Chapter 8

Design at the Register Transfer Level

8.1 INTRODUCTION

A digital system is a sequential logic system constructed with flip-flops and gates. Sequential circuits can be specified by means of state tables as shown in Chapter 5. To specify a large digital system with a state table is very difficult, because the number of states would be enormous. To overcome this difficulty, digital systems are designed via a modular approach. The system is partitioned into modular subsystems, each of which performs some function. The modules are constructed from such digital devices as registers, decoders, multiplexers, arithmetic elements, and control logic. The various modules are interconnected with datapaths and control signals to form a digital system. In this chapter, we will introduce a design methodology for describing and designing large, complex digital systems.

8.2 REGISTER TRANSFER LEVEL (RTL) NOTATION

The modules of a digital system are best defined by a set of registers and the operations that are performed on the binary information stored in them. Examples of register operations are *shift*, *count*, *clear*, and *load*. Registers are assumed to be the basic components of the digital system. The information flow and processing performed on the data stored in the registers are referred to as *register transfer operations*. We'll see subsequently how a hardware description language includes operators that correspond to the register transfer operations of a digital system. A digital system is represented at the *register transfer level* (RTL) when it is specified by the following three components:

1. The set of registers in the system.
2. The operations that are performed on the data stored in the registers.
3. The control that supervises the sequence of operations in the system.

A register is a group of flip-flops that stores binary information and has the capability of performing one or more elementary operations. A register can load new information or shift the information to the right or the left. A counter is considered a register that increments a number by a fixed value (e.g., 1). A flip-flop is considered a one-bit register that can be set, cleared, or complemented. In fact, the flip-flops and associated gates of any sequential circuit can be called registers by this definition.

The operations executed on the information stored in registers are elementary operations that are performed in parallel on a data word consisting of bits during one clock cycle. The data produced by the operation may replace the binary information that was in the register before the operation executed. Alternatively, the result may be transferred to another register (i.e., an operation on a register may leave its contents unchanged). The digital circuits introduced in Chapter 6 are registers that implement elementary operations. A counter with a parallel load is able to perform the increment-by-one and load operations. A bidirectional shift register is able to perform the shift-right and shift-left operations.

The operations in a digital system are controlled by timing signals that sequence the operations in a prescribed manner. Certain conditions that depend on results of previous operations may determine the sequence of future operations. The outputs of the control logic are binary variables that initiate the various operations in the system's registers.

Information transfer from one register to another is designated in symbolic form by means of a replacement operator. The statement

$$R2 \leftarrow R1$$

denotes a transfer of the contents of register *R1* into register *R2*—that is, a replacement of the contents of register *R2* by the contents of register *R1*. By definition, the contents of the source register *R1* do not change after the transfer. They are merely copied to *R1*. The arrow symbolizes the transfer and its direction; it points from the register whose contents are being transferred and towards the register that will receive the contents. A control signal would determine when the operation actually executes.

The controller in a digital system is a finite state machine whose outputs are the control signals governing the register operations. In synchronous machines, the operations are synchronized by the system clock.

A statement that specifies a register transfer operation implies that a datapath (i.e., a set of circuit connections) is available from the outputs of the source register to the inputs of the destination register and that the destination register has a parallel load capability. Data can be transferred serially between registers, too, by repeatedly shifting their contents along a single wire, one bit at a time. Normally, we want a register transfer operation to occur, not with every clock cycle, but only under a predetermined condition. A conditional statement governing a register transfer operation is symbolized with an if-then statement such as

$$\text{If } (T1 = 1) \text{ then } (R2 \leftarrow R1)$$

where *T1* is a control signal generated in the control section. Note that the clock is not included as a variable in the register transfer statements. It is assumed that all transfers occur at a clock-edge transition (i.e., a transition from 0 to 1 or from 1 to 0). Although a control condition such as *T1* may become true before the clock transition, the actual transfer does not occur until the clock transition does.

A comma may be used to separate two or more operations that are executed at the same time (concurrently). Consider the statement

$$\text{If } (T3 = 1) \text{ then } (R2 \leftarrow R1,\ R1 \leftarrow R2)$$

This statement specifies an operation that exchanges the contents of two registers; moreover, the operation in both registers is triggered by the same clock edge, provided that $T3 = 1$. This simultaneous operation is possible with registers that have edge-triggered flip-flops controlled by a common clock (synchronizing signal). Other examples of register transfers are as follows:

$R1 \leftarrow R1 + R2$	Add contents of $R2$ to $R1$ ($R1$ gets $R1 + R2$)
$R3 \leftarrow R3 + 1$	Increment $R3$ by 1 (count upwards)
$R4 \leftarrow \text{shr } R4$	Shift right $R4$
$R5 \leftarrow 0$	Clear $R5$ to 0

In hardware, addition is done with a binary parallel adder, incrementing is done with a counter, and the shift operation is implemented with a shift register. The type of operations most often encountered in digital systems can be classified into four categories:

1. Transfer operations, which transfer (i.e., copy) data from one register to another.
2. Arithmetic operations, which perform arithmetic on data in registers.
3. Logic operations, which perform bit manipulation (e.g., logical OR) of nonnumeric data in registers.
4. Shift operations, which shift data between registers.

The transfer operation does not change the information content of the data being moved from the source register to the destination register. The other three operations change the information content during the transfer. The register transfer notation and the symbols used to represent the various register transfer operations are not standardized. In this text, we employ two types of notation. The notation introduced in this section will be used informally to specify and explain digital systems at the register transfer level. The next section introduces the RTL symbols used in the Verilog HDL.

8.3 REGISTER TRANSFER LEVEL IN HDL

Digital systems can be described at the register transfer level by means of a hardware description language (HDL). In Verilog, descriptions of RTL operations use a combination of behavioral and dataflow constructs and are employed to specify the register operations and the combinational logic functions implemented by hardware. Register transfers are specified by means of procedural assignment statements within an edge-sensitive cyclic behavior. Combinational circuit functions are specified at the RTL level by means of continuous assignment statements or by procedural assignment statements within a level-sensitive cyclic behavior. The symbol used to designate a register transfer is either an equals sign (=) or an arrow (<=); the symbol used to specify a combinational circuit function is an equals sign. Synchronization

with the clock is represented by associating with an **always** statement an event control expression in which sensitivity to the clock event is qualified by **posedge** or **negedge**. The **always** keyword indicates that the associated block of statements will be executed repeatedly, for the life of the simulation. The @ operator and the event control expression preceding the block of statements synchronize the execution of the statements to the clock event.

The following examples show the various ways to specify a register transfer operation in Verilog:

```
(a) assign  S = A + B;          // Continuous assignment for addition operation
(b) always @ (A, B)             // Level-sensitive cyclic behavior
      S = A + B;                // Combinational logic for addition operation
(c) always @ (negedge clock)    // Edge-sensitive cyclic behavior
      begin
        RA = RA + RB;           // Blocking procedural assignment for addition
        RD = RA;                        // Register transfer operation
      end
(d) always @ (negedge clock)    // Edge-sensitive cyclic behavior
      begin
        RA <= RA + RB;          // Nonblocking procedural assignment for addition
        RD <= RA;                       // Register transfer operation
      end
```

Continuous assignments are used to represent and specify combinational logic circuits. In simulation, a continuous assignment statement executes when the expression on the right-hand side changes. The effect of execution is immediate. (The variable on the left-hand side is updated.) Similarly, a level-sensitive cyclic behavior executes when a change is detected by its event control expression (sensitivity list). The effect of assignments made by the = operator are immediate. The continuous assignment statement (**assign**) describes a binary adder with inputs A and B and output S. The target operand in a continuous assignment statement (S in this case) cannot be a register data type, but must be a type of net, for example, **wire**. The procedural assignment made in the level-sensitive cyclic behavior in the second example shows an alternative way of specifying a combinational circuit for addition. Within the cyclic behavior, the mechanism of the sensitivity list ensures that the output, S, will be updated whenever A, or B, or both change.

There are two kinds of procedural assignments: *blocking* and *nonblocking*. The two are distinguished by the symbols that they use and by their operation. Blocking assignments use the equals symbol (=) as the assignment operator, and nonblocking assignments use the left arrow (<=) as the operator. Blocking assignment statements are executed *sequentially* in the order that they are listed in a sequential block; when they execute, they have an immediate effect on the contents of memory before the next statement can be executed. Nonblocking assignments are made *concurrently*. This feature is implemented by evaluating the expression on the right-hand side of each statement in the list of statements before making the assignment to their left-hand sides. Consequently, there is no interaction between the result of any assignment and the evaluation of an expression affecting another assignment. Also, the statements associated with an edge-sensitive cyclic behavior do not execute until the indicated edge condition occurs.

Consider the two examples given. In the blocking procedural assignment, the first statement transfers the sum to *RA* and the second statement transfers the new value of *RA* into *RD*. At the completion of the operation, both *RA* and *RD* have the same value. In the nonblocking procedural assignment, the two operations are performed concurrently, so that *RD* receives the original value of *RA*. The activity in both examples is launched by the clock undergoing a falling edge transition.

The registers in a system are clocked simultaneously (concurrently). The *D*-input of each flip-flop determines the value that will be assigned to its output, independently of the input to any other flip-flop. To ensure synchronous operations in RTL design, and to ensure a match between an HDL model and the circuit synthesized from the model, it is necessary that nonblocking procedural assignments be used for all variables that are assigned a value within an edge-sensitive cyclic behavior (**always** clocked). The nonblocking assignment that appears in an edge-sensitive cyclic behavior models the behavior of the hardware of a synchronous sequential circuit accurately.

HDL Operators

The Verilog HDL operators and their symbols used in RTL design are listed in Table 8.1. The arithmetic, logic, and shift operators describe register transfer operations. The logical and relational operators specify control conditions and have Boolean expressions as their arguments.

The operands of the arithmetic operators are numbers. The +, −, *, and / operators form the sum, difference, product, and quotient, respectively, of a pair of operands. The exponentiation operator (**) was added to the language in 2001 and forms a double-precision floating-point value from a base and exponent having a real, integer, or signed value. Negative numbers are represented in 2's-complement form. The modulus operator produces the remainder from the division of two numbers. For example, 14 % 3 evaluates to 2.

There are two types of logic operators for binary words: bitwise and reduction. The bitwise operators perform a bit-by-bit operation on two vector operands to form a vector result. They take each bit in one operand and perform the operation with the corresponding bit in the other operand. Negation (~) is a unary operator; it complements the bits of a single vector operand to form a vector result. The reduction operators are also unary, acting on a single operand and producing a scalar (one-bit) result. They operate pairwise on the bits of a word, from right to left, and yield a one-bit result. For example, the reduction NOR (~|) results in 0 with operand 00101 and in 1 with operand 00000. The result of applying the NOR operation on the first two bits is used with the third bit, and so forth. Negation is not used as a reduction operator. Truth tables for the bitwise operators are the same as those listed in Table 4.9 in Section 4.12 for the corresponding Verilog primitive (e.g., the **and** primitive and the **&** bitwise operator have the same truth table). The output of an AND gate with two scalar inputs is the same as the result produced by operating on the two bits with the **&** operator.

The logical and relational operators are used to form Boolean expressions and can take variables or expressions as operands. (*Note*: A variable is also an expression.) Used basically for determining true or false conditions, the logical and relational operators evaluate to 1 if the condition expressed is true and to 0 if the condition is false. If the condition is ambiguous, they evaluate to x. An operand that is a variable evaluates to 0 if the value of the variable is equal to zero and

Table 8.1
Verilog 2001 HDL Operators

Operator Type	Symbol	Operation Performed
Arithmetic	+	addition
	–	subtraction
	*	multiplication
	/	division
	%	modulus
	**	exponentiation
Logic (bitwise or reduction)	~	negation (complement)
	&	AND
	|	OR
	^	exclusive-OR (XOR)
Logical	!	negation
	&&	AND
	||	OR
Shift	>>	logical right shift
	<<	logical left shift
	>>>	arithmetic right shift
	<<<	arithmetic left shift
	{ , }	concatenation
Relational	>	greater than
	<	less than
	==	equality
	!=	inequality
	===	case equality
	!==	case inequality
	>=	greater than or equal
	<=	less than or equal

to 1 if the value is not equal to zero. For example, if $A = 1010$ and $B = 0000$, then the expression A has the Boolean value 1 (the number in question is not equal to 0) and the expression B has the Boolean value 0. Results of other operations with these values are as follows:

```
A && B = 0          // logical AND
A || B = 1          // logical OR
!A = 0              // logical complement
!B = 1              // logical complement
```

```
(A > B) = 1          // is greater than
(A == B) = 0         // identity (equality)
```

The relational operators === and !== test for bitwise equality (identity) and inequality in Verilog's four-valued logic system. For example, if $A = 0xx0$ and $B = 0xx0$, the test $A === B$ would evaluate to true, but the test $A == B$ would evaluate to x.

Verilog 2001 has logical and arithmetic shift operators. The logical shift operators shift a vector operand to the right or the left by a specified number of bits. The vacated bit positions are filled with zeros. For example, if $R = 11010$, then the statement

```
R = R >> 1;
```

shifts R to the right one position. The value of R that results from the logical right-shift operation is 01101. In contrast, the arithmetic right-shift operator fills the vacated cell (the most significant bit (MSB)) with its original contents when the word is shifted to the right. The arithmetic left-shift operator fills the vacated cell with a 0 when the word is shifted to the left. The arithmetic right-shift operator is used when the sign extension of a number is important. If $R = 11010$, then the statement

```
R >>> 1;
```

produces the result $R = 11101$; if $R = 01101$, it produces the result $R = 00110$. There is no distinction between the logical left-shift and the arithmetic left-shift operators.

The concatenation operator provides a mechanism for appending multiple operands. It can be used to specify a shift, including the bits transferred into the vacant positions. This aspect of its operation was shown in HDL Example 6.1 for the shift register.

Expressions are evaluated from left to right, and their operators associate from left to right (with the exception of the conditional operator) according to the precedence shown in Table 8.2. For example, in the expression $A + B - C$, the value of B is added to A, and then C is subtracted from the result. In the expression $A + B/C$, the value of B is divided by C, and then the result is added to A because the division operator (/) has a higher precedence than the addition operator (+). Use parentheses to establish precedence. For example, the expression $(A + B)/C$ is not the same as the expression $A + B/C$.

Loop Statements

Verilog HDL has four types of loops that execute procedural statements repeatedly: *repeat*, *forever*, *while*, and *for*. All looping statements must appear inside an **initial** or **always** block.

The **repeat** loop executes the associated statements a specified number of times. The following is an example that was used previously:

```
initial
 begin
  clock = 1'b0;
  repeat (16)
   #5 clock = ~ clock;
 end
```

This code produces eight clock cycles with a cycle time of 10 time units.

Table 8.2
Verilog Operator Precedence

+ − ! ~ & ~ & | ~| ^ ~^ ^~ (unary)	Highest precedence
**	
*/%	
+− (binary)	
<< >> <<< >>>	
< <= > >=	
== != === !==	
& (binary)	
^ ^~ ~^ (binary)	
| (binary)	
&&	
||	
?: (conditional operator)	
{} {{}}	Lowest precedence

The **forever** loop causes unconditional, repetitive execution of a procedural statement or a block of procedural statements. For example, the following loop produces a continuous clock having a cycle time of 20 time units:

```
initial
 begin
  clock = 1'b0;
  forever
   #10 clock = ~ clock;
 end
```

The **while** loop executes a statement or a block of statements repeatedly while an expression is true. If the expression is false to begin with, the statement is never executed. The following example illustrates the use of the **while** loop:

```
integer count;
 initial
  begin
   count = 0;
   while (count < 64)
    #5 count = count + 1;
  end
```

The value of count is incremented from 0 to 63. Each increment is delayed by five time units, and the loop exits at the count of 64.

In dealing with looping statements, it is sometimes convenient to use the **integer** data type to index the loop. Integers are declared with the keyword **integer**, as in the previous example. Although it is possible to use a **reg** variable to index a loop, sometimes it is more convenient to declare an **integer** variable, rather than a **reg**, for counting purposes. Variables declared as data type **reg** are stored as unsigned numbers. Those declared as data type **integer** are store as signed numbers in 2's-complement format. The default width of an integer is a minimum of 32 bits.

The **for** loop contains three parts separated by two semicolons:

- An initial condition.
- An expression to check for the terminating condition.
- An assignment to change the control variable.

The following is an example of a **for** loop:

```
for (j = 0; j < 8; j = j + 1)
  begin
    // procedural statements go here
  end
```

The **for** loop statement repeats the execution of the procedural statements eight times. The control variable is *j*, the initial condition is $j = 0$, and the loop is repeated as long as *j* is less than 8. After each execution of the loop statement, the value of *j* is incremented by 1.

A description of a two-to-four-line decoder using a **for** loop is shown in HDL Example 8.1. Since output *Y* is evaluated in a procedural statement, it must be declared as type **reg**. The control variable for the loop is the **integer** *k*. When the loop is expanded (unrolled), we get the following four conditions (*IN* and *Y* are in binary, and the index for *Y* is in decimal):

```
if IN = 00 then Y(0) = 1; else Y(0) = 0;
if IN = 01 then Y(1) = 1; else Y(1) = 0;
if IN = 10 then Y(2) = 1; else Y(2) = 0;
if IN = 11 then Y(3) = 1; else Y(3) = 0;
```

HDL Example 8.1

```
// Description of 2 x 4 decoder using a for loop statement
module decoder (IN, Y);
  input       [1: 0] IN;      // Two binary inputs
  output      [3: 0] Y;       // Four binary outputs
  reg         [3: 0] Y;
  integer            k;       // Control (index) variable for loop

  always @ (IN)
    for (k = 0; k <= 3; k = k + 1)
```

```
    if (IN == k) Y[k] = 1;
    else Y[k] = 0;
endmodule
```

Logic Synthesis

Logic synthesis is the automatic process by which a computer-based program (i.e., a synthesis tool) transforms an HDL model of a logic circuit into an optimized netlist of gates that perform the operations specified by the source code. There are various target technologies that implement the synthesized design in hardware. The effective use of an HDL description requires that designers adopt a vendor-specific style suitable for the particular synthesis tools. The type of ICs that implement the design may be an application-specific integrated circuit (ASIC), a programmable logic device (PLD), or a field-programmable gate array (FPGA). Logic synthesis is widely used in industry to design and implement large circuits efficiently, correctly, and rapidly.

Logic synthesis tools interpret the source code of the hardware description language and translate it into an optimized gate structure, accomplishing (correctly) all of the work that would be done by manual methods using Karnaugh maps. Designs written in Verilog or a comparable language for the purpose of logic synthesis tend to be at the register transfer level. This is because the HDL constructs used in an RTL description can be converted into a gate-level description in a straightforward manner. The following examples discuss how a logic synthesizer can interpret an HDL construct and convert it into a gate structure.

The continuous assignment (**assign**) statement is used to describe combinational circuits. In an HDL, it represents a Boolean equation for a logic circuit. A continuous assignment with a Boolean expression for the right-hand side of the assignment statement is synthesized into the corresponding gate circuit implementing the expression. An expression with an addition operator (+) is interpreted as a binary adder with full-adder circuits. An expression with a subtraction operator (−) is converted into a gate-level subtractor consisting of full adders and exclusive-OR gates (Fig. 4.13). A statement with a conditional operator such as

```
assign Y = S ? In_1 : In_0;
```

translates into a two-to-one-line multiplexer with control input *S* and data inputs *In_1* and *In_0*. A statement with multiple conditional operators specifies a larger multiplexer.

A cyclic behavior (**always** ...) may imply a combinational or sequential circuit, depending on whether the event control expression is level sensitive or edge sensitive. A synthesis tool will interpret as combinational logic a level-sensitive cyclic behavior whose event control expression is sensitive to every variable that is referenced within the behavior (e.g., by the variable's appearing in the right-hand side of an assignment statement). The event control expression in a description of combinational logic may not be sensitive to an edge of any signal. For example,

```
always @ (In_1 or In_0 or S)
  if (S) Y = In_1;
  else   Y = In_0;
```

translates into a two-to-one-line multiplexer. As an alternative, the **case** statement may be used to imply large multiplexers. The **casex** statement treats the logic values x and z as don't-cares when they appear in either the case expression or a case item.

An edge-sensitive cyclic behavior (e.g., **always @ (posedge** clock)) specifies a synchronous (clocked) sequential circuit. The implementation of the corresponding circuit consists of *D* flip-flops and the gates that implement the synchronous register transfer operations specified by the statements associated with the event control expression. Examples of such circuits are registers and counters. A sequential circuit description with a **case** statement translates into a control circuit with *D* flip-flops and gates that form the inputs to the flip-flops. Thus, each statement in an RTL description is interpreted by the synthesizer and assigned to a corresponding gate and flip-flop circuit. For synthesizable sequential circuits, the event control expression must be sensitive to the positive or the negative edge of the clock (synchronizing signal), but not to both.

A simplified flowchart of the process used by industry to design digital systems is shown in Fig. 8.1. The RTL description of the HDL design is simulated and checked for proper

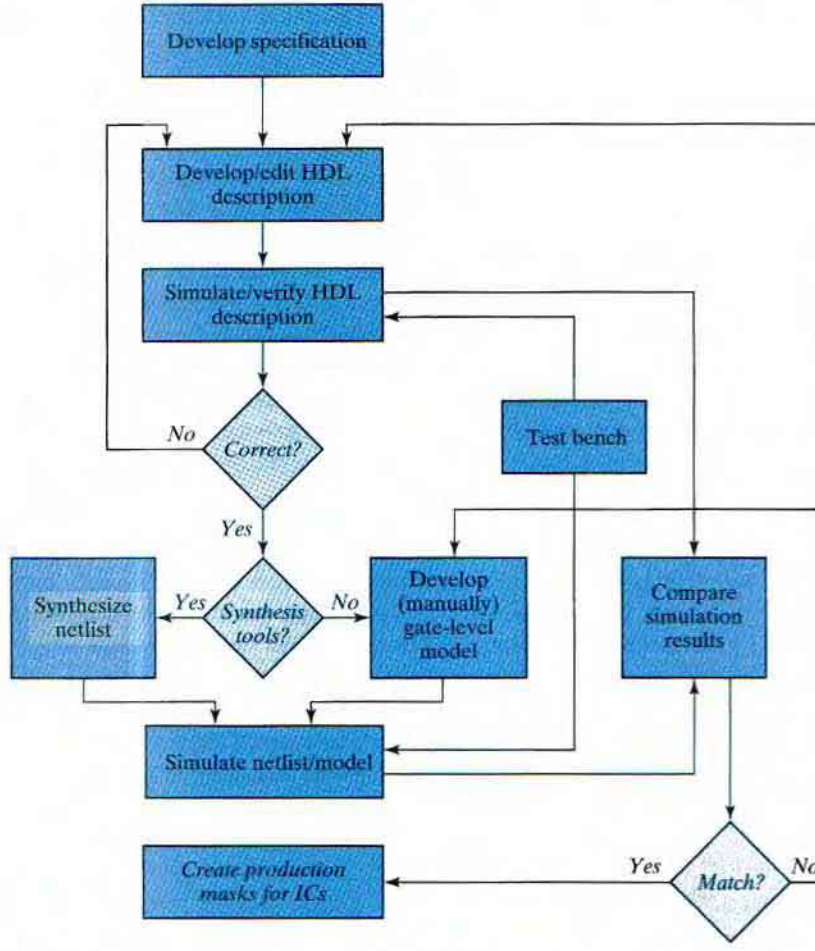


FIGURE 8.1
A simplified flowchart for HDL-based modeling, verification, and synthesis

operation. Its operational features must match those given in the specification for the behavior of the circuit. The test bench provides the stimulus signals to the simulator. If the result of the simulation is not satisfactory, the HDL description is corrected and checked again. After the simulation run shows a valid design, the RTL description is ready to be compiled by the logic synthesizer. All errors (syntax and functional) in the description must be eliminated before synthesis. The synthesis tool generates a netlist equivalent to a gate-level description of the design as it is represented by the model. If the model fails to express the functionality of the specification, the circuit will fail to do so also. The gate-level circuit is simulated with the same set of stimuli used to check the RTL design. If any corrections are needed, the process is repeated until a satisfactory simulation is achieved. The results of the two simulations are compared to see if they match. If they do not, the designer must change the RTL description to correct any errors in the design. Then the description is again compiled by the logic synthesizer to generate a new gate-level description. Once the designer is satisfied with the results of all simulation tests, the design of the circuit is ready for physical implementation in a technology. In practice, additional testing will be performed to verify that the timing specifications of the circuit can be met in the chosen hardware technology. That issue is not within the scope of this text.

Logic synthesis provides several advantages to the designer. It takes less time to write an HDL description and synthesize a gate-level realization than it does to develop the circuit by manual entry from schematic diagrams. The ease of changing the description facilitates exploration of design alternatives. It is faster, easier, less expensive, and less risky to check the validity of the design by simulation than it is to produce a hardware prototype for evaluation. A schematic and the database for fabricating the integrated circuit can be generated automatically by synthesis tools. The HDL model can be compiled by different tools into different technologies (e.g., ASIC cells or FPGAs), providing multiple returns on the investment to create the model.

8.4 ALGORITHMIC STATE MACHINES (ASMs)

The binary information stored in a digital system can be classified as either data or control information. Data are discrete elements of information (binary words) that are manipulated by performing arithmetic, logic, shift, and other similar data-processing operations. These operations are implemented with digital components such as adders, decoders, multiplexers, counters, and shift registers. Control information provides command signals that coordinate and execute the various operations in the data section in order to accomplish the desired data-processing tasks.

The logic design of a digital system can be divided into two distinct parts. One part is concerned with the design of the digital circuits that perform the data-processing operations. The other part is concerned with the design of the control circuits that determine the sequence in which the various actions are performed.

