the multiplier is stored in register Q, and the partial product is formed in register A and stored in A and Q. A parallel adder adds the contents of register B to register A. The C flip-flop stores the carry after the addition. The counter P is initially set to hold a binary number equal to the number of bits in the multiplier. This counter is decremented after the formation of each partial product. When the content of the counter reaches zero, the product is formed in the double register A and Q, and the process stops. The control logic stays in an initial state until *Start* becomes 1. The system then performs the multiplication. The sum of A and B forms the n most significant bits of the partial product, which is transferred to A. The output carry from the addition, whether 0 or 1, is transferred to C. Both the partial product in A and the multiplier in Q are shifted to the right. The least significant bit of A is shifted into the most significant position of Q, the carry from C is shifted into the most significant position of A, and 0 is shifted into C. After the shift-right operation, one bit of the partial product is transferred into Q while the multiplier bits in Q are shifted one position to the right. In this manner, the least significant bit of register Q, designated by *Q[0]*, holds the bit of the multiplier that must be inspected next. The control logic determines whether to add or not on the basis of this input bit. The control logic also receives a signal, *Zero*, from a circuit that checks counter P for zero. *Q[0]* and *Zero* are status inputs for the control unit. The input signal *Start* is an external control input. The outputs of the control logic launch the required operations in the registers of the datapath unit.

The interface between the controller and the datapath consists of the status signals and the output signals of the controller. The control signals govern the synchronous register operations of the datapath. Signal *Load_regs* loads the internal registers of the datapath, *Shift_regs* causes the shift register to shift, *Add_regs* forms the sum of the multiplicand and register A, and *Decr_P* decrements the counter. The controller also forms output *Ready* to signal to the host environment that the machine is ready to multiply. The contents of the register holding the product vary during execution, so it is useful to have a signal indicating that its contents are valid. Note, again, that the state of the control is not an interface signal between the control unit and the datapath. Only the signals needed to control the datapath are included in the interface. Putting the state in the interface would require a decoder in the datapath, and require a wider and more active bus than the control signals alone. Not good.

ASMD Chart

The ASMD chart for the binary multiplier is shown in Fig. 8.15. The intermediate form in Fig. 8.15(a) annotates the ASM chart of the controller with the register operations, and the completed chart in Fig. 8.15(b) identifies the Moore and Mealy outputs of the controller. Initially, the multiplicand is in B and the multiplier in Q. As long as the circuit is in the initial state and $Start = 0$, no action occurs and the system remains in state *S_idle* with *Ready* asserted. The multiplication process is launched when $Start = 1$. Then, (1) control goes to state *S_add*, (2) register A and carry flip-flop C are cleared to 0, (3) registers B and Q are loaded with the multiplicand and the multiplier, respectively, and (4) the sequence counter P is set to a binary number n, equal to the number of bits in the multiplier. In state *S_add*, the multiplier bit in *Q[0]* is checked, and if it is equal to 1, the multiplicand in B is added to the partial product in A. The carry from the addition is transferred to C. The partial product

