sus programmable logic) provide more or less flexibility in how flops and logic gates are placed together. In many cases, custom logic (e.g., an ASIC) is more efficient with a binary encoded FSM, whereas programmable logic performs better with moderately sized one-hot FSMs.

An FSM can be explicitly written as one-hot, but most leading HDL synthesis tools (e.g., Cadence BuildGates, Exemplar Leonardo Spectrum, Synplicity Synplify, or Synopsys Design Compiler and FPGA Express) have options to automatically evaluate a binary encoded FSM in either its native encoding or a one-hot scheme and then pick the best result. These tools save the engineer the manual time of trying various encoding styles. Better yet, if an FSM works well in one-hot encoding at one phase during the design, and then better as binary encoded due to later design changes, the synthesis tool will automatically change the encoding style without manual intervention.

If a situation arises in which it is advantageous to manually write an FSM using one-hot encoding, the technique is straightforward. Once the number of required state flops has been counted, RTL can be written as shown in Fig. 10.20 using the previous example of an asynchronous bus slave FSM.

There are two significant syntactical elements that enable the one-hot FSM encoding. The first is using the case (1) construct wherein the individual test cases become the individual state vector bits as indexed by the defined state identifiers. Next is the actual NextState assignment, made by left shifting the constant 1 by the state identifiers. Left shifting by the state vector index of the desired next state causes that next state's bit, and only that bit, to be set while the other bits are cleared.

```
// Define bit positions of each state in the set of state bits

`define IDLE  0  // state = 01
`define ACK   1  // state = 10

always @(State[1:0] or CS_Sync_ or Rd_ or Wr_)
begin
  NextState[1:0] = State[1:0];

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

  case (1)

    State[`IDLE] :
       if (!CS_Sync_) begin
         NextState[1:0] = 1 << `ACK;
         ClrAck         = 1´b1;
         CpuDataOE      = !Rd_;
         WriteEnable    = !Wr_;
       end

    State[`ACK] :
       if (CS_Sync_) begin
         NextState[1:0] = 1 << `IDLE;
         SetAck         = 1´b1;
       end

  endcase
end
```

FIGURE 10.20 One-hot encoded FSM.

10.7 PIPELINING

Pipelining is a significant method of improving FSM timing in a way similar to normal logic. A complex FSM contains a large set of inputs that feeds the next state logic. As the number of branch variables across the entire FSM increases, long timing paths quickly develop. It is advantageous to precompute many branch conditions and reduce a condition to a binary variable: true or false. In doing so, the branch variables are moved away from the FSM logic and behind a pipeline flop. The logic delay penalty of evaluating the branch condition is hidden by the pipeline flop. This simplifies the FSM logic because it has to evaluate only the true/false condition result, which is a function of one variable, instead of the whole branch condition, which can be a function of many variables.

An example of using pipelining to improve FSM timing is whereby an FSM is counting events and waits in a particular state until the event has occurred N times, at which point the FSM branches to a new state. In this situation, a counter lies off to the side of the FSM, and the FSM asserts a signal to increment the counter each time a particular event occurs. Without pipelining, the FSM would compare the counter bits against the constant, N, as a branch condition. If the counter is eight bits wide, there is a function of eight variables plus any other qualifiers. If the counter is 16 bits wide, at least 16 variables are present. Many such Boolean equations in the same FSM can lead to timing problems.

Pipelining can be implemented by performing the comparison outside of the FSM with the true/false result being stored in a flop that is directly evaluated by the FSM, thereby reducing the evaluation to a function of only one variable. The trick to making this work properly is that the latency of the pipelined comparison needs to be built into both the comparison logic and the FSM itself.

In a counter situation without pipelining, the FSM increments the counter when an event occurs and simultaneously evaluates the current count value to determine if N events have occurred. Therefore, the FSM would have to make its comparison against N – 1 instead of against N if it wanted to react at the proper time. When the counter is at N – 1 and the event occurs, the FSM knows that this event is the Nth event in the sequence.

Inserting a pipeline flop into the counter comparison logic adds a cycle of latency to the comparison. The FSM cannot simply look ahead an additional cycle to N – 2, because it cannot tell the future: it does not ordinarily know if an event will occur in the next cycle. This problem can be addressed in the pipelined comparison logic. The comparison logic must be triggered at N – 2 so that the FSM can observe the asserted signal on the following cycle when the counter is at N – 1, which enables the FSM to operate as before to recognize the Nth event. Because an ultimate comparison to N – 1 is desired, the pipelined logic can evaluate the expression, "If count equals N – 2 and increment is true, then count will equal N – 1 in the next cycle." In the next cycle, the FSM will have the knowledge that count equals N – 1 from a single flop, and it can qualify this in the same way it did without the pipelining with the end same result. Verilog code fragments to implement this scheme are shown in Fig. 10.21, where the FSM completes its task when 100 events have been observed.

Pipelining should be carefully considered when an FSM is evaluating long bit vectors if it is believed that proper timing will be difficult to achieve. Counter comparisons to constants are not the only candidates for pipelining. Valid pipelining candidates include counter comparisons to other registers and arithmetic embedded within branch conditions. Arithmetic expressions such as one shown in Fig. 10.22 can add long timing paths to an already complex FSM.

Adding two vectors and then comparing their sum to a third vector is a substantial quantity of logic. Such situations are not uncommon in an FSM that must negotiate complex algorithms. It would be highly advantageous to perform the arithmetic and comparison in a pipelined manner so that the FSM would have to evaluate only a single flop. The complexity involved in pipelining such an expression is highly dependent on the application's characteristics. Pipelining can get tricky when it becomes necessary to look ahead one cycle and deal with possible exceptions to the look-ahead