The relationship between the control logic and the data-processing operations in a digital system is shown in Fig. 8.2. The data-processing path, commonly referred to as the *datapath unit*, manipulates data in registers according to the system's requirements. The *control unit* issues a sequence of commands to the datapath unit. Note that an internal feedback path from the datapath unit to the control unit provides status conditions that the control unit uses together with the external (primary) inputs to determine the sequence of control signals (outputs of the control

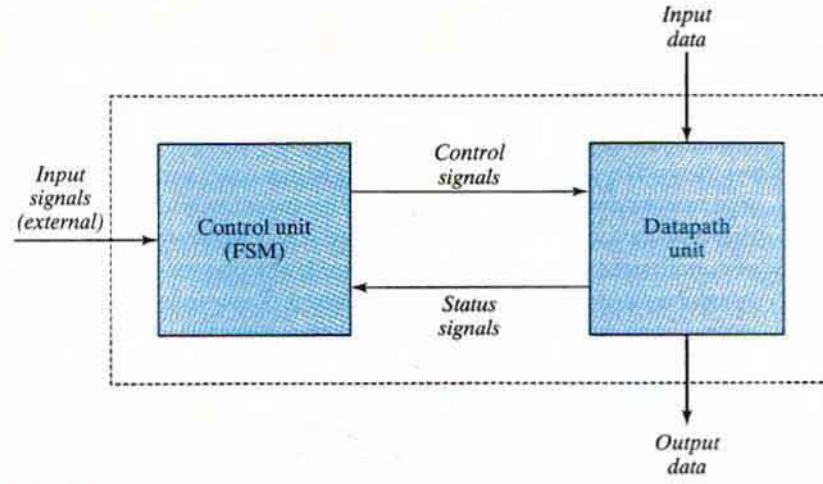


FIGURE 8.2
Control and datapath interaction

unit) that direct the operation of the datapath unit. We'll see later that understanding how to model this feedback relationship with an HDL is very important.

The control logic that generates the signals for sequencing the operations in the datapath unit is a finite state machine (FSM), i.e., a synchronous sequential circuit. The control commands for the system are produced by the FSM as functions of the primary inputs, the status signals, and the state of the machine. In a given state, the outputs of the controller are the inputs to the datapath unit and determine the operations that it will execute. Depending on status conditions and other external inputs, the FSM goes to its next state to initiate other operations. The digital circuits that act as the control logic provide a time sequence of signals for initiating the operations in the datapath and also determine the next state of the control subsystem itself.

The control sequence and datapath tasks of a digital system are specified by means of a hardware algorithm. An algorithm consists of a finite number of procedural steps that specify how to obtain a solution to a problem. A hardware algorithm is a procedure for solving the problem with a given piece of equipment. The most challenging and creative part of digital design is the formulation of hardware algorithms for achieving required objectives. The goal to implement the algorithms in silicon as an integrated circuit.

A flowchart is a convenient way to specify the sequence of procedural steps and decision paths for an algorithm. A flowchart for a hardware algorithm translates the verbal instructions to an information diagram that enumerates the sequence of operations together with the conditions necessary for their execution. A flowchart that has been developed specifically to define digital hardware algorithms is called an *algorithmic state machine* (ASM) chart. A *state machine* is another term for a sequential circuit, which is the basic structure of a digital system.

ASM Chart

The ASM chart resembles a conventional flowchart, but is interpreted somewhat differently. A conventional flowchart describes the procedural steps and decision paths of an algorithm in

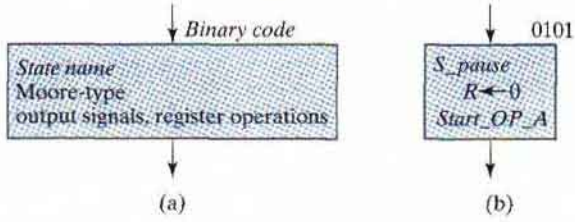


FIGURE 8.3
ASM chart state box

a sequential manner, without taking into consideration their time relationship. The ASM chart describes the sequence of events, as well as the timing relationship between the states of a sequential controller and the events that occur while going from one state to the next (i.e., the events that are synchronous with changes in the state). The chart is adapted to specify accurately the control sequence and datapath operations in a digital system, taking into consideration the constraints of digital hardware.

The ASM chart is composed of three basic elements: the state box, the decision box, and the conditional box. The boxes themselves are connected by directed edges indicating the sequential precedence and evolution of the states as the machine operates. There are various ways to attach information to an ASM chart. In one, a state in the control sequence is indicated by a state box, as shown in Fig. 8.3(a). The shape of the state box is a rectangle within which are written register operations or the names of output signals that the control generates while being in the indicated state. The state is given a symbolic name, which is placed within the upper left corner of the box. The binary code assigned to the state is placed at the upper right corner. (The state symbol and code can be placed in other places as well.) Figure 8.3(b) gives an example of a state box. The state has the symbolic name *S_pause*, and the binary code assigned to it is 0101. Inside the box is written the register operation $R \leftarrow 0$, which indicates that register R is to be cleared to 0. The name *Start_OP_A* inside the box indicates, for example, a Moore-type output signal that is asserted while the machine is in state *S_pause* and that launches a certain operation in the datapath unit.

The style of state box shown in Fig. 8.3(b) is sometimes used in ASM charts, but it can lead to confusion about when the register operation $R \leftarrow 0$ is to execute. Although the operation is written inside the state box, it actually occurs when the machine makes a transition from *S_pause* to its next state. In fact, writing the register operation within the state box is a way (albeit possibly confusing) to indicate that the controller must assert a signal that will cause the register operation to occur when the machine changes state. Later we'll introduce a chart and notation that are more suited to digital design and that will eliminate any ambiguity about the register operations controlled by a state machine.

The decision box of an ASM chart describes the effect of an input (i.e., a primary, or external, input or a status, or internal, signal) on the control subsystem. The box is diamond shaped and has two or more exit paths, as shown in Fig. 8.4. The input condition to be tested is written inside the box. One or the other exit path is taken, depending on the evaluation of the condition. In the binary case, one path is taken if the condition is true and another when the condition is false. When an input condition is assigned a binary value, the two paths are indicated by 1 and 0, respectively.

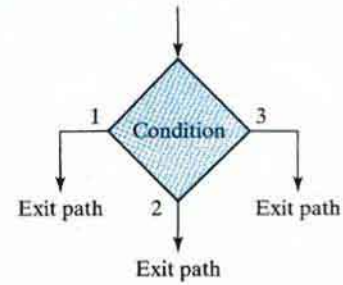


FIGURE 8.4
ASM chart decision box

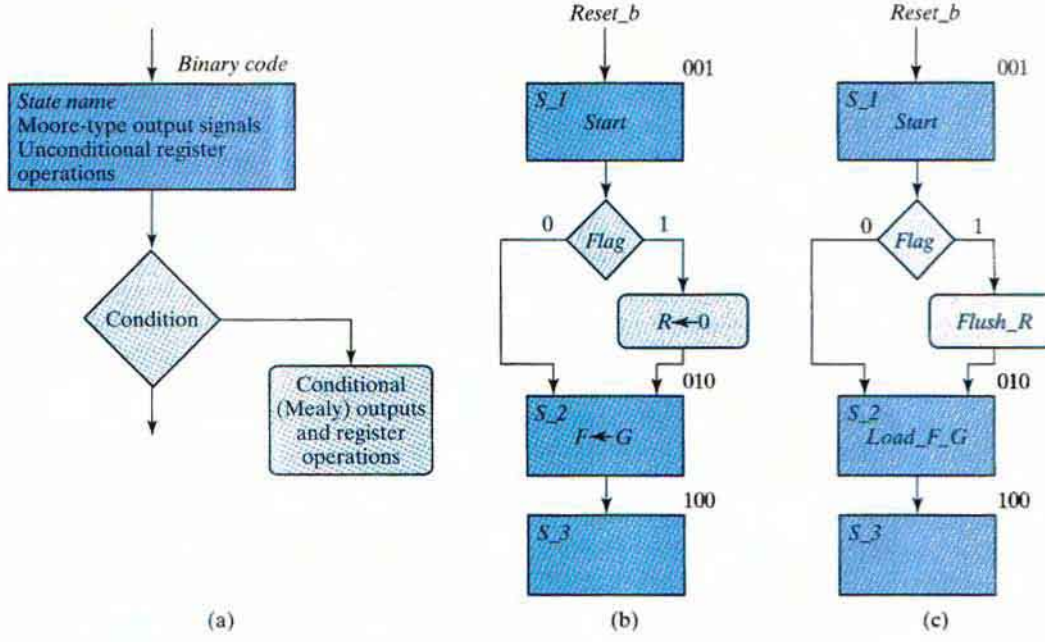


FIGURE 8.5
ASM chart conditional box

The state and decision boxes of an ASM chart are similar to those used in conventional flowcharts. The third element, the conditional box, is unique to the ASM chart. The shape of the conditional box is shown in Fig. 8.5(a). Its rounded corners differentiate it from the state box. The input path to the conditional box must come from one of the exit paths of a decision box. The outputs listed inside the conditional box are generated as Mealy-type signals during a given state; the register operations listed in the conditional box are associated with a transition from the state. Figure 8.5(b) shows an example with a conditional box. The control generates the output signal *Start* when in state *S_1* and checks the status of input *Flag*. If $Flag = 1$,

then R is cleared to 0; otherwise, R remains unchanged. In either case, the next state is S_2. A register operation is associated with S_2. We again note that this style of chart can be a source of confusion, because the state machine does not execute the indicated register operation $R \leftarrow 0$ when it is in S_1 or the operation $F \leftarrow G$ when it is in S_2. The notation actually indicates that when the controller is in S_1, it must assert a Mealy-type signal that will cause the register operation $R \leftarrow 0$ to execute in the datapath unit, subject to the condition that $Flag = 0$. Likewise, in state S_2, the controller must generate a Moore-type output signal that causes the register operation $F \leftarrow G$ to execute in the datapath unit. The operations in the datapath unit are synchronized to the clock edge that causes the state to move from S_1 to S_2 and from S_2 to S_3, respectively. Thus, the control signal generated in a given state affects the operation of a register in the datapath when the next clock transition occurs. The result of the operation is apparent in the next state.

The ASM chart in Fig. 8.5(b) mixes descriptions of the datapath and the controller. An ASM chart for only the controller is shown in Fig. 8.5(c), in which the register operations are omitted. In their place are the control signals that must be generated by the control unit to launch the operations of the datapath unit. This chart is useful for describing the controller, but it does not contain adequate information about the datapath. (We'll address this issue later.)

ASM Block

An ASM block is a structure consisting of one state box and all the decision and conditional boxes connected to its exit path. An ASM block has one entrance and any number of exit paths represented by the structure of the decision boxes. An ASM chart consists of one or more interconnected blocks. An example of an ASM block is given in Fig. 8.6. Associated with state

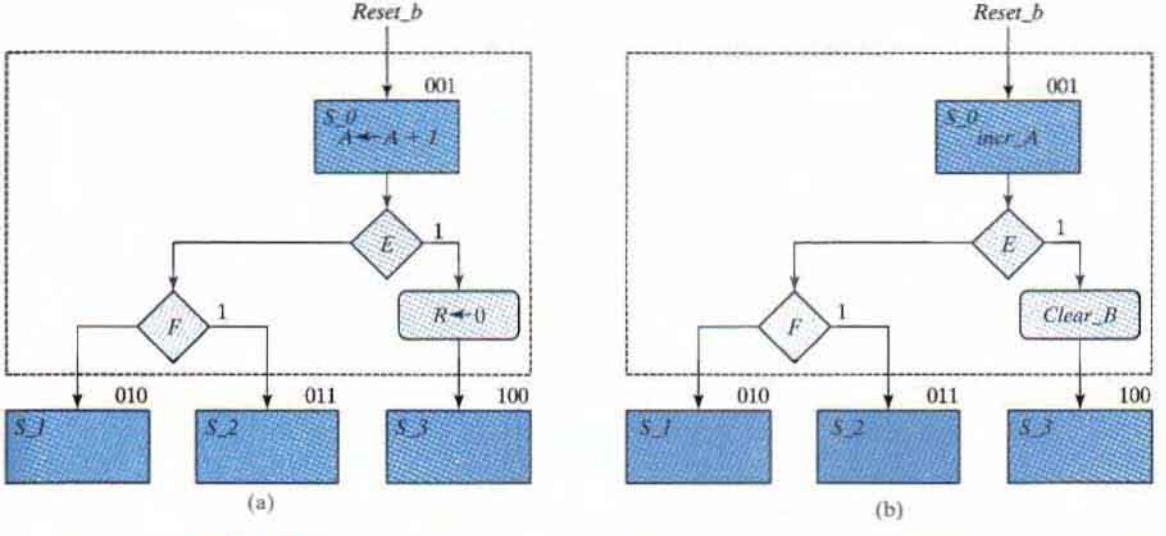


FIGURE 8.6
ASM block

S_0 are two decision boxes and one conditional box. The diagram distinguishes the block with dashed lines around the entire structure, but this is not usually done, since the ASM chart uniquely defines each block from its structure. A state box without any decision or conditional boxes constitutes a simple block.

Each block in the ASM chart describes the state of the system during one clock-pulse interval (i.e., the interval between two successive active edges of the clock). The operations within the state and conditional boxes in Fig. 8.6(a) are initiated by a common clock pulse when the state of the controller transitions from *S_0* to its next state. The same clock pulse transfers the system controller to one of the next states, *S_1*, *S_2*, or *S_3*, as dictated by the binary values of *E* and *F*. The ASM chart for the controller alone is shown in Fig. 8.6(b). The Moore-type signal *incr_A* is asserted while the machine is in *S_0*; the Mealy-type signal *Clear_R* is generated conditionally when the state is *S_0* and *E* is asserted. In general, the Moore-type outputs of the controller are generated unconditionally and are indicated within a state box; the Mealy-type outputs are generated conditionally and are indicated in the conditional boxes connected to the edges that leave a decision box.

The ASM chart is similar to a state diagram. Each state block is equivalent to a state in a sequential circuit. The decision box is equivalent to the binary information written along the directed lines that connect two states in a state diagram. As a consequence, it is sometimes convenient to convert the chart into a state diagram and then use sequential circuit procedures to design the control logic. As an illustration, the ASM chart of Fig. 8.6 is drawn as a state diagram in Fig. 8.7. The states are symbolized by circles, with their binary values written inside. The directed lines indicate the conditions that determine the next state. The unconditional and conditional operations that must be performed in the datapath unit are not indicated in the state diagram.

Simplifications

A binary decision box of an ASM chart can be simplified by labeling only the edge corresponding to the asserted decision variable and leaving the other edge without a label. A further simplification is to omit the edges corresponding to the state transitions that occur when a reset condition is asserted. Output signals that are not asserted are not shown on the chart; the presence of the name of an output signal indicates that it is asserted.

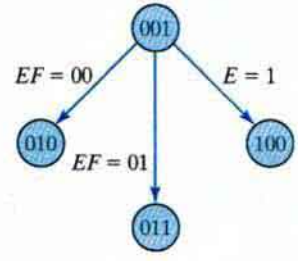


FIGURE 8.7
State diagram equivalent to the ASM chart of Fig. 8.6

Timing Considerations

The timing for all registers and flip-flops in a digital system is controlled by a master-clock generator. The clock pulses are applied not only to the registers of the datapath, but also to all the flip-flops in the state machine implementing the control unit. Inputs are also synchronized to the clock, because they are normally generated as outputs of another circuit that uses the same clock signals. If the input signal changes at an arbitrary time independently of the clock, we call it an asynchronous input. Asynchronous inputs may cause a variety of problems, as discussed in Chapter 9. To simplify the design, we will assume that all inputs are synchronized with the clock and change state in response to an edge transition.

The major difference between a conventional flowchart and an ASM chart is in interpreting the time relationship among the various operations. For example, if Fig. 8.6 were a conventional flowchart, then the operations listed would be considered to follow one after another in sequence: First register A is incremented, and only then is E evaluated. If $E = 1$, then register R is cleared and control goes to state *S_3*. Otherwise (if $E = 0$), the next step is to evaluate F and go to state *S_1* or *S_2*. In contrast, an ASM chart considers the entire block as one unit. All the register operations that are specified within the block must occur in synchronism at the edge transition of the same clock pulse while the system changes from *S_0* to the next state. This sequence of events is presented pictorially in Fig. 8.8. We assume positive-edge triggering of all flip-flops. An asserted asynchronous reset signal (*reset_b*) transfers the control circuit into state *S_0*. While in state *S_0*, the control circuits check inputs E and F and generate appropriate signals accordingly. If *reset_b* is not asserted, the following operations occur simultaneously at the next positive edge of the clock:

1. Register A is incremented.
2. If $E = 1$, register R is cleared.
3. Control transfers to the next state, as specified in Fig. 8.7.

Note that the two operations in the datapath and the change of state in the control logic occur at the same time. Note also that the ASM chart in Fig. 8.6(a) indicates the register operations that must occur in the datapath unit, but does not indicate the control signal that is to be formed by the control unit. Conversely, the chart in Fig. 8.6(b) indicates the control signals, but not the datapath operations. We will now present an ASMD chart to provide the clarity and complete information needed by logic designers.

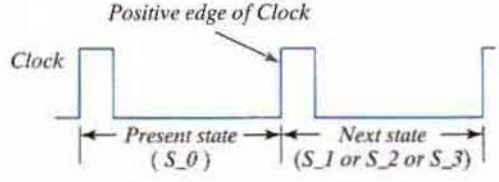


FIGURE 8.8
Transition between states

ASMD Chart

Algorithmic state machine and datapath (ASMD) charts were developed to clarify the information displayed by ASM charts and to provide an effective tool for designing a control unit for a given datapath unit. An ASMD chart differs from an ASM chart in three important ways: (1) An ASMD chart does not list register operations within a state box, (2) the edges of an ASMD chart are annotated with register operations that are concurrent with the state transition indicated by the edge, and (3) an ASMD chart includes conditional boxes identifying the signals which control the register operations that annotate the edges of the chart. Thus, *an ASMD chart associates register operations with state transitions rather than with states.*

Designers form an ASMD chart in a three-step process that creates an annotated and completely specified ASM chart for the controller of a datapath unit. The steps are to (1) form an ASM chart displaying only how the inputs to the controller determine its state transitions, (2) convert the ASM chart to an ASMD chart by annotating the edges of the ASM chart to indicate the concurrent register operations of the datapath unit, and (3) modify the ASMD chart to identify the control signals that are generated by the controller and that cause the indicated register operations in the datapath unit. The ASMD chart produced by this process clearly and completely specifies the finite state machine of the controller and identifies the register operations of the given datapath.

One important use of a state machine is to control register operations on a datapath in a sequential machine that has been partitioned into a controller and a datapath. An ASMD chart links the ASM chart of the controller to the datapath it controls in a manner that serves as a universal model representing all synchronous digital hardware design. ASMD charts help clarify the design of a sequential machine by separating the design of its datapath from the design of the controller, while maintaining a clear relationship between the two units. Register operations that occur concurrently with state transitions are annotated on a path of the chart, rather than in state boxes or in conditional boxes on the path, because these registers are not part of the controller. The outputs generated by the controller are the signals that control the registers of the datapath and cause the register operations annotated on the ASMD chart.

8.5 DESIGN EXAMPLE

We will now present a simple example demonstrating the use of the ASMD chart and the register transfer representation. We start from the initial specifications of a system and proceed with the development of an appropriate ASMD chart from which the digital hardware is then designed.

The datapath unit is to consist of two *JK* flip-flops E and F, and one four-bit binary counter $A[3:0]$. The individual flip-flops in A are denoted by A_3, A_2, A_1, and A_0, with A_3 holding the most significant bit of the count. A signal, *Start*, initiates the system's operation by clearing the counter A and flip-flop F. At each subsequent clock pulse, the counter is incremented by 1 until the operations stop. Counter bits A_2 and A_3 determine the sequence of operations:

If $A_2 = 0$, E is cleared to 0 and the count continues.

If $A_2 = 1$, E is set to 1; then, if $A_3 = 0$, the count continues, but if $A_3 = 1$, F is set to 1 on the next clock pulse and the system stops counting.

Then, if *Start* = 0, the system remains in the initial state, but if *Start* = 1, the operation cycle repeats.

A block diagram of the system's architecture is shown in Fig. 8.9(a), with (1) the registers of the datapath unit, (2) the external (primary) input signals, (3) the status signals fed back from the datapath unit to the control unit, and (4) the control signals generated by the control unit and input to the datapath unit. Note that the names of the control signals clearly indicate the operations that they cause to be executed in the datapath unit. For example, *clr_A_F* clears registers *A* and *F*. The name of the signal *reset_b* (alternatively, *reset_bar*) indicates that the reset action is active low. The internal details of each unit are not shown.

ASMD Chart

An ASMD chart for the system is shown in Fig. 8.9(b) for asynchronous reset action and in Fig. 8.9(c) for synchronous reset action. The chart shows the state transitions of the controller and the datapath operations associated with those transitions. The chart is not in its final form, for it does not identify the control signals generated by the controller. The nonblocking Verilog operator (<=) is shown instead of the arrow (←) for register transfer operations because we will ultimately use the ASMD chart to write a Verilog description of the system.

When the reset action is synchronous, the transition to the reset state is synchronous with the clock. This transition is shown in the diagram, but *all other synchronous reset paths are omitted for clarity*. The system remains in the reset state, *S_idle*, until *Start* is asserted. When that happens (i.e., *Start* = 1), the state moves to *S_1*. *At the next clock edge*, depending on the values of A_2 and A_3 (decoded in a priority order), the state returns to *S_1* or goes to *S_2*. From *S_2*, it moves unconditionally to *S_idle*, where it awaits another assertion of *Start*.

The edges of the chart represent the state transitions that occur at the active (i.e., synchronizing) edge of the clock (e.g., the rising edge) and are annotated with the register operations that are to occur in the datapath. With *Start* asserted in *S_idle*, the state will transition to *S_1* and the registers *A* and *F* will be cleared. Note that, on the one hand, if a register operation is annotated on the *edge* leaving a state box, the operation occurs unconditionally and will be controlled by a Moore-type signal. For example, register *A* is incremented at every clock edge that occurs while the machine is in the state *S_1*. On the other hand, the register operation setting register *E* annotates the edge leaving the *decision box* for A_2. The signal controlling the operation will be a Mealy-type signal asserted when the system is in state *S_1* and A_2 has the value 1. Likewise, the control signal clearing *A* and *F* is asserted conditionally: The system is in state *S_idle* and *Start* is asserted.

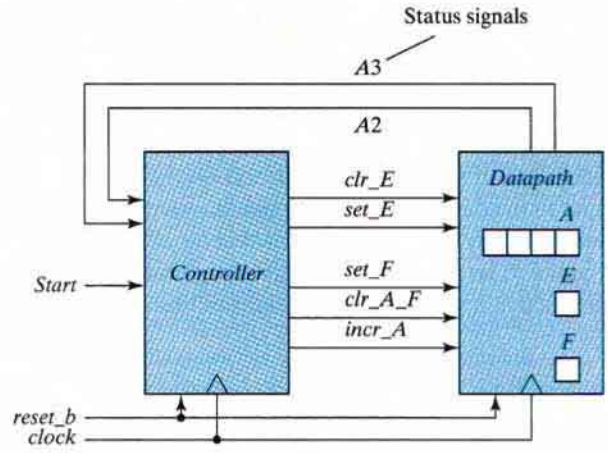
In addition to showing that the counter is incremented in state *S_1*, the annotated paths show that other operations occur conditionally with the same clock edge:

Either *E* is cleared and control stays in state *S_1* ($A_2 = 0$) or

E is set and control stays in state *S_1* ($A_2A_3 = 10$) or

E is set and control goes to state *S_2* ($A_2A_3 = 11$).

When control is in state *S_2*, a Moore-type control signal must be asserted to set flip-flop *F* to 1, and the state returns to *S_idle* at the next active edge of the clock.



(a)

Note: A3 denotes A[3],
A2 denotes A[2],
<= denotes nonblocking assignment
reset_b denotes active-low reset condition

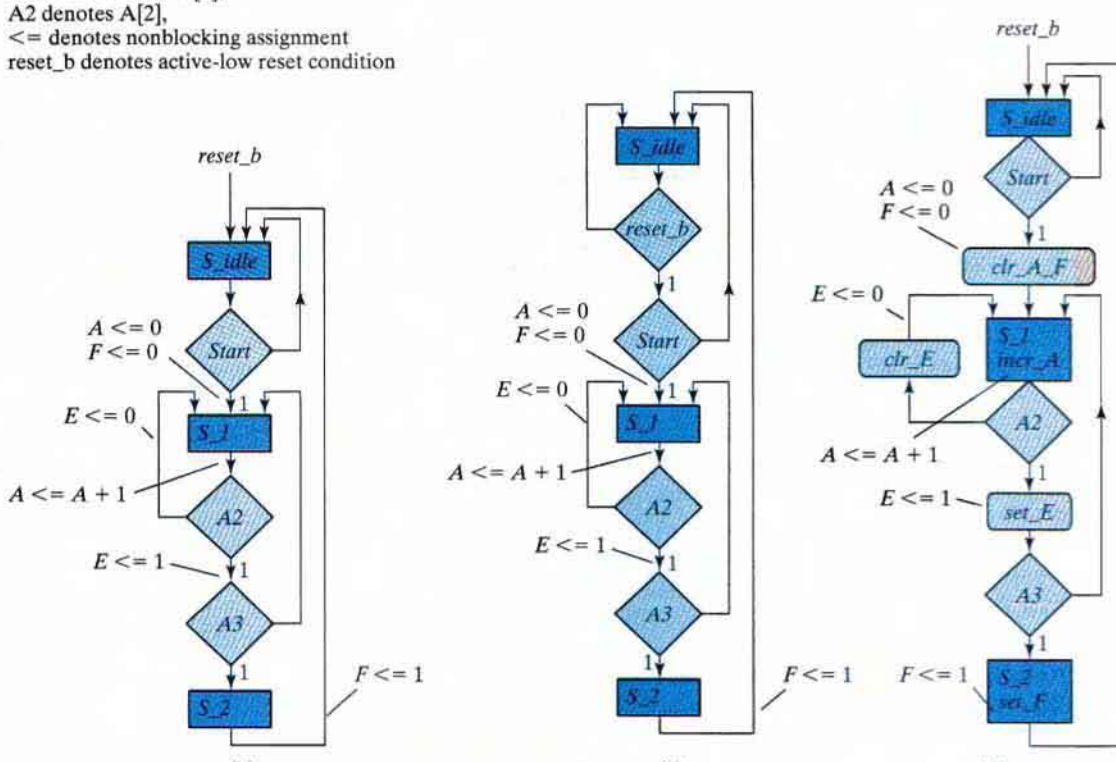


FIGURE 8.9
(a) Block diagram for design example
(b) ASMD chart for controller state transitions, asynchronous reset
(c) ASMD chart for controller state transitions, synchronous reset
(d) ASMD chart for a completely specified controller, asynchronous reset

The third and final step in creating the ASMD chart is to insert conditional boxes for the signals generated by the controller or to insert Moore-type signals in the state boxes, as shown in Fig. 8.9(d). The signal *clr_A_F* is generated conditionally in state *S_idle*, *incr_A* is generated unconditionally in *S_1*, *clr_E* and *set_E* are generated conditionally in *S_1*, and *set_F* is generated unconditionally in *S_2*. The ASM chart has three states and three blocks. The block associated with *S_idle* consists of the state box, one decision box, and one conditional box. The block associated with *S_2* consists of only the state box. In addition to *clock* and *reset_b*, the control logic has one external input, *Start*, and two status inputs, A_2 and A_3.

