chapter

7

DIGITAL SYSTEM DESIGN

CHAPTER OBJECTIVES

In this chapter you will learn about aspects of digital system design, including

- Bus structure
- A simple processor
- Usage of ASM charts
- Clock synchronization and timing issues
- Flip-flop timing at the chip level

In the previous chapters we showed how to design many types of simple circuits, such as multiplexers, decoders, flip-flops, registers, and counters, which can be used as building blocks. In this chapter we provide examples of more complex circuits that can be constructed using the building blocks as subcircuits. Such larger circuits form a *digital system*. For practical reasons our examples of digital systems will not be large, but the design techniques presented are applicable to systems of any size.

A digital system consists of two main parts, called the datapath circuit and the control circuit. The *datapath circuit* is used to store and manipulate data and to transfer data from one part of the system to another. Datapath circuits comprise building blocks such as registers, shift registers, counters, multiplexers, decoders, adders, and so on. The *control circuit* controls the operation of the datapath circuit. In Chapter 6 we referred to the control circuits as finite state machines. We will give several examples of digital systems and show how to design their datapath and control circuits.

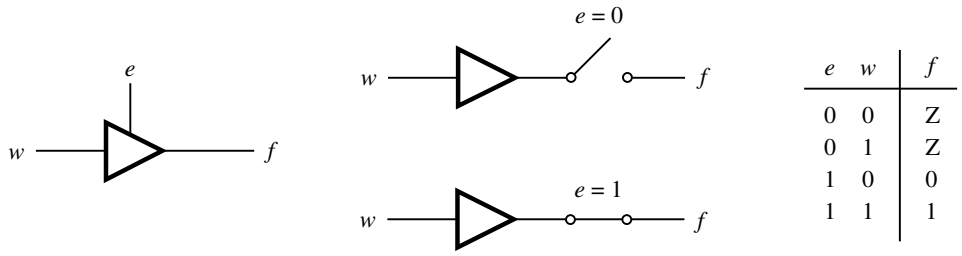
7.1 BUS STRUCTURE

Digital systems often contain a set of registers used to store data. As already discussed in Chapter 6, such registers can be interconnected via an interconnection network as illustrated in Figure 6.10. There are many ways of implementing an interconnection network. In this section we will present one of the commonly used schemes.

Suppose that a digital system has a number of n-bit registers, and that it must be possible to transfer data from any register to any other register. A simple way of providing the desired interconnectivity may be to connect each register to a common set of n wires, which are used to transfer data into and out of the registers. This common set of wires is usually called a *bus*. If common paths are used to transfer data from multiple sources to multiple destinations, it is necessary to ensure that only one register acts as a source at any given time and other registers do not interfere. We will present two plausible arrangements for implementing the bus structure.

7.1.1 USING TRI-STATE DRIVERS TO IMPLEMENT A BUS

Outputs of two ordinary logic gates cannot be connected together, because a short circuit would result if one gate forces the output value 1 while the other gate forces the value 0. Therefore, some special gates are needed if the outputs of two registers are to be connected to a common set of wires. A commonly used circuit element for this purpose is shown in Figure 7.1*a*. It has a data input w, an output f, and an enable input e. Its operation is illustrated by the equivalent circuit in part (*b*). The triangular symbol in the figure represents a noninverting driver, which is a circuit that performs no logic operation and its output simply replicates the input signal. Its purpose is to provide additional electrical driving capability. In conjunction with the output switch, it behaves as indicated in Figure 7.1*c*. When $e = 1$, the output reflects the logic value on the data input. But, when $e = 0$, the output is electrically disconnected from the data input, which is referred to as a *high impedance* state and it is usually denoted by the letter Z (or z). Because this circuit element exhibits three distinct states, 0, 1, and Z, it is called a *tri-state* driver (or buffer). Appendix B explains how it can be implemented with transistors.



(a) Symbol (b) Equivalent circuit (c) Truth table

Figure 7.1 Tri-state driver.

Consider a system that contains k n-bit registers, $R1$ to Rk. Figure 7.2 shows how these registers can be connected using tri-state drivers to implement the bus structure. The data outputs of each register are connected to tri-state drivers. When selected by their enable signals, the drivers place the contents of the corresponding register onto the bus wires. We showed in Figure 5.56 how an enable input can be added to a register. If the enable input is set to 1, then the contents of the register will be changed on the next active edge of the clock. In Figure 7.2 the enable input on each register is labeled L, which stands for *load*. The signal that controls the load input for a register Rj is denoted as Rj_{in}, while the signal that controls the associated tri-state driver is called Rj_{out}. These signals are generated by a control circuit.

In addition to registers, in a real system other types of circuit blocks would be connected to the bus. The figure shows how n bits of data from an external source can be placed on the bus, using the control input *Extern*.

It is essential to ensure that only one circuit block attempts to place data onto the bus wires at any given time. The control circuit must ensure that only one of the tri-state driver enable signals, $R1_{out}, \ldots, Rk_{out}$, is asserted at a given time. The control circuit also produces the signals $R1_{in}, \ldots, Rk_{in}$, which determine when data is loaded into each register. In general, the control circuit could perform a number of functions, such as transferring the data stored in one register into another register and controlling the processing of data in various functional units of the system. Figure 7.2 shows an input signal named *Function* that instructs the control circuit to perform a particular task. The control circuit is synchronized by a clock input, which is the same clock signal that controls the k registers.

Figure 7.3 provides a more detailed view of how the registers from Figure 7.2 can be connected to a bus. To keep the picture simple, 2 two-bit registers are shown, but the same scheme can be used for larger registers. For register $R1$, two tri-state drivers enabled by $R1_{out}$ are used to connect each flip-flop output to a wire in the bus. The D input on each flip-flop is connected to a 2-to-1 multiplexer, whose select input is controlled by $R1_{in}$. If $R1_{in} = 0$, the flip-flops are loaded from their Q outputs; hence the stored data does not change. But if $R1_{in} = 1$, data is loaded into the flip-flops from the bus.

The system in Figure 7.2 can be used in many different ways, depending on the design of the control circuit and on how many registers and other circuit blocks are connected to

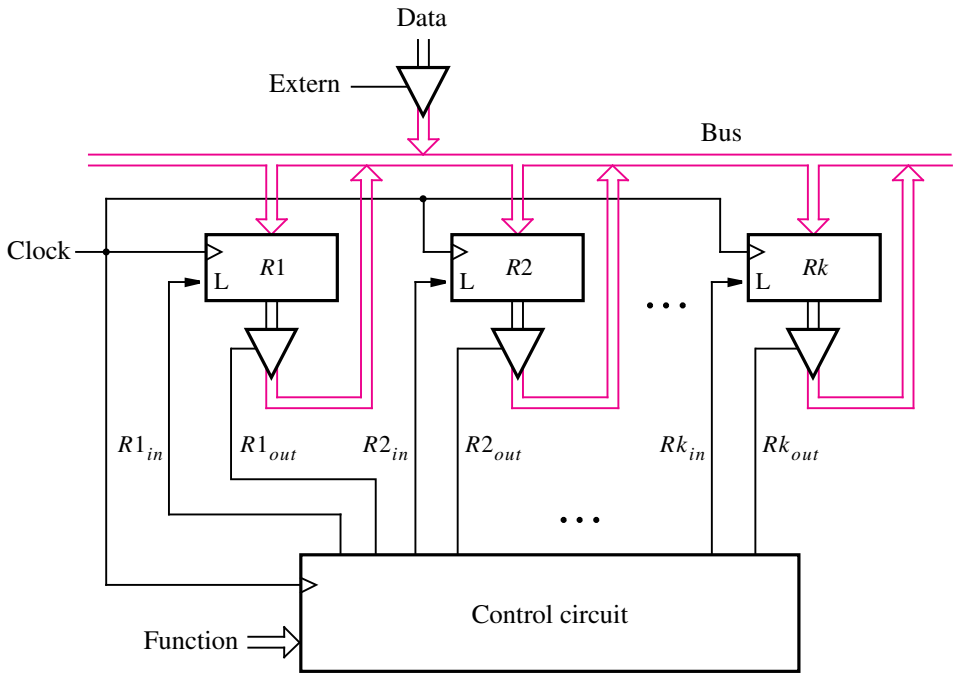


Figure 7.2 A digital system with k registers.

the bus. As a simple example, consider a system that has three registers, $R1$, $R2$, and $R3$. We will specify a control circuit that performs a single function—it swaps the contents of registers $R1$ and $R2$, using $R3$ for temporary storage.

The required swapping is done in three steps, each needing one clock cycle. In the first step the contents of $R2$ are transferred into $R3$. Then the contents of $R1$ are transferred into $R2$. Finally, the contents of $R3$, which are the original contents of $R2$, are transferred into $R1$. We have already designed the control circuit for this task in Example 6.1, in the form of a finite state machine.

Note that we say that the contents of one register, R_i, are "transferred" into another register, R_j. This jargon is commonly used to indicate that the new contents of R_j will be a copy of the contents of R_i. The contents of R_i are not changed as a result of the transfer. Therefore, it would be more precise to say that the contents of R_i are "copied" into R_j.

7.1.2 USING MULTIPLEXERS TO IMPLEMENT A BUS

In Figure 7.2 we used tri-state drivers to control access to the bus. An alternative approach is to use multiplexers, as depicted in Figure 7.4. The outputs of each register are connected to a multiplexer. This multiplexer's output is connected to the inputs of the registers, thus realizing the bus. The multiplexer select inputs determine which register's contents appear

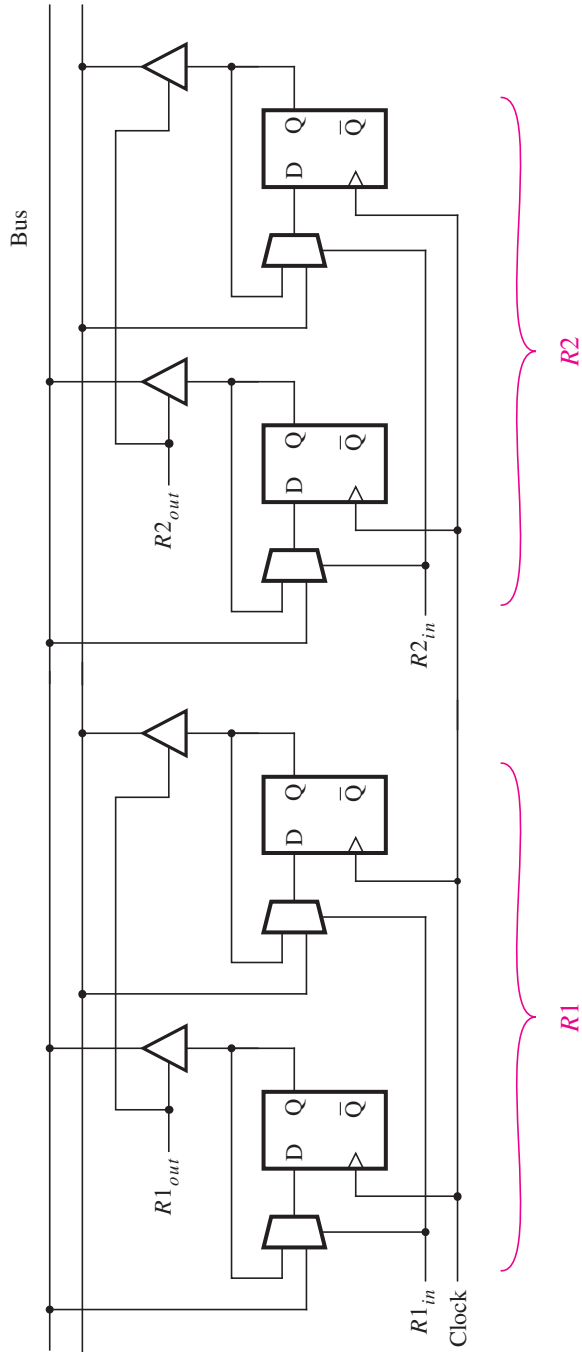


Figure 7.3 Details for connecting registers to a bus.

