```
   --=================================================
   process(clk)
   begin
      if (clk'event and clk='1') then
         btn_reg <= btn(1);
      end if;
   end process;
   btn_tick <= (not btn_reg) and btn(1);

   --=================================================
   -- two counters
   --=================================================
   clr <= btn(0);
   process(clk)
   begin
      if (clk'event and clk='1') then
         q1_reg <= q1_next;
         q0_reg <= q0_next;
      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
              q1_reg;
   q0_next <= (others=>'0') when clr='1' else
              q0_reg + 1 when db_tick='1' else
              q0_reg;
   -- counter output
   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

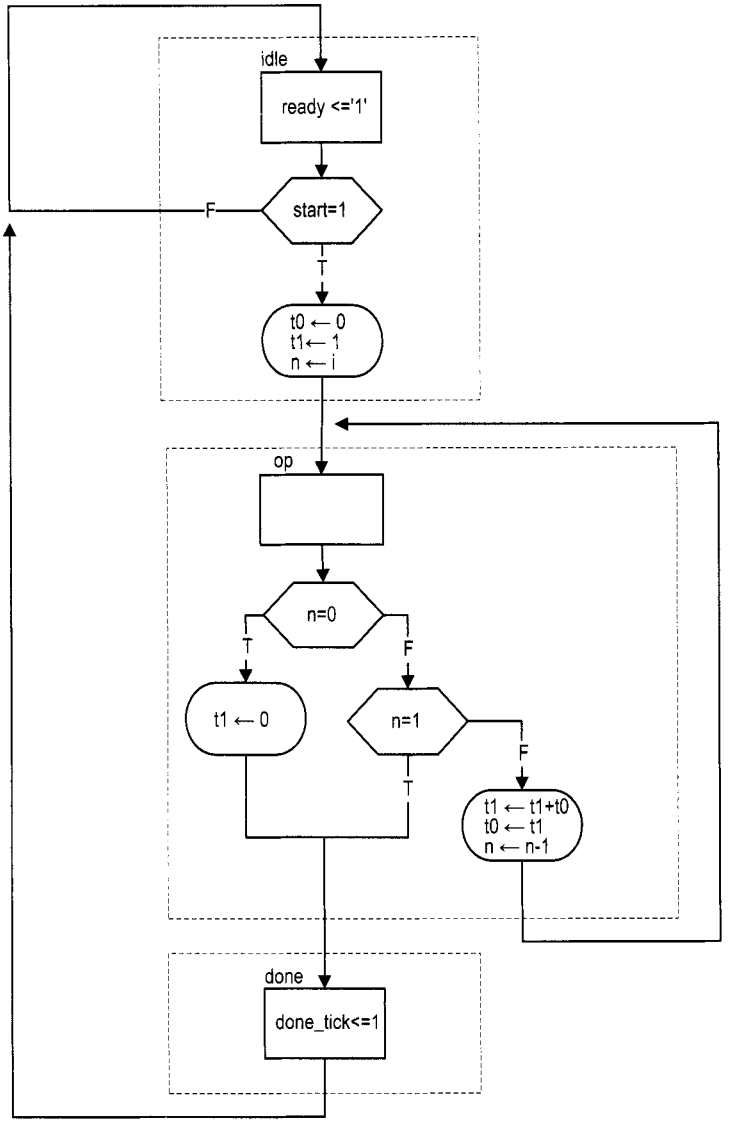


Figure 6.9 ASMD chart of a Fibonacci circuit.

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

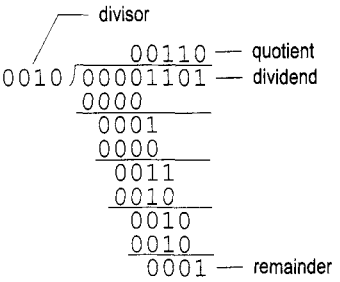


Figure 6.10 Long division of two 4-bit unsigned integers.