In this example, we have shown how a verbal (text) description (specification) of a design is translated into an ASMD chart that completely describes the controller for the datapath, indicating the control signals and their associated register operations. This design example does not have a practical application, and in general, depending on the interpretation, the ASMD chart produced by the three-step design process for the controller may be simplified and formulated differently. However, once the ASMD chart is established, the procedure for designing the circuit is straightforward. *In practice, designers use the ASMD chart to write Verilog models of the controller and the datapath and then synthesize a circuit directly from the Verilog description.* We will first design the system manually and then write the HDL description, keeping synthesis as an optional step for those who have access to synthesis tools.

Timing Sequence

Every block in an ASMD chart specifies the signals which control the operations that are to be initiated by one common clock pulse. The control signals specified within the state and conditional boxes in the block are formed while the controller is in the indicated state, and the annotated operations occur in the datapath unit when the state makes a transition along an edge that exits the state. The change from one state to the next is performed in the control logic. In order to appreciate the timing relationship involved, we will list the step-by-step sequence of operations after each clock edge, beginning with an assertion of the signal *Start* until the system returns to the reset (initial) state, *S_idle*.

Table 8.3 shows the binary values of the counter and the two flip-flops after every clock pulse. The table also shows separately the status of A_2 and A_3, as well as the present state of the controller. We start with state *S_1* right after the input signal *Start* has caused the counter and flip-flop *F* to be cleared. We will assume that the machine had been running before it entered *S_idle*, instead of entering it from a reset condition. Therefore, the value of *E* is assumed to be 1, because *E* is set to 1 when the machine enters *S_2*, before moving to *S_idle* (as shown at the bottom of the table), and because *E* does not change during the transition from *S_idle* to *S_1*. The system stays in state *S_1* during the next 13 clock pulses. Each pulse increments the counter and either clears or sets *E*. Note the relationship between the time at which A_2 becomes a 1 and the time at which *E* is set to 1. When $A = (A_3\ A_2\ A_1\ A_0)$ 0011, the next (4th) clock pulse increments the counter to 0100, but that same clock edge sees the value of A_2 as 0, so *E* remains cleared. The next (5th) pulse changes the counter from 0100 to 0101, and because A_2 is equal to 1 *before* the clock pulse arrives, *E* is set to 1. Similarly, *E* is cleared to 0 not when the count goes from 0111 to 1000, but when it goes from 1000 to 1001, which is when A_2 is 0 in the *present* value of the counter.

Table 8.3
Sequence of Operations for Design Example

Counter				Flip-Flops			
A_3	A_2	A_1	A_0	E	F	Conditions	State
0	0	0	0	1	0	$A_2 = 0, A_3 = 0$	*S_1*
0	0	0	1	0	0		
0	0	1	0	0	0		
0	0	1	1	0	0		
0	1	0	0	0	0	$A_2 = 1, A_3 = 0$	
0	1	0	1	1	0		
0	1	1	0	1	0		
0	1	1	1	1	0		
1	0	0	0	1	0	$A_2 = 0, A_3 = 1$	
1	0	0	1	0	0		
1	0	1	0	0	0		
1	0	1	1	0	0		
1	1	0	0	0	0	$A_2 = 1, A_3 = 1$	
1	1	0	1	1	0		*S_2*
1	1	0	1	1	1		*S_idle*

When the count reaches 1100, both A_2 and A_3 are equal to 1. The next clock edge increments A by 1, sets E to 1, and transfers control to state *S_2*. Control stays in *S_2* for only one clock period. The clock edge associated with the path leaving *S_2* sets flip-flop F to 1 and transfers control to state *S_idle*. The system stays in the initial state *S_idle* as long as *Start* is equal to 0.

From an observation of Table 8.3, it may seem that the operations performed on E are delayed by one clock pulse. This is the difference between an ASMD chart and a conventional flowchart. If Fig. 8.9(d) were a conventional flowchart, we would assume that A is first incremented and the incremented value would have been used to check the status of A_2. The operations that are performed in the digital hardware as specified by a block in the ASMD chart occur during the same clock cycle and not in a sequence of operations following each other in time, as is the usual interpretation in a conventional flowchart. Thus, the value of A_2 to be considered in the decision box is taken from the value of the counter in the present state and before it is incremented. This is because the decision box for E belongs with the same block as state *S_1*. The digital circuits in the control unit generate the signals for all the operations specified in the present block *prior to the arrival of the next clock pulse*. The next clock edge executes all the operations in the registers and flip-flops, including the flip-flops in the controller that determine the next state, using the present values of the output signals of the controller. Thus, the signals that control the operations in the datapath unit are formed in the controller in the clock cycle (control state) *preceding* the clock edge at which the operations execute.

Controller and Datapath Hardware Design

The ASMD chart provides all the information needed to design the digital system–the datapath and the controller. The actual boundary between the hardware of the controller and that of the datapath can be arbitrary, but we advocate, first, that the datapath unit contain only the hardware associated with its operations and the logic required, perhaps, to form status signals used by the controller, and, second, that the control unit contain all of the logic required to generate the signals that control the operations of the datapath unit. The requirements for the design of the datapath are indicated by the control signals inside the state and conditional boxes of the ASMD chart and are specified by the annotations of the edges indicating datapath operations. The control logic is determined from the decision boxes and the required state transitions. The hardware configuration of the datapath and controller is shown in Fig. 8.10.

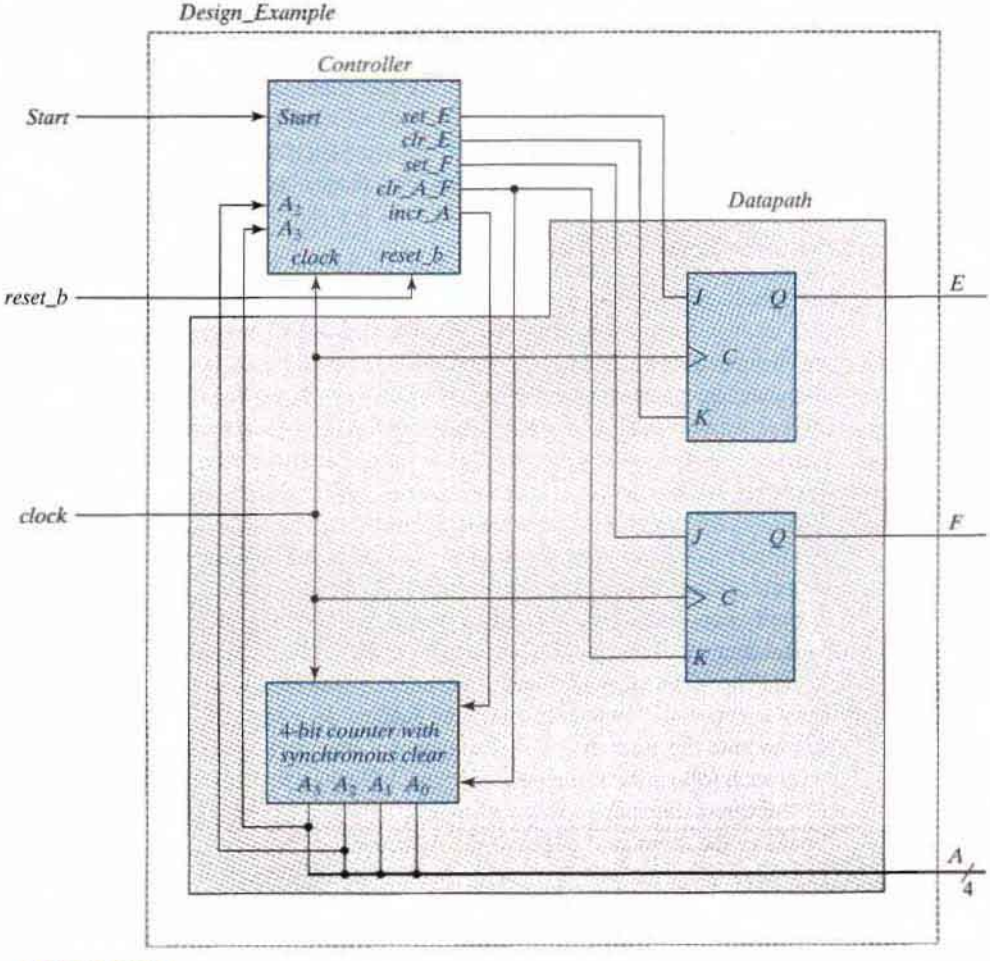


FIGURE 8.10
Datapath and controller for design example

Note that the input signals of the control unit are the external (primary) inputs (*Start*, *reset_b*, and *clock*) and the status signals from the datapath (A_2 and A_3). The status signals provide information about the present condition of the datapath. This information, together with the primary inputs and information about the present state of the machine, is used to form the output of the controller and the value of the next state. The outputs of the controller are inputs to the datapath and determine which operations will be executed when the clock undergoes a transition. Note, also, that the state of the control is not an output of the control unit, even if the entire design is encapsulated in only one module.

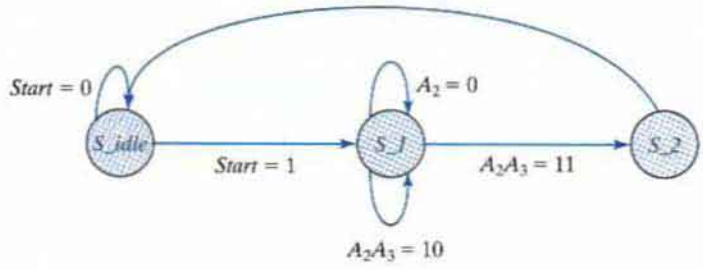
The control subsystem is shown in Fig. 8.10 with only its inputs and outputs, with names matching those of the ASMD chart. The detailed design of the controller is considered subsequently. The datapath unit consists of a four-bit binary counter and two *JK* flip-flops. The counter is similar to the one shown in Fig. 6.12, except that additional internal gates are required for the synchronous clear operation. The counter is incremented with every clock pulse when the controller state is *S_1*. It is cleared only when control is at state *S_idle* and *Start* is equal to 1. The logic for the signal *clr_A_F* will be included in the controller and requires an AND gate to guarantee that both conditions are present. Similarly, we can anticipate that the controller will use AND gates to form signals *set_E* and *clr_E*. Depending on whether the controller is in state *S_1* and whether A_2 is asserted, *set_F* controls flip-flop *F* and is asserted unconditionally during state *S_2*. Note that all flip-flops and registers, including the flip-flops in the control unit, use a common clock.

Register Transfer Representation

A digital system is represented at the register transfer level by specifying the registers in the system, the operations performed, and the control sequence. The register operations and control information can be specified with an ASMD chart. It is convenient to separate the control logic and the register operations for the datapath. The ASMD chart provides this separation and a clear sequence of steps to design a controller for a datapath. The control information and register transfer operations can also be represented separately, as shown in Fig. 8.11. The state diagram specifies the control sequence, and the register operations are represented by the register transfer notation introduced in Section 8.2. The state transition and the signal controlling the register operation are shown with the operation. This representation is an alternative to the representation of the system described in the ASMD chart of Fig. 8.9(d). Only the ASMD chart is really needed, but the state diagram for the controller is an alternative representation that is useful in manual design. The information for the state diagram is taken directly from the ASMD chart. The state names are specified in each state box. The conditions that cause a change of state are specified inside the diamond-shaped decision boxes of the ASMD chart and are used to annotate the state diagram. The directed lines between states and the condition associated with each follow the same path as in the ASMD chart. The register transfer operations for each of the three states are listed following the name of the state. They are taken from the state boxes or the annotated edges of the ASMD chart.

State Table

The state diagram can be converted into a state table from which the sequential circuit of the controller can be designed. First, we must assign binary values to each state in the ASMD chart. For *n* flip-flops in the control sequential circuit, the ASMD chart can accommodate up



(a)

S_idle ⟶ S_1, clr_A_F: $A \leftarrow 0, F \leftarrow 0$

S_1 ⟶ S_1, incr_A: $A \leftarrow A + 1$

if ($A_2 = 1$) then set_E: $E \leftarrow 1$

if ($A_2 = 0$) then clr_E: $E \leftarrow 0$

S_2 ⟶ S_idle, set_F: $F \leftarrow 1$

(b)

FIGURE 8.11
Register transfer-level description of design example

to 2^n states. A chart with 3 or 4 states requires a sequential circuit with two flip-flops. With 5 to 8 states, there is a need for three flip-flops. Each combination of flip-flop values represents a binary number for one of the states.

A *state table* for a controller is a list of present states and inputs and their corresponding next states and outputs. In most cases, there are many don't-care input conditions that must be included, so it is advisable to arrange the state table to take those conditions into consideration. We assign the following binary values to the three states: *S_idle* = 00, *S_1* = 01, and *S_2* = 11. Binary state 10 is not used and will be treated as a don't-care condition. The state table corresponding to the state diagram is shown in Table 8.4. Two flip-flops are needed, and they are

Table 8.4
State Table for the Controller of Fig. 8.10

	Present State		Inputs			Next State		Outputs				
Present-State Symbol	G_1	G_0	*Start*	A_2	A_3	G_1	G_0	*set_E*	*clr_E*	*set_F*	*clr_A_F*	*incr_A*
S_idle	0	0	0	X	X	0	0	0	0	0	0	0
S_idle	0	0	1	X	X	0	1	0	0	0	1	0
S_1	0	1	X	0	X	0	1	0	1	0	0	1
S_1	0	1	X	1	0	0	1	1	0	0	0	1
S_1	0	1	X	1	1	1	1	1	0	0	0	1
S_2	1	1	X	X	X	0	0	0	0	1	0	0

labeled G_1 and G_0. There are three inputs and five outputs. The inputs are taken from the conditions in the decision boxes. The outputs depend on the inputs and the present state of the control. Note that there is a row in the table for each possible transition between states. Initial state 00 goes to state 01 or stays in 00, depending on the value of input *Start*. The other two inputs are marked with don't-care X's, as they do not determine the next state in this case. While the system is in binary state 00 with $Start = 1$, the control unit provides an output labeled *clr_A_F* to initiate the required register operations. The transition from binary state 01 depends on inputs A_2 and A_3. The system goes to binary state 11 only if $A_2A_3 = 11$; otherwise, it remains in binary state 01. Finally, binary state 11 goes to 00 independently of the input variables.

Control Logic

The procedure for designing a sequential circuit starting from a state table was presented in Chapter 5. If this procedure is applied to Table 8.4, we need to use five-variable maps to simplify the input equations. This is because there are five variables listed under the present-state and input columns of the table. Instead of using maps to simplify the input equations, we can obtain them directly from the state table by inspection. To design the sequential circuit of the controller with D flip-flops, it is necessary to go over the next-state columns in the state table and derive all the conditions that must set each flip-flop to 1. From Table 8.4, we note that the next-state column of G_1 has a single 1 in the fifth row. The D input of flip-flop G_1 must be equal to 1 during present state *S_1* when both inputs A_2 and A_3 are equal to 1. This condition is expressed with the D flip-flop input equation

$$D_{G1} = S\_1A_2A_3$$

Similarly, the next-state column of G_0 has four 1's, and the condition for setting this flip-flop is

$$D_{G0} = Start\ S\_idle + S\_1$$

To derive the five output functions, we can exploit the fact that binary state 10 is not used, which simplifies the equation for *clr_A_F* and enables us to obtain the following simplified set of output equations:

$$\begin{aligned} set\_E &= S\_1A_2 \\ clr\_E &= S\_1A_2' \\ set\_F &= S\_2 \\ clr\_A\_F &= Start\ S\_idle \\ incr\_A &= S\_1 \end{aligned}$$

The logic diagram showing the internal detail of the controller of Fig. 8.10 is drawn in Fig. 8.12. Note that although we derived the output equations from Table 8.4, they can also be obtained directly by inspection of Fig. 8.9(d). This simple example illustrates the manual design of a controller for a datapath, using an ASMD chart as a starting point. The fact that synthesis tools automatically execute these steps should be appreciated.

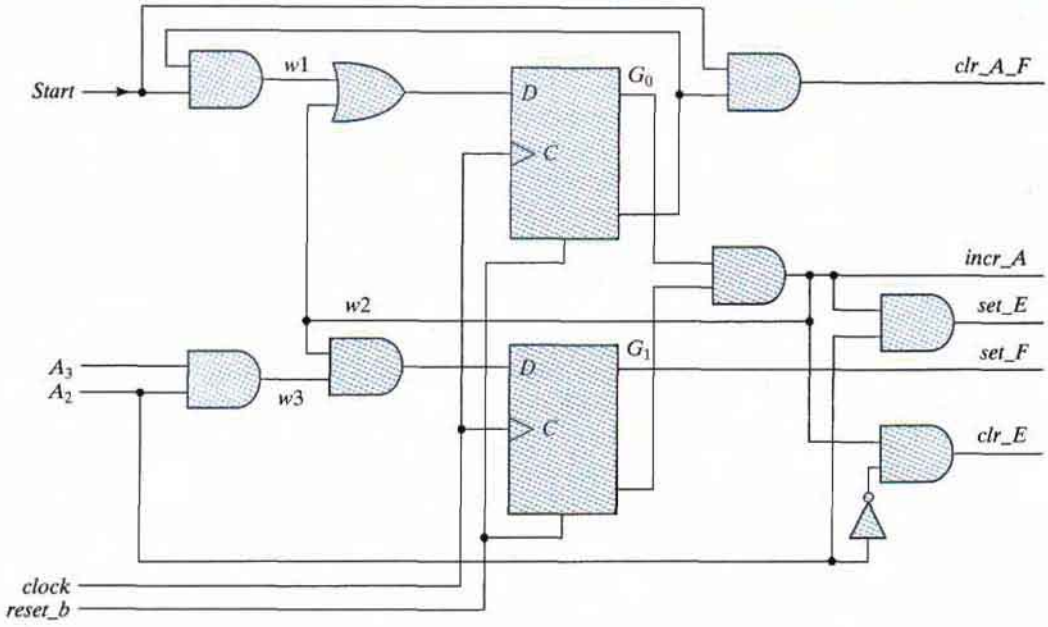


FIGURE 8.12
Logic diagram of the control unit for Fig. 8.10

8.6 HDL DESCRIPTION OF DESIGN EXAMPLE

In previous chapters, we gave examples of HDL descriptions of combinational circuits, sequential circuits, and various standard components such as multiplexers, counters, and registers. We are now in a position to incorporate these components into the description of a specific design. As mentioned previously, a design can be described either at the structural or behavioral level. Behavioral descriptions may be classified as being either at the register transfer level or at an abstract algorithmic level. Consequently, we now consider three levels of design: structural description, RTL description, and algorithmic-based behavioral description.

The *structural* description is the lowest and most detailed level. The digital system is specified in terms of the physical components and their interconnection. The various components may include gates, flip-flops, and standard circuits such as multiplexers and counters. The design is hierarchically decomposed into functional units, and each unit is described by an HDL module. A top-level module combines the entire system by instantiating all the lower level modules. This style of description requires that the designer have sufficient experience not only to understand the functionality of the system, but also to implement it by selecting and connecting other functional elements.

The *RTL* description specifies the digital system in terms of the registers, the operations performed, and the control that sequences the operations. This type of description simplifies the design process because it consists of procedural statements that determine the relationship between the various operations of the design without reference to any specific structure. The

RTL description implies a certain hardware configuration among the registers, allowing the designer to create a design that can be synthesized automatically, rather than manually, into standard digital components.

The *algorithmic-based behavioral* description is the most abstract level, describing the function of the design in a procedural, algorithmic form similar to a programming language. It does not provide any detail on how the design is to be implemented with hardware. The algorithmic-based behavioral description is most appropriate for simulating complex systems in order to verify design ideas and explore tradeoffs. Descriptions at this level are accessible to nontechnical users who understand programming languages. Some algorithms, however, might not be synthesizable.

We will now illustrate the RTL and structural descriptions by using the design example of the previous section. The design example will serve as a model of coding style for future examples and will exploit alternative syntax options supported by revisions to the Verilog language. (An algorithmic-based description is illustrated in Section 8.9.)

RTL Description

The block diagram in Fig. 8.10 describes the design example. An HDL description of the design example can be written as a single RTL description in a Verilog module or as a top-level module having instantiations of separate modules for the controller and the datapath. The former option simply ignores the boundaries between the functional units; the modules in the latter option establish the boundaries shown in Fig. 8.9(a) and Fig. 8.10. We advocate the second option, because, in general, it distinguishes more clearly between the controller and the datapath. This choice also allows one to easily substitute alternative controllers for a given datapath (e.g., replace an RTL model by a structural model). The RTL description of the design example is shown in HDL Example 8.2. The description follows the ASMD chart of Fig. 8.9(d), which contains a complete description of the controller, the datapath, and the interface between them (i.e., the outputs of the controller and the status signals). Likewise, our description has three modules: *Design_Example_RTL*, *Controller_RTL*, and *Datapath_RTL*. The descriptions of the controller and the datapath units are taken directly from Fig. 8.9(d). *Design_Example_RTL* declares the input and output ports of the module and instantiates *Controller_RTL* and *Datapath_RTL*. At this stage of the description, it is important to remember to declare *A* as a vector. Failure to do so will produce *port mismatch* errors when the descriptions are compiled together. Note that the status signals *A[2]* and *A[3]* are passed to the controller. The primary (external) inputs to the controller are *Start*, *clock* (to synchronize the system), and *reset_b*. The active-low input signal *reset_b* is needed to initialize the state of the controller to *S_idle*. Without that signal, the controller could not be placed in a known initial state.

The controller is described by three cyclic (**always**) behaviors. An edge-sensitive behavior updates the state at the positive edge of the clock, depending on whether a reset condition is asserted. Two level-sensitive behaviors describe the combinational logic for the next state and the outputs of the controller, as specified by the ASMD chart. Notice that the description includes default assignments to all of the outputs (e.g., *set_E* = 0). This approach allows the code of the **case** logic to be simplified by expressing only explicit assertions of the variables (i.e., values are assigned by exception). The approach also ensures that every path through the assignment logic assigns a value to every variable. Thus, a synthesis tool will interpret the

logic to be combinational; failure to assign a value to every variable on every path of logic implies the need for a transparent latch (memory) to implement the logic. Synthesis tools will provide the latch, wasting silicon area.

The three states of the controller are given symbolic names and are encoded into binary values. Only three of the possible two-bit patterns are used, so the **case** statement for the next-state logic includes a **default** assignment to handle the possibility that one of the three assigned codes is not detected. The alternative is to allow the hardware to make an arbitrary assignment to the next state (*next_state = 2'bx;*). Also, the first statement of the next-state logic assigns *next_state = S_idle* to guarantee that the next state is assigned in every thread of the logic. This is a precaution against accidentally forgetting to make an assignment to the next state in every thread of the logic, with the result that the description implies the need for memory, which a synthesis tool will implement with a transparent latch.

The description of *Datapath_RTL* is written by testing for an assertion of each control signal from *Controller_RTL*. The register transfer operations are displayed in the ASMD chart (Fig. 8.9(d)). Note that nonblocking assignments are used (with symbol <=) for the register transfer operations. This ensures that the register operations and state transitions are concurrent, a feature that is especially crucial during control state *S_1*. In this state, *A* is incremented by 1 and the value of *A2* (*A[2]*) is checked to determine the operation to execute at register *E* at the next clock. To accomplish a valid synchronous design, it is necessary to ensure that *A[2]* is checked before *A* is incremented. If blocking assignments were used, one would have to place the two statements that check *E* first and the *A* statement that increments last. However, by using nonblocking assignments, we accomplish the required synchronization without being concerned about the order in which the statements are listed. The counter *A* in *Datapath_RTL* is cleared synchronously because *clr_A_F* is synchronized to the clock.

The cyclic behaviors of the controller and the datapath interact in a chain reaction: At the active edge of the clock, the state and datapath registers are updated. A change in the state, a primary input, or a status input causes the level-sensitive behaviors of the controller to update the value of the next state and the outputs. The updated values are used at the next active edge of the clock to determine the state transition and the updates of the datapath.

