcd2_reg);
   bcd3 <= std_logic_vector(bcd3_reg);
end arch;
```

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

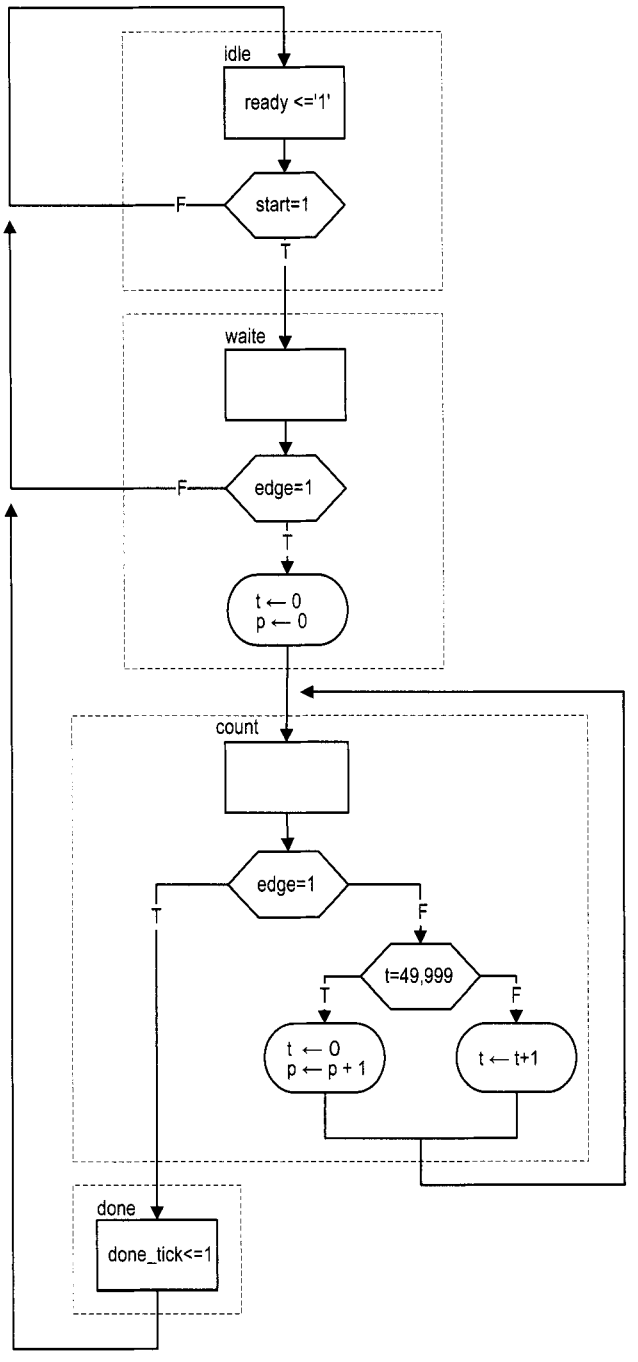


Figure 6.12 ASMD chart of a period counter.

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

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity period_counter is
   port(
      clk, reset: in std_logic;
      start, si: in std_logic;
      ready, done_tick: out std_logic;
      prd: out std_logic_vector(9 downto 0)
   );
end period_counter;

architecture arch of period_counter is
   constant CLK_MS_COUNT: integer := 50000; -- 1 ms tick
   type state_type is (idle, waite, count, done);
   signal state_reg, state_next: state_type;
   signal t_reg, t_next: unsigned(15 downto 0);
   signal p_reg, p_next: unsigned(9 downto 0);
   signal delay_reg: std_logic;
   signal edge: std_logic;
begin
   -- state and data register
   process(clk,reset)
   begin
      if reset='1' then
         state_reg <= idle;
         t_reg <= (others=>'0');
         p_reg <= (others=>'0');
         delay_reg <= '0';
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
         t_reg <= t_next;
         p_reg <= p_next;
         delay_reg <= si;
      end if;
   end process;

   -- edge detection circuit
   edge <= (not delay_reg) and si;

   -- fsmd next-state logic / data path operations
   process(start,edge,state_reg,t_reg,t_next,p_reg)
   begin
      ready <= '0';
      done_tick <= '0';
```

```
      state_next <= state_reg;
      p_next <= p_reg;
      t_next <= t_reg;
      case state_reg is
         when idle =>
            ready <= '1';
            if (start='1') then
               state_next <= waite;
            end if;
         when waite => -- wait for the first edge
            if (edge='1') then
               state_next <= count;
               t_next <= (others=>'0');
               p_next <= (others=>'0');
            end if;
         when count =>
            if (edge='1') then    -- 2nd edge arrived
               state_next <= done;
            else -- otherwise count
               if t_reg = CLK_MS_COUNT-1 then -- 1ms tick
                  t_next <= (others=>'0');
                  p_next <= p_reg + 1;
               else
                  t_next <= t_reg + 1;
               end if;
            end if;
         when done =>
            done_tick <= '1';
            state_next <= idle;
      end case;
   end process;
   prd <= std_logic_vector(p_reg);
end arch;
```