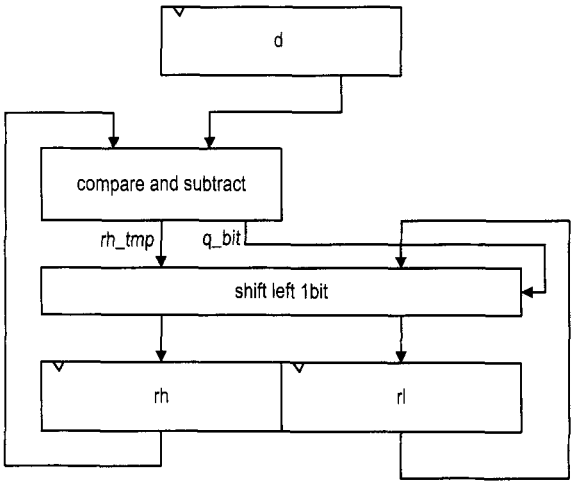


Figure 6.11 Sketch of division circuit's data path.

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

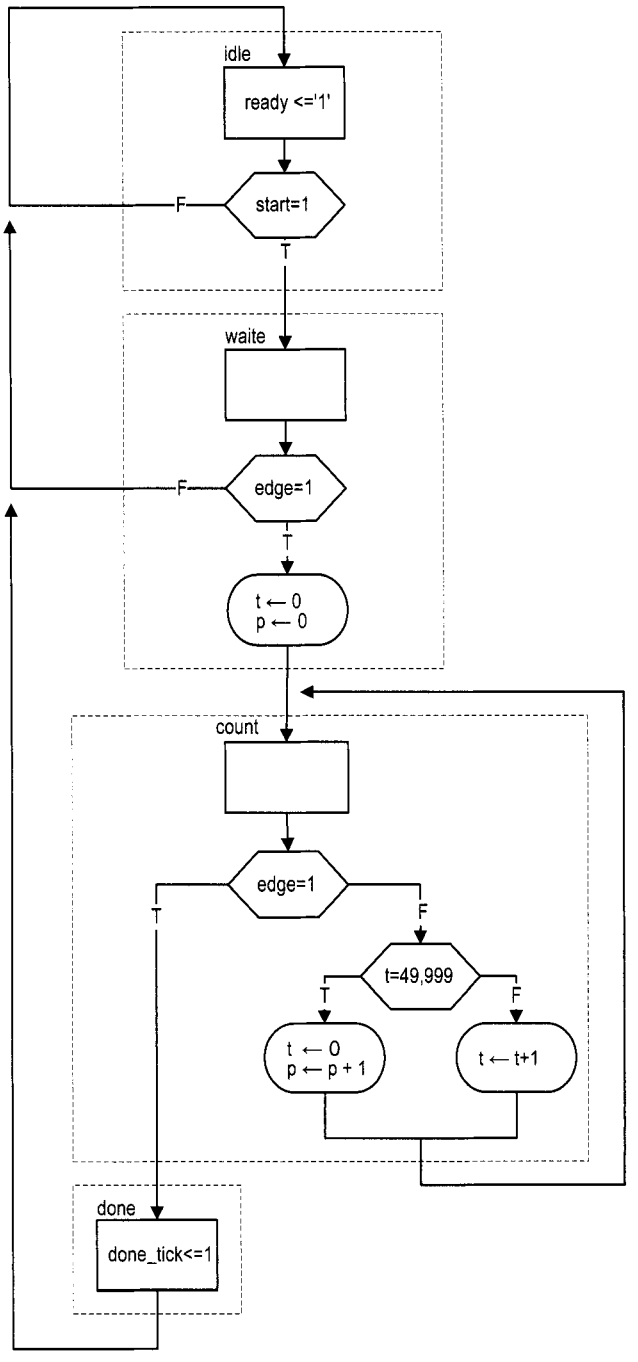


Figure 6.12 ASMD chart of a period counter.

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.