Note that the manual method of design developed (1) a block diagram (Fig. 8.9(a)) showing the interface between the datapath and the controller, (2) an ASMD chart for the system (Fig. 8.9(d)), (3) the logic equations for the inputs to the flip-flops of the controller, and (4) a circuit that implements the controller (Fig. 8.12). In contrast, an RTL model describes the state transitions of the controller and the operations of the datapath as a step towards automatically synthesizing the circuit that implements them. The descriptions of the datapath and controller are derived directly from the ASMD chart in both cases.

HDL Example 8.2

```
// RTL description of design example (see Fig. 8.11)
module Design_Example_RTL (A, E, F, Start, clock, reset_b);
  // Specify ports of the top-level module of the design
  // See block diagram, Fig. 8.10
  output [3: 0] A;
  output        E, F;
```

```
 input        Start, clock, reset_b;
 // Instantiate controller and datapath units
 Controller_RTL M0 (set_E, clr_E, set_F, clr_A_F, incr_A, A[2], A[3], Start, clock,
  reset_b );
 Datapath_RTL  M1 (A, E, F, set_E, clr_E, set_F, clr_A_F, incr_A, clock);
endmodule
module Controller_RTL (set_E, clr_E, set_F, clr_A_F, incr_A, A2, A3, Start, clock,
 reset_b);
 output reg      set_E, clr_E, set_F, clr_A_F, incr_A;
 input           Start, A2, A3, clock, reset_b;
 reg [1: 0]      state, next_state;
 parameter       S_idle = 2'b00, S_1 = 2'b01, S_2 = 2'b11;     // State codes
 always @ (posedge clock or negedge reset_b)      // State transitions (edge sensitive)
  if (reset_b == 0) state <= S_idle;
  else state <= next_state;
 // Code next-state logic directly from ASMD chart (Fig. 8.9d)
 always @ (state, Start, A2, A3) begin                    // Next-state logic (level sensitive)
  next_state = S_idle;
  case (state)
    S_idle:              if (Start) next_state = S_1; else next_state = S_idle;
    S_1:                 if (A2 & A3) next_state = S_2; else next_state = S_1;
    S_2:                 next_state = S_idle;
    default:             next_state = S_idle;
  endcase
 end
 // Code output logic directly from ASMD chart (Fig. 8.9d)
 always @ (state, Start, A2) begin
  set_E   = 0;   // default assignments; assign by exception
  clr_E   = 0;
  set_F   = 0;
  clr_A_F = 0;
  incr_A  = 0;
  case (state)
   S_idle:               if (Start) clr_A_F = 1;
   S_1:                  begin incr_A = 1; if (A2) set_E = 1; else clr_E = 1; end
   S_2:                  set_F = 1;
  endcase
 end
endmodule
module  Datapath_RTL  (A, E, F, set_E, clr_E, set_F, clr_A_F, incr_A, clock);
 output reg [3: 0] A;                                     // register for counter
 output reg      E, F;                                    // flags
 input           set_E, clr_E, set_F, clr_A_F, incr_A, clock;
 // Code register transfer operations directly from ASMD chart (Fig. 8.9(d))
```

```
always @ (posedge clock) begin
  if (set_E)                    E <= 1;
  if (clr_E)                    E <= 0;
  if (set_F)                    F <= 1;
  if (clr_A_F)                  begin A <= 0; F <= 0; end
  if (incr_A)                   A <= A + 1;
 end
endmodule
```

Testing the Design Description

The sequence of operations for the design example was investigated in the previous section. Table 8.3 shows the values of E and F while register A is incremented. It is instructive to devise a test that checks the circuit to verify the validity of the HDL description. The test bench in HDL Example 8.3 provides such a module. (The procedure for writing test benches is explained in Section 4.12.) The test module generates signals for *Start*, *clock*, and *reset_b*, and checks the results obtained from registers A, E, and F. Initially, the *reset_b* signal is set to 0 to initialize the controller, and *Start* and *clock* are set to 0. At time $t = 5$, the *reset_b* signal is deasserted by setting it to 1, the *Start* input is asserted by setting it to 1, and the clock is then repeated for 16 cycles. The **$monitor** statement displays the values of A, E, and F every 10 ns. The output of the simulation is listed in the example under the simulation log. Initially, at time $t = 0$, the values of the registers are unknown, so they are marked with the symbol **x**. The first positive clock transition, at time = 10, clears A and F, but does not affect E, so E is unknown at this time. The rest of the table is identical to Table 8.3. Note that since *Start* is still equal to 1 at time = 160, the last entry in the table shows that A and F are cleared to 0, and E does not change and remains at 1. This occurs during the second transition, from *S_idle* to *S_1*.

HDL Example 8.3

```
// Test bench for design example
module t_Design_Example_RTL;
 reg       Start, clock, reset_b;
 wire [3: 0] A;
 wire      E, F;
 // Instantiate design example
 Design_Example_RTL M0 (A, E, F, Start, clock, reset_b);
 // Describe stimulus waveforms
 initial #500 $finish;              // Stopwatch
 initial
  begin
   reset_b = 0;
   Start = 0;
   clock = 0;
   #5 reset_b = 1; Start = 1;
   repeat (32)
```

```
    begin
      #5 clock = ~ clock;  // Clock generator
    end
  end
 initial
  $monitor ("A = %b E = %b F = %b time = %0d", A, E, F, $time);
endmodule
```

Simulation log:

```
A = xxxx  E = x F = x time = 0
A = 0000 E = x F = 0 time = 10
A = 0001 E = 0 F = 0 time = 20
A = 0010 E = 0 F = 0 time = 30
A = 0011 E = 0 F = 0 time = 40
A = 0100 E = 0 F = 0 time = 50
A = 0101 E = 1 F = 0 time = 60
A = 0110 E = 1 F = 0 time = 70
A = 0111 E = 1 F = 0 time = 80
A = 1000 E = 1 F = 0 time = 90
A = 1001 E = 0 F = 0 time = 100
A = 1010 E = 0 F = 0 time = 110
A = 1011 E = 0 F = 0 time = 120
A = 1100 E = 0 F = 0 time = 130
A = 1101 E = 1 F = 0 time = 140
A = 1101 E = 1 F = 1 time = 150
A = 0000 E = 1 F = 0 time = 160
```

Waveforms produced by a simulation of *Design_Example_RTL* with the test bench are shown in Fig. 8.13. Numerical values are shown in hexadecimal format. The results are annotated to call attention to the relationship between a control signal and the operation that it causes to execute. For example, the controller asserts *set_E* for one clock cycle *before* the clock edge at which *E* is set to 1. Likewise, *set_F* asserts during the clock cycle before the edge at which *F* is set to 1. Also, *clr_A_F* is formed in the cycle before *A* and *F* are cleared. A more thorough verification of *Design_Example_RTL* would confirm that the machine recovers from a reset on the fly (i.e., a reset that is asserted randomly after the machine is operating). Note that the signals in the output of the simulation have been listed in groups showing (1) *clock* and *reset_b*, (2) *Start* and the status inputs, (3) the state, (4) the control signals, and (5) the datapath registers. It is strongly recommended that the state always be displayed, because this information is essential for verifying that the machine is operating correctly and for debugging its description when it is not. For the chosen binary state code, $S\_idle = 00_2 = 0_H$, $S\_1 = 01_2 = 1_H$, and $S\_2 = 11_2 = 3_H$.

Structural Description

The RTL description of a design consists of procedural statements that determine the functional behavior of the digital circuit. This type of description can be compiled by HDL synthesis tools, from which it is possible to obtain the equivalent gate-level circuit of the design. It is also

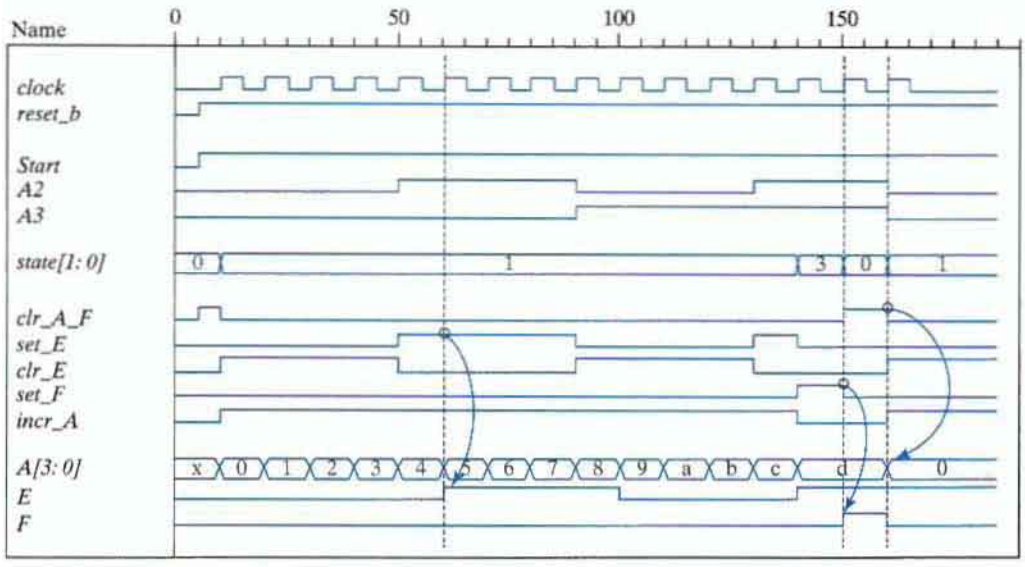


FIGURE 8.13
Simulation results for design example

possible to describe the design by its structure rather than its function. A structural description of a design consists of instantiations of components that define the circuit elements and their interconnections. In this regard, a structural description is equivalent to a schematic diagram or a block diagram of the circuit. Contemporary design practice relies heavily on RTL descriptions, but we will present a structural description here to contrast the two approaches.

For convenience, the circuit is again decomposed into two parts: the controller and the datapath. The block diagram of Fig. 8.10 shows the high-level partition between these units, and Fig. 8.12 provides additional underlying structural details of the controller. The structure of the datapath is evident in Fig. 8.10 and consists of the flip-flops and the four-bit counter with synchronous clear. The top level of the Verilog description replaces *Design_Example_RTL*, *Controller_RTL*, and *Datapath_RTL* by *Design_Example_STR*, *Controller_STR*, and *Datapath_STR*, respectively. The descriptions of *Controller_STR* and *Datapath_STR* will be structural.

HDL Example 8.4 presents the structural description of the design example. It consists of a nested hierarchy of modules and gates describing (1) the top-level module, *Design_Example_STR*, (2) the modules describing the controller and the datapath, (3) the modules describing the flip-flops and counters, and (4) gates implementing the logic of the controller. For simplicity, the counter and flip-flops are described by RTL models.

The top-level module (see Fig. 8.10) encapsulates the entire design by (1) instantiating the controller and the datapath modules, (2) declaring the primary (external) input signals, (3) declaring the output signals, (4) declaring the control signals generated by the controller and connected to the datapath unit, and (5) declaring the status signals generated by the datapath unit and connected to the controller. The port list is identical to the list used in the RTL description. The outputs are declared as **wire** type here because they serve merely to connect the outputs

of the datapath module to the outputs of the top-level module, with their logic value being determined within the datapath module.

The control module describes the circuit of Fig. 8.12. The outputs of the two flip-flops *G1* and *G0* are declared as **wire** data type. *G1* and *G0* cannot be declared as **reg** data type because they are outputs of an instantiated *D* flip-flop. *DG1* and *DG0* are undeclared identifiers, i.e., implicit wires. The name of a variable is local to the module or procedural block in which it is declared. Nets may not be declared within a procedural block (e.g., **begin** ... **end**). The rule to remember is that a variable must be a declared register type (e.g., **reg**) if and only if its value is assigned by a procedural statement (i.e., a blocking or nonblocking assignment statement within a procedural block in cyclic or single-pass behavior or in the output of a sequential UDP). The instantiated gates specify the combinational part of the circuit. There are two flip-flop input equations and three output equations. The outputs of the flip-flops *G1* and *G0* and the input equations *DG1* and *DG0* replace output *Q* and input *D* in the instantiated flip-flops. The *D* flip-flop is then described in the next module. The structure of the datapath unit has direct inputs to the *JK* flip-flops. Note the correspondence between the modules of the HDL description and the structures in Figs. 8.9, 8.10, and 8.12.

HDL Example 8.4

```
// Structural description of design example (Figs. 8.9(a), 8.12)
module Design_Example_STR
 ( output     [3: 0]   A,                                  // V 2001 port syntax
   output              E, F,
   input               Start, clock, reset_b
 );

 Controller_STR M0 (clr_A_F, set_E, clr_E, set_F, incr_A, Start, A[2], A[3], clock,
   reset_b );
 Datapath_STR  M1 (A, E, F, clr_A_F, set_E, clr_E, set_F, incr_A, clock);
endmodule

module Controller_STR
( output clr_A_F, set_E, clr_E, set_F, incr_A,
  input  Start, A2, A3, clock, reset_b
);

 wire                G0, G1;
 parameter           S_idle = 2'b00, S_1 = 2'b01, S_2 = 2'b11;
 wire                w1, w2, w3;

 not (G0_b, G0);
 not (G1_b, G1);
 buf (incr_A, w2);
 buf (set_F, G1);
 not (A2_b, A2);
```

```
  or  (D_G0, w1, w2);
  and (w1, Start, G0_b);
  and (clr_A_F, G0_b, Start);
  and (w2, G0, G1_b);
  and (set_E, w2, A2);
  and (clr_E, w2, A2_b);
  and (D_G1, w3, w2);
  and (w3, A2, A3);
  D_flip_flop_AR M0 (G0, D_G0, clock, reset_b);
  D_flip_flop_AR M1 (G1, D_G1, clock, reset_b);
endmodule

// datapath unit

module Datapath_STR
( output [3: 0] A,
  output        E, F,
  input         clr_A_F, set_E, clr_E, set_F, incr_A, clock
);

  JK_flip_flop_2 M0 (E, E_b, set_E, clr_E, clock);
  JK_flip_flop_2 M1 (F, F_b, set_F, clr_A_F, clock);
  Counter_4      M2 (A, incr_A, clr_A_F, clock);
endmodule

// Counter with synchronous clear

module Counter_4 (output reg [3: 0] A, input incr, clear, clock);
  always @ (posedge clock)
    if (clear)     A <= 0; else if (incr) A <= A + 1;
endmodule

module D_flip_flop_AR (Q, D, CLK, RST);
  output        Q;
  input         D, CLK, RST;
  reg           Q;

  always @ (posedge CLK, negedge RST)
    if (RST == 0) Q <= 1'b0;
    else Q <= D;
endmodule

// Description of JK flip-flop

module JK_flip_flop_2 (Q, Q_not, J, K, CLK);
  output        Q, Q_not;
```

```
 input        J, K, CLK;
 reg          Q;
 assign       Q_not = ~Q;
 always @ (posedge CLK)
  case ({J, K})
    2'b00:     Q <= Q;
    2'b01:     Q <= 1'b0;
    2'b10:     Q <= 1'b1;
    2'b11:     Q <= ~Q;
  endcase
endmodule

module t_Design_Example_STR;
 reg Start, clock, reset_b;
 wire [3: 0] A;
 wire        E, F;

 // Instantiate design example

 Design_Example_STR M0 (A, E, F, Start, clock, reset_b);

 // Describe stimulus waveforms

 initial #500 $finish;              // Stopwatch
 initial
  begin
    reset_b = 0;
    Start = 0;
    clock = 0;
    #5 reset_b = 1; Start = 1;
    repeat (32)
      begin
        #5 clock = ~ clock;         // Clock generator
      end
  end
 initial
  $monitor ("A = %b E = %b F = %b time = %0d", A, E, F, $time);
endmodule
```

The structural description was tested with the test bench that verified the RTL description to produce the results shown in Fig. 8.13. The only change necessary is the replacement of the instantiation of the example from *Design_Example_RTL* to *Design_Example_STR*. The simulation results for *Design_Example_STR* matched those for *Design_Example_RTL*. However, a comparison of the two descriptions indicates that the RTL style is easier to write and will lead to results faster if synthesis tools are available to automatically synthesize the registers, the combinational logic, and their interconnections.

8.7 SEQUENTIAL BINARY MULTIPLIER

This section introduces a second design example. It presents a hardware algorithm for binary multiplication, proposes the register configuration for its implementation, and then shows how to use an ASMD chart to design datapath and its controller.

The system we will examine multiplies two unsigned binary numbers. The hardware algorithm that was developed in Section 4.7 to execute multiplication resulted in a combinational circuit multiplier with many adders and AND gates, requiring a large area of silicon for the implementation of the algorithm as an integrated circuit. In contrast, in this section, a more efficient hardware algorithm results in a sequential multiplier that uses only one adder and a shift register. The savings in hardware and silicon area come about from a trade-off in the space (hardware)–time domain. A parallel adder uses more hardware, but forms its result in one cycle of the clock; a sequential adder uses less hardware, but takes multiple clock cycles to form its result.

The multiplication of two binary numbers is done with paper and pencil by successive (i.e., sequential) additions and shifting. The process is best illustrated with a numerical example. Let us multiply the two binary numbers 10111 and 10011:

23	10111	multiplicand
19	10011	multiplier
	10111	
	10111	
	00000	
	00000	
	10111	
437	110110101	product

The process consists of successively adding and shifting copies of the multiplicand. Successive bits of the multiplier are examined, least significant bit first. If the multiplier bit is 1, the multiplicand is copied down; otherwise, 0's are copied down. The numbers copied in successive lines are shifted one position to the left from the previous number. Finally, the numbers are added and their sum forms the product. The product obtained from the multiplication of two binary numbers of n bits each can have up to $2n$ bits. It is apparent that the operations of addition and shifting are executed by the algorithm.

When the multiplication process is implemented with digital hardware, it is convenient to change the process slightly. First, we note that, in the context of synthesizing a sequential machine, the add-and-shift algorithm for binary multiplication can be executed in a single clock cycle or over multiple clock cycles. On the one hand, a choice to form the product in the time span of a single clock cycle will synthesize the circuit of a parallel multiplier like the one discussed in Section 4.7. On the other hand, an RTL model of the algorithm adds shifted copies of the multiplicand to an accumulated partial product. The values of the multiplier, multiplicand, and partial product are stored in registers, and the operations of shifting and adding their contents are executed under the control of a state machine. Among the many possibilities for distributing the effort of multiplication over multiple clock cycles, we will consider that in which only one partial product is formed and accumulated in a single cycle of the clock. (One alternative would be to use additional hardware

to form and accumulate two partial products in a clock cycle, but this would require more logic gates and either faster circuits or a slower clock.) Instead of providing digital circuits to store and add simultaneously as many binary numbers as there are 1's in the multiplier, it is less expensive to provide only the hardware needed to sum two binary numbers and accumulate the partial products in a register. Second, instead of shifting the multiplicand to the left, the partial product being formed is shifted to the right. This leaves the partial product and the multiplicand in the required relative positions. Third, when the corresponding bit of the multiplier is 0, there is no need to add all 0's to the partial product, since doing so will not alter its resulting value.

Register Configuration

A block diagram for the sequential binary multiplier is shown in Fig. 8.14(a), and the register configuration of the datapath is shown in Fig. 8.14(b). The multiplicand is stored in register *B*,

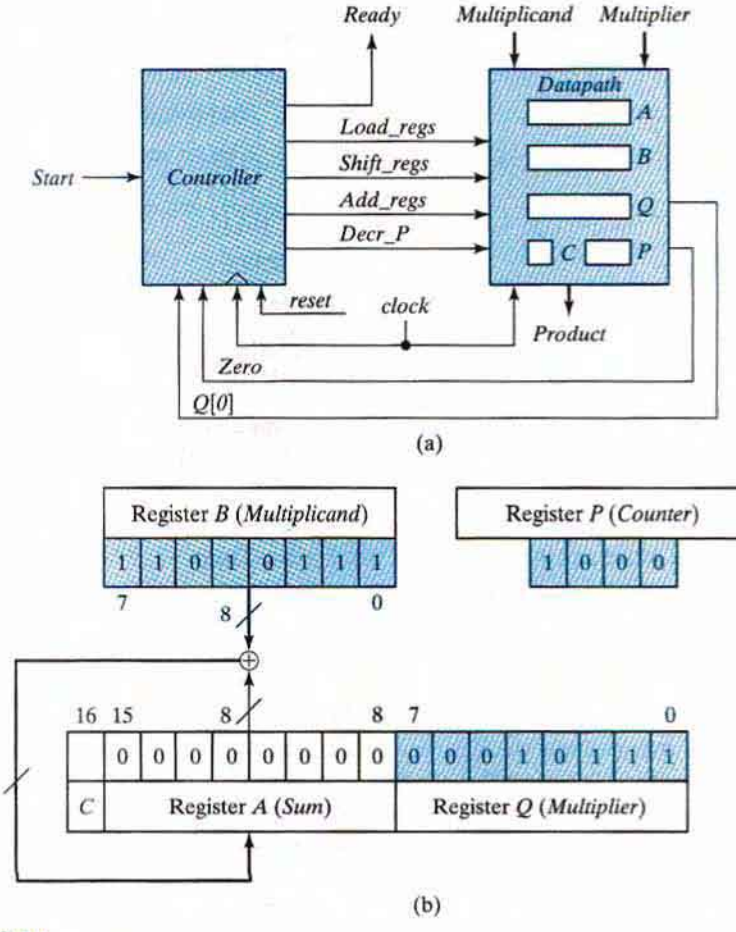


FIGURE 8.14
(a) Block diagram and (b) datapath of a binary multiplier

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

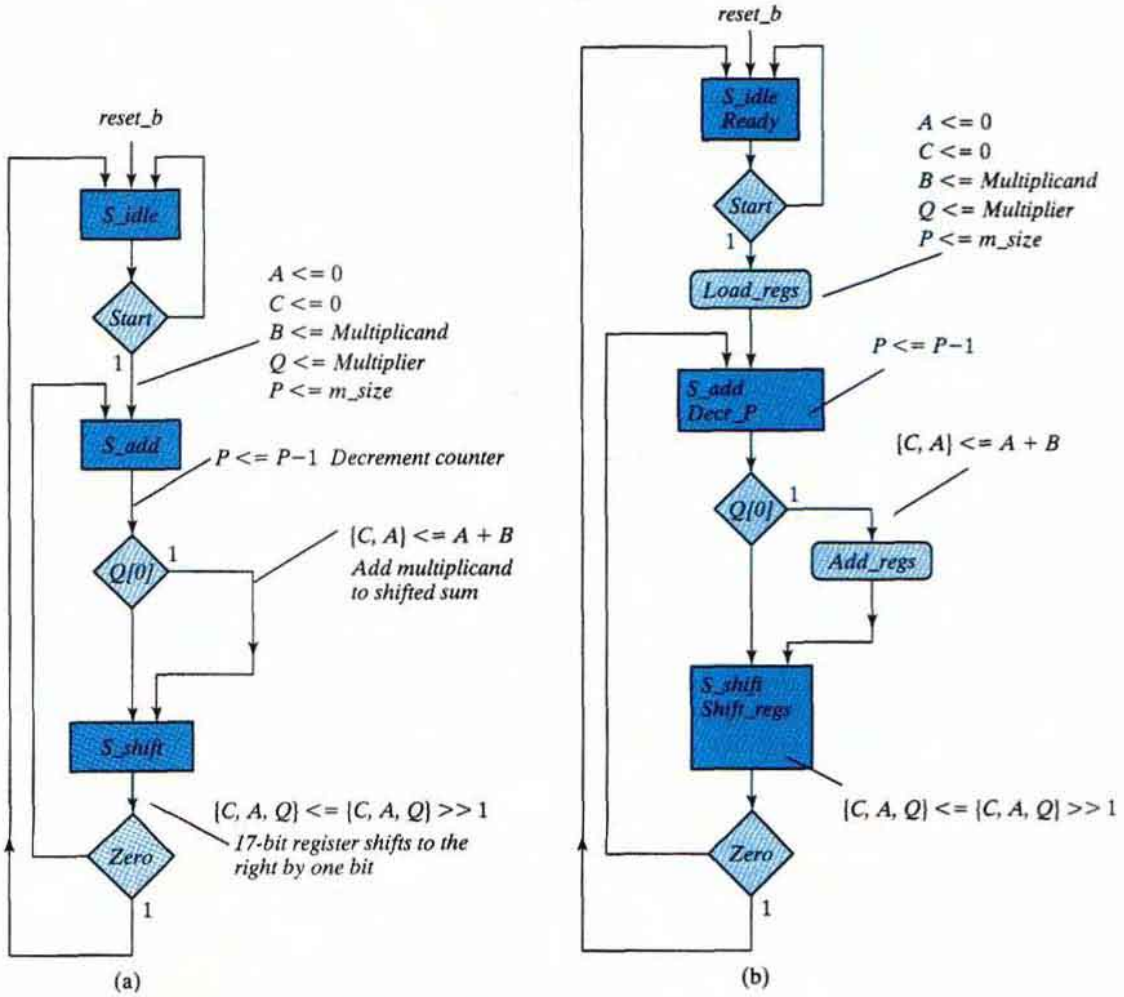


FIGURE 8.15
ASMD chart for binary multiplier

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:

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

In terms of individual register symbols, the shift operation can be described by the following register operations:

$$A \leftarrow \text{shr } A,\ A_{n-1} \leftarrow C$$
$$Q \leftarrow \text{shr } Q,\ Q_{n-1} \leftarrow A_0$$
$$C \leftarrow 0$$

Both registers A and Q are shifted right. The leftmost bit of A, designated by A_{n-1}, receives the carry from C. The leftmost bit of Q, or Q_{n-1}, receives the bit from the rightmost position of A in A_0, and C is reset to 0. In essence, this is a long shift of the composite register CAQ with 0 inserted into the serial input, which is at C.

The value in counter P is checked after the formation of each partial product. If the contents of P are different from zero, status bit *Zero* is set equal to 0 and the process is repeated to form a new partial product. The process stops when the counter reaches 0 and the controller's status input *Zero* is equal to 1. Note that the partial product formed in A is shifted into Q one bit at a time and eventually replaces the multiplier. The final product is available in A and Q, with A holding the most significant bits and Q the least significant bits of the product.