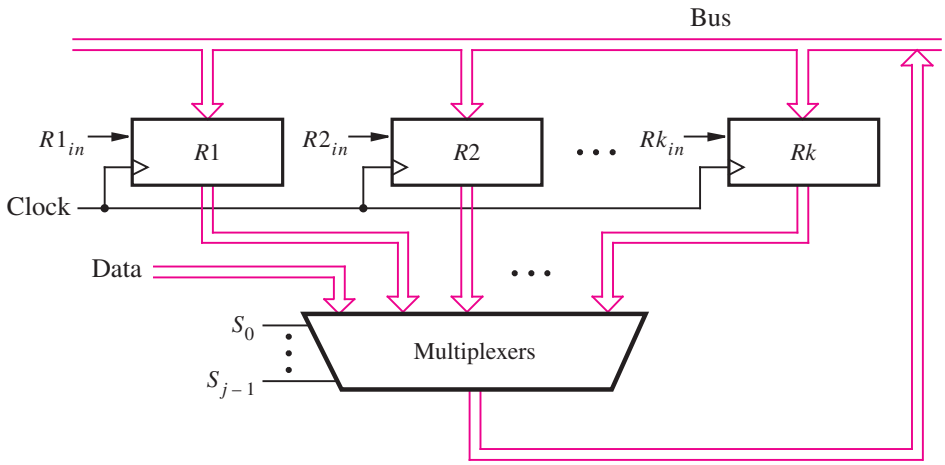


Figure 7.4 Using multiplexers to implement a bus.

on the bus. Although the figure shows just one multiplexer symbol, we actually need one multiplexer for each bit in the registers. For example, assume that there are 4 eight-bit registers, $R1$ to $R4$, plus the externally-supplied eight-bit *Data*. To interconnect them, we need eight 5-to-1 multiplexers.

The control circuit can be designed as a finite state machine as mentioned in the previous subsection. However, instead of using the control signals Rj_{out} to place the contents of register Rj onto the bus, we have to generate the select inputs for the multiplexers.

The tri-state driver and multiplexer approaches for implementing a bus are both equally valid. However, some types of chips, such as FPGAs, do not contain a sufficient number of tri-state drivers to realize even moderately large buses. In such chips the multiplexer-based approach is the only practical alternative. In practice, circuits are designed with CAD tools. If the designer describes the circuit using tri-state drivers, but there are not enough such drivers in the target device, then the CAD tools automatically produce an equivalent circuit that uses multiplexers.

7.1.3 VERILOG CODE FOR SPECIFICATION OF BUS STRUCTURES

This section presents Verilog code for our circuit example that swaps the contents of two registers. We first give the code for the style of circuit in Figure 7.2 that uses tri-state drivers to implement the bus and then give the code for the style of circuit in Figure 7.4 that uses multiplexers. Figure 7.5 gives the code for an n-bit register of the type in Figure 7.3. The number of bits in the register is set by the parameter n, which has the default value of 8. The register is specified such that if the input $L = 1$, then the flip-flops are loaded from the n-bit input R. Otherwise, the flip-flops retain their presently stored values.

Figure 7.6 gives the code for a subcircuit that represents n tri-state drivers, each enabled by the input E. The inputs to the drivers are the n-bit signal Y, and the outputs are the n-bit

```
module regn (R, L, Clock, Q);
   parameter n = 8;
   input [n–1:0] R;
   input L, Clock;
   output reg [n–1:0] Q;

   always @(posedge Clock)
      if (L)
         Q <= R;

endmodule
```

Figure 7.5 Code for an n-bit register of the type in Figure 7.3.

```
module trin (Y, E, F);
   parameter n = 8;
   input [n–1:0] Y;
   input E;
   output wire [n–1:0] F;

   assign F = E ? Y : 'bz;

endmodule
```

Figure 7.6 Code for an n-bit tri-state module.

signal F. The conditional assignment statement specifies that the output of each driver is set to $F = Y$ if $E = 1$; otherwise, the output is set to the high impedance value z. The conditional assignment statement uses an unsized number to define the high impedance case. The Verilog compiler will make the size of this number the same as the size of vector Y, namely n. We cannot define the number as n'bz because the size of a sized number cannot be given as a parameter.

The code in Figure 7.7 represents a digital system like the one in Figure 7.2, with 3 eight-bit registers, $R1$, $R2$, and $R3$. The circuit in Figure 7.2 includes tri-state drivers that are used to place n bits of externally supplied *Data* on the bus. In Figure 7.7, these drivers are instantiated in the module *tri_ext*. Each of the eight drivers is enabled by the input signal *Extern*, and the data inputs on the drivers are attached to the eight-bit signal *Data*. When *Extern* $= 1$, the value of *Data* is placed on the bus, which is represented by the signal *BusWires*. The *BusWires* vector represents the circuit's output as well as the internal bus wiring. We declared this vector to be of **tri** type rather than of **wire** type. The keyword **tri** is treated in the same way as the keyword **wire** by the Verilog compiler. The designation **tri** makes it obvious to a reader that the synthesized connections will have tri-state capability.

```
module swap (Resetn, Clock, w, Data, Extern, RinExt1, RinExt2, RinExt3, BusWires, Done);
   parameter n = 8;
   input Resetn, Clock, w, Extern, RinExt1, RinExt2, RinExt3;
   input [n–1:0] Data;
   output tri [n–1:0] BusWires;
   output Done;
   wire [n–1:0] R1, R2, R3;
   wire R1in, R1out, R2in, R2out, R3in, R3out;
   reg [2:1] y, Y;
   parameter [2:1] A = 2'b00, B = 2'b01, C = 2'b10, D = 2'b11;

   // Define the next state combinational circuit for FSM
   always @(w, y)
      case (y)
         A: if (w)    Y = B;
            else      Y = A;
         B:           Y = C;
         C:           Y = D;
         D:           Y = A;
      endcase

   // Define the sequential block for FSM
   always @(negedge Resetn, posedge Clock)
      if (Resetn == 0)   y < = A;
      else   y < = Y;

   // Define outputs of FSM
   assign R2out = (y == B);
   assign R3in = (y == B);
   assign R1out = (y == C);
   assign R2in = (y == C);
   assign R3out = (y == D);
   assign R1in = (y == D);
   assign Done = (y == D);

   // Instantiate registers
   regn reg_1 (BusWires, RinExt1 | R1in, Clock, R1);
   regn reg_2 (BusWires, RinExt2 | R2in, Clock, R2);
   regn reg_3 (BusWires, RinExt3 | R3in, Clock, R3);
   // Instantiate tri-state drivers
   trin tri_ext (Data, Extern, BusWires);
   trin tri_1 (R1, R1out, BusWires);
   trin tri_2 (R2, R2out, BusWires);
   trin tri_3 (R3, R3out, BusWires);
endmodule
```

Figure 7.7 A digital system like the one in Figure 7.2.

We assume that three control signals named *RinExt1*, *RinExt2*, and *RinExt3* exist, which allow the externally supplied data to be loaded from the bus into register *R*1, *R*2, or *R*3. These signals are not shown in Figure 7.2, to keep the figure simple, but they would be generated by the same external circuit block that produces *Extern* and *Data*. When *RinExt1* = 1, the data on the bus is loaded into register *R*1; when *RinExt2* = 1, the data is loaded into *R*2; and when *RinExt3* = 1, the data is loaded into *R*3.

The FSM that implements the control circuit for our swapping task example is included in Figure 7.7. Since our control circuit performs only one operation, we have not included the *Function* input from Figure 7.2. We have assumed that an input signal named *w* exists, which is set to 1 for one clock cycle to start the swapping task. We have also assumed that at the end of the swapping task, which is indicated by the *Done* signal being asserted, the control circuit returns to the starting state.

Verilog Code Using Multiplexers

Figure 7.8 shows how the code in Figure 7.7 can be modified to use multiplexers instead of tri-state drivers. Using the circuit structure shown in Figure 7.4, the bus is implemented with eight 4-to-1 multiplexers. Three of the data inputs on each 4-to-1 multiplexer are connected to one bit from registers *R*1, *R*2, and *R*3. The fourth data input is connected to one bit of the *Data* input signal to allow externally supplied data to be written into the registers.

The same FSM control circuit is used. However, the control signals *R*1*out*, *R*2*out*, and *R*3*out* are not needed because tri-state drivers are not used. Instead, the required multiplexers are defined in an **if-else** statement by specifying the source of data based on the state of the FSM. Hence, when the FSM is in state *A*, the selected input to the multiplexers is *Data*. When the state is *B*, the register *R*2 provides the input data to the multiplexers, and so on.

7.2 SIMPLE PROCESSOR

A second example of a digital system like the one in Figure 7.2 is shown in Figure 7.9. It has four *n*-bit registers, *R*0, . . . , *R*3, that are connected to the bus with tri-state drivers. External data can be loaded into the registers from the *n*-bit *Data* input, which is connected to the bus using tri-state drivers enabled by the *Extern* control signal. The system also includes an adder/subtractor module. One of its data inputs is provided by an *n*-bit register, *A*, that is attached to the bus, while the other data input, *B*, is directly connected to the bus. If the *AddSub* signal has the value 0, the module generates the sum $A + B$; if $AddSub = 1$, the module generates the difference $A - B$. To perform the subtraction, we assume that the adder/subtractor includes the required XOR gates to form the 2's complement of *B*, as discussed in Section 3.3. The register *G* stores the output produced by the adder/subtractor. The *A* and *G* registers are controlled by the signals A_{in}, G_{in}, and G_{out}.

