```
module PatternMatch (
  Clk,
  Reset_,
  Trigger,
  Data,
  Match
);

input          Clk, Reset_, Trigger;
input   [7:0] Data;
output         Match;

reg            Match, NextMatch;
reg     [2:0] State, NextState;

`define IDLE    3´h0
`define WAIT01  3´h1
`define WAIT02  3´h2
`define WAIT03  3´h3
`define WAIT04  3´h4

// sequential state vector block

always @(posedge Clk)
begin
  if (!Reset_)
    State[2:0] <= `IDLE;
  else
    State[2:0] <= NextState[2:0];
end

// register output of FSM
// could be combined with above state vector block if desired

always @(posedge Clk)
begin
  if (!Reset_)
    Match <= 1´b0;
  else
    Match <= NextMatch;
end

// FSM combinatorial logic

always @(State[2:0] or Trigger or Data[7:0])
begin
  NextState[2:0] = State[2:0]; // default values prevent latches
  NextMatch      = 1´b0;
  case (State[2:0])

    `IDLE :
       if (Trigger && (Data[7:0] == 8´h01))
         NextState[2:0] = `WAIT02;
       else if (Trigger)  // data != 0x01
         NextState[2:0] = `WAIT01;
```

FIGURE 10.17 Pattern-matching FSM logic.

```
    `WAIT01 :
       if (Data[7:0] == 8´h01)
         NextState[2:0] = `WAIT02;
       // else not required due to default assignment above

    `WAIT02 :
       if (Data[7:0] == 8´h01)
         NextState[2:0] = `WAIT02;
       else if (Data[7:0] == 8´h02)
         NextState[2:0] = `WAIT03;
       else // data != 0x01 or 0x02
         NextState[2:0] = `WAIT01;

    `WAIT03 :
       if (Data[7:0] == 8´h01)
         NextState[2:0] = `WAIT02;
       else if (Data[7:0] == 8´h03)
         NextState[2:0] = `WAIT04;
       else // data != 0x01 or 0x03
         NextState[2:0] = `WAIT01;

    `WAIT04 :
       if (Data[7:0] == 8´h01)
         NextState[2:0] = `WAIT02;
       else if (Data[7:0] == 8´h04) begin
         NextState[2:0] = `IDLE;
         NextMatch      = 1´b1;
       end
       else // data != 0x01 or 0x04
         NextState[2:0] = `WAIT01;

    default : NextState[2:0] = `IDLE; // handle unknown state vectors

  endcase
end

endmodule
```

FIGURE 10.17 (continued) Pattern-matching FSM logic.

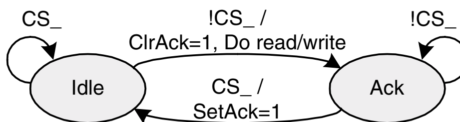

FIGURE 10.18 Asynchronous bus slave FSM bubble diagram.

secondary transaction on behalf of the microprocessor and then transition to a state that waits until the secondary transaction completes before acknowledging the microprocessor.

It may appear that writing a formal FSM when the state vector is a single flop is too cumbersome. Equivalent functionality can be obtained with a simple if…else structure instead of a case statement and separate vector assignment block. The advantages to coding the FSM in a formal manner are consistency and ease of maintenance. If the logic grows in complexity to include more states, it will be relatively easy to augment an existing FSM structure rather than rewriting one from an if…else