The previous numerical example is repeated in Table 8.5 to clarify the multiplication process. The procedure follows the steps outlined in the ASMD chart. The data shown in the table can be compared with simulation results.

The type of registers needed for the data processor subsystem can be derived from the register operations listed in the ASMD chart. Register A is a shift register with parallel load to accept the sum from the adder and must have a synchronous clear capability to reset the register to 0. Register Q is a shift register. The counter P is a binary down counter with a facility

Table 8.5
Numerical Example For Binary Multiplier

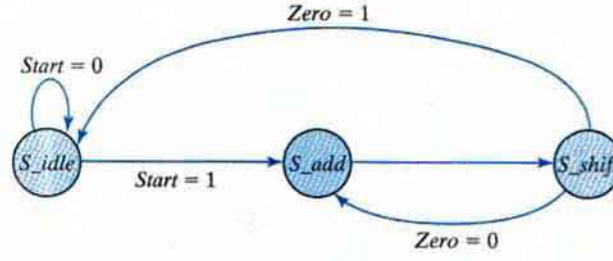
Multiplicand $B = 10111_2 = 17_H = 23_{10}$	**Multiplier $Q = 10011_2 = 13_H = 19_{10}$**			
	C	**A**	**Q**	**P**
Multiplier in Q	0	00000	10011	101
$Q_0 = 1$; add B		10111		
First partial product	0	10111		100
Shift right CAQ	0	01011	11001	
$Q_0 = 1$; add B		10111		
Second partial product	1	00010		011
Shift right CAQ	0	10001	01100	
$Q_0 = 0$; shift right CAQ	0	01000	10110	010
$Q_0 = 0$; shift right CAQ	0	00100	01011	001
$Q_0 = 1$; add B		10111		
Fifth partial product	0	11011		
Shift right CAQ	0	01101	10101	000
Final product in $AQ = 0110110101_2 = 1b5_H$				

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

ASMD chart of Fig. 8.15. The three states *S_idle* through *S_shift* are taken from the rectangular state boxes. The inputs *Start* and *Zero* are taken from the diamond-shaped decision boxes. The register transfer operations for each of the three states are listed in Fig. 8.16(b) and are taken from the corresponding state and conditional boxes in the ASMD chart. Establishing the state transitions is the initial focus, so the outputs of the controller are not shown.

There are two distinct aspects with which we have to deal when implementing the control logic: Establish the required sequence of states and provide signals to control the register operations. The sequence of states is specified in the ASMD chart or the state diagram. The signals for controlling the operations in the registers are specified in the register transfer statements annotated on the ASMD chart or listed in tabular format. For the multiplier, these signals are *Load_regs* (for parallel loading the registers in the datapath unit), *Decr_P* (for decrementing the counter), *Add_regs* (for adding the multiplicand and the partial product), and *Shift_regs* (for shifting register *CAQ*). The block diagram of the control unit is shown in Fig. 8.14(b). The inputs to the controller are *Start*, *Q[0]*, and *Zero*, and the outputs are *Ready*, *Load_regs*, *Decr_P*, *Add_regs*, and *Shift_regs*, as specified in the ASMD chart. We note that *Q[0]* affects only the output of the controller, not its state transitions. The machine transitions from *S_add* to *S_shift* unconditionally.

An important step in the design is the assignment of coded binary values to the states. The simplest assignment is the sequence of binary numbers, as shown in Table 8.6. A similar assignment is the Gray code, according to which only one bit changes when going from one number to the next. A state assignment often used in control design is the *one-hot* assignment. This assignment uses as many bits as there are states in the circuit. At any given time, only one bit is equal to 1 (the one that is hot) while all others are kept at 0 (all cold). This type of assignment uses a flip-flop for each state. Indeed, one-hot encoding uses more flip-flops than other types of coding, but it usually leads to simpler decoding logic for the next state and the output of the machine. Because the decoding logic does not become more complex as states are added to the machine, the speed at which the machine can operate is not limited by the time required to decode the state.

Since the controller is a sequential circuit, it can be designed manually by the sequential logic procedure outlined in Chapter 5. However, in most cases this method is difficult to carry out manually because of the large number of states and inputs that a typical control circuit may have. As a consequence, it is necessary to use specialized methods for control logic design that may be considered as variations of the classical sequential logic method. We will now present two such design procedures. One uses a sequence register and decoder, and the other uses one flip-flop per state. The method will be presented for a small circuit, but it applies to larger circuits as well. Of course, the need for these methods is eliminated if one has software that automatically synthesizes the circuit from an HDL description.

Table 8.6
State Assignment for Control

State	Binary	Gray Code	One-Hot
S_idle	00	00	001
S_add	01	01	010
S_shift	10	11	100

Sequence Register and Decoder

The sequence-register-and-decoder (manual) method, as the name implies, uses a register for the control states and a decoder to provide an output corresponding to each of the states. (The decoder is not needed if a one-hot code is used.) A register with n flip-flops can have up to 2^n states, and an n-to-2^n-line decoder has up to 2^n outputs. An n-bit sequence register is essentially a circuit with n flip-flops, together with the associated gates that effect their state transitions.

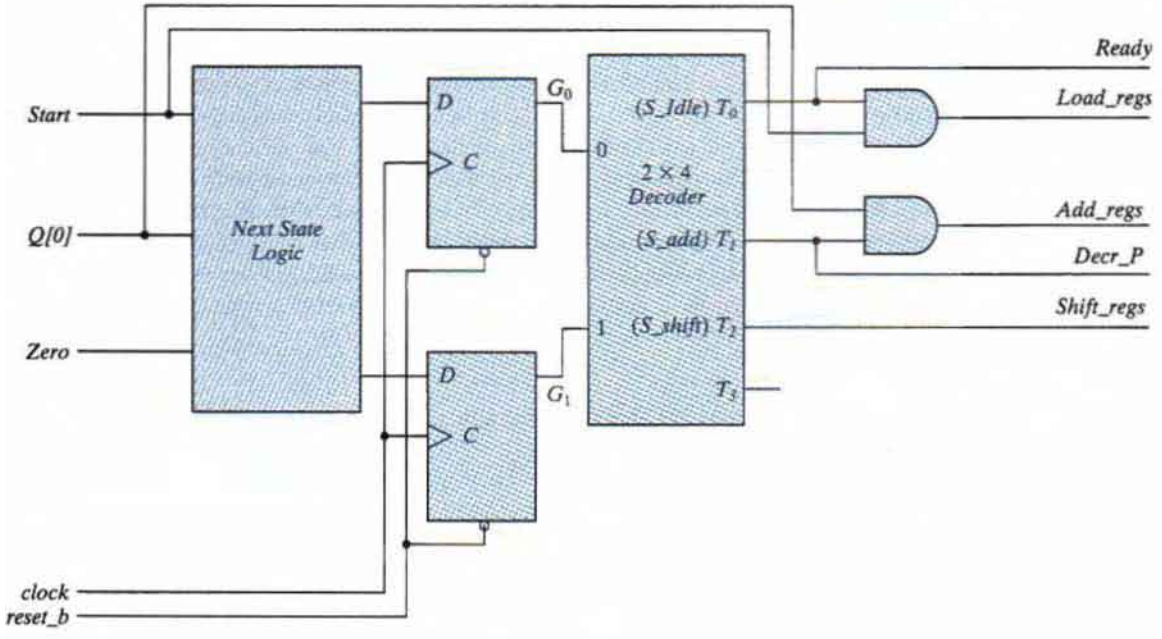
The ASMD chart and the state diagram for the controller of the binary multiplier have three states and two inputs. (There is no need to consider *Q[0]*.) To implement the design with a sequence register and decoder, we need two flip-flops for the register and a two-to-four-line decoder. The outputs of the decoder will form the Moore-type outputs of the controller directly. The Mealy-type outputs will be formed from the Moore outputs and the inputs.

The state table for the finite state machine of the controller is shown in Table 8.7. It is derived directly from the ASMD chart of Fig. 8.15(b) or the state diagram of Fig. 8.16(a). We designate the two flip-flops as G_1 and G_0 and assign the binary states 00, 01, and 10 to *S_idle*, *S_add*, and *S_shift*, respectively. Note that the input columns have don't-care entries whenever the input variable is not used to determine the next state. The outputs of the control circuit are designated by the names given in the ASMD chart. The particular Moore-type output variable that is equal to 1 at any given time is determined from the equivalent binary value of the present state. Those output variables are shaded in Table 8.7. Thus, when the present state is $G_1G_0 = 00$, output *Ready* must be equal to 1, while the other outputs remain at 0. Since the Moore-type outputs are a function of only the present state, they can be generated with a decoder circuit having the two inputs G_1 and G_0 and using three of the decoder outputs T_0 through T_2, as shown in Fig. 8.17(a), which does not include the wiring for the state feedback.

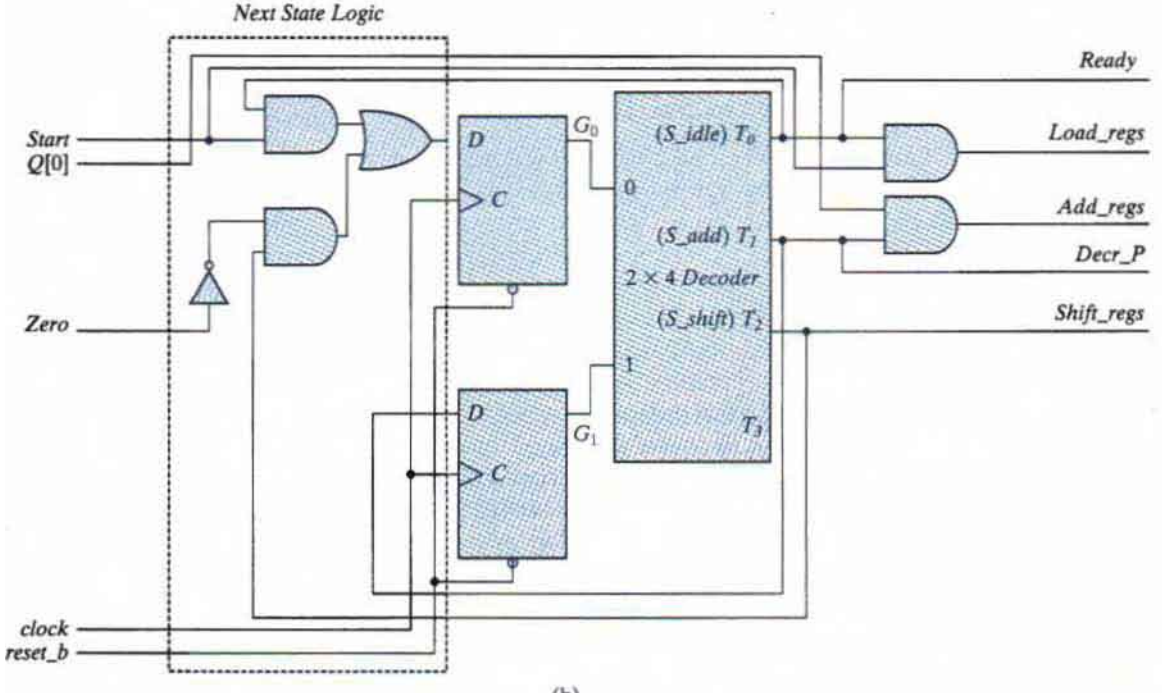
The state machine of the controller can be designed from the state table by means of the classical procedure presented in Chapter 5. This example has a small number of states and inputs, so we could use maps to simplify the Boolean functions. In most control logic applications, the

Table 8.7
State Table for Control Circuit

	Present State		Inputs			Next State						
Present-State Symbol	G_1	G_0	*Start*	*Q[0]*	*Zero*	G_1	G_0	*Ready*	*Load_regs*	*Decr_P*	*Add_regs*	*Shift_regs*
S_idle	0	0	0	X	X	0	0	1	0	0	0	0
S_idle	0	0	1	X	X	0	1	1	1	0	0	0
S_add	0	1	X	0	X	1	0	0	0	1	0	0
S_add	0	1	X	1	X	1	0	0	0	1	1	0
S_shift	1	0	X	X	0	0	1	0	0	0	0	1
S_shift	1	0	X	X	1	0	0	0	0	0	0	1



(a)



(b)

FIGURE 8.17
Logic diagram of control for binary multiplier using a sequence register and decoder

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

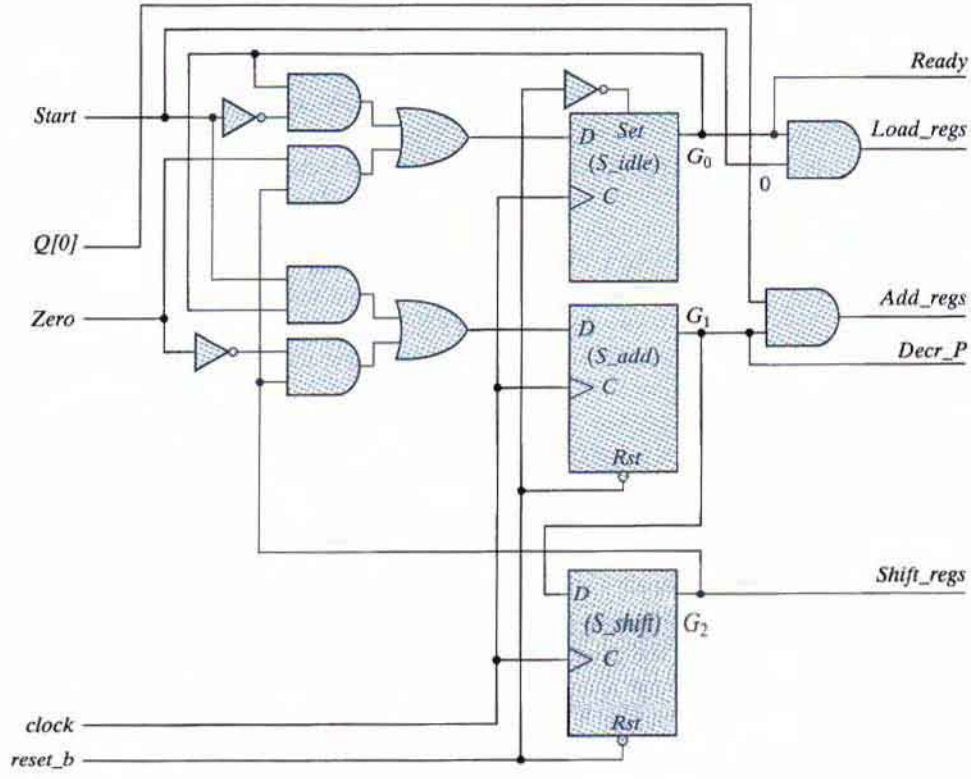


FIGURE 8.18
Logic diagram for one-hot state controller

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.)

HDL Example 8.5

```
module Sequential_Binary_Multiplier (Product, Ready, Multiplicand, Multiplier, Start,
clock, reset_b);
// Default configuration: five-bit datapath
  parameter                          dp_width = 5;       // Set to width of datapath
  output      [2*dp_width -1: 0]     Product;
  output                             Ready;
  input       [dp_width -1: 0]       Multiplicand, Multiplier;
  input                              Start, clock, reset_b;

  parameter                          BC_size = 3;        // Size of bit counter
  parameter                          S_idle =  3'b001, // one-hot code
                                     S_add =   3'b010,
                                     S_shift = 3'b100;
  reg   [2: 0]                       state, next_state;
  reg   [dp_width -1: 0]     A, B, Q;          // Sized for datapath
  reg                        C;
  reg   [BC_size -1: 0]      P;
  reg                        Load_regs, Decr_P, Add_regs, Shift_regs;

// Miscellaneous combinational logic

  assign                     Product = {A, Q};
  wire                       Zero = (P == 0);          // counter is zero
                             // Zero = ~|P;             // alternative
  wire                       Ready = (state == S_idle); // controller status
// control unit
  always @ (posedge clock, negedge reset_b)
    if (~reset_b) state <= S_idle; else state <= next_state;

  always @ (state, Start, Q[0], Zero) begin
    next_state = S_idle;
    Load_regs = 0;
    Decr_P = 0;
    Add_regs = 0;
    Shift_regs = 0;
    case (state)
      S_idle:    begin if (Start) next_state = S_add; Load_regs = 1; end
      S_add:     begin next_state = S_shift; Decr_P = 1; if (Q[0]) Add_regs = 1; end
      S_shift:   begin Shift_regs = 1; if (Zero) next_state = S_idle;
                 else next_state = S_add; end
      default:   next_state = S_idle;
```

```
    endcase
  end
 // datapath unit
  always @ (posedge clock) begin
   if (Load_regs) begin
    P <= dp_width;
    A <= 0;
    C <= 0;
    B <= Multiplicand;
    Q <= Multiplier;
   end
   if (Add_regs) {C, A} <= A + B;
   if (Shift_regs) {C, A, Q} <= {C, A, Q} >> 1;
   if (Decr_P) P <= P -1;
  end
endmodule
```

Testing the Multiplier

HDL Example 8.6 shows a test bench for testing the multiplier. The inputs and outputs are the same as those shown in the block diagram of Fig. 8.14(a). It is naive to conclude that an HDL description of a system is correct on the basis of the output it generates under the application of a few input signals. A more strategic approach to testing and verification exploits the partition of the design into its datapath and control unit. This partition supports separate verification of the controller and the datapath. A separate test bench can be developed to verify that the datapath executes each operation and generates status signals correctly. After the datapath unit is verified, the next step is to verify that each control signal is formed correctly by the control unit. A separate test bench can verify that the control unit exhibits the complete functionality specified by the ASMD chart (i.e., that it makes the correct state transitions and asserts its outputs in response to the external inputs and the status signals).

A verified control unit and a verified datapath unit together do not guarantee that the system will operate correctly. The final step in the design process is to integrate the verified models within a parent module and verify the functionality of the overall machine. The interface between the controller and the datapath must be examined in order to verify that the ports are connected correctly. For example, a mismatch in the listed order of signals may not be detected by the compiler. After the datapath unit and the control unit have been verified, a third test bench should verify the specified functionality of the complete system. In practice, this requires writing a comprehensive test plan identifying that functionality. For example, the test plan would identify the need to verify that the sequential multiplier asserts the signal *Ready* in state *S_idle*. The exercise to write a test plan is not academic: The quality and scope of the test plan determine the worth of the verification effort. The test plan guides the development of the test bench and increases the likelihood that the final design will match its specification.

Testing and verifying an HDL model usually requires access to more information than the inputs and outputs of the machine. Knowledge of the state of the control unit, the control signals, the status signals, and the internal registers of the datapath might all be necessary for debugging. Fortunately, Verilog provides a mechanism to hierarchically de-reference identifiers so that any variable at any level of the design hierarchy can be visible to the test bench. Procedural statements can display the information required to support efforts to debug the machine. Simulators use this mechanism to display waveforms of any variable in the design hierarchy. To use the mechanism, we reference the variable by its hierarchical path name. For example, the register *P* within the datapath unit is not an output port of the multiplier, but it can be referenced as *M0.P*. The hierarchical path name consists of the sequence of module identifiers or block names, separated by periods and specifying the location of the variable in the design hierarchy. We also note that simulators commonly have a graphical user interface that displays all levels of the hierarchy of a design.

The first test bench in HDL Example 8.6 uses the system task **$strobe** to display the result of the computations. This task is similar to the **$display** and **$monitor** tasks explained in Section 4.12. The **$strobe** system task provides a synchronization mechanism to ensure that data are displayed only after all assignments in a given time step are executed. This is very useful in synchronous sequential circuits, where the time step begins at a clock edge and multiple assignments may occur at the same time step of simulation. When the system is synchronized to the positive edge of the clock, using **$strobe** after the ***always @ (posedge*** *clock)* statement ensures that the display shows values of the signal after the clock pulse.

The test bench module *t_Sequential_Binary_Multiplier* in HDL Example 8.6 instantiates the module *Sequential Binary_Multiplier* of HDL Example 8.5. Both modules must be included as source files when simulating the multiplier with a Verilog HDL simulator. The result of this simulation displays a simulation log with numbers identical to the ones in Table 8.5. The code includes a second test bench to exhaustively multiply five-bit values of the multiplicand and the multiplier. Waveforms for a sample of simulation results are shown in Fig. 8.19. The numerical values of *Multiplicand*, *Multiplier*, and *Product* are displayed in decimal and hexadecimal formats. Insight can be gained by studying the displayed waveforms of the control state, the control signals, the status signals, and the register operations. Enhancements to the multiplier and its test bench are considered in the problems at the end of this chapter. In this example, $19_{10} \times 23_{10} = 437_{10}$, and $17_H + 0b_H = 02_H$ with C = 1. Note the need for the carry bit.

HDL Example 8.6

```
// Test bench for the binary multiplier
module t_Sequential_Binary_Multiplier;
 parameter                    dp_width = 5;                 // Set to width of datapath
 wire       [2*dp_width -1: 0] Product;                     // Output from multiplier
 wire                         Ready;
 reg        [dp_width -1: 0]  Multiplicand, Multiplier;     // Inputs to multiplier
 reg                          Start, clock, reset_b;
```

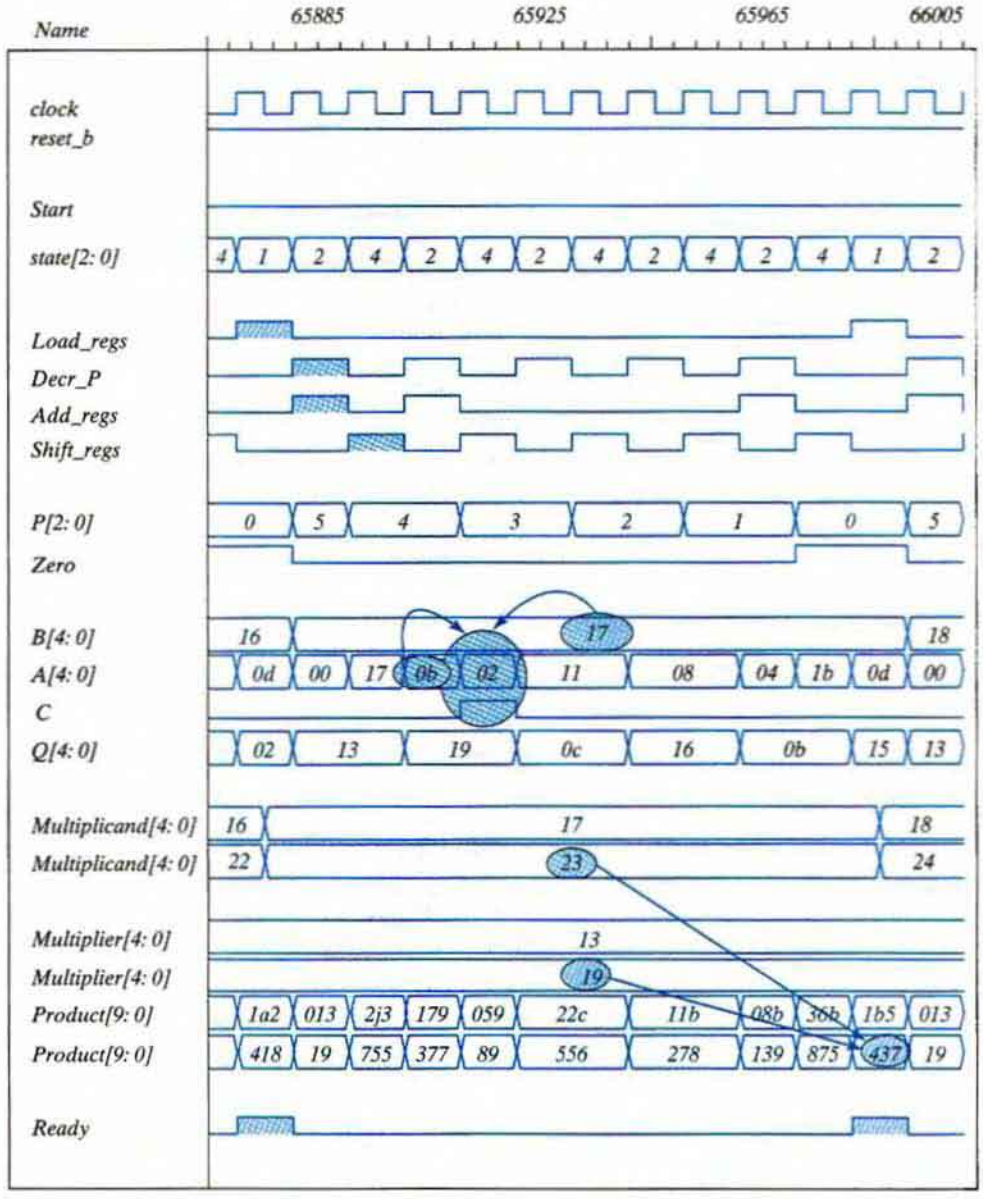


FIGURE 8.19
Simulation waveforms for one-hot state controller

```
 // Instantiate multiplier
 Sequential_Binary_Multiplier M0 (Product, Ready, Multiplicand, Multiplier, Start, clock,
  reset_b);
 // Generate stimulus waveforms
 initial #200 $finish;
 initial
  begin
   Start = 0;
   reset_b = 0;
   #2 Start = 1; reset_b = 1;
   Multiplicand = 5'b10111;     Multiplier = 5'b10011;
   #10 Start = 0;
  end
 initial
  begin
   clock = 0;
   repeat (26) #5 clock = ~clock;
  end
 // Display results and compare with Table 8.5
 always @ (posedge clock)
  $strobe ("C=%b  A=%b  Q=%b  P=%b  time=%0d",M0.C,M0.A,M0.Q,M0.P, $time);
endmodule
```

Simulation log:

```
C=0  A=00000  Q=10011  P=101  time=5
C=0  A=10111  Q=10011  P=100  time=15
C=0  A=01011  Q=11001  P=100  time=25
C=1  A=00010  Q=11001  P=011  time=35
C=0  A=10001  Q=01100  P=011  time=45
C=0  A=10001  Q=01100  P=010  time=55
C=0  A=01000  Q=10110  P=010  time=65
C=0  A=01000  Q=10110  P=001  time=75
C=0  A=00100  Q=01011  P=001  time=85
C=0  A=11011  Q=01011  P=000  time=95
C=0  A=01101  Q=10101  P=000  time=105
C=0  A=01101  Q=10101  P=000  time=115
C=0  A=01101  Q=10101  P=000  time=125
```

```
/* Test bench for exhaustive simulation
module t_Sequential_Binary_Multiplier;
 parameter                          dp_width = 5;              // Width of datapath
 wire      [2 * dp_width -1: 0]     Product;
 wire                               Ready;
 reg       [dp_width -1: 0]         Multiplicand, Multiplier;
 reg                                Start, clock, reset_b;
```

```
 Sequential_Binary_Multiplier M0 (Product, Ready, Multiplicand, Multiplier, Start,
   clock, reset_b);
 initial #1030000 $finish;
 initial begin clock = 0; #5 forever #5 clock = ~clock; end
 initial fork
   reset_b = 1;
   #2 reset_b = 0;
   #3 reset_b = 1;
 join
 initial begin #5 Start = 1; end
 initial begin
   #5 Multiplicand = 0;
   Multiplier = 0;
   repeat (32) #10 begin Multiplier = Multiplier + 1;
     repeat (32) @ (posedge M0.Ready) 5 Multiplicand = Multiplicand + 1;
   end
 end
endmodule
*/
```

Behavioral Description of a Parallel Multiplier

Structural modeling implicitly specifies the functionality of a digital machine by prescribing an interconnection of gate-level hardware units. In this form of modeling, a synthesis tool performs Boolean optimization and translates the HDL description of a circuit into a netlist of gates in a particular technology, e.g., CMOS. Hardware design at this level often requires cleverness and accrued experience. It is the most tedious and detailed form of modeling. In contrast, behavioral RTL modeling specifies functionality abstractly, in terms of HDL operators. The RTL model does not specify a gate-level implementation of the registers or the logic to control the operations that manipulate their contents—those tasks are accomplished by a synthesis tool. RTL modeling implicitly schedules operations by explicitly assigning them to clock cycles. The most abstract form of behavioral modeling describes only an algorithm, without any reference to a physical implementation, a set of resources, or a schedule for their use. Thus, algorithmic modeling allows a designer to explore trade-offs in the space (hardware) and time domains, trading processing speed for hardware complexity.