The system in Figure 7.9 can perform various functions, depending on the design of the control circuit. As an example, we will design a control circuit that can perform the four operations listed in Table 7.1. The left column in the table shows the name of an operation and its operands; the right column indicates the function performed in the operation. For

```
module swapmux (Resetn, Clock, w, Data, RinExt1, RinExt2, RinExt3, BusWires, Done);
    parameter n = 8;
    input Resetn, Clock, w, RinExt1, RinExt2, RinExt3;
    input [n–1:0] Data;
    output reg [n–1:0] BusWires;
    output Done;
    wire [n–1:0] R1, R2, R3;
    wire R1in, R2in, R3in;
    reg [2:1] y, Y;
    parameter [2:1] A = 2'b00, B = 2'b01, C = 2'b10, D = 2'b11;

    // Define the next state combinational circuit for FSM
    always @(w, y)
        case (y)
            A: if (w)    Y = B;
               else      Y = A;
            B:           Y = C;
            C:           Y = D;
            D:           Y = A;
        endcase

    // Define the sequential block for FSM
    always @(negedge Resetn, posedge Clock)
        if (Resetn == 0)   y < = A;
        else   y < = Y;

    // Define control signals
    assign R3in = (y == B);
    assign R2in = (y == C);
    assign R1in = (y == D);
    assign Done = (y == D);

    // Instantiate registers
    regn reg_1 (BusWires, RinExt1 | R1in, Clock, R1);
    regn reg_2 (BusWires, RinExt2 | R2in, Clock, R2);
    regn reg_3 (BusWires, RinExt3 | R3in, Clock, R3);

    // Define the multiplexers
    always @(y, Data, R1, R2, R3)
        if (y == A) BusWires = Data;
        else if (y == B) BusWires = R2;
        else if (y == C) BusWires = R1;
        else BusWires = R3;
endmodule
```

Figure 7.8 Using multiplexers to implement the bus structure.

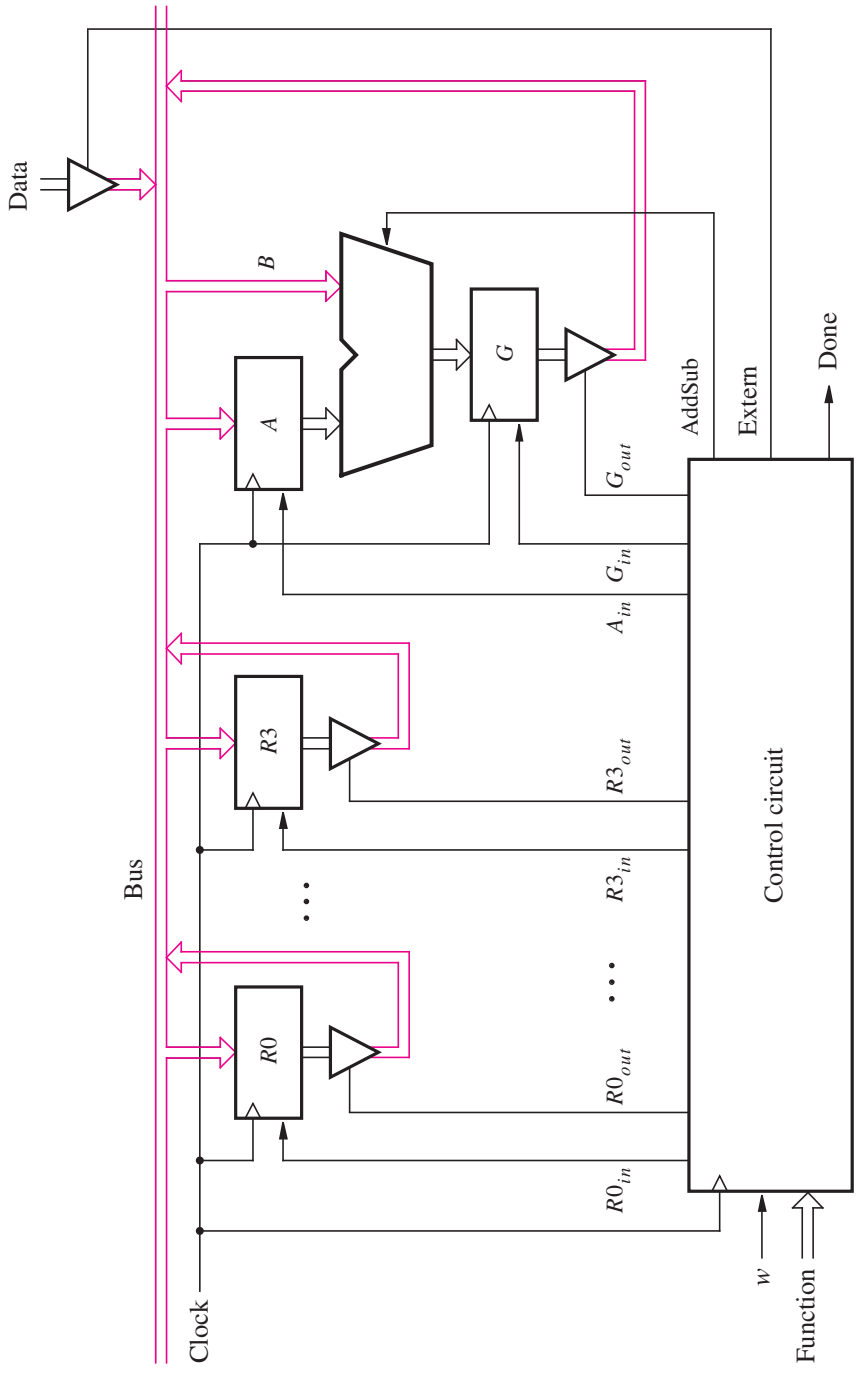


Figure 7.9 A digital system that implements a simple processor.

Table 7.1 Operations performed in the processor.

Operation	Function performed
Load *Rx*, *Data*	$Rx \leftarrow Data$
Move *Rx*, *Ry*	$Rx \leftarrow [Ry]$
Add *Rx*, *Ry*	$Rx \leftarrow [Rx] + [Ry]$
Sub *Rx*, *Ry*	$Rx \leftarrow [Rx] - [Ry]$

the *Load* operation the meaning of $Rx \leftarrow Data$ is that the data on the external *Data* input is transferred across the bus into any register, *Rx*, where *Rx* can be *R*0 to *R*3. The *Move* operation copies the data stored in register *Ry* into register *Rx*. In the table the square brackets, as in [*Rx*], refer to the *contents* of a register. Since only a single transfer across the bus is needed, both the *Load* and *Move* operations require only one step (clock cycle) to be completed. The *Add* and *Sub* operations require three steps, as follows: In the first step the contents of *Rx* are transferred across the bus into register *A*. Then in the next step, the contents of *Ry* are placed onto the bus. The adder/subtractor module performs the required function, and the results are stored in register *G*. Finally, in the third step the contents of *G* are transferred into *Rx*.

A digital system that performs the types of operations listed in Table 7.1 is usually called a *processor*. The specific operation to be performed at any given time is indicated using the control circuit input named *Function*. The operation is initiated by setting the *w* input to 1, and the control circuit asserts the *Done* output when the operation is completed.

In Figure 7.2 we used an FSM to implement the control circuit. It is possible to use a similar design for the system in Figure 7.9. To illustrate another approach, we will base the design of the control circuit on a counter. This circuit has to generate the required control signals in each step of each operation. Since the longest operations (*Add* and *Sub*) need three steps (clock cycles), a two-bit counter can be used. Figure 7.10 shows a two-bit up-counter connected to a 2-to-4 decoder. Decoders are discussed in Chapter 4. The decoder is enabled at all times by setting its enable (*En*) input permanently to the value 1. Each of the decoder outputs represents a step in an operation. When no operation is currently being performed, the count value is 00; hence the T_0 output of the decoder is asserted. In the first step of an operation, the count value is 01, and T_1 is asserted. During the second and third steps of the *Add* and *Sub* operations, T_2 and T_3 are asserted, respectively.

In each of steps T_0 to T_3, various control signal values have to be generated by the control circuit, depending on the operation being performed. Figure 7.11 shows that the operation is specified with six bits, which form the *Function* input. The two left-most bits, $F = f_1 f_0$, are used as a two-bit number that identifies the operation. To represent *Load*, *Move*, *Add*, and *Sub*, we use the codes $f_1 f_0 = 00, 01, 10$, and 11, respectively. The inputs $Rx_1 Rx_0$ are a binary number that identifies the *Rx* operand, while $Ry_1 Ry_0$ identifies the *Ry* operand. The *Function* inputs are stored in a six-bit Function Register when the FR_{in} signal is asserted.

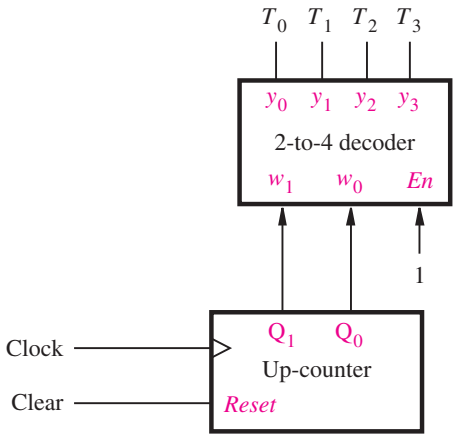


Figure 7.10 A part of the control circuit for the processor.

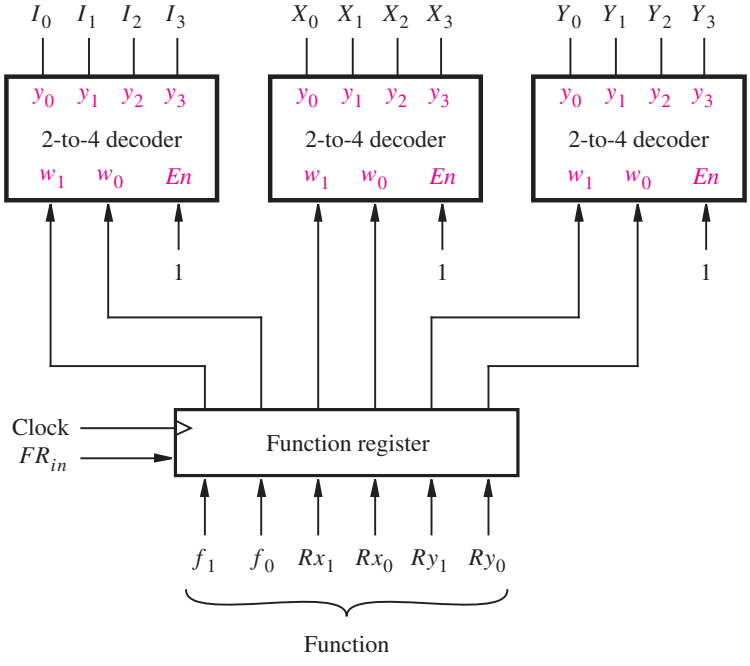


Figure 7.11 The function register and decoders.

Figure 7.11 also shows three 2-to-4 decoders that are used to decode the information encoded in the F, Rx, and Ry inputs. We will see shortly that these decoders are included as a convenience because their outputs provide simple-looking logic expressions for the various control signals.

