```
            begin
               db_level = 1'b1;
               if (~sw)
                  begin
                     state_next = wait0;
                     q_load = 1'b1;
                  end
            end
         wait0:
            begin
               db_level = 1'b1;
               if (~sw)
                  begin
                     q_dec = 1'b1;
                     if (q_zero)
                        state_next = zero;
                  end
               else // sw==1
                  state_next = one;
            end
         default: state_next = zero;
      endcase
   end

endmodule
```

6.2.3 Code with implicit data path components

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

Listing 6.2 Debouncing circuit with an implicit data path component

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

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

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

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

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

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

         end
      default: state_next = zero;
   endcase
end

endmodule
```

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

6.2.4 Comparison

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

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