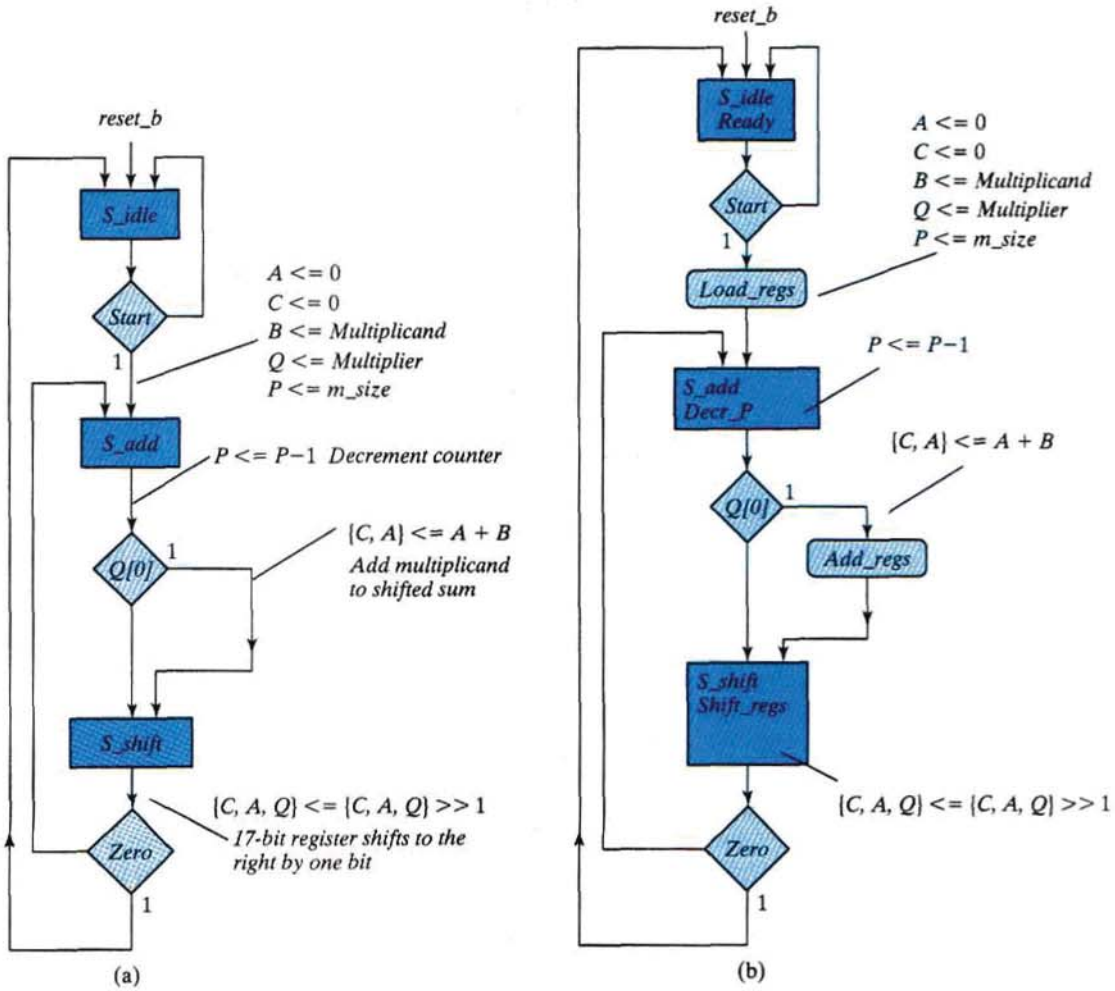


FIGURE 8.15
ASMD chart for binary multiplier

in A and C is left unchanged if $Q[0] = 0$. The counter P is decremented by 1 regardless of the value of $Q[0]$, so *Decr_P* is formed in state *S_add* as a Moore output of the controller. In both cases, the next state is *S_shift*. Registers C, A, and Q are combined into one composite register CAQ, denoted by the concatenation $\{C, A, Q\}$, and its contents are shifted once to the right to obtain a new partial product. This shift operation is symbolized in the flowchart with the Verilog logical right-shift operator, >>. It is equivalent to the following statement in register transfer notation:

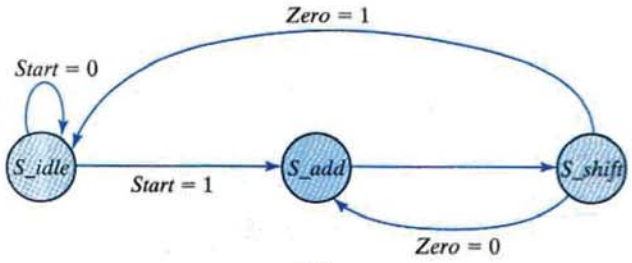
$$\text{Shift right } CAQ,\ C \leftarrow 0$$

to parallel load a binary constant. The C flip-flop must be designed to accept the input carry and have a synchronous clear. Registers B and Q need a parallel load capability in order to receive the multiplicand and multiplier prior to the start of the multiplication process.

8.8 CONTROL LOGIC

The design of a digital system can be divided into two parts: the design of the register transfers in the datapath unit and the design of the control logic of the control unit. The control logic is a finite state machine; its Mealy- and Moore-type outputs control the operations of the datapath. The inputs to the control unit are the primary (external) inputs and the internal status signals fed back from the datapath to the controller. The design of the system can be synthesized from an RTL description derived from the ASMD chart. Alternatively, a manual design must derive the logic governing the inputs to the flip-flops holding the state of the controller. The information needed to form the state diagram of the controller is already contained in the ASMD chart, since the rectangular blocks that designate state boxes are the states of the sequential circuit. The diamond-shaped blocks that designate decision boxes determine the logical conditions for the next state transition in the state diagram.