The circuits in Figures 7.10 and 7.11 form a part of the control circuit. Using the input w and the signals $T_0, \ldots, T_3$, $I_0, \ldots, I_3$, $X_0, \ldots, X_3$, and $Y_0, \ldots, Y_3$, we will show how to derive the rest of the control circuit. It has to generate the outputs *Extern*, *Done*, A_{in}, G_{in}, G_{out}, *AddSub*, $R0_{in}, \ldots, R3_{in}$, and $R0_{out}, \ldots, R3_{out}$. The control circuit also has to generate the *Clear* and FR_{in} signals used in Figures 7.10 and 7.11.

Clear and FR_{in} are defined in the same way for all operations. *Clear* is used to ensure that the count value remains at 00 as long as $w = 0$ and no operation is being executed. Also, it is used to clear the count value to 00 at the end of each operation. Hence an appropriate logic expression is

$$Clear = \overline{w}\, T_0 + Done$$

The FR_{in} signal is used to load the values on the *Function* inputs into the Function Register when w changes to 1. Hence

$$FR_{in} = wT_0$$

The rest of the outputs from the control circuit depend on the specific step being performed in each operation. The values that have to be generated for each signal are shown in Table 7.2. Each row in the table corresponds to a specific operation, and each column represents one time step. The *Extern* signal is asserted only in the first step of the *Load* operation. Therefore, the logic expression that implements this signal is

$$Extern = I_0 T_1$$

Done is asserted in the first step of *Load* and *Move*, as well as in the third step of *Add* and *Sub*. Hence

$$Done = (I_0 + I_1)T_1 + (I_2 + I_3)T_3$$

Table 7.2 Control signals asserted in each operation/time step.

	T_1	T_2	T_3
(Load): I_0	*Extern*, $R_{in} = X$, *Done*		
(Move): I_1	$R_{in} = X$, $R_{out} = Y$, *Done*		
(Add): I_2	$R_{out} = X$, A_{in}	$R_{out} = Y$, G_{in}, $AddSub = 0$	G_{out}, $R_{in} = X$, *Done*
(Sub): I_3	$R_{out} = X$, A_{in}	$R_{out} = Y$, G_{in}, $AddSub = 1$	G_{out}, $R_{in} = X$, *Done*

The A_{in}, G_{in}, and G_{out} signals are asserted in the *Add* and *Sub* operations. A_{in} is asserted in step T_1, G_{in} is asserted in T_2, and G_{out} is asserted in T_3. The *AddSub* signal has to be set to 0 in the *Add* operation and to 1 in the *Sub* operation. This is achieved with the following logic expressions

$$\begin{aligned} A_{in} &= (I_2 + I_3)T_1 \\ G_{in} &= (I_2 + I_3)T_2 \\ G_{out} &= (I_2 + I_3)T_3 \\ AddSub &= I_3 \end{aligned}$$

The values of $R0_{in}, \ldots, R3_{in}$ are determined using either the $X_0, \ldots, X_3$ signals or the $Y_0, \ldots, Y_3$ signals. In Table 7.2 these actions are indicated by writing either $R_{in} = X$ or $R_{in} = Y$. The meaning of $R_{in} = X$ is that $R0_{in} = X_0$, $R1_{in} = X_1$, and so on. Similarly, the values of $R0_{out}, \ldots, R3_{out}$ are specified using either $R_{out} = X$ or $R_{out} = Y$.

We will develop the expressions for $R0_{in}$ and $R0_{out}$ by examining Table 7.2 and then show how to derive the expressions for the other register control signals. The table shows that $R0_{in}$ is set to the value of X_0 in the first step of both the *Load* and *Move* operations and in the third step of both the *Add* and *Sub* operations, which leads to the expression

$$R0_{in} = (I_0 + I_1)T_1X_0 + (I_2 + I_3)T_3X_0$$

Similarly, $R0_{out}$ is set to the value of Y_0 in the first step of *Move*. It is set to X_0 in the first step of *Add* and *Sub* and to Y_0 in the second step of these operations, which gives

$$R0_{out} = I_1T_1Y_0 + (I_2 + I_3)(T_1X_0 + T_2Y_0)$$

The expressions for $R1_{in}$ and $R1_{out}$ are the same as those for $R0_{in}$ and $R0_{out}$ except that X_1 and Y_1 are used in place of X_0 and Y_0. The expressions for $R2_{in}$, $R2_{out}$, $R3_{in}$, and $R3_{out}$ are derived in the same way.

The circuits shown in Figures 7.10 and 7.11, combined with the circuits represented by the above expressions, implement the control circuit in Figure 7.9.

Processors are extremely useful circuits that are widely used. We have presented only the most basic aspects of processor design. However, the techniques presented can be extended to design realistic processors, such as modern microprocessors. The interested reader can refer to books on computer organization for more details on processor design [1–2].

Verilog Code

In this section we give two different styles of Verilog code for describing the system in Figure 7.9. The first style uses tri-state drivers to represent the bus, and it gives the logic expressions shown above for the outputs of the control circuit. The second style of code uses multiplexers to represent the bus, and it uses **case** statements that correspond to Table 7.2 to describe the outputs of the control circuit.

Verilog code for a two-bit up-counter, named *upcount*, is shown in Figure 7.12. It has a synchronous reset input, which is active high. Other subcircuits that we use in the Verilog code for the processor are the *dec2to4*, *regn*, and *trin* modules in Figures 4.31, 7.5, and 7.6.

```
module upcount (Clear, Clock, Q);
   input Clear, Clock;
   output reg [1:0] Q;

   always @(posedge Clock)
      if (Clear)
         Q <= 0;
      else
         Q <= Q + 1;

endmodule
```

Figure 7.12 A two-bit up-counter with synchronous reset.

Complete code for the processor is given in Figure 7.13. The instantiated modules *counter* and *decT* represent the subcircuits in Figure 7.10. Note that we have assumed that the circuit has an active-high reset input, *Reset*, which is used to initialize the counter to 00. The statement **assign** Func = {F, Rx, Ry} uses the concatenate operator to create the six-bit signal *Func*, which represents the inputs to the Function Register in Figure 7.11. The *functionreg* module represents the Function Register with the data inputs *Func* and the outputs *FuncReg*. The instantiated modules *decI*, *decX*, and *decY* represent the decoders in Figure 7.11. Following these statements the previously derived logic expressions for the outputs of the control circuit are given. For $R0_{in}, \ldots, R3_{in}$ and $R0_{out}, \ldots, R3_{out}$, a **for** loop is used to produce the expressions.

At the end of the code, the adder/subtractor module is defined and the tri-state drivers and registers in the processor are instantiated.

Using Multiplexers and Case Statements

We showed in Figure 7.4 that a bus can be implemented with multiplexers, rather than tri-state drivers. Verilog code that describes the processor using this approach is shown in Figure 7.14. The code illustrates a different way of describing the control circuit in the processor. It does not give logic expressions for the signals *Extern*, *Done*, and so on, as in Figure 7.13. Instead, **case** statements are used to represent the information shown in Table 7.2. Each control signal is first assigned the value 0 as a default. This is required because the **case** statements specify the values of the control signals only when they should be asserted, as we did in Table 7.2. As explained in Chapter 5, when the value of a signal is not specified, the signal retains its current value. This implied memory results in a feedback connection, representing a latch, in the synthesized circuit. We avoid this problem by providing the default value of 0 for each of the control signals involved in the **case** statements.

In Figure 7.13 the decoders *decT* and *decI* are used to decode the *Count* signal and the stored values of the *F* input, respectively. The *decT* decoder has the outputs $T_0, \ldots, T_3$, and *decI* produces $I_0, \ldots, I_3$. In Figure 7.14 these two decoders are not used, because they do not serve a useful purpose in this code. Instead, the signal *I* is defined as a two-bit

```
module proc (Data, Reset, w, Clock, F, Rx, Ry, Done, BusWires);
    input [7:0] Data;
    input Reset, w, Clock;
    input [1:0] F, Rx, Ry;
    output wire [7:0] BusWires;
    output Done;
    reg [0:3] Rin, Rout;
    reg [7:0] Sum;
    wire Clear, AddSub, Extern, Ain, Gin, Gout, FRin;
    wire [1:0] Count;
    wire [0:3] T, I, Xreg, Y;
    wire [7:0] R0, R1, R2, R3, A, G;
    wire [1:6] Func, FuncReg;
    integer k;

    upcount counter (Clear, Clock, Count);
    dec2to4 decT (Count, 1'b1, T);

    assign Clear = Reset | Done | (~w & T[0]);
    assign Func = {F, Rx, Ry};
    assign FRin = w & T[0];

    regn functionreg (Func, FRin, Clock, FuncReg);
        defparam functionreg.n = 6;
    dec2to4 decI (FuncReg[1:2], 1'b1, I);
    dec2to4 decX (FuncReg[3:4], 1'b1, Xreg);
    dec2to4 decY (FuncReg[5:6], 1'b1, Y);

    assign Extern = I[0] & T[1];
    assign Done = ((I[0] | I[1]) & T[1]) | ((I[2] | I[3]) & T[3]);
    assign Ain = (I[2] | I[3]) & T[1];
    assign Gin = (I[2] | I[3]) & T[2];
    assign Gout = (I[2] | I[3]) & T[3];
    assign AddSub = I[3];
```

. . . continued in Part *b*.

Figure 7.13 Code for the processor (Part *a*).

signal, and the two-bit signal *Count* is used instead of T. These signals are used in the **case** statements. The code sets I to the value of the two left-most bits in the Function Register, which correspond to the stored values of the input F.

There are two nested levels of **case** statements. The first one enumerates the possible values of *Count*. For each alternative in this **case** statement, which represents a column in Table 7.2, there is a nested **case** statement that enumerates the four values of I. As

```
    // RegCntl
    always @(I, T, Xreg, Y)
        for (k = 0; k < 4; k = k+1)
        begin
            Rin[k] = ((I[0] | I[1]) & T[1] & Xreg[k]) |
                ((I[2] | I[3]) & T[3] & Xreg[k]);
            Rout[k] = (I[1] & T[1] & Y[k]) | ((I[2] | I[3]) &
                ((T[1] & Xreg[k]) | (T[2] & Y[k])));
        end

    trin tri_ext (Data, Extern, BusWires);
    regn  reg_0 (BusWires, Rin[0], Clock, R0);
    regn  reg_1 (BusWires, Rin[1], Clock, R1);
    regn  reg_2 (BusWires, Rin[2], Clock, R2);
    regn  reg_3 (BusWires, Rin[3], Clock, R3);

    trin  tri_0 (R0, Rout[0], BusWires);
    trin  tri_1 (R1, Rout[1], BusWires);
    trin  tri_2 (R2, Rout[2], BusWires);
    trin  tri_3 (R3, Rout[3], BusWires);
    regn  reg_A (BusWires, Ain, Clock, A);

    // alu
    always @(AddSub, A, BusWires)
        if (!AddSub)
            Sum = A + BusWires;
        else
            Sum = A – BusWires;

    regn  reg_G (Sum, Gin, Clock, G);
    trin  tri_G (G, Gout, BusWires);

endmodule
```

Figure 7.13 Code for the processor (Part *b*).

indicated by the comments in the code, the nested **case** statements correspond exactly to the information given in Table 7.2.