HDL Example 8.7 presents an RTL model and an algorithmic model of a binary multiplier. Both use a level-sensitive cyclic behavior. The RTL model expresses the functionality of a multiplier in a single statement. A synthesis tool will associate with the multiplication operator a gate-level circuit equivalent to that shown in Section 4.7. In simulation, when either the multiplier or the multiplicand changes, the product will be updated. The time required to form the product will depend on the propagation delays of the gates available in the library of standard cells used by the synthesis tool. The second model is an algorithmic description of the multiplier. A synthesis tool will unroll the loop of the algorithm and infer the need for a gate-level circuit equivalent to that shown in Section 4.7.

Be aware that a synthesis tool may not be able to synthesize a given algorithmic description, even though the associated HDL model will simulate and produce correct results. One difficulty is that the sequence of operations implied by an algorithm might not be physically realizable in a single clock cycle. It then becomes necessary to distribute the operations over multiple clock cycles. A tool for synthesizing RTL logic will not be able to automatically accomplish the required distribution of effort, but a tool that is designed to synthesize algorithms should be successful. In effect, a behavioral synthesis tool would have to allocate the registers and adders to implement multiplication. If only a single adder is to be shared by all of the operations that form a partial sum, the activity must be distributed over multiple clock cycles and in the correct sequence, ultimately leading to the sequential binary multiplier for which we have explicitly designed the controller for its datapath. Behavioral synthesis tools require a different and more sophisticated style of modeling and are not within the scope of this text.

HDL Example 8.7

```
// Behavioral (RTL) description of a parallel multiplier (n = 8)
module Mult (Product, Multiplicand, Multiplier);
  input [7: 0]          Multiplicand, Multiplier;
  output reg [15: 0] Product;
  always @ (Multiplicand, Multiplier)
    Product = Multiplicand * Multiplier;
endmodule
module Algorithmic_Binary_Multiplier #(parameter dp_width = 5) (
   output [2*dp_width -1: 0] Product, input [dp_width -1: 0] Multiplicand, Multiplier);
   reg [dp_width -1: 0]              A, B, Q;          // Sized for datapath
   reg                               C;
   integer                           k;
   assign                            Product = {C, A, Q};
   always @ (Multiplier, Multiplicard) begin
     Q = Multiplier;
     B = Multiplicand;
     C = 0;
     A = 0;
     for (k = 0; k <= dp_width -1; k = k + 1) begin
     if (Q[0]) {C, A} = A + B;
     {C, A, Q} = {C, A, Q} >> 1;
   end
  end
endmodule
module t_Algorithmic_Binary_Multiplier;
  parameter                         dp_width = 5;     // Width of datapath
  wire [2* dp_width -1: 0]          Product;
  reg [dp_width -1: 0]              Multiplicand, Multiplier;
  integer                           Exp_Value;
```

```
reg                                    Error;
 Algorithmic_Binary_Multiplier M0 (Product, Multiplicand, Multiplier);
// Error detection
 initial # 1030000 finish;
 always @ (Product) begin
  Exp_Value = Multiplier * Multiplicand;
  // Exp_Value = Multiplier * Multiplicand +1; // Inject error to confirm detection
  Error = Exp_Value ^ Product;
 end
// Generate multiplier and multiplicand exhaustively for 5 bit operands
 initial begin
  #5 Multiplicand = 0;
  Multiplier = 0;
  repeat (32) #10 begin Multiplier = Multiplier + 1;
   repeat (32) #5 Multiplicand = Multiplicand + 1;
  end
 end
endmodule
```

8.10 DESIGN WITH MULTIPLEXERS

The sequence-register-and-decoder scheme for the design of a controller has three parts: the flip-flops that hold the binary state value, the decoder that generates the control outputs, and the gates that determine the next-state and output signals. In Section 4.11, it was shown that a combinational circuit can be implemented with multiplexers instead of individual gates. Replacing the gates with multiplexers results in a regular pattern of three levels of components. The first level consists of multiplexers that determine the next state of the register. The second level contains a register that holds the present binary state. The third level has a decoder that asserts a unique output line for each control state. These three components are predefined standard cells in many integrated circuits.

Consider, for example, the ASM chart of Fig. 8.20, consisting of four states and four control inputs. We are interested in only the control signals governing the state sequence. These signals are independent of the register operations of the datapath, so the edges of the graph are not annotated with datapath register operations, and the graph does not identify the output signals of the controller. The binary assignment for each state is indicated at the upper right corner of the state boxes. The decision boxes specify the state transitions as a function of the four control inputs: w, x, y, and z. The three-level control implementation, shown in Fig. 8.21, consists of two multiplexers, MUX1 and MUX2; a register with two flip-flops, G_1 and G_0; and a decoder with four outputs—d_0, d_1, d_2, and d_3, corresponding to *S_0*, *S_1*, *S_2*, and *S_3*, respectively. The outputs of the state-register flip-flops are applied to the decoder inputs and also to the select inputs of the multiplexers. In this way, the present state of the register is used to select one of the inputs from each multiplexer. The outputs of the multiplexers are then applied to the D inputs of G_1 and G_0. The purpose of each multiplexer is to produce an input to its corresponding flip-flop equal to the binary value of that bit of the next-state vector. The inputs of

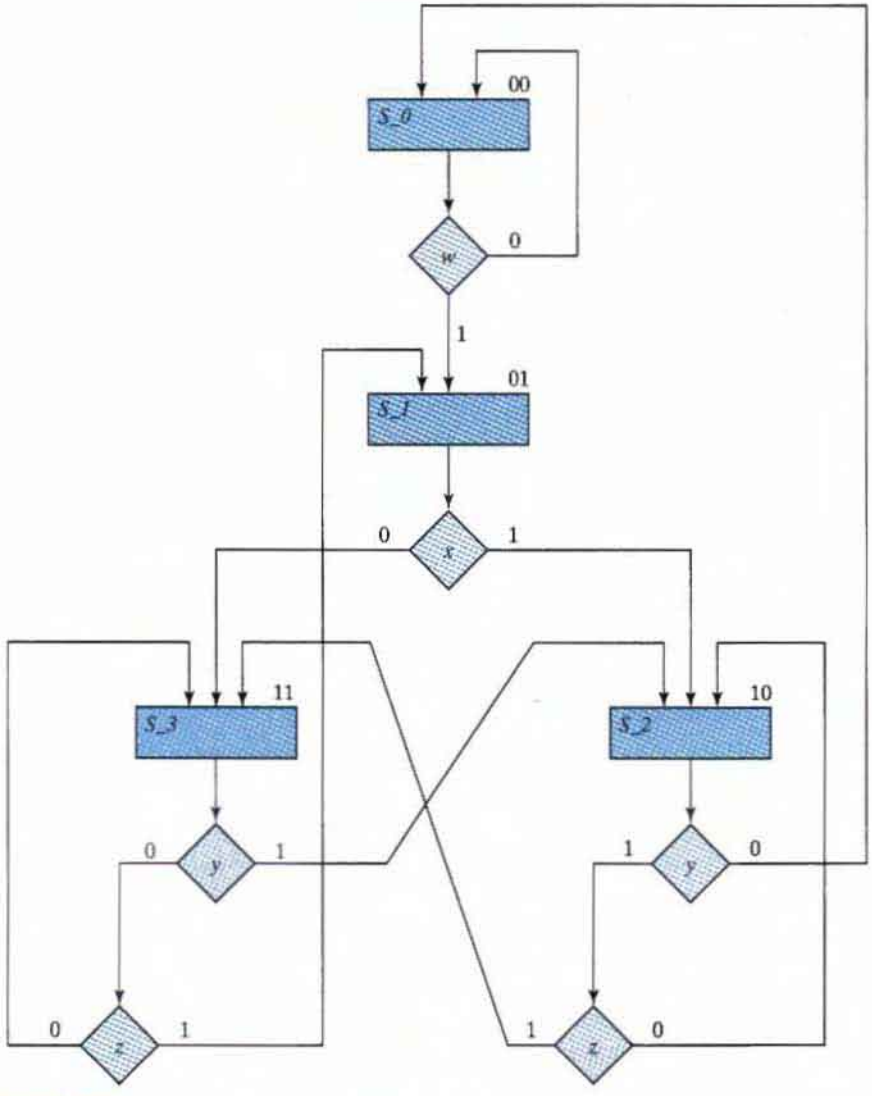


FIGURE 8.20
Example of ASM chart with four control inputs

the multiplexers are determined from the decision boxes and state transitions given in the ASM chart. For example, state 00 stays at 00 or goes to 01, depending on the value of input w. Since the next state of G_1 is 0 in either case, we place a signal equivalent to logic 0 in MUX1 input 0. The next state of G_0 is 0 if $w = 0$ and 1 if $w = 1$. Since the next state of G_0 is equal to w, we apply control input w to MUX2 input 0. This means that when the select inputs of the multiplexers are equal to present state 00, the outputs of the multiplexers provide the binary value that is transferred to the register at the next clock pulse.

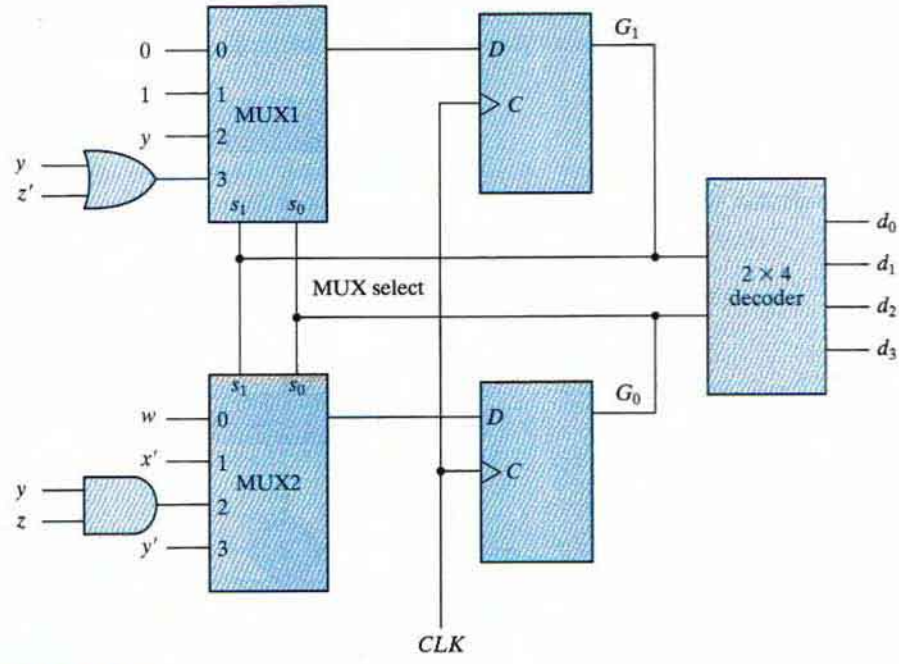


FIGURE 8.21
Control implementation with multiplexers

To facilitate the evaluation of the multiplexer inputs, we prepare a table showing the input conditions for each possible state transition in the ASM chart. Table 8.8 gives this information for the ASM chart of Fig. 8.20. There are two transitions from present state 00 or 01 and three from present state 10 or 11. The sets of transitions are separated by horizontal lines across the table. The input conditions listed in the table are obtained from the decision boxes in the ASM chart. For example, from Fig. 8.20, we note that present state 01 will go to next state 10 if $x = 1$ or to next state 11 if $x = 0$. In the table, we mark these input conditions as x and x', respectively. The two columns under "multiplexer inputs" in the table specify the input values that must be applied to MUX1 and MUX2. The multiplexer input for each present state is determined from the input conditions when the next state of the flip-flop is equal to 1. Thus, after present state 01, the next state of G_1 is always equal to 1 and the next state of G_0 is equal to the complement of x. Therefore, the input of MUX1 is made equal to 1 and that of MUX2 to x' when the present state of the register is 01. As another example, after present state 10, the next state of G_1 must be equal to 1 if the input conditions are yz' or yz. When these two Boolean terms are ORed together and then simplified, we obtain the single binary variable y, as indicated in the table. The next state of G_0 is equal to 1 if the input conditions are $yz = 11$. If the next state of G_1 remains at 0 after a given present state, we place a 0 in the multiplexer input, as shown in present state 00 for MUX1. If the next state of G_1 is always 1, we place a 1 in the multiplexer input, as shown in present state 01 for MUX1. The other entries for MUX1 and MUX2 are derived in a similar

Table 8.8
Multiplexer Input Conditions

Present State		Next State		Input Condition	Inputs	
G_1	G_0	G_1	G_0	s	MUX1	MUX2
0	0	0	0	w'		
0	0	0	1	w	0	w
0	1	1	0	x		
0	1	1	1	x'	1	x'
1	0	0	0	y'		
1	0	1	0	yz'		
1	0	1	1	yz	$yz' + yz = y$	yz
1	1	0	1	$y'z$		
1	1	1	0	y		
1	1	1	1	$y'z'$	$y + y'z' = y + z'$	$y'z + y'z' = y'$

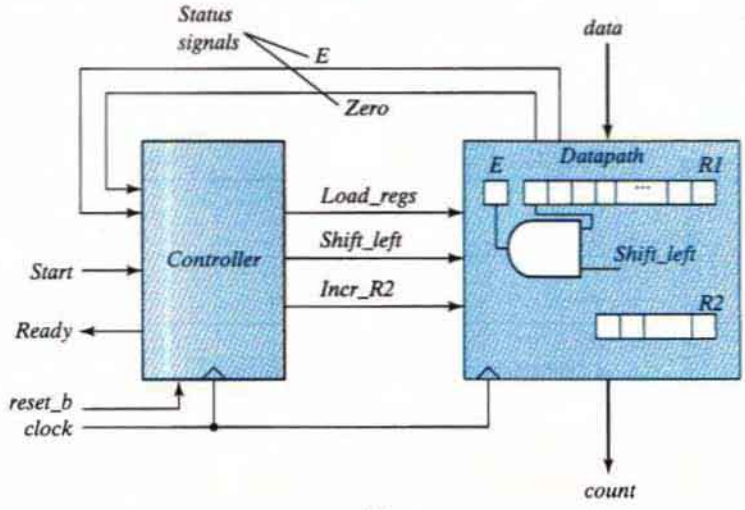
manner. The multiplexer inputs from the table are then used in the control implementation of Fig. 8.21. Note that if the next state of a flip-flop is a function of two or more control variables, the multiplexer may require one or more gates in its input. Otherwise, the multiplexer input is equal to the control variable, the complement of the control variable, 0, or 1.

Design Example: Count the Number of Ones in a Register

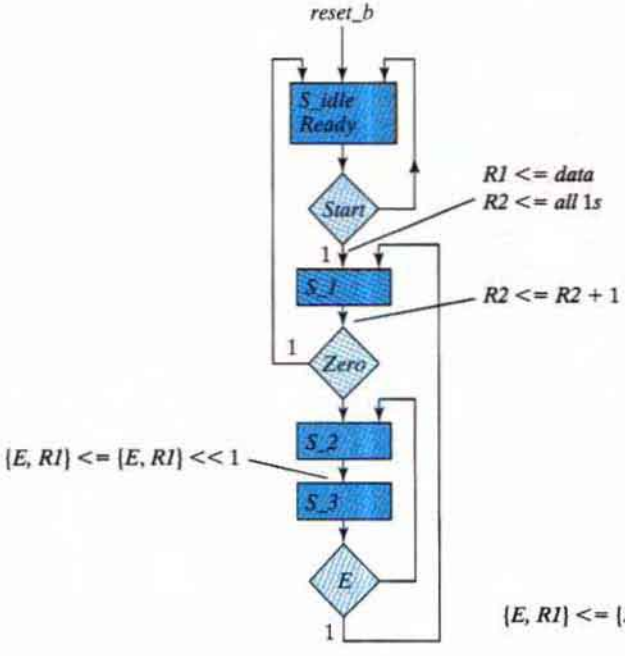
We will demonstrate the multiplexer implementation of the logic for a control unit by means of a design example—a system that is to count the number of 1's in a word of data. The example will also demonstrate the formulation of the ASMD chart and the implementation of the datapath subsystem.

From among various alternatives, we will consider a ones counter consisting of two registers *R1* and *R2*, and a flip-flop *E*. (A more efficient implementation is considered in the problems at the end of the chapter.) The system counts the number of 1's in the number loaded into register *R1* and sets register *R2* to that number. For example, if the binary number loaded into *R1* is 10111001, the circuit counts the five 1's in *R1* and sets register *R2* to the binary count 101. This is done by shifting each bit from register *R1* one at a time into flip-flop *E*. The value in *E* is checked by the control, and each time it is equal to 1, register *R2* is incremented by 1.

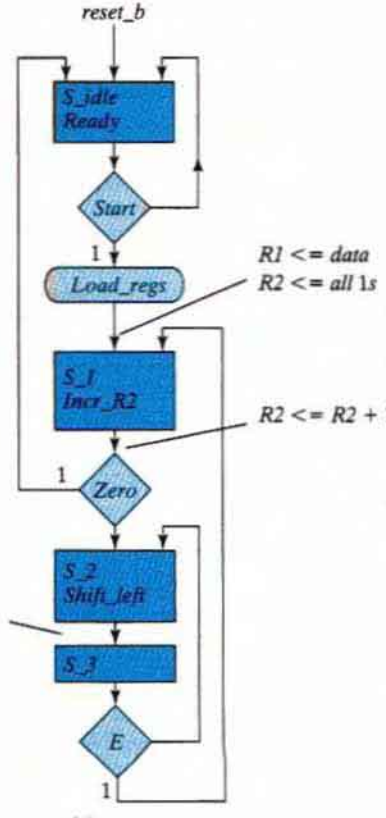
The block diagram of the datapath and controller are shown in Fig. 8.22(a). The datapath contains registers *R1*, *R2*, and *E*, as well as logic to shift the leftmost bit of *R1* into *E*. The unit also contains logic (a NOR gate to detect whether *R1* is 0, but that detail is omitted in the figure). The external input signal *Start* launches the operation of the machine; *Ready* indicates the status of the machine to the external environment. The controller has status input signals *E* and *Zero* from the datapath. These signals indicate the contents of a register holding the MSB of the data word and the condition that the data word is 0, respectively. *E* is the output of the flip-flop. *Zero* is the output of a circuit that checks the contents of register *R1* for all 0's. The circuit produces an output $Zero = 1$ when *R1* is equal to 0 (i.e., when *R1* is empty of 1's).



(a)



(b)



(c)

FIGURE 8.22
Block diagram and ASMD chart for count-of-ones circuit

A preliminary ASMD chart showing the state sequence and the register operations is illustrated in Fig. 8.22(b), and the complete ASMD chart in Fig. 8.22(c). Asserting *Start* with the controller in *S_idle* transfers the state to *S_1*, concurrently loads register *R1* with the binary data word, and fills the cells of *R2* with 1's. Note that incrementing a number with all 1's in a counter register produces a number with all 0's. Thus, the first transition from *S_1* to *S_2* will clear *R2*. Subsequent transitions will have *R2* holding a count of the bits of data that have been processed. The content of *R1*, as indicated by *Zero*, will also be examined in *S_1*. If *R1* is empty, *Zero* = 1, and the state returns to *S_idle*, where it asserts *Ready*. In state *S_1*, *Incr_R2* is asserted to cause the datapath unit to increment *R2* at each clock pulse. If *R1* is not empty of 1's, then *Zero* = 0, indicating that there are some 1's stored in the register. The number in *R1* is shifted and its leftmost bit is transferred into *E*. This is done as many times as necessary, until a 1 is transferred into *E*. For every 1 detected in *E*, register *R2* is incremented and register *R1* is checked again for more 1's. The major loop is repeated until all the 1's in *R1* are counted. Note that the state box of *S_3* has no register operations, but the block associated with it contains the decision box for *E*. Note also that the serial input to shift register *R1* must be equal to 0 because we don't want to shift external 1's into *R1*. The register *R1* in Fig. 8.22(a) is a shift register. Register *R2* is a counter with parallel load. The multiplexer input conditions for the control are determined from Table 8.9. The input conditions are obtained from the ASMD chart for each possible binary state transition. The four states are assigned binary values 00 through 11. The transition from present state 00 depends on *Start*. The transition from present state 01 depends on *Zero*, and the transition from present state 11 on *E*. Present state 10 goes to next state 11 unconditionally. The values under MUX1 and MUX2 in the table are determined from the Boolean input conditions for the next state of G_1 and G_0, respectively.

The control implementation of the design example is shown in Fig. 8.23. This is a three-level implementation, with the multiplexers in the first level. The inputs to the multiplexers are obtained from Table 8.9. The Verilog description in HDL Example 8.8 instantiates structural models of the controller and the datapath. The listing of code includes the lower level modules

Table 8.9
Multiplexer Input Conditions for Design Example

Present State		Next State		Input Conditions	Multiplexer Inputs	
G_1	G_0	G_1	G_0		MUX1	MUX2
0	0	0	0	*Start'*		
0	0	0	1	*Start*	0	*Start*
0	1	0	0	*Zero*		
0	1	1	0	*Zero'*	*Zero'*	0
1	0	1	1	None	1	1
1	1	1	0	*E'*		
1	1	0	1	*E*	*E'*	*E*

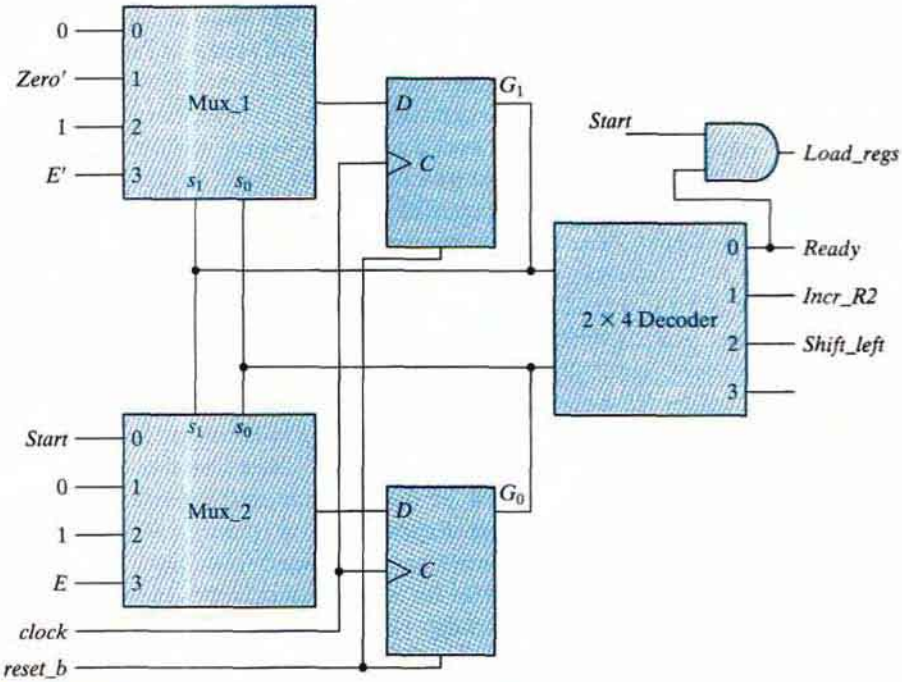


FIGURE 8.23
Control implementation for count-of-ones circuit

implementing their structures. Note that the datapath unit does not have a reset signal to clear the registers, but the models for the flip-flop, shift register, and counter have an active-low reset. This illustrates the use of Verilog data type **supply1** to hardwire those ports to logic value 1 in their instantiation within *Datapath_STR*. Note also that the test bench uses hierarchical dereferencing to access the state of the controller to make the debug and verification tasks easier, without having to alter the module ports to provide access to the internal signals. Another detail to observe is that the serial input to the shift register is hardwired to 0. The lower level models are described behaviorally for simplicity.