As an example, the control state diagram for the binary multiplier developed in the previous section is shown in Fig. 8.16(a). The information for the diagram is taken directly from the



(a)

State Transition		Register Operations
From	To	
S_idle		Initial state
S_idle	*S_add*	$A <= 0, C <= 0, P <= dp\_width$
S_add	*S_shift*	$P <= P - 1$ if $(Q[0])$ then $(A <= A + B, C <= C_{out})$
S_shift		shift right $\{CAQ\}, C <= 0$

(b)

FIGURE 8.16
Control specifications for binary multiplier

number of states and inputs is much larger. In general, the application of the classical method requires an excessive amount of work to obtain the simplified input equations for the flip-flops and is prone to error. The design can be simplified if we take into consideration the fact that the decoder outputs are available for use in the design. Instead of using flip-flop outputs as the present-state conditions, *we use the outputs of the decoder to indicate the present-state condition of the sequential circuit*. Moreover, instead of using maps to simplify the flip-flop equations, we can obtain them directly by inspection of the state table. For example, from the next-state conditions in the state table, we find that the next state of G_1 is equal to 1 when the present state is *S_add* and is equal to 0 when the present state is *S_idle* or *S_shift*. These conditions can be specified by the equation

$$D_{G1} = T_1$$

where D_{G1} is the D input of flip-flop G_1. Similarly, the D input of G_0 is

$$D_{G0} = T_0\, Start + T_2\, Zero'$$

When deriving input equations by inspection from the state table, we cannot be sure that the Boolean functions have been simplified in the best possible way. (Synthesis tools take care of this detail automatically.) In general, it is advisable to analyze the circuit to ensure that the equations derived do indeed produce the required state transitions.

The logic diagram of the control circuit is drawn in Fig. 8.17(b). It consists of a register with two flip-flops G_1 and G_0 and a 2 × 4 decoder. The outputs of the decoder are used to generate the inputs to the next-state logic as well as the control outputs. The outputs of the controller should be connected to the datapath to activate the required register operations.

One-Hot Design (One Flip-Flop per State)

Another method of control logic design is the one-hot assignment, which results in a sequential circuit with one flip-flop per state. Only one of the flip-flops contains a 1 at any time; all others are reset to 0. The single 1 propagates from one flip-flop to another under the control of decision logic. In such a configuration, each flip-flop represents a state that is present only when the control bit is transferred to it.

This method uses the maximum number of flip-flops for the sequential circuit. For example, a sequential circuit with 12 states requires a minimum of four flip-flops. By contrast, with the method of one flip-flop per state, the circuit requires 12 flip-flops, one for each state. At first glance, it may seem that this method would increase system cost, since more flip-flops are used. But the method offers some advantages that may not be apparent. One advantage is the simplicity with which the logic can be designed by inspection of the ASMD chart or the state diagram. No state or excitation tables are needed if D-type flip-flops are employed. The one-hot method offers a savings in design effort, an increase in operational simplicity, and a possible decrease in the total number of gates, since a decoder is not needed.

The design procedure will be demonstrated by obtaining the control circuit specified by the state diagram of Fig. 8.16(a). Since there are three states in the state diagram, we choose three D flip-flops and label their outputs G_0, G_1, and G_2, *corresponding to S_idle*, *S_add*, and *S_shift*, respectively. The input equations for setting each flip-flop to 1 are determined from the present state and

the input conditions along the corresponding directed lines going into the state. For example, D_{G0}, the input to flip-flop G_0, is set to 1 if the machine is in state G_0 and *Start* is not asserted, or if the machine is in state G_2 and *Zero* is asserted. These conditions are specified by the input equation:

$$D_{G0} = G_0\, Start' + G_2\, Zero$$

In fact, the condition for setting a flip-flop to 1 is obtained directly from the state diagram, from the condition specified in the directed lines going into the corresponding flip-flop state ANDed with the previous flip-flop state. If there is more than one directed line going into a state, all conditions must be ORed. Using this procedure for the other three flip-flops, we obtain the remaining input equations:

$$D_{G1} = G_0\, Start + G_2\, Zero'$$

$$D_{G2} = G_1$$

The logic diagram of the one-hot controller (with one flip-flop per state) is shown in Fig. 8.18. The circuit consists of three *D* flip-flops labeled G_0 through G_2, together with the associated gates