At the end of Figure 7.14, the bus structure is described with an **if-else** statement which represents multiplexers that place the appropriate data onto *BusWires*, depending on the values of R_{out}, G_{out}, and *Extern*.

We synthesized a circuit to implement the code in Figure 7.14 in a chip. Figure 7.15 gives an example of the results of a timing simulation. Each clock cycle in which $w = 1$ in this timing diagram indicates the start of an operation. In the first such operation, at 250 ns

```
module proc (Data, Reset, w, Clock, F, Rx, Ry, Done, BusWires);
    input [7:0] Data;
    input Reset, w, Clock;
    input [1:0] F, Rx, Ry;
    output reg [7:0] BusWires;
    output reg Done;
    reg [7:0] Sum;
    reg [0:3] Rin, Rout;
    reg Extern, Ain, Gin, Gout, AddSub;
    wire [1:0] Count, I;
    wire [0:3] Xreg, Y;
    wire [7:0] R0, R1, R2, R3, A, G;
    wire [1:6] Func, FuncReg, Sel;

    wire Clear = Reset | Done | (~w & ~Count[1] & ~Count[0]);
    upcount counter (Clear, Clock, Count);
    assign Func = {F, Rx, Ry};
    wire FRin = w & ~Count[1] & ~Count[0];
    regn functionreg (Func, FRin, Clock, FuncReg);
        defparam functionreg.n = 6;
    assign I = FuncReg[1:2];
    dec2to4 decX (FuncReg[3:4], 1'b1, Xreg);
    dec2to4 decY (FuncReg[5:6], 1'b1, Y);

    always @(Count, I, Xreg, Y)
    begin
        Extern = 1'b0; Done = 1'b0; Ain = 1'b0; Gin = 1'b0;
        Gout = 1'b0; AddSub = 1'b0; Rin = 4'b0; Rout = 4'b0;
        case (Count)
            2'b00: ; //no signals asserted in time step T0
            2'b01:   //define signals in time step T1
                case (I)
                    2'b00: begin //Load
                                Extern = 1'b1; Rin = Xreg; Done = 1'b1;
                           end
                    2'b01: begin //Move
                                Rout = Y; Rin = Xreg; Done = 1'b1;
                           end
                    default: begin //Add, Sub
                                Rout = Xreg; Ain = 1'b1;
                             end
                endcase
```

. . . continued in Part *b*.

Figure 7.14 Alternative code for the processor (Part *a*).

```
            2'b10: //define signals in time step T2
               case(I)
                  2'b10: begin //Add
                              Rout = Y; Gin = 1'b1;
                         end
                  2'b11: begin //Sub
                              Rout = Y; AddSub = 1'b1; Gin = 1'b1;
                         end
                  default: ; //Add, Sub
               endcase
            2'b11:
               case (I)
                  2'b10, 2'b11: begin
                                   Gout = 1'b1; Rin = Xreg; Done = 1'b1;
                                end
                  default: ; //Add, Sub
               endcase
         endcase
      end

      regn  reg_0 (BusWires, Rin[0], Clock, R0);
      regn  reg_1 (BusWires, Rin[1], Clock, R1);
      regn  reg_2 (BusWires, Rin[2], Clock, R2);
      regn  reg_3 (BusWires, Rin[3], Clock, R3);
      regn  reg_A (BusWires, Ain, Clock, A);
```

. . . continued in Part *c*.

Figure 7.14 Alternative code for the processor (Part *b*).

in the simulation time, the values of both inputs F and Rx are 00. Hence the operation corresponds to "*Load R*0, *Data*." The value of *Data* is 2A, which is loaded into $R0$ on the next positive clock edge. The next operation loads 55 into register $R1$, and the subsequent operation loads 22 into $R2$. At 850 ns the value of the input F is 10, while $Rx = 01$ and $Ry = 00$. This operation is "*Add R*1, *R*0." In the following clock cycle, the contents of $R1$ (55) appear on the bus. This data is loaded into register A by the clock edge at 950 ns, which also results in the contents of $R0$ (2A) being placed on the bus. The adder/subtractor module generates the correct sum (7F), which is loaded into register G at 1050 ns. After this clock edge the new contents of G (7F) are placed on the bus and loaded into register $R1$ at 1150 ns. Two more operations are shown in the timing diagram. The one at 1250 ns ("*Move R*3, *R*1") copies the contents of $R1$ (7F) into $R3$. Finally, the operation starting at 1450 ns ("*Sub R*3, *R*2") subtracts the contents of $R2$ (22) from the contents of $R3$ (7F), producing the correct result, $7F - 22 = 5D$.

```
// alu
always @(AddSub, A, BusWires)
begin
    if (!AddSub)
        Sum = A + BusWires;
    else
        Sum = A –  BusWires;
end

regn reg_G (Sum, Gin, Clock, G);
assign Sel = {Rout, Gout, Extern};

always @(Sel, R0, R1, R2, R3, G, Data)
begin
    if (Sel == 6'b100000)
        BusWires = R0;
    else if (Sel == 6'b010000)
        BusWires = R1;
    else if (Sel == 6'b001000)
        BusWires = R2;
    else if (Sel == 6'b000100)
        BusWires = R3;
    else if (Sel == 6'b000010)
        BusWires = G;
    else BusWires = Data;
end

endmodule
```

Figure 7.14 Alternative code for the processor (Part c).

7.3 A BIT-COUNTING CIRCUIT

We introduced algorithmic state machine (ASM) charts in Section 6.10 and showed how they can be used to describe finite state machines. ASM charts can also be used to describe digital systems that include both datapath and control circuits. We will illustrate how the ASM charts can be used as an aid in designing digital systems by using them in the remaining examples in this chapter.

Suppose that we wish to count the number of bits in a register, A, that have the value 1. Figure 7.16 shows pseudo-code for a step-by-step procedure, or *algorithm*, that can be used to perform the required task. It assumes that A is stored in a register that can shift its contents in the left-to-right direction. The answer produced by the algorithm is stored in

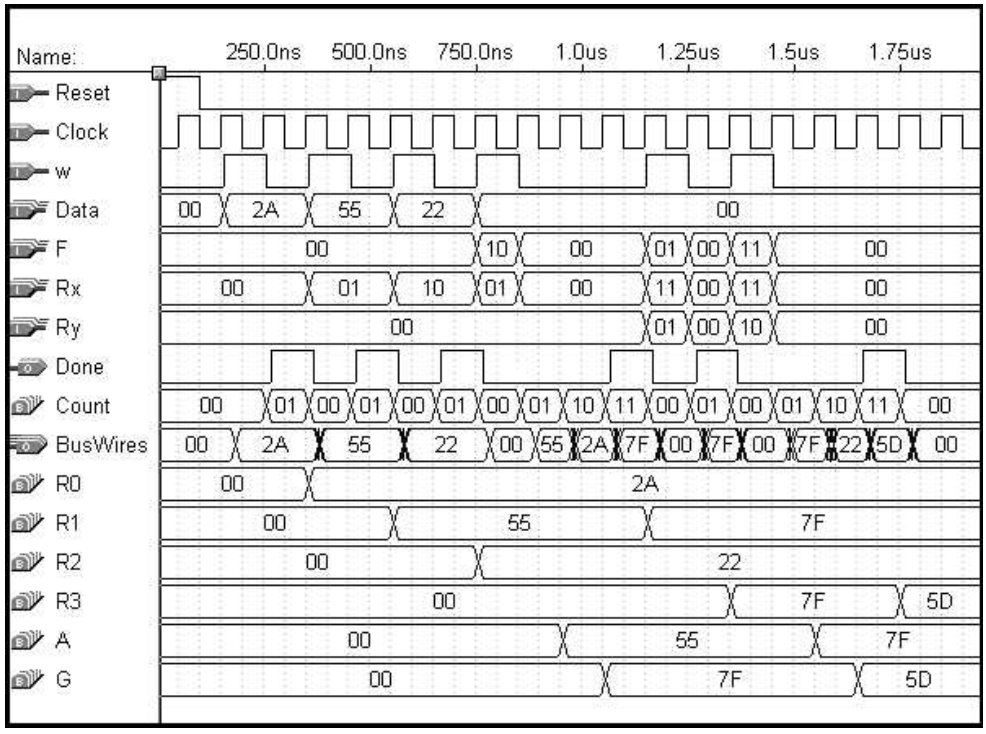


Figure 7.15 Timing simulation for the Verilog code in Figure 7.14.

```
B = 0;
while A ≠ 0 do
        if a0 = 1 then
                B = B+1;
        end if;
        Right-shift A;
end while;
```

Figure 7.16 Pseudo-code for the bit counter.

the variable named B. The algorithm terminates when A does not contain any more 1s, that is when $A = 0$. In each iteration of the while loop, if the least-significant bit (LSB) of A is 1, then B is incremented by 1; otherwise, B is not changed. The contents of A are shifted one bit to the right at the end of each loop iteration.

Figure 7.17 gives an ASM chart that represents the algorithm in Figure 7.16. The state box for the starting state, $S1$, specifies that B is initialized to 0. We assume that an input signal, s, exists, which is used to indicate when the data to be processed has been loaded

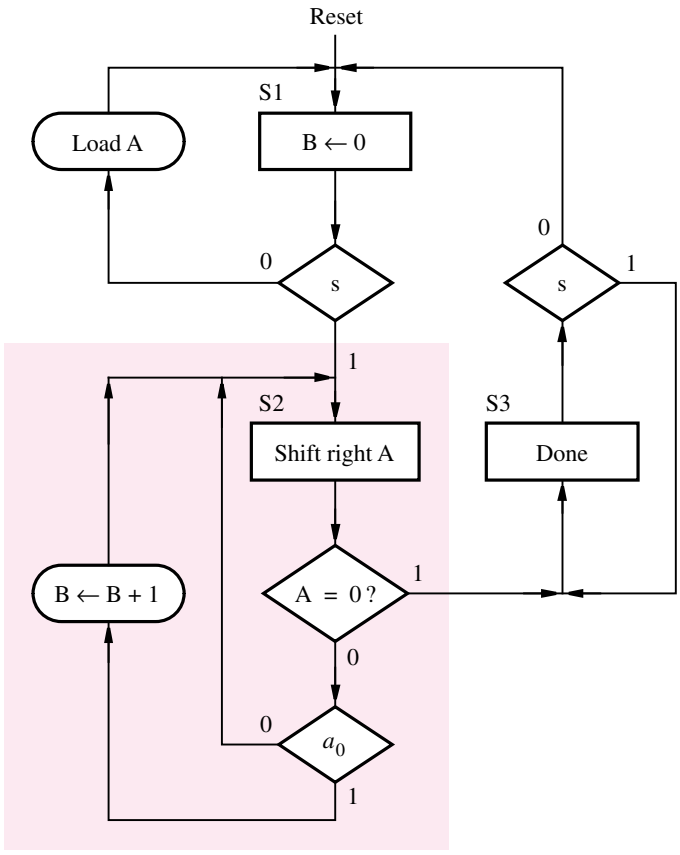


Figure 7.17 ASM chart for the pseudo-code in Figure 7.16.

into A, so that the machine can start. The decision box labeled s stipulates that the machine remains in state $S1$ as long as $s = 0$. The conditional output box with *Load A* written inside it indicates that A is loaded from external data inputs if $s = 0$ in state $S1$.

