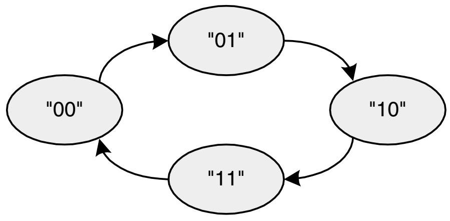

FIGURE 10.15 Two-bit counter bubble diagram.

conditional or the default if no other conditional arcs are valid. Because this is a simple counter, each state has one unconditional arc that leads to the next state.

To illustrate basic FSM concepts, consider a stream of bytes being driven onto a data interface and a task to detect a certain pattern of data within that stream when a trigger signal is asserted. That pattern is the consecutive set of data 0x01, 0x02, 0x03, and 0x04. When this consecutive pattern has been detected, an output detection flag should be set to indicate a match. The logic should never miss detecting this pattern, no matter what data precedes or follows it. After some consideration, the bubble diagram in Fig. 10.16 can be developed. It is common to name states with useful names such as "Idle" rather than binary numbers for readability. FSMs often sit in an idle or quiescent state while they wait for a triggering event that starts their execution.

The FSM waits in the Idle state until trigger is asserted. If the data value at this point is already 0x01, the FSM skips directly to the Wait02 state. Otherwise, it branches to Wait01, where it remains indefinitely until the value 0x01 is observed and it branches to Wait02. There are three possible arcs coming from Wait02. If the value 0x02 is observed, the FSM can advance to the third matching state, Wait03. If 0x01 is observed again, the FSM remains in Wait02, because 0x02 is the next value in the sequence following 0x01. If neither of these values is observed, the FSM branches back to Wait01 to start over again. Wait03 is similar, although there is no arc to remain in the same state. If the first sequence value, 0x01, is observed, the next state is Wait02. If Wait03 does succeed in immediately observing 0x03, the FSM can advance to the final matching state, Wait04. If Wait04 immediately observes 0x04, a match is declared by asserting the Match output flag, and the FSM completes its function by returning to Idle. Otherwise, like in Wait03, the FSM branches back to Wait01 or Wait02.

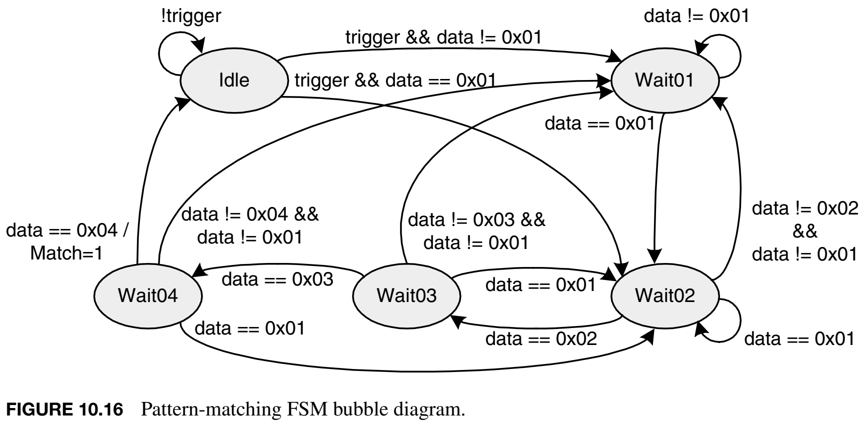

FIGURE 10.16 Pattern-matching FSM bubble diagram.

FSMs are formally classified into one of two types: *Moore* and *Mealy*. A Moore FSM is one wherein the output signals are functions of only the current state vector. A Mealy FSM is one in which the output signals are functions of both the current state as well as the inputs to the FSM. The pattern-matching FSM is a Mealy FSM, because Match is asserted when in state Wait04 and the input data is equal to 0x04. It could be converted into a Moore FSM by adding an additional state, perhaps called Matched, that would be inserted in the arc between Wait04 and Idle. The benefit of doing this would be to reduce logic complexity, because the output signals are functions of only the state vector rather than the inputs as well. However, this comes at the cost of adding an extra state. As FSMs get fairly complex, it may be too cumbersome to rigidly conform to a Moore design.

An FSM is typically coded in HDL using two blocks: a sequential (clocked) state vector block and a combinatorial state and output assignment block. The pattern-matching FSM can be written in Verilog as shown in Fig. 10.17. Rather than calling out numerical state values directly in the RTL, constants are defined to make the code more readable. Verilog supports the *`define* construct that associates a literal text string with a text identifier. The output, Match, is explicitly registered before leaving the module so that downstream logic timing will not be slowed down by incurring the penalty of the FSM logic. Registering the signal means that the output is that of a flop directly, rather than through an arbitrary number of logic gates that create additional timing delays.

In reading the FSM code, it can be seen that the state vector is three bits wide, because there are five states in total. However, 2^3 equals 8, indicating that three of the possible state vector values are invalid. Although none of these invalid values should ever arise, system reliability is increased by inserting the *default* clause into the case statement. This assignment ensures that any invalid state will result in the FSM returning to Idle. Without a default clause, any invalid state would not cause any action to be taken, and the FSM would remain in the same state indefinitely. It is good practice to insert default clauses into case statements when writing FSMs to guard against a hung condition. It is admittedly unlikely that the state vector will assume an invalid value, but if a momentary glitch were to happen, the design would benefit from a small amount of logic to restore the FSM to a valid state.

10.5 FSM BUS CONTROL

FSMs are well suited to managing arbitrarily complex bus interfaces. Microprocessors, memory devices, and I/O controllers can have interfaces with multiple states to handle arbitration between multiple requestors, wait states, and multiword transfers (e.g., a burst SDRAM). Each of these bus states can be represented by an FSM state, thereby breaking a seemingly complex problem into quite manageable pieces.

A simple example of CPU bus control is an asynchronous CPU interface with request/acknowledge signaling in which the FSM runs on a different clock from that of the CPU itself. Asynchronous CPU buses tend to implement four-corner handshaking to prevent the CPU and peripheral logic from getting out of step with one another. Two asynchronous control signals, chip select (CS_) and acknowledge (ACK_), can be managed with an FSM by breaking the distinct phases of a CPU access into separate states as shown in Fig. 10.18. The FSM waits in Idle until a synchronized CS_ is observed, at which point it executes the transaction, asserts ACK_, and then transitions to the Ack state. Once in Ack, the FSM waits until CS_ is deasserted before deasserting ACK_.

The logic is shown implemented in Verilog in Fig. 10.19. This FSM logic is simplistic in that it assumes that all transactions can be executed and acknowledged immediately. In reality, certain applications will make such rapid execution impossible. A memory controller will need to insert latency while data is fetched, for example. Such situations can be handled by adding states to the FSM that handle such cases. Rather than *Idle* immediately returning an acknowledge, it could initiate a