HDL Example 8.8

```
module Count_Ones_STR_STR (count, Ready, data, Start, clock, reset_b);
// Mux – decoder implementation of control logic
// controller is structural
// datapath is structural

 parameter R1_size = 8, R2_size = 4;
 output        [R2_size -1: 0]    count;
 output                           Ready;
```

```
 input [R1_size -1: 0]          data;
 input                          Start, clock, reset_b;
 wire                           Load_regs, Shift_left, Incr_R2, Zero, E;

 Controller_STR M0 (Ready, Load_regs, Shift_left, Incr_R2, Start, E, Zero,
  clock, reset_b);
 Datapath_STR  M1 (count, E, Zero, data, Load_regs, Shift_left, Incr_R2,
  clock);
endmodule

module Controller_STR (Ready, Load_regs, Shift_left, Incr_R2, Start, E, Zero, clock,
 reset_b);
 output        Ready;
 output        Load_regs, Shift_left, Incr_R2;
 input         Start;
 input         E, Zero;
 input         clock, reset_b;
 supply0       GND;
 supply1       PWR;
 parameter     S0 = 2'b00, S1 = 2'b01, S2 = 2'b10, S3 = 2'b11;  // Binary code
 wire                         Load_regs, Shift_left, Incr_R2;
 wire                         G0, G0_b, D_in0, D_in1, G1, G1_b;
 wire                         Zero_b = ~Zero;
 wire                         E_b = ~E;
 wire [1: 0]                  select = {G1, G0};
 wire [0: 3]                  Decoder_out;
 assign                       Ready = ~Decoder_out[0];
 assign                       Incr_R2 = ~Decoder_out[1];
 assign                       Shift_left = ~Decoder_out[2];
 and                          (Load_regs, Ready, Start);
 mux_4x1_beh          Mux_1   (D_in1, GND, Zero_b, PWR, E_b, select);
 mux_4x1_beh          Mux_0   (D_in0, Start, GND, PWR, E, select);
 D_flip_flop_AR_b     M1      (G1, G1_b, D_in1, clock, reset_b);
 D_flip_flop_AR_b     M0      (G0, G0_b, D_in0, clock, reset_b);
 decoder_2x4_df       M2      (Decoder_out, G1, G0, GND);

endmodule

module Datapath_STR (count, E, Zero, data, Load_regs, Shift_left, Incr_R2, clock);
 parameter                    R1_size = 8, R2_size = 4;
 output [R2_size -1: 0]  count;
 output                       E, Zero;
 input  [R1_size -1: 0]       data;
 input                        Load_regs, Shift_left, Incr_R2, clock;
```

```
 wire   [R1_size -1: 0]          R1;
 wire                            Zero;
 supply0                         Gnd;
 supply1                         Pwr;
 assign Zero = (R1 == 0);        // implicit combinational logic
 Shift_Reg              M1       (R1, data, Gnd, Shift_left, Load_regs, clock, Pwr);
 Counter                M2       (count, Load_regs, Incr_R2, clock, Pwr);
 D_flip_flop_AR         M3       (E, w1, clock, Pwr);
 and                             (w1, R1[R1_size -1], Shift_left);
endmodule

module Shift_Reg (R1, data, SI_0, Shift_left, Load_regs, clock, reset_b);
 parameter R1_size = 8;
 output    [R1_size -1: 0]       R1;
 input     [R1_size -1: 0]       data;
 input                           SI_0, Shift_left, Load_regs;
 input                           clock, reset_b;
 reg       [R1_size -1: 0]       R1;
 always @ (posedge clock, negedge reset_b)
  if (reset_b == 0) R1 <= 0;
  else begin
   if (Load_regs) R1 <= data; else
    if (Shift_left) R1 <= {R1[R1_size -2: 0], SI_0}; end
endmodule
module Counter (R2, Load_regs, Incr_R2, clock, reset_b);
 parameter                       R2_size = 4;
 output [R2_size -1: 0]          R2;
 input                           Load_regs, Incr_R2;
 input                           clock, reset_b;
 reg    [R2_size -1: 0]          R2;
 always @ (posedge clock, negedge reset_b)
  if (reset_b == 0) R2 <= 0;
  else if (Load_regs) R2 <= {R2_size {1'b1}};        // Fill with 1
   else if (Incr_R2 == 1) R2 <= R2 + 1;
endmodule
module D_flip_flop_AR (Q, D, CLK, RST);
 output         Q;
 input          D, CLK, RST;
 reg            Q;
 always @ (posedge CLK, negedge RST)
  if (RST == 0) Q <= 1'b0;
  else Q <= D;
endmodule
```

```
module D_flip_flop_AR_b (Q, Q_b, D, CLK, RST);
 output          Q, Q_b;
 input           D, CLK, RST;
 reg             Q;
 assign          Q_b = ~Q;
 always @ (posedge CLK, negedge RST)
  if (RST == 0) Q <= 1'b0;
  else Q <= D;
endmodule
// Behavioral description of four-to-one line multiplexer
// Verilog 2005 port syntax
module mux_4x1_beh
(output reg     m_out,
 input          in_0, in_1, in_2, in_3,
 input [1: 0]   select
);
 always @ (in_0, in_1, in_2, in_3, select)   // Verilog 2005 syntax
  case (select)
    2'b00:      m_out = in_0;
    2'b01:      m_out = in_1;
    2'b10:      m_out = in_2;
    2'b11:      m_out = in_3;
  endcase
 endmodule

// Dataflow description of two-to-four-line decoder
// See Fig. 4.19. Note: The figure uses symbol E, but the
// Verilog model uses enable to indicate functionality clearly.
module decoder_2x4_df (D, A, B, enable);
 output          [0: 3]    D;
 input                     A, B;
 input                     enable;

 assign          D[0] = ~(~A & ~B & ~enable),
                 D[1] = ~(~A & B & ~enable),
                 D[2] = ~(A & ~B & ~enable),
                 D[3] = ~(A & B & ~enable);
endmodule

module t_Count_Ones;
 parameter R1_size = 8, R2_size = 4;
 wire            [R2_size -1: 0]    R2;
 wire            [R2_size -1: 0]    count;
```

```
wire                          Ready;
reg          [R1_size -1: 0]  data;
reg                           Start, clock, reset_b;
wire         [1: 0]           state;          // Use only for debug
assign state = {M0.M0.G1, M0.M0.G0};
Count_Ones_STR_STR M0 (count, Ready, data, Start, clock, reset_b);
initial #650 $finish;
initial begin clock = 0; #5 forever #5 clock = ~clock; end
initial fork
  #1 reset_b = 1;
  #3 reset_b = 0;
  #4 reset_b = 1;
  #27 reset_b = 0;
  #29 reset_b = 1;
  #355 reset_b = 0;
  #365 reset_b = 1;
  #4 data = 8'Hff;
  #145 data = 8'haa;
  # 25 Start = 1;
  # 35  Start = 0;
  #55 Start = 1;
  #65 Start = 0;
  #395 Start = 1;
  #405 Start = 0;
 join
endmodule
```

Testing the Ones Counter

The test bench in HDL Example 8.8 was used to produce the simulation results in Fig. 8.24. Annotations have been added for clarification. In Fig. 8.24(a), *reset_b* is toggled low at $t = 3$ to drive the controller into *S_idle*, but with *Start* not yet having an assigned value. (The default is x.) Consequently, the controller enters an unknown state (the shaded waveform) at the next clock, and its outputs are unknown. When *reset_b* is asserted (low) again at $t = 27$, the state enters *S_idle*. Then, with *Start* = 1 at the first clock after *reset_b* is deasserted, (1) the controller enters *S_1*, (2) *Load_regs* causes *R1* to be set to the value of *data*, namely, 8'Hff, and (3) *R2* is filled with 1's. At the next clock, *R2* starts counting from 0. *Shift_left* is asserted while the controller is in state *S_2*, and *incr_R2* is asserted while the controller is in state *S_1*. Notice that *R2* is incremented in the next cycle after *incr_R2* is asserted. No output is asserted in state *S_3*. The counting sequence continues in Fig. 8.24(b) until *Zero* is asserted, with *E* holding the last 1 of the data word. The next clock produces *count* = 8, and *state* returns to *S_idle*. (Additional testing is addressed in the problems at the end of the chapter.)

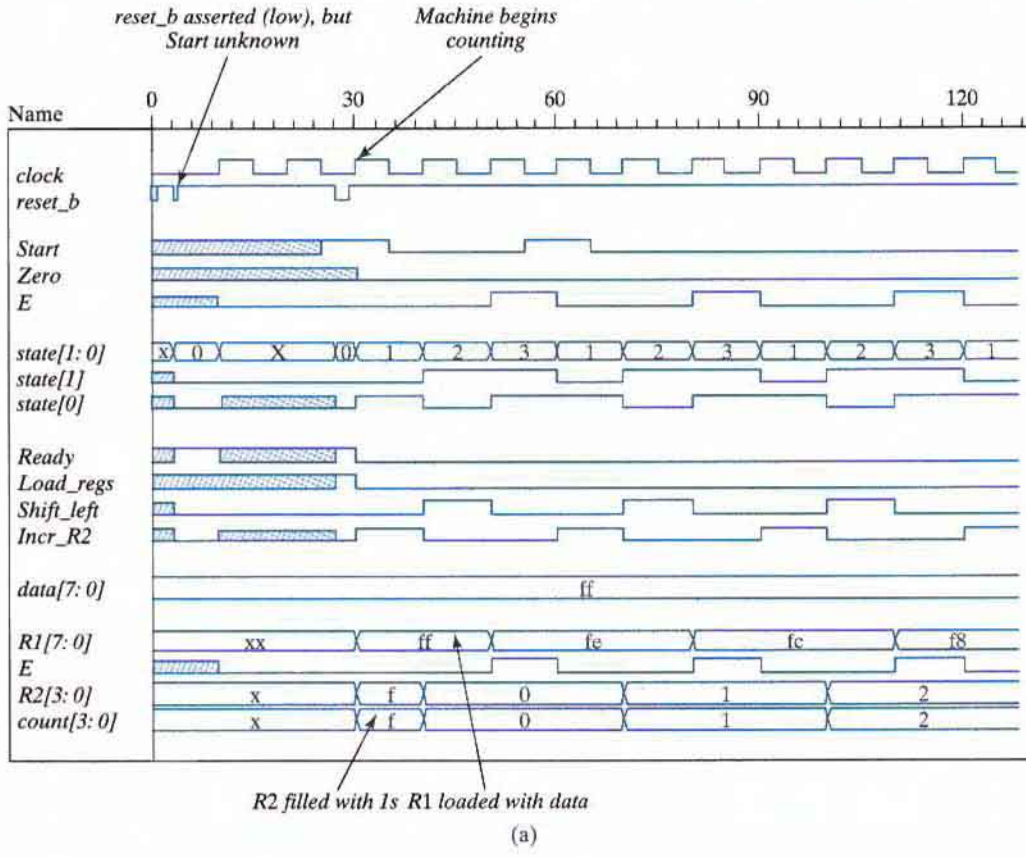
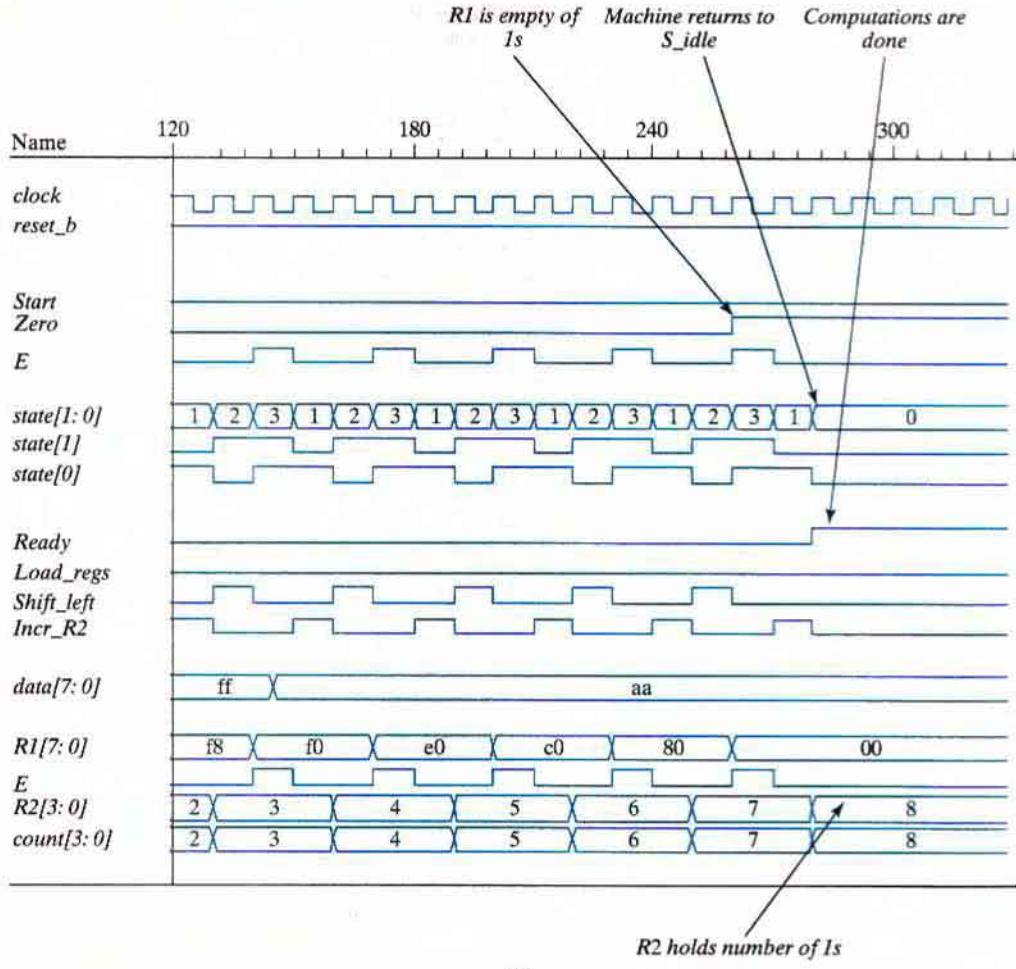


FIGURE 8.24
Simulation waveforms for count-of-ones circuit

8.11 RACE-FREE DESIGN

Once a circuit has been synthesized, either manually or with tools, it is necessary to verify that the simulation results produced by the HDL behavioral model match those of the netlist of the gates (standard cells) of the physical circuit. It is important to resolve any mismatch, because the behavioral model was presumed to be correct. There are various potential sources of mismatch between the results of a simulation, but we will consider one that typically happens in HDL-based design methodology. Three realities contribute to the potential problem: (1) A physical feedback path exists between a datapath unit and a control unit whose inputs include status signals fed back from the datapath unit; (2) blocked procedural assignments execute immediately, and behavioral models simulate with 0 propagation delays, effectively creating immediate changes in the outputs of combinational logic when its inputs change (i.e., changes in the inputs and the outputs are scheduled in the same time step of the simulation); and (3) the



(b)

FIGURE 8.24 (Continued)

order in which a simulator executes multiple blocked assignments to the same variable at a given time step of the simulation is indeterminate (i.e., unpredictable).

Now consider a sequential machine with an HDL model in which all assignments are made with the blocked assignment operator. At a clock pulse, the register operations in the datapath, the state transitions in the controller, the updates of the next state and output logic of the controller, and the updates to the status signals in the datapath are all scheduled to occur at the same time step of the simulation. Which executes first? Suppose that when a clock pulse occurs, the state of the controller changes before the register operations execute. The change in the state could change the outputs of the control unit. The new values of the outputs would be used by the datapath when it finally executes its assignments at that same clock pulse. The

result might not be the same as it would have been if the datapath had executed its assignments before the control unit updated its state and outputs. Conversely, suppose that when the clock pulse occurs, the datapath unit executes its operations and updates its status signals first. The updated status signals could cause a change in the value of the next state of the controller, which would be used to update the state. The result could differ from that which would result if the state had been updated before the edge-sensitive operations in the datapath executed. In either case, the timing of register transfer operations and state transitions in the different representations of the system might not match. Fortunately, there is a solution to this dilemma.

A designer can eliminate the *software race conditions* just described by observing the rule of modeling combinational logic with blocked assignments and modeling state transitions and edge-sensitive register operations with nonblocking assignments. A software race cannot happen if nonblocking operators are used as shown in all of the examples in this text, because the sampling mechanism of the nonblocking operator breaks the feedback path between a state transition or edge-sensitive datapath operation and the combinational logic that forms the next state or inputs to the registers in the datapath unit. The mechanism does this because simulators evaluate the expressions on the right-hand side of their nonblocking assignment statements before any blocked assignments are made. Thus, the nonblocking assignments cannot be affected by the results of the blocked assignments. In sum, always use the blocking operator to model combinational logic, and use the nonblocking operator to model edge-sensitive register operations and state transitions.

It also might appear that the physical structure of a datapath and the controller together create a physical (i.e., hardware), race condition, because the status signals are fed back to the controller and the outputs of the controller are fed forward to the datapath. However, timing analysis can verify that a change in the output of the controller will not propagate through the datapath logic and then through the input logic of the controller in time to have an effect on the output of the controller until the next clock pulse. The state cannot update until the next edge of the clock, even though the status signals update the value of the next state. The flip-flop cuts the feedback path between clock cycles. In practice, timing analysis verifies that the circuit will operate at the specified clock frequency, or it identifies signal paths whose propagation delays are problematic. Remember the design must implement the correct logic and operate at the speed prescribed by the clock.

8.12 LATCH-FREE DESIGN

Continuous assignments model combinational logic implicitly. A feedback-free continuous assignment will synthesize to combinational logic, and the input-output relationship of the logic is automatically sensitive to all of the inputs of the circuit. In simulation, the simulator monitors the right-hand sides of all continuous assignments, detects a change in any of the referenced variables, and updates the left-hand side of an affected assignment statement. Unlike a continuous assignment, a cyclic behavior is not necessarily completely sensitive to all of the variables that are referenced by its assignments statements. If a level-sensitive cyclic behavior is used to describe combinational logic, it is essential that the sensitivity list include every

variable that is referenced on the left-hand side of an assignment statement in the behavior. If the list is incomplete, the logic described by the behavior will be synthesized with latches at the outputs of the logic. This implementation wastes silicon area and may have a mismatch between the simulation of the behavioral model and the synthesized circuit. These difficulties can be avoided by ensuring that the sensitivity list is complete, but, in large circuits, it is easy to fail to include every referenced variable in the sensitivity list of a level-sensitive cyclic behavior. Consequently, Verilog 2001 included a new operator to reduce the risk of accidentally synthesizing latches.

In Verilog 2001, the tokens @ and * can be combined as @* or @(*) and are used without a sensitivity list to indicate that execution of the associated statement is sensitive to every variable that is referenced on the right-hand side of an assignment statement in the logic. In effect, the operator @* indicates that the logic is to be interpreted as level-sensitive combinational logic; the logic has an implicit sensitivity list composed of all of the variables that are referenced by the procedural assignments. Using the @* operator will prevent accidental synthesis of latches.

HDL Example 8.9

The following level-sensitive cyclic behavior will synthesize a two-channel multiplexer:

```
module mux_2_V2001 (output reg [31: 0] y, input [31: 0] a, b, input sel);
 always @*
 y = sel ? a: b;
endmodule
```

The cyclic behavior has an implicit sensitivity list consisting of *a*, *b*, and *sel*.

8.13 OTHER LANGUAGE FEATURES

The examples in this text have used only those features of the Verilog HDL that are appropriate for an introductory course in logic design. Verilog 2001 contains features that are very useful to designers, but which are not considered here. Among them are multidimensional arrays, variable part selects, array bit and part selects, signed reg, net, and port declarations, and local parameters. These enhancements are treated in more advanced texts using Verilog 2001 and Verilog 2005.

PROBLEMS

Answers to problems marked with * appear at the end of the book.

8.1* Explain in words and write HDL statements for the operations specified by the following register transfer notation:

(a) $R2 \leftarrow R2 + 1, R1 \leftarrow R$

(b) $R3 \leftarrow R3 - 1$

(c) If $(S_1 = 1)$ then $(R0 \leftarrow R1)$ else if $(S_2 = 1)$ then $(R0 \leftarrow R2)$

8.2 Draw (1) a block diagram showing the controller, datapath unit (with internal registers), and signals, and (2) the portion of an ASMD chart starting from an initial state. There are two control signals: x and y. If $xy = 01$, register R is incremented by 1 and control goes to a second state. If $xy = 10$, register R is cleared to zero and control goes from the initial state to a third state. Otherwise, control stays in the initial state. Assume active-low synchronous reset.

8.3 Draw the ASMD charts for the following state transitions:
(a) If $x = 1$, control goes from state S_1 to state S_2; if $x = 0$, generate a conditional operation $R <= R + 2$ and go from S_1 to S_2.
(b) If $x = 1$, control goes from S_1 to S_2 and then to S_3; if $x = 0$, control goes from S_1 to S_3.
(c) Start from state S_1; then if $xy = 00$, go to S_2; if $xy = 10$, go to S_3; and if $xy = 01$, go to S_1; otherwise, go to S_3.

8.4 Show the eight exit paths in an ASM block emanating from the decision boxes that check the eight possible binary values of three control variables x, y, and z.

8.5 Explain how the ASM and ASMD charts differ from a conventional flowchart. Using Fig. 8.5 as an illustration, show the difference in interpretation.

8.6 Construct a block diagram and an ASMD chart for a digital system that counts the number of people in a room. The one door through which people enter the room has a photocell that changes a signal x from 1 to 0 when the light is interrupted. They leave the room from a second door with a similar photocell that changes a signal y from 1 to 0 when the light is interrupted. The datapath circuit consists of an up–down counter with a display that shows how many people are in the room.

8.7* Draw a block diagram and an ASMD chart for a circuit with two eight-bit registers *RA* and *RB* that receive two unsigned binary numbers. The circuit performs the subtraction operation

$$RA \leftarrow RA - RB$$

Use the method for subtraction described in Section 1.5, and set a borrow flip-flop to 1 if the answer is negative. Write and verify an HDL model of the circuit.

8.8* Design a digital circuit with three 16-bit registers *AR*, *BR*, and *CR* that perform the following operations:
(a) Transfer two 16-bit signed numbers (in 2's-complement representation) to *AR* and *BR*.
(b) If the number in *AR* is negative, divide the number in *AR* by 2 and transfer the result to register *CR*.
(c) If the number in *AR* is positive but nonzero, multiply the number in *BR* by 2 and transfer the result to register *CR*.
(d) If the number in *AR* is zero, clear register *CR* to 0.
(e) Write and verify a behavioral model of the circuit.

8.9* Design the controller whose state diagram is given by Fig. 8.11(a). Use one flip-flop per state (a one-hot assignment). Write, simulate, verify, and compare RTL and structural models of the controller.

8.10 The state diagram of a control unit is shown in Fig. P8.10. It has four states and two inputs x and y. Draw the equivalent ASM chart. Write and verify a Verilog model of the controller.

8.11* Design the controller whose state diagram is shown in Fig. P8.10. Use *D* flip-flops.

8.12 Design the four-bit counter with synchronous clear specified in Fig. 8.10.

8.13 Simulate *Design_Example_STR* (see HDL Example 8.4), and verify that its behavior matches that of the RTL description. Obtain state information by displaying *G0* and *G1* as a concatenated vector for the state.

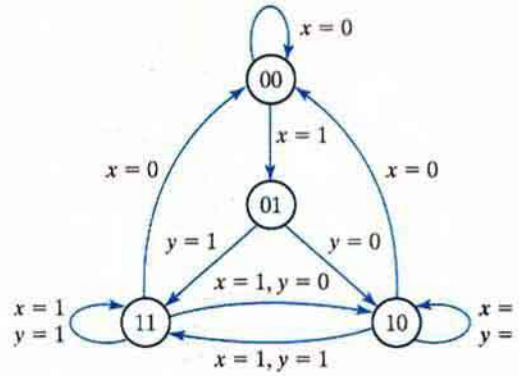


FIGURE P8.10
Control state diagram for Problems 8.10 and 8.11

8.14 What, if any, are the consequences of the machine in *Design_Example_RTL* (see HDL Example 8.2) entering an unused state?

8.15 Simulate *Design_Example_RTL*, and verify that it recovers from an unexpected reset condition during its operation, i.e., a "running reset" or a "reset on-the-fly."

8.16* Develop a block diagram and an ASMD chart for a digital circuit that multiplies two binary numbers by the repeated-addition method. For example, to multiply 5×4, the digital system evaluates the product by adding the multiplicand four times: $5 + 5 + 5 + 5 = 20$. Design the circuit. Let the multiplicand be in register *BR*, the multiplier in register *AR*, and the product in register *PR*. An adder circuit adds the contents of *BR* to *PR*. A zero-detection signal indicates whether *AR* is 0. Write and verify a Verilog behavioral model of the circuit.

8.17* Prove that the multiplication of two n-bit numbers gives a product of length less than or equal to $2n$ bits.

8.18* In Fig. 8.14, the Q register holds the multiplier and the B register holds the multiplicand. Assume that each number consists of 16 bits.
(a) How many bits can be expected in the product, and where is it available?
(b) How many bits are in the P counter, and what is the binary number loaded into it initially?
(c) Design the circuit that checks for zero in the P counter.