When s becomes 1, the machine changes to state $S2$. The decision box below the state box for $S2$ checks whether $A = 0$. If so, the bit-counting operation is complete; hence the machine should change to state $S3$. If not, the FSM remains in state $S2$. The decision box at the bottom of the chart checks the value of a_0. If $a_0 = 1$, B is incremented, which is indicated in the chart as $B \leftarrow B + 1$. If $a_0 = 0$, then B is not changed. In state $S3$, B contains the result, which is the number of bits in A that were 1. An output signal, *Done*, is set to 1 to indicate that the algorithm is finished; the FSM stays in $S3$ until s goes back to 0.

ASM Chart Implied Timing Information

In Chapter 6 we said that ASM charts are similar to traditional flowcharts, except that the ASM chart implies timing information. We can use the bit-counting example to illustrate this concept. Consider the ASM block for state $S2$, which is shaded in blue in Figure 7.17.

In a traditional flowchart, when state $S2$ is entered, the value of A would first be shifted to the right. Then we would examine the value of A and if A's LSB is 1, we would immediately add 1 to B. But, since the ASM chart represents a sequential circuit, changes in A and B, which represent the outputs of flip-flops, take place after the active clock edge. The same clock signal that controls changes in the state of the machine also controls changes in A and B. When the machine is in state $S1$, the next active clock edge will only perform the action specified inside the state box for $S1$, which is $B \leftarrow 0$. Hence in state $S2$, the decision box that tests whether $A = 0$, as well as the box that checks the value of a_0, check the bits in A before they are shifted. If $A = 0$, then the FSM will change to state $S3$ on the next clock edge (this clock edge also shifts A, which has no effect because A is already 0 in this case.) On the other hand, if $A \neq 0$, then the FSM does not change to $S3$, but remains in $S2$. At the same time, A is still shifted, and B is incremented if a_0 has the value 1. These timing issues are illustrated in Figure 7.21, which represents a simulation result for a circuit that implements the ASM chart. We show how the circuit is designed in the following discussion.

Datapath Circuit

By examining the ASM chart for the bit-counting circuit, we can infer the type of circuit elements needed to implement its datapath. We need a shift register that shifts left-to-right to implement A. It must have the parallel-load capability because of the conditional output box in state $S1$ that loads data into the register. An enable input is also required because shifting should occur only in state $S2$. A counter is needed for B, and it needs a parallel-load capability to initialize the count to 0 in state $S1$. It is not wise to rely on the counter's reset input to clear B to 0 in state $S1$. In practice, the reset signal is used in a digital system for only two purposes: to initialize the circuit when power is first applied, or to recover from an error. The machine changes from state $S3$ to $S1$ as a result of $s = 0$; hence we should not assume that the reset signal is used to clear the counter.

The datapath circuit is depicted in Figure 7.18. The serial input to the shift register, w, is connected to 0, because it is not needed. The load and enable inputs on the shift register are driven by the signals LA and EA. The parallel input to the shift register is named $Data$, and its parallel output is A. An n-input NOR gate is used to test whether $A = 0$. The output of this gate, z, is 1 when $A = 0$. Note that the figure indicates the n-input NOR gate by showing a single input connection to the gate, with the label n attached to it. The counter has $log_2(n)$ bits, with parallel inputs connected to 0 and parallel outputs named B. It also has a parallel-load input LB and enable input EB control signals.

Control Circuit

For convenience we can draw a second ASM chart that represents only the FSM needed for the control circuit, as shown in Figure 7.19. The FSM has the inputs s, a_0, and z and generates the outputs EA, LB, EB, and $Done$. In state $S1$, LB is asserted, so that 0 is loaded in parallel into the counter. Note that for the control signals, like LB, instead of writing $LB = 1$, we simply write LB to indicate that the signal is asserted. We assume that external circuitry drives LA to 1 when valid data is present at the parallel inputs of the shift register, so that the shift register contents are initialized before s changes to 1. In state $S2$, EA is asserted to cause a shift operation, and the count enable for B is asserted only if $a_0 = 1$.

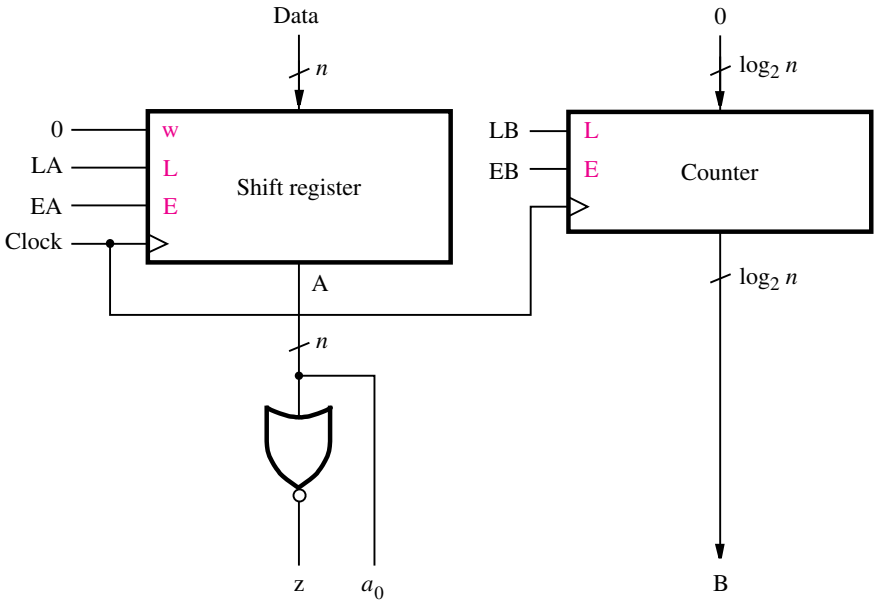


Figure 7.18 Datapath for the ASM chart in Figure 7.17.

Verilog Code

The bit-counting circuit can be described in Verilog code as shown in Figure 7.20. We have chosen to define A as an eight-bit vector and B as a four-bit vector signal. The ASM chart in Figure 7.19 can be directly translated into code that describes the required control circuit. The signal y is used to represent the present state of the FSM, and Y represents the next state. The FSM is described with three **always** blocks: the block labeled *State_table* specifies the state transitions, the block labeled *State_flipflops* represents the state flip-flops, and the block labeled *FSM_outputs* specifies the generated outputs in each state. A default value is specified at the beginning of the *FSM_outputs* block for each output signal, and then individual output values are specified in the **case** statement. For example, the default value of *EA* is 0, and *EA* is set to 1 only in state $S2$. Thus, A will be shifted only when the FSM is in state $S2$ and an active clock edge occurs.

The fourth **always** block defines the up-counter that implements B. The shift register for A is instantiated at the end of the code; its specification is given in Figure 5.60, but the parameter value has to be $n = 8$. Finally, the z signal is defined using the reduction NOR operator.

We implemented the code in Figure 7.20 in a chip and performed a timing simulation. Figure 7.21 gives the results of the simulation for $A = 00111011$. After the circuit is reset, the input signal *LA* is set to 1, and the desired data, $(3B)_{16}$, is placed on the *Data* input. When s changes to 1, the next active clock edge causes the FSM to change to state $S2$. In this state, each active clock edge increments B if a_0 is 1, and shifts A. When $A = 0$, the

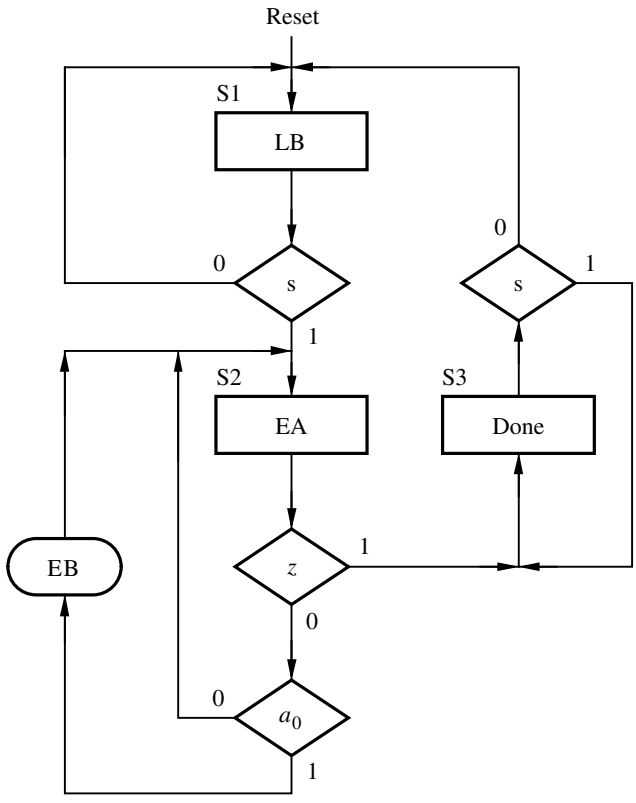


Figure 7.19 ASM chart for the bit counter control circuit.

next clock edge causes the FSM to change to state $S3$, where *Done* is set to 1 and B has the correct result, $B = 5$. To check more thoroughly that the circuit is designed correctly, we should try different values of input data.

7.4 SHIFT-AND-ADD MULTIPLIER

We presented a circuit that multiplies two unsigned n-bit binary numbers in Figure 3.35. The circuit uses a two-dimensional array of identical subcircuits, each of which contains a full-adder and an AND gate. For large values of n, this approach may not be appropriate because of the large number of gates needed. Another approach is to use a shift register in combination with an adder to implement the traditional method of multiplication that is done by "hand." Figure 7.22*a* illustrates the manual process of multiplying two binary numbers. The product is formed by a series of addition operations. For each bit i in the

```
module bitcount (Clock, Resetn, LA, s, Data, B, Done);
    input Clock, Resetn, LA, s;
    input [7:0] Data;
    output reg [3:0] B;
    output reg Done;
    wire [7:0] A;
    wire z;
    reg [1:0] Y, y;
    reg EA, EB, LB;

// control circuit

    parameter S1 = 2'b00, S2 = 2'b01, S3 = 2'b10;

    always @(s, y, z)
    begin: State_table
        case (y)
            S1: if (!s) Y = S1;
                else Y = S2;
            S2: if (z == 0) Y = S2;
                else Y = S3;
            S3: if (s) Y = S3;
                else Y = S1;
            default: Y = 2'bxx;
        endcase
    end

    always @(posedge Clock, negedge Resetn)
    begin: State_flipflops
        if (Resetn == 0)
            y <= S1;
        else
            y <= Y;
    end
```

. . . continued in Part *b*.

