```
reg  State, NextState;
reg  CS_Input_, CS_Sync_, CpuAck_;

`define IDLE  1'h0
`define ACK   1'h1

always @(posedge Clk)
begin
  if (!Reset_)
    State <= `IDLE;
  else
    State <= NextState;
end

// FSM support logic: synchronization and registered output

always @(posedge Clk)
begin
  if (!Reset_) begin
    CS_Input_ <= 1'b1; // active low signals reset to high
    CS_Sync_  <= 1'b1;
    CpuAck_   <= 1'b1;
  end
  else begin
    CS_Input_ <= CS_;       // first synchronizer stage
    CS_Sync_  <= CS_Input_; // second synchronizer stage

    if (SetAck)
      CpuAck_ <= 1'b1;

    else if (ClrAck)
      CpuAck_ <= 1'b0;
  end
end

// FSM logic assumes supporting logic:
//
// CpuDataOE enables tristate output for reads
// WriteEnable enables writes to registers decoded from address inputs

always @(State or CS_Sync_ or Rd_ or Wr_)
begin
  NextState = State; // default values prevent latches

  ClrAck      = 1'b0;
  CpuDataOE   = 1'b0;
  SetAck      = 1'b0;
  WriteEnable = 1'b0;

  case (State)

    `IDLE :
       if (!CS_Sync_) begin
         NextState   = `ACK;
         ClrAck      = 1'b1;
         CpuDataOE   = !Rd_;
         WriteEnable = !Wr_;
       end

    `ACK :
       if (CS_Sync_) begin  // wait for CS_ deassertion
         NextState   = `IDLE;
         SetAck      = 1'b1;
       end

  endcase
end
```

FIGURE 10.19 Asynchronous bus slave logic.

construct. Real bus control FSMs typically requires additional states to handle more complex transaction. In the end, the decision is a matter of individual preference and style.

10.6 FSM OPTIMIZATION

FSM complexity can rapidly get out of hand when designing logic to execute a complex algorithm. Two related ways in which complexity manifests itself are excessively large FSMs and timing problems. FSMs with dozens of states can, on their own, lead to timing problems resulting from the many levels of logic necessary to map the full set of inputs and the current state vector to the full set of outputs and the next state vector. Yet, even FSMs with relatively few states can exhibit timing problems if the branch conditions get overly complex.

When complex branch conditions are combined with a very large FSM, the result can be a timing nightmare if suitable design decisions are not made from the beginning. A conceivably poor result could be logic that needs to run at 66 MHz being barely capable of 20-MHz operation. In most instances, an acceptable level of performance can be obtained by properly optimizing the FSM and its supporting logic from the start. If this is not true, chances are that a fundamental change is necessary in either the implementation technology (e.g., faster logic circuits) or the overall architectural approach to solving the problem (e.g., more parallelism in dividing a task into smaller pieces). FSM timing optimization techniques include partitioning, state vector encoding methods, and pipelining.

Proper partitioning of FSMs, and logic in general, is a major factor in successful systems development. Even the best optimization schemes can fail if the underlying FSM is improperly structured. It is usually better to design a system with multiple smaller FSMs instead of a few large ones. Smaller logic structures will tend to have fewer inputs, thereby reducing the complexity of the logic and improving its performance. When the interfaces between multiple FSMs are registered, long timing paths are broken to isolate each FSM from a timing perspective. Without registered interfaces, it is possible for multiple FSMs to form a large loop of dense logic as one feeds back to another. This would defeat a primary benefit of designing smaller FSMs.

Partitioning functionality across smaller FSMs not only makes it easier to improve their timing, it also makes it easier to design and debug the logic. Each smaller section of logic can be assigned to a different engineer for concurrent development, or the sections can be developed serially in a progressive manner by designing and testing each element sequentially. By the time the entire design has been completed, the daunting task of simulating everything at once can be substantially minimized, because each section has already been tested individually. Bugs are likely to arise when all the pieces are put together, but the overall magnitude of the debugging process should be reduced.

State-vector encoding methods are most often considered as a choice between two options: *binary encoding* and *one-hot encoding*. A binary encoded FSM is equivalent to the examples presented earlier in this chapter: a state vector is chosen with N flops such that 2^N is greater than or equal to the total number of states in the FSM. Each state is assigned a unique value in the range of 0 to $2^N - 1$. A one-hot FSM allocates one state flop for each unique state and adheres to a rule that only one flop is set (hot) on any single cycle. The benefits of a one-hot FSM include reduced complexity of output logic and reduced power consumption. Output logic complexity is often reduced by one-hot encoding, because the entire state vector does not have to be decoded. Instead, only those state flops that directly map to an output signal are included in the Boolean expression. Power consumption is reduced, because only two flops change state at a time (the old state flop transitions from high to low, and the new state flop transitions from low to high) instead of many or all state bits. The decision to implement one-hot encoding varies according to the size of the FSM and the constraints of the implementation technology. Beyond a certain size, one-hot encoding becomes too unwieldy and may actually result in more logic than a binary encoded version. Different technologies (e.g., custom ver-