8.19 List the contents of registers C, A, Q, and P in a manner similar to Table 8.5 during the process of multiplying the two numbers 11011 (multiplicand) and 10111 (multiplier).

8.20* Determine the time it takes to process the multiplication operation in the binary multiplier described in Section 8.8. Assume that the Q register has n bits and the clock cycle is t nanoseconds.

8.21 Design the control circuit of the binary multiplier specified by the state diagram of Fig. 8.16, using multiplexers, a decoder, and a register.

8.22 Figure P8.22 shows an alternative ASMD chart for a sequential binary multiplier. Write and verify an RTL model of the system. Compare this design with that described by the ASMD chart in Fig. 8.15(b).

8.23 Figure P8.23 shows an alternative ASMD chart for a sequential binary multiplier. Write and verify an RTL model of the system. Compare this design with that described by the ASMD chart in Fig. 8.15(b).

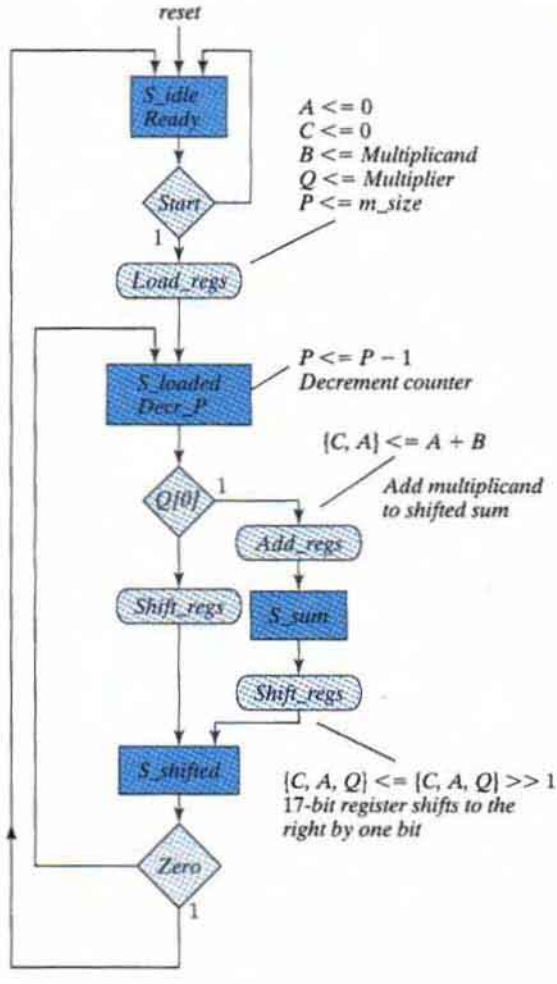


FIGURE P8.22
ASMD chart for Problem 8.22

8.24 The HDL description of a sequential binary multiplier given in HDL Example 8.5 encapsulates the descriptions of the controller and the datapath in a single Verilog module. Write and verify a model that encapsulates the controller and datapath in separate modules.

8.25 The sequential binary multiplier described by the ASMD chart in Fig. 8.15 does not consider whether the multiplicand or the shifted multiplier is 0. Therefore, it executes for a fixed number of clock cycles, independently of the data.

(a) Develop an ASMD chart for a more efficient multiplier that will terminate execution as soon as either word is found to be zero.

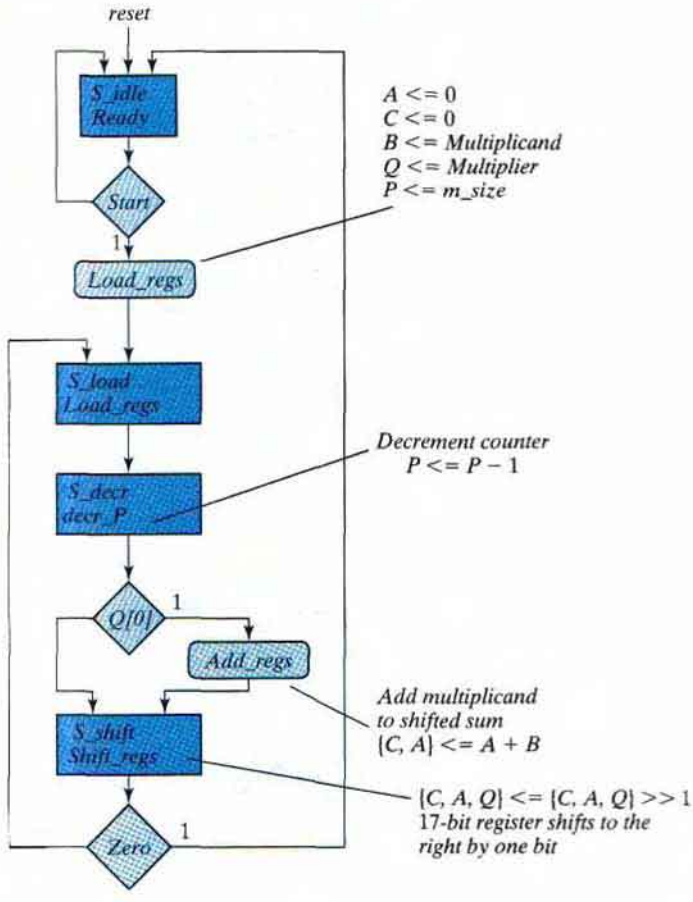


FIGURE P8.23
ASMD chart for Problem 8.23

(b) Write an HDL description of the circuit. The controller and datapath are to be encapsulated in separate Verilog modules.
(c) Write a test plan and a test bench, and verify the circuit.

8.26 Modify the ASMD chart of the sequential binary multiplier shown in Fig. 8.15 to add and shift in the same clock cycle. Write and verify an RTL description of the system.

8.27 The second test bench given in HDL Example 8.6 generates a product for all possible values of the multiplicand and multiplier. Verifying that each result is correct would not be practical, so modify the test bench to include a statement that forms the expected product. Write additional statements to compare the result produced by the RTL description with the expected result. Your simulation is to produce an error signal indicating the result of the comparison. Repeat for the structural model of the multiplier.

8.28 Write the HDL structural description of the multiplier designed in Section 8.8. Use the block diagram of Fig. 8.14(a) and the control circuit of Fig. 8.18. Simulate the design and verify its functionality by using the test bench of HDL Example 8.6.

8.29 An ASMD chart for a finite state machine is shown in Fig. P8.29. The register operations are not specified, because we are interested only in designing the control logic.

(a) Draw the equivalent state diagram.
(b) Design the control unit with one flip-flop per state.
(c) List the state table for the control unit.
(d) Design the control unit with three D flip-flops, a decoder, and gates.

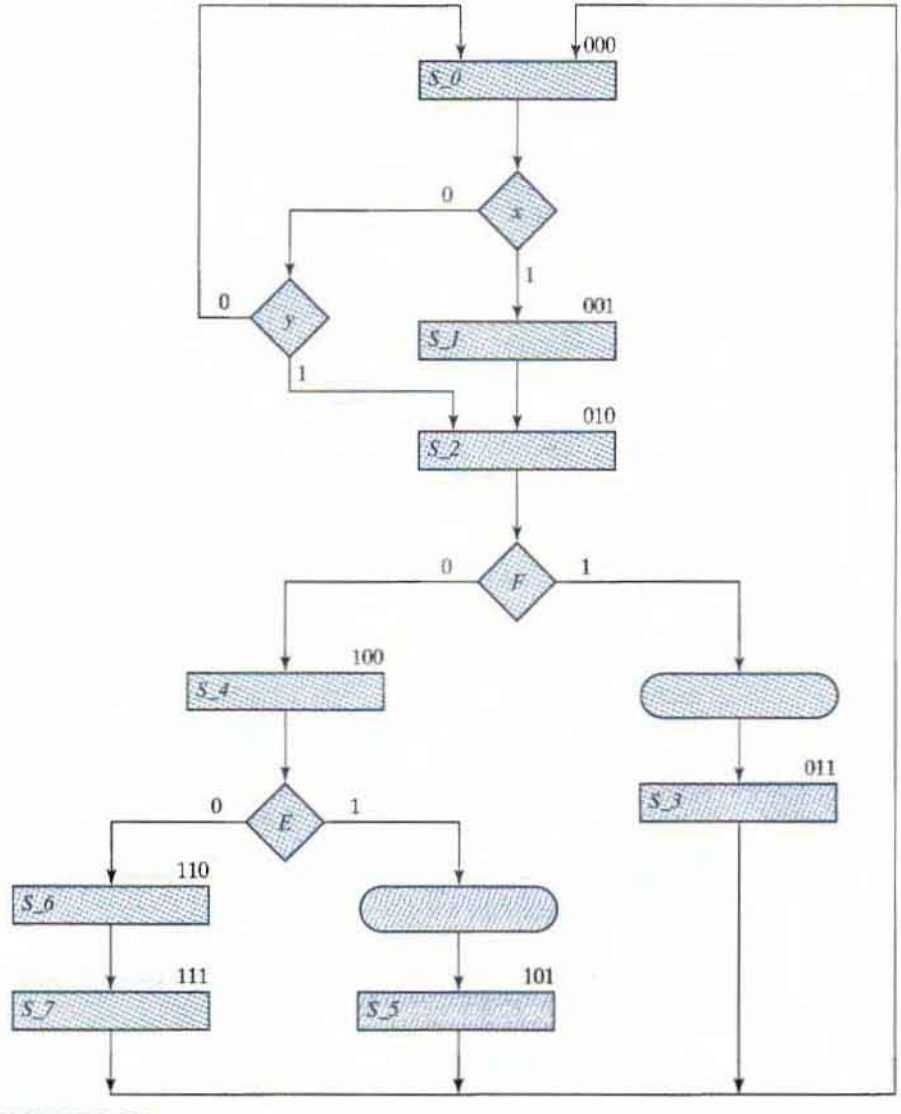


FIGURE P8.29
ASMD chart for Problem 8.29

(e) Derive a table showing the multiplexer input conditions for the control unit.
(f) Design the control unit with three multiplexers, a register with three flip-flops, and a 3×8 decoder.
(g) Using the results of (f), write and verify a structural model of the controller.
(h) Write and verify an RTL description of the controller.

8.30* What is the value of E in each HDL block, assuming that $RA = 1$?

(a)
```
RA = RA - 1;
if (RA == 0) E = 1;
else E = 0;
```

(b)
```
RA <= RA - 1;
if (RA == 0) E <= 1;
else E <= 0;
```

8.31* Using the Verilog HDL operators listed in Table 8.2, assume that $A = 4'b0110$, $B = 4'b0010$, and $C = 4'b0000$ and evaluate the result of the following operations:

```
A * B; A + B; A - B; ~C; A & B; A | B; A ^ B; & A; ~|C; A || B; A && C; |A; A < B; A > B;
A != B;
```

8.32 Consider the following always block:

```
always @ (posedge CLK)
  if (S1) R1 <= R1 + R2;
  else if (S2) R1 <= R1 + 1;
  else R1 <= R1;
```

Using a four-bit counter with parallel load for *R1* (as in Fig. 6.15) and a four-bit adder, draw a block diagram showing the connections of components and control signals for a possible synthesis of the block.

8.33 The multilevel case statement is often translated by a logic synthesizer into hardware multiplexers. How would you translate the following **case** block into hardware (assume registers of eight bits each)?

```
case (state)
  S0: R4 = R0;
  S1: R4 = R1;
  S2: R4 = R2;
  S3: R4 = R3;
endcase
```

8.34 The design of a circuit that counts the number of ones in a register is carried out in Section 8.10. The block diagram for the circuit is shown in Fig. 8.22(a), a complete ASMD chart for the circuit appears in Fig. 8.22(c), and structural HDL models of the datapath and controller are given in HDL Example 8.8. Using the operations and signal names indicated on the ASMD chart,

(a) Write *Datapath_BEH*, an RTL description of the datapath unit of the ones counter. Write a test plan specifying the functionality that will be tested, and write a test bench to implement the plan. Execute the test plan to verify the functionality of the datapath unit, and produce annotated simulation results relating the test plan to the waveforms produced in a simulation.
(b) Write *Controller_BEH*, an RTL description of the control unit of the ones counter. Write a test plan specifying the functionality that will be tested, and write a test bench to implement the plan. Execute the test plan to verify the functionality of the control unit, and produce annotated simulation results relating the test plan to the waveforms produced in a simulation.
(c) Write *Count_Ones_BEH_BEH*, a top-level module encapsulating and integrating *Controller_BEH* and *Datapath_BEH*. Write a test plan and a test bench, and verify the

description. Produce annotated simulation results relating the test plan to the waveforms produced in a simulation.

(d) Write *Controller_BEH_1Hot*, an RTL description of a one-hot controller implementing the ASMD chart of Fig. 8.22(c). Write a test plan specifying the functionality that will be tested, and write a test bench to implement the plan. Execute the test plan and produce annotated simulation results relating the test plan to the waveforms produced in a simulation.

(e) Write *Count_Ones_BEH_1_Hot*, a top-level module encapsulating the module *Controller_BEH_1_Hot* and *Datapath_BEH*. Write a test plan and a test bench, and verify the description. Produce annotated simulation results relating the test plan to the waveforms produced in a simulation.

8.35 The HDL description and test bench for a circuit that counts the number of ones in a register are given in HDL Example 8.8. Modify the test bench and simulate the circuit to verify that the system operates correctly for the following patterns of data: 8 8'hff, 8'h0f, 8'hf0, 8'h00, 8'haa, 8'h0a, 8'ha0, 8'h55, 8'h05, 8'hf50, 8'ha5, and 8'h5a.

8.36 The design of a circuit that counts the number of ones in a register is carried out in Section 8.10. The block diagram for the circuit is shown in Fig. 8.22(a), a complete ASMD chart for this circuit appears in Fig. 8.22(c), and structural HDL models of the datapath and controller are given in HDL Example 8.8. Using the operations and signal names indicated on the ASMD chart,

(a) Design the control logic, employing one flip-flop per state (a one-hot assignment). List the input equations for the four flip-flops.

(b) Write *Controller_Gates_1_Hot*, a gate-level HDL structural description of the circuit, using the control designed in part (a) and the signals shown in the block diagram of Fig. 8.22(a).

(c) Write a test plan and a test bench, and then verify the controller.

(d) Write *Count_Ones_Gates_1_Hot_STR*, a top-level module encapsulating and integrating instantiations of *Controller_Gates_1_Hot* and *Datapath_STR*. Write a test plan and a test bench to verify the description. Produce annotated simulation results relating the test plan to the waveforms produced in a simulation.

8.37 Compared with the circuit presented in HDL Example 8.8, a more efficient circuit that counts the number of ones in a data word is described by the block diagram and the partially completed ASMD chart in Fig. P8.37. This circuit accomplishes addition and shifting in the same clock cycle and adds the LSB of the data register to the counter register at every clock cycle.

(a) Complete the ASMD chart.

(b) Using the ASMD chart, write an RTL description of the circuit. A top-level Verilog module, *Count_of_ones_2_Beh* is to instantiate separate modules for the datapath and control units.

(c) Design the control logic, using one flip-flop per state (a one-hot assignment). List the input equations for the flip-flops.

(d) Write the HDL structural description of the circuit, using the controller designed in part (b) and the block diagram of Fig. P8.37(a).

(e) Write a test bench to test the circuit. Simulate the circuit to verify the operation described in both the RTL and the structural programs.

8.38 The addition of two signed binary numbers in the signed-magnitude representation follows the rules of ordinary arithmetic: If the two numbers have the same sign (both positive or both negative), the two magnitudes are added and the sum has the common sign; if the two numbers have opposite signs, the smaller magnitude is subtracted from the larger and the result has the sign of the larger magnitude. Write an HDL behavioral description for adding two 8-bit signed numbers in signed-magnitude representation and verify. The leftmost bit of the number holds the sign and the other seven bits hold the magnitude.

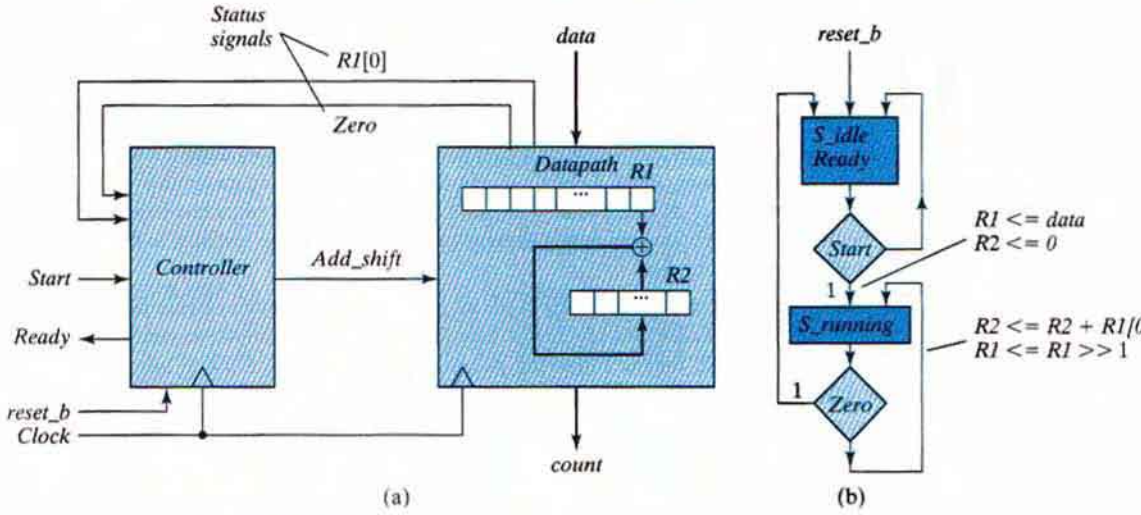


FIGURE P8.37
(a) Alternative circuit for a ones counter
(b) ASMD Chart for Problem 8.37

8.39* For the circuit designed in Problem 8.16,

(a) Write and verify a structural HDL description of the circuit. The datapath and controller are to be described in separate units.

(b) Write and verify an RTL description of the circuit. The datapath and controller are to be described in separate units.

8.40 Modify the block diagram of the sequential multiplier given in Fig. 8.14(a) and the ASMD chart in Fig. 8.15(b) to describe a system that multiplies 32-bit words, but with 8-bit (bytewide) external datapaths. The machine is to assert *Ready* in the (initial) reset state. When *Start* is asserted, the machine is to fetch the data bytes from a single 8-bit data bus in consecutive clock cycles (multiplicand bytes first, followed by multiplier bytes, least significant byte first) and store the data in datapath registers. *Got_Data* is to be asserted for one cycle of the clock when the transfer is complete. When *Run* is asserted, the product is to be formed sequentially. *Done_Product* is to be asserted for one clock cycle when the multiplication is complete. When a signal *Send_Data* is asserted, each byte of the product is to be placed on an 8-bit output bus for one clock cycle, in sequence, beginning with the least significant byte. The machine is to return to the initial state after the product has been transmitted. Consider safeguards, such as not attempting to send or receive data while the product is being formed. Consider also other features that might eliminate needless multiplication by 0. For example, do not continue to multiply if the shifted multiplier is empty of 1's.

8.41 The block diagram and partially completed ASMD chart in Fig. P8.41 describe the behavior of a two-stage pipeline that acts as a 2:1 decimator with a parallel input and output. Decimators are used in digital signal processors to move data from a datapath with a high clock rate to a datapath with a lower clock rate, converting data from a parallel format to a serial format in the process. In the datapath shown, entire words of data can be transferred into the pipeline at twice the rate at which the contents of the pipeline must be dumped into a holding register or consumed by some processor. The contents of the holding register *R0* can be shifted out serially, to accomplish

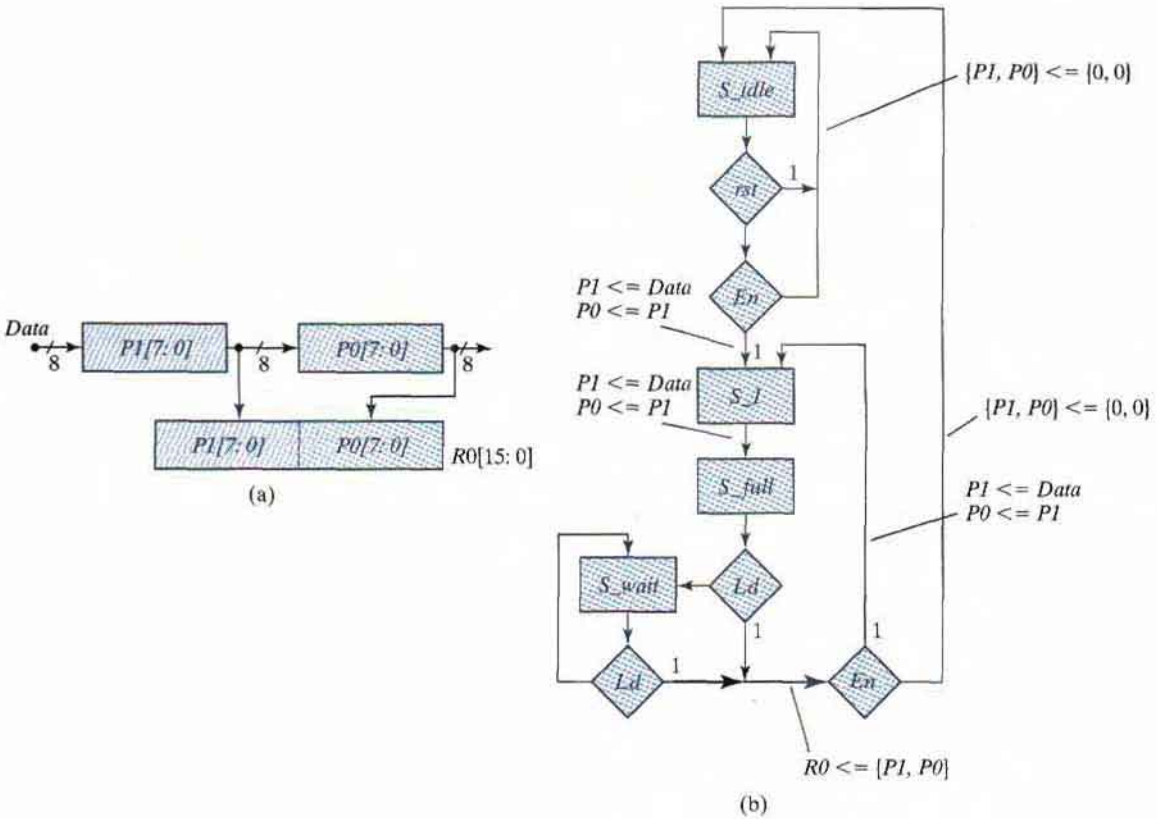


FIGURE P8.41
Two-stage pipeline register: Datapath unit and ASMD chart

an overall parallel-to-serial conversion of the data stream. The ASMD chart indicates that the machine has synchronous reset to *S_idle*, where it waits until *rst* is de-asserted and *En* is asserted. Note that synchronous transitions which would occur from the other states to *S_idle* under the action of *rst* are not shown. With *En* asserted, the machine transitions from *S_idle* to *S_1*, accompanied by concurrent register operations that load the MSByte of the pipe with *Data* and move the content of *P1* to the LSByte (*P0*). At the next clock, the state goes to *S_full*, and now the pipe is full. If *Ld* is asserted at the next clock, the machine moves to *S_1* while dumping the pipe into a holding register *R0*. If *Ld* is not asserted, the machine enters *S_wait* and remains there until *Ld* is asserted, at which time it dumps the pipe and returns to *S_1* or to *S_idle*, depending on whether *En* is asserted, too. The data rate at R_0 is one-half the rate at which data are supplied to the unit from an external datapath.

(a) Develop the complete ASMD chart.
(b) Using the ASMD chart developed in (a), write and verify an HDL model of the datapath.
(c) Write and verify a Verilog behavioral model of the control unit.
(d) Encapsulate the datapath and controller in a top-level module, and verify the integrated system.

REFERENCES

1. ARNOLD, M. G. 1999. *Verilog Digital Computer Design*. Upper Saddle River, NJ: Prentice Hall.
2. BHASKER, J. 1997. *A Verilog HDL Primer*. Allentown, PA: Star Galaxy Press.
3. BHASKER, J. 1998. *Verilog HDL Synthesis*. Allentown, PA: Star Galaxy Press.
4. CILETTI, M. D. 1999. *Modeling, Synthesis, and Rapid Prototyping with Verilog HDL*. Upper Saddle River, NJ: Prentice Hall.
5. CILETTI, M. D. 2003. *Modeling, Synthesis, and Rapid Prototyping with Verilog HDL*. Upper Saddle River, NJ: Prentice Hall.
6. CLARE, C. R. 1971. *Designing Logic Systems Using State Machines*. New York: McGraw-Hill.
7. HAYES, J. P. 1993. *Introduction to Digital Logic Design*. Reading, MA: Addison-Wesley.
8. *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language* (IEEE Std 1364-2001). 2001. New York: Institute of Electrical and Electronics Engineers.
9. MANO, M. M. 1993. *Computer System Architecture*, 3d ed. Upper Saddle River, NJ: Prentice Hall.
10. MANO, M. M., and C. R. KIME. 2000. *Logic and Computer Design Fundamentals*, 3rd ed. Upper Saddle River, NJ: Prentice Hall.
11. PALNITKAR, S. 2003. *Verilog HDL: A Guide to Digital Design and Synthesis*. Mountain View, CA: SunSoft Press (a Prentice Hall Title).
12. SMITH, D. J. 1996. *HDL Chip Design*. Madison, AL: Doone Publications.
13. THOMAS, D. E., and P. R. MOORBY. 2002. *The Verilog Hardware Description Language*, 5th ed. Boston: Kluwer Academic Publishers.
14. WINKLER, D., and F. PROSSER. 1987. *The Art of Digital Design*, 2d ed. Englewood Cliffs, NJ: Prentice-Hall.