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.

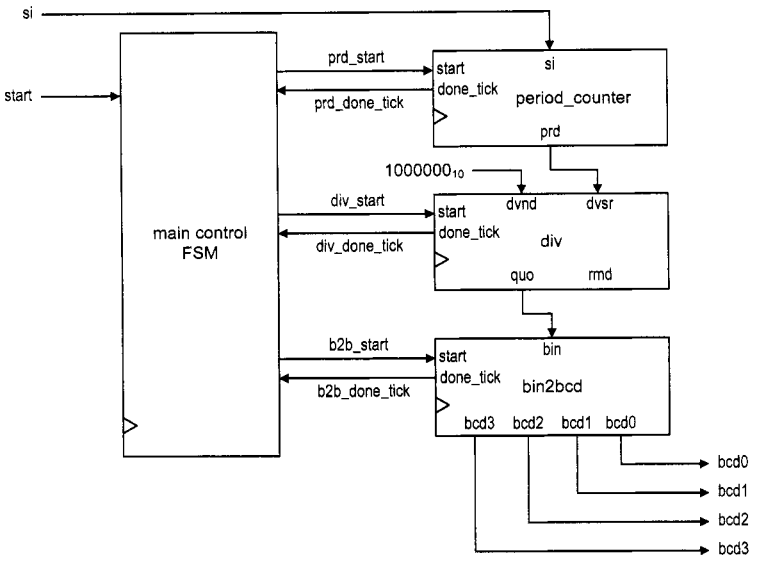
We can use the period counter, division circuit, and binary-to-BCD converter to perform the three tasks and create another FSM as the master control to sequence and coordinate the operation of the three circuits. The block diagram is shown in Figure 6.13(a), and the ASM chart of the master control is 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.