Figure 7.20 Verilog code for the bit-counting circuit (Part *a*).

multiplier that is 1, we add to the product the value of the multiplicand shifted to the left i times. This algorithm can be described in pseudo-code as shown in Figure 7.22*b*, where A is the multiplicand, B is the multiplier, and P is the product.

An ASM chart that represents the algorithm in Figure 7.22*b* is given in Figure 7.23. We assume that an input s is used to control when the machine begins the multiplication process. As long as s is 0, the machine stays in state $S1$ and the data for A and B can be

```
    always @(y, A[0])
    begin: FSM_outputs
        // defaults
        EA = 0; LB = 0; EB = 0; Done = 0;
        case (y)
            S1:  LB = 1;
            S2:  begin
                     EA = 1;
                     if (A[0]) EB = 1;
                     else EB = 0;
                 end
            S3:  Done = 1;
        endcase
    end

// datapath circuit

    // counter B
    always @(negedge Resetn, posedge Clock)
        if (!Resetn)
            B <= 0;
        else if (LB)
            B <= 0;
        else if (EB)
            B <= B + 1;

    shiftrne  ShiftA (Data, LA, EA, 1'b0, Clock, A);
    assign   z = ~| A;

endmodule
```

Figure 7.20 Verilog code for the bit-counting circuit (Part *b*).

loaded from external inputs. In state $S2$ we test the value of the LSB of B, and if it is 1, we add A to P. Otherwise, P is not changed. The machine moves to state $S3$ when B contains 0, because P has the final product in this case. For each clock cycle in which the machine is in state $S2$, we shift the value of A to the left, as specified in the pseudo-code in Figure 7.22*b*. We shift the contents of B to the right so that in each clock cycle b_0 can be used to decide whether or not A should be added to P.

Datapath Circuit

We can now define the datapath circuit. To implement A we need a right-to-left shift register that has $2n$ bits. A $2n$-bit register is needed for P, and it must have an enable input because the assignment $P \leftarrow P + A$ in state $S2$ is inside a conditional output box. A $2n$-bit

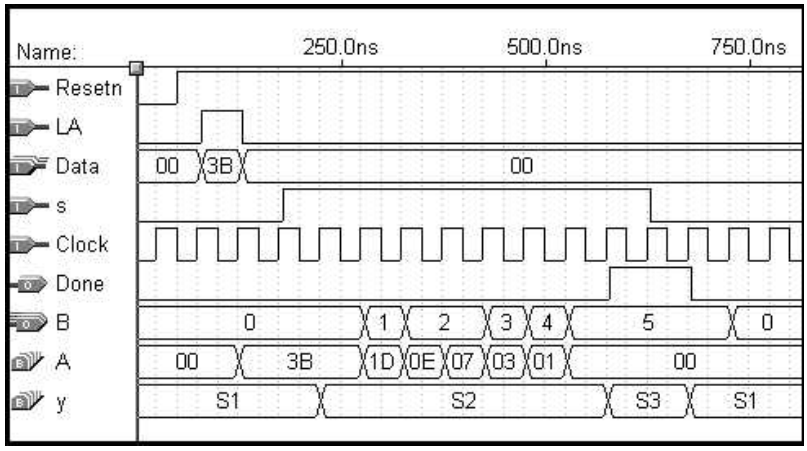


Figure 7.21 Simulation results for the bit-counting circuit.

Decimal	Binary	
13	1 1 0 1	Multiplicand
× 11	× 1 0 1 1	Multiplier
13	1 1 0 1	
13	1 1 0 1	
143	0 0 0 0	
	1 1 0 1	
	1 0 0 0 1 1 1 1	Product

(a) Manual method

$P = 0$;
for $i = 0$ to $n - 1$ do
    if $b_i = 1$ then
        $P = P + A$;
    end if;
    Left-shift A;
end for;

(b) Pseudo-code

Figure 7.22 An algorithm for multiplication.

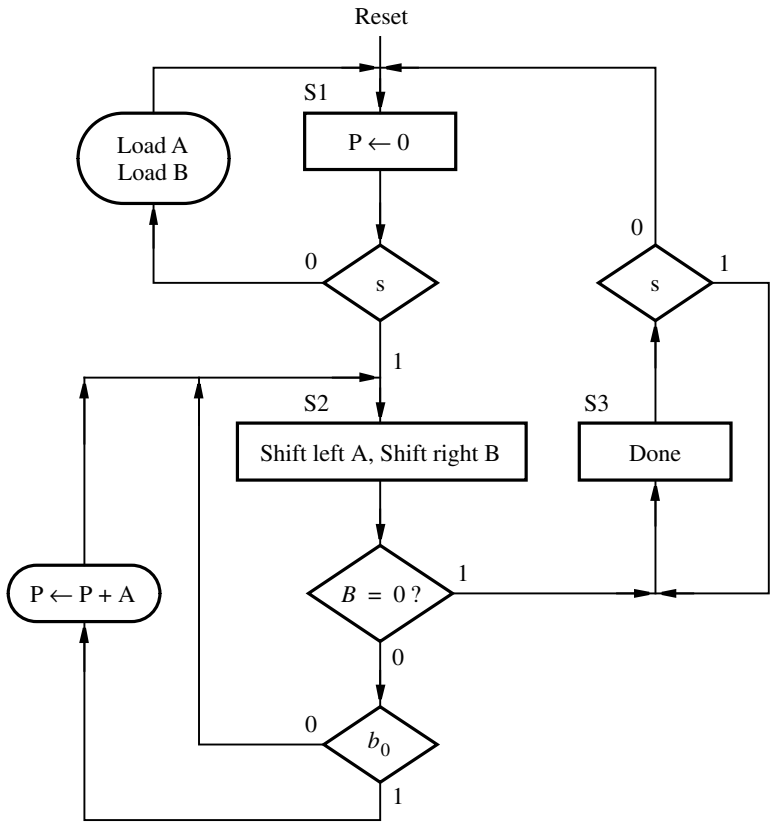


Figure 7.23 ASM chart for the multiplier.

adder is needed to produce $P + A$. Note that P is loaded with 0 in state $S1$, and P is loaded from the output of the adder in state $S2$. We cannot assume that the reset input is used to clear P, because the machine changes from state $S3$ back to $S1$ based on the s input, not the reset input. Hence a 2-to-1 multiplexer is needed for each input to P, to select either 0 or the appropriate sum bit from the adder. An n-bit left-to-right shift register is needed for B, and an n-input NOR gate can be used to test whether $B = 0$.

Figure 7.24 shows the datapath circuit and labels the control signals for the shift registers. The input data for the shift register that holds A is named *DataA*. Since the shift register has $2n$ bits, the most-significant n data inputs are connected to 0. A single multiplexer symbol is shown connected to the register that holds P. This symbol represents $2n$ 2-to-1 multiplexers that are each controlled by the *Psel* signal.

Control Circuit

An ASM chart that represents only the control signals needed for the multiplier is given in Figure 7.25. In state $S1$, *Psel* is set to 0 and *EP* is asserted, so that register P is cleared. When $s = 0$, parallel data can be loaded into shift registers A and B by an external circuit

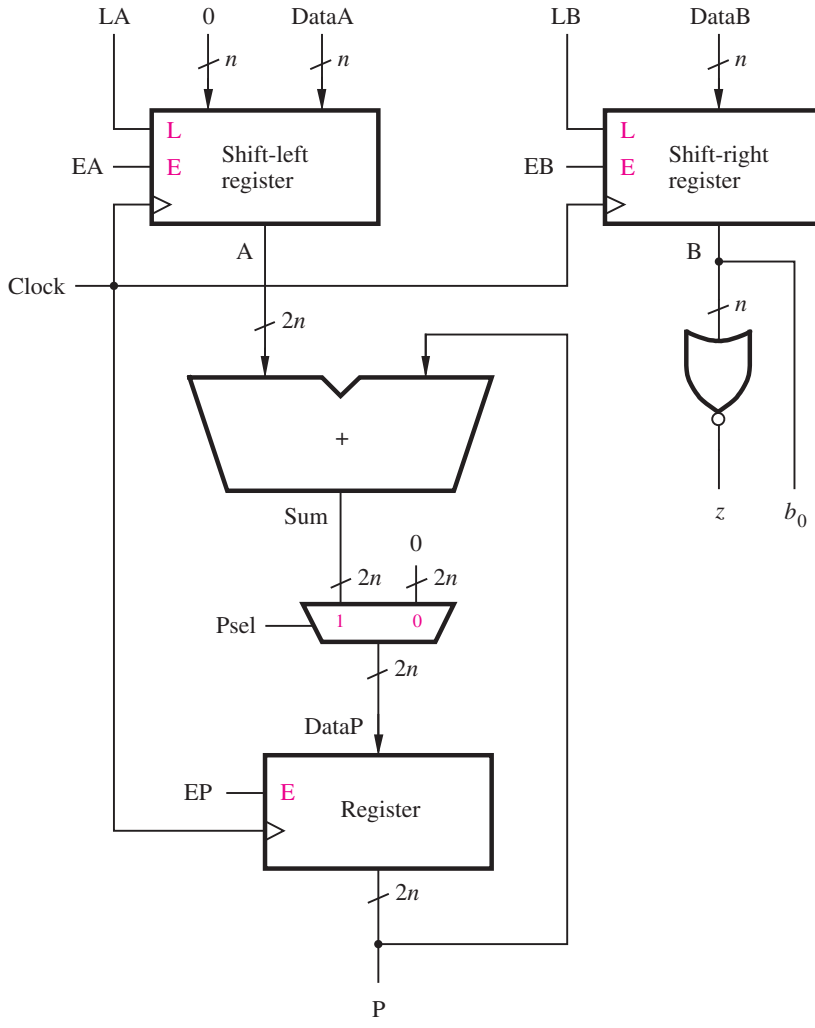


Figure 7.24 Datapath circuit for the multiplier.

that controls their parallel-load inputs *LA* and *LB*. When $s = 1$, the machine changes to state $S2$, where *Psel* is set to 1 and shifting of *A* and *B* is enabled. If $b_0 = 1$, the enable for *P* is asserted. The machine changes to state $S3$ when $z = 1$, and then remains in $S3$ and sets *Done* to the value 1 as long as $s = 1$.

Verilog Code

Verilog code for the multiplier is given in Figure 7.26. The number of bits in *A* and *B* is set by the parameter *n*. For registers that are $2n$ bits wide, the number of bits is set to $n + n$. By changing the value of the parameters, the code can be used for numbers of any size.