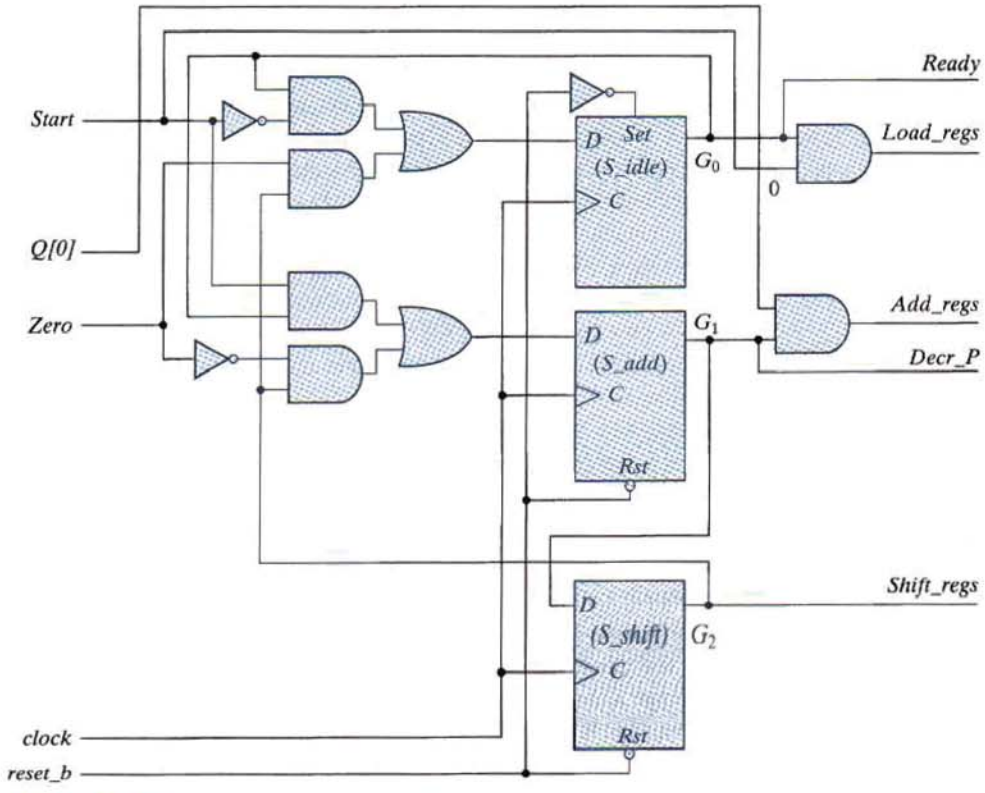


FIGURE 8.18
Logic diagram for one-hot state controller

specified by the input equations. Initially, flip-flop G_0 must be set to 1 and all other flip-flops must be reset to 0, so that the flip-flop representing the initial state is enabled. This can be done by using an asynchronous preset on flip-flop G_0 and an asynchronous clear for the other flip-flops. Once started, the controller with one flip-flop per state will propagate from one state to the other in the proper manner. Only one flip-flop will be set to 1 with each clock edge; all others are reset to 0, because their D inputs are equal to 0.

8.9 HDL DESCRIPTION OF BINARY MULTIPLIER

A second example of an HDL description of an RTL design is given in HDL Example 8.5. The example is of the binary multiplier designed in Section 8.7. For simplicity, the entire description is "flattened" and encapsulated in one module. Comments will identify the controller and the datapath. The first part of the description declares all of the inputs and outputs as specified in the block diagram of Fig. 8.14(a). The machine will be parameterized for a five-bit datapath to enable a comparison between its simulation data and the result of the multiplication with the numerical example listed in Table 8.5. The same model can be used for a datapath having a different size merely by changing the value of the parameters. The second part of the description declares all registers in the controller and the datapath, as well as the one-hot encoding of the states. The third part specifies implicit combinational logic (continuous assignment statements) for the concatenated register *CAQ*, the *Zero* status signal, and the *Ready* output signal. The continuous assignments for *Zero* and *Ready* are accomplished by assigning a Boolean expression to their **wire** declarations. The next section describes the control unit, using a single edge-sensitive cyclic behavior to describe the state transitions, and a level-sensitive cyclic behavior to describe the combinational logic for the next state and the outputs. Again, note that default assignments are made to *next_state*, *Load_regs*, *Decr_P*, *Add_regs*, and *Shift_regs*. The subsequent logic of the case statement assigns their value by exception. The state transitions and the output logic are written directly from the ASMD chart of Fig. 8.15(b).

The datapath unit describes the register operations within a separate edge-sensitive cyclic behavior. (For clarity, separate cyclic behaviors are used; we do not mix the description of the datapath with the description of the controller.) Each control input is decoded and is used to specify the associated operations. The addition and subtraction operations will be implemented in hardware by combinational logic. Signal *Load_regs* causes the counter and the other registers to be loaded with their initial values, etc. Because the controller and datapath have been partitioned into separate units, the control signals completely specify the behavior of the datapath; explicit information about the state of the controller is not needed and is not made available to the datapath unit.

The next-state logic of the controller includes a default case item to direct a synthesis tool to map any of the unused codes to *S_idle*. The default case item and the default assignments preceding the **case** statement ensure that the machine will recover if it somehow enters an unused state. They also prevent unintentional synthesis of latches. (Remember, a synthesis tool will synthesize latches when what was intended to be combinational logic in fact fails to completely specify the input–output function of the logic.)