Listing 6.8 Low-frequency counter

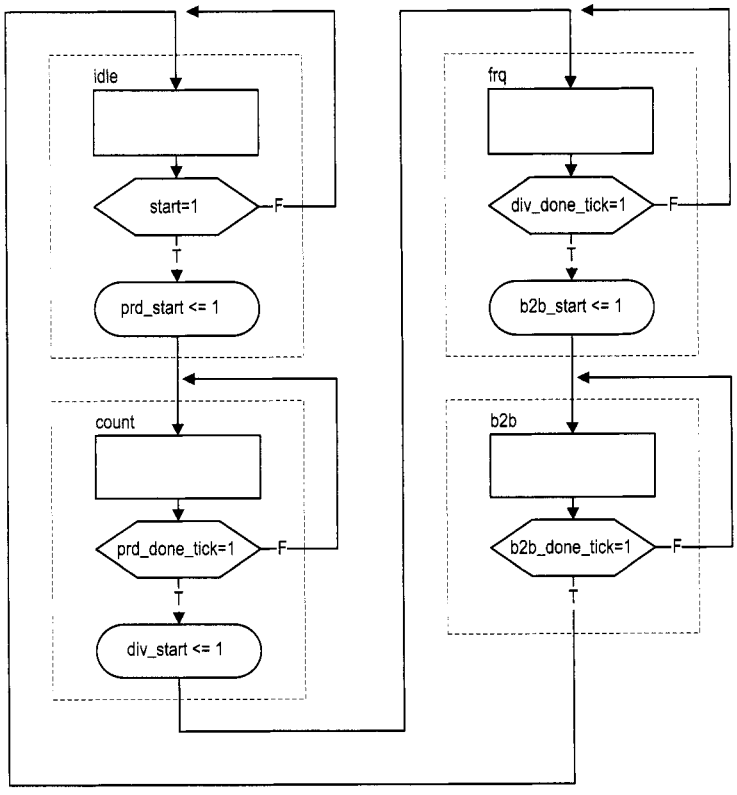
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity low_freq_counter is
   port(
      clk, reset: in std_logic;
      start: in std_logic;
      si: in std_logic;
      bcd3,bcd2,bcd1,bcd0: out std_logic_vector(3 downto 0)
   );
end low_freq_counter;

architecture arch of low_freq_counter is
   type state_type is (idle, count, frq, b2b);
   signal state_reg, state_next: state_type;
   signal prd: std_logic_vector(9 downto 0);
   signal dvsr, dvnd, quo: std_logic_vector(19 downto 0);
   signal prd_start, div_start, b2b_start: std_logic;
   signal prd_done_tick, div_done_tick, b2b_done_tick:
          std_logic;
begin
   --================================================
   -- component instantiation
   --================================================
   -- instantiate period counter
   prd_count_unit: entity work.period_counter
   port map(clk=>clk, reset=>reset, start=>prd_start, si=>si,
            ready=>open, done_tick=>prd_done_tick, prd=>prd);
   -- instantiate division circuit
   div_unit: entity work.div
   generic map(W=>20, CBIT=>5)
   port map(clk=>clk, reset=>reset, start=>div_start,
            dvsr=>dvsr, dvnd=>dvnd, quo=>quo, rmd=>open,
            ready=>open, done_tick=>div_done_tick);
   -- instantiate binary-to-BCD convertor
   bin2bcd_unit: entity work.bin2bcd
   port map
      (clk=>clk, reset=>reset, start=>b2b_start,
       bin=>quo(12 downto 0), ready=>open,
       done_tick=>b2b_done_tick,
       bcd3=>bcd3, bcd2=>bcd2, bcd1=>bcd1, bcd0=>bcd0);
   -- signal width extension
   dvnd <= std_logic_vector(to_unsigned(1000000, 20));
   dvsr <= "0000000000" & prd;

```



(a) Top-level block diagram



(b) ASM chart of main control

Figure 6.13 Accurate low-frequency counter.

```
   --=================================================
   -- master FSM
   --=================================================
   process(clk,reset)
50 begin
      if reset='1' then
         state_reg <= idle;
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
55    end if;
   end process;

   process(state_reg,start,
           prd_done_tick,div_done_tick,b2b_done_tick)
60 begin
      state_next <= state_reg;
      prd_start <='0';
      div_start <='0';
      b2b_start <='0';
65    case state_reg is
         when idle =>
            if start='1' then
               state_next <= count;
               prd_start <='1';
70          end if;
         when count =>
            if (prd_done_tick='1') then
               div_start <='1';
               state_next <= frq;
75          end if;
         when frq =>
            if (div_done_tick='1') then
               b2b_start <='1';
               state_next <= b2b;
80          end if;
         when b2b =>
            if (b2b_done_tick='1') then
               state_next <= idle;
            end if;
85    end case;
   end process;
end arch;
```

6.4 BIBLIOGRAPHIC NOTES

The bibliographic information for this chapter is similar to that for Chapter 3.

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 the BCD format.

The first design approach is to follow the procedure 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 previous "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 returning 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 of 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.