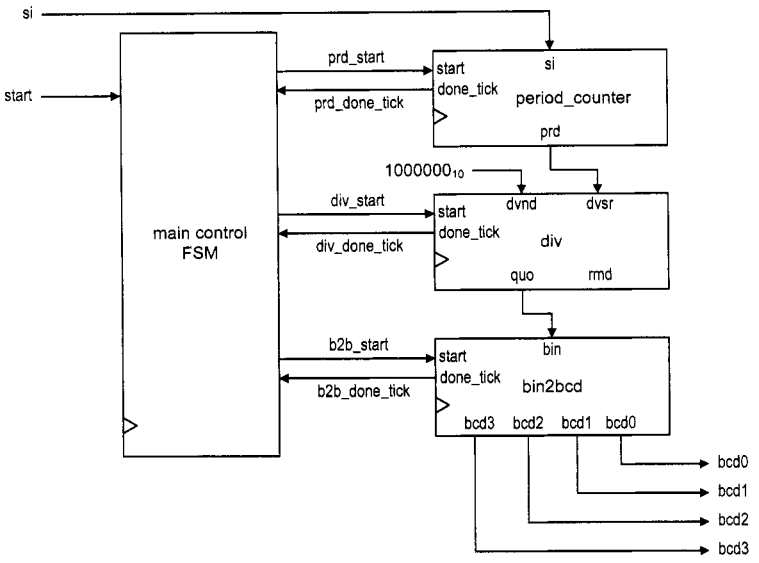
We can use the period counter, division circuit, and binary-to-BCD converter to perform the three tasks and create another FSM as the master control to sequence and coordinate the operation of the three circuits. The block diagram is shown in Figure 6.13(a), and the ASM chart of the master control is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

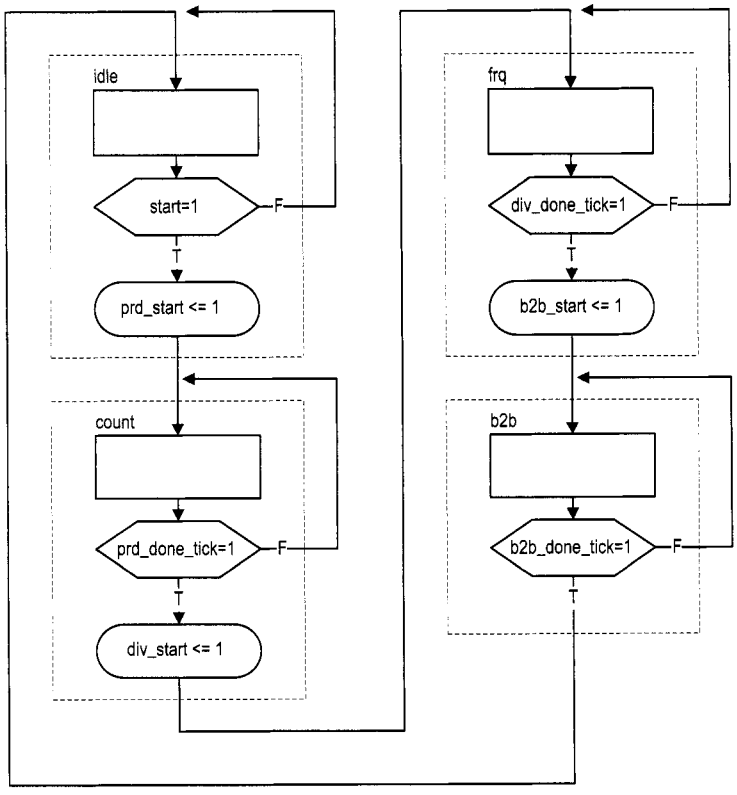
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity low_freq_counter is
   port(
      clk, reset: in std_logic;
      start: in std_logic;
      si: in std_logic;
      bcd3,bcd2,bcd1,bcd0: out std_logic_vector(3 downto 0)
   );
end low_freq_counter;

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.