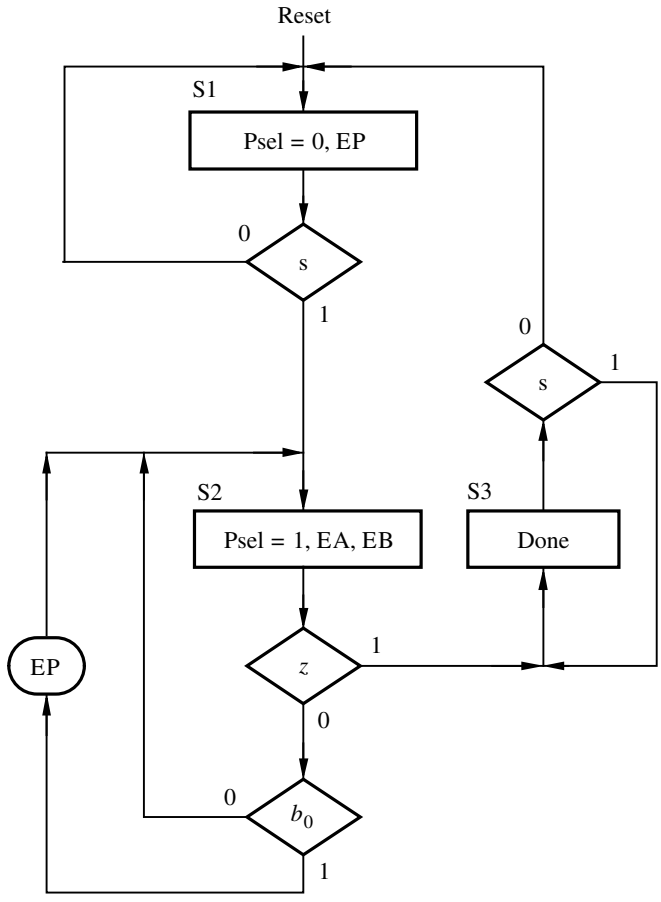


Figure 7.25 ASM chart for the multiplier control circuit.

The **always** blocks labeled *State_table* and *State_flipflops* define the state transitions and state flip-flops, respectively. The control circuit outputs are specified in the **always** block labeled *FSM_outputs*. The parallel data input on the shift register *A* is 2*n* bits wide, but *DataA* is only *n* bits wide. Hence the concatenate operation {{n{1'b0}}, DataA} is used to prepend *n* zeros onto *DataA* for loading into the shift register. The multiplexer needed for register *P* is defined using a **for** loop that defines 2*n* 2-to-1 multiplexers. Figure 7.27 gives a simulation result for the circuit generated from the code. After the circuit is reset, *LA* and *LB* are set to 1, and the numbers to be multiplied are placed on the *DataA* and *DataB* inputs. After *s* is set to 1, the FSM (*y*) changes to state *S*2, where it remains until $B = 0$. For each clock cycle in state *S*2, *A* is shifted to the left, and *B* is shifted to the right. In three of the clock cycles in state *S*2, the contents of *A* are added to *P*, corresponding to the three bits in *B* that have the value 1. When $B = 0$, the FSM changes to state *S*3 and *P* contains the

```
module multiply (Clock, Resetn, LA, LB, s, DataA, DataB, P, Done);
    parameter n = 8;
    input Clock, Resetn, LA, LB, s;
    input [n–1:0] DataA, DataB;
    output [n+n–1:0] P;
    output reg Done;
    wire z;
    reg [n+n–1:0] DataP;
    wire [n+n–1:0] A, Sum;
    reg [1:0] y, Y;
    wire [n–1:0] B;
    reg EA, EB, EP, Psel;
    integer k;

// control circuit

    parameter S1 = 2'b00, S2 = 2'b01, S3 = 2'b10;

    always @(s, y, z)
    begin: State_table
        case (y)
            S1: if (s == 0) Y = S1;
                else Y = S2;
            S2: if (z == 0) Y = S2;
                else Y = S3;
            S3: if (s == 1) Y = S3;
                else Y = S1;
            default: Y = 2'bxx;
        endcase
    end

    always @(posedge Clock, negedge Resetn)
    begin: State_flipflops
        if (Resetn == 0)
            y <= S1;
        else
            y <= Y;
        end
```

. . . continued in Part *b*.

Figure 7.26 Verilog code for the multiplier circuit (Part *a*).

```
    always @(s, y, B[0])
    begin: FSM_outputs
        // defaults
        EA = 0; EB = 0; EP = 0; Done = 0; Psel = 0;
        case (y)
            S1:  EP = 1;
            S2:  begin
                     EA = 1; EB = 1; Psel = 1;
                     if (B[0]) EP = 1;
                     else EP = 0;
                 end
            S3:  Done = 1;
        endcase
    end

//datapath circuit

    shiftrne  ShiftB (DataB, LB, EB, 1'b0, Clock, B);
        defparam  ShiftB.n = 8;
    shiftlne  ShiftA ({{n{1'b0}}, DataA}, LA, EA, 1'b0, Clock, A);
        defparam  ShiftA.n = 16;

    assign   z = (B == 0);
    assign   Sum = A + P;

    // define the 2n 2-to-1 multiplexers
    always @(Psel, Sum)
        for (k = 0; k < n+n; k = k+1)
            DataP[k] = Psel ? Sum[k] : 1'b0;

    regne  RegP (DataP, Clock, Resetn, EP, P);
        defparam RegP.n = 16;

endmodule
```

Figure 7.26 Verilog code for the multiplier circuit (Part *b*).

correct product, which is $(64)_{16} \times (19)_{16} = (9C4)_{16}$. The decimal equivalent of this result is $100 \times 25 = 2500$.

The number of clock cycles that the circuit requires to generate the final product is determined by the left-most digit in B that is 1. It is possible to reduce the number of clock cycles needed by using more complex shift registers for A and B. If the two right-most bits in B are both 0, then both A and B could be shifted by two bit positions in one clock cycle. Similarly, if the three lowest digits in B are 0, then a three bit-position shift can be done,

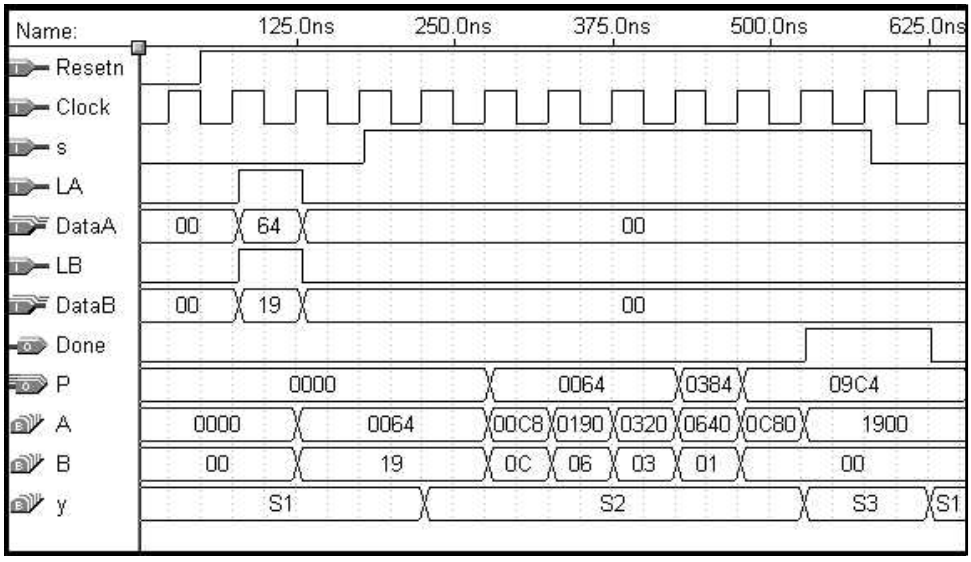


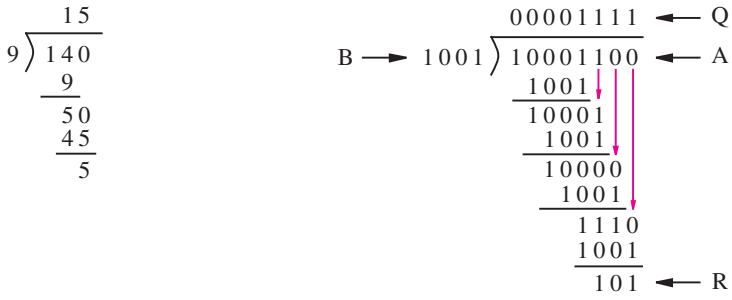
Figure 7.27 Simulation results for the multiplier circuit.

and so on. A shift register that can shift by multiple bit positions at once can be built using a *barrel shifter*. We leave it as an exercise for the reader to modify the multiplier to make use of a barrel shifter.

7.5 DIVIDER

The preceding example implements the traditional method of performing multiplication by hand. In this example we will design a circuit that implements the traditional long-hand division. Figure 7.28*a* gives an example of long-hand division. The first step is to try to divide the divisor 9 into the first digit of the dividend 1, which does not work. Next, we try to divide 9 into 14, and determine that 1 is the first digit in the quotient. We perform the subtraction $14 - 9 = 5$, bring down the last digit from the dividend to form 50, and then determine that the next digit in the quotient is 5. The remainder is $50 - 45 = 5$, and the quotient is 15. Using binary numbers, as illustrated in Figure 7.28*b*, involves the same process, with the simplification that each digit of the quotient can be only 0 or 1.

Given two unsigned n-bit numbers A and B, we wish to design a circuit that produces two n-bit outputs Q and R, where Q is the quotient A/B and R is the remainder. The procedure illustrated in Figure 7.28*b* can be implemented by shifting the digits in A to the left, one digit at a time, into a shift register R. After each shift operation, we compare R with B. If $R \geq B$, a 1 is placed in the appropriate bit position in the quotient and B is subtracted from R. Otherwise, a 0 bit is placed in the quotient. This algorithm is described using



(a) An example using decimal numbers

(b) Using binary numbers

```
R = 0;
for i = 0 to n − 1 do
    Left-shift R||A;
    if R ≥ B then
        q_i = 1;
        R = R − B;
    else
        q_i = 0 ;
    end if;
end for;
```

(c) Pseudo-code

Figure 7.28 An algorithm for division.

pseudo-code in Figure 7.28*c*. The notation $R||A$ is used to represent a $2n$-bit shift register formed using R as the left-most n bits and A as the right-most n bits.

In the long-division pseudo-code, each loop iteration results in setting a digit q_i to either 1 or 0. A straightforward way to accomplish this is to shift 1 or 0 into the least-significant bit of Q in each loop iteration. An ASM chart that represents the divider circuit is shown in Figure 7.29. The signal C represents a counter that is initialized to $n - 1$ in the starting state $S1$. In state $S2$, both R and A are shifted to the left, and then in state $S3$, B is subtracted from R if $R \geq B$. The machine changes to state $S4$ when $C = 0$.

Datapath Circuit

We need n-bit shift registers that shift right to left for A, R, and Q. An n-bit register is needed for B, and a subtractor is needed to produce $R - B$. We can use an adder module in which the carry-in is set to 1 and B is complemented. The carry-out, c_{out}, of this module has the value 1 if the condition $R \geq B$ is true. Hence the carry-out can be connected to the serial input of the shift register that holds Q, so that it is shifted into Q in state $S3$. Since R is loaded with 0 in state $S1$ and from the outputs of the adder in state $S3$, a multiplexer is needed for the parallel data inputs on R. The datapath circuit is depicted in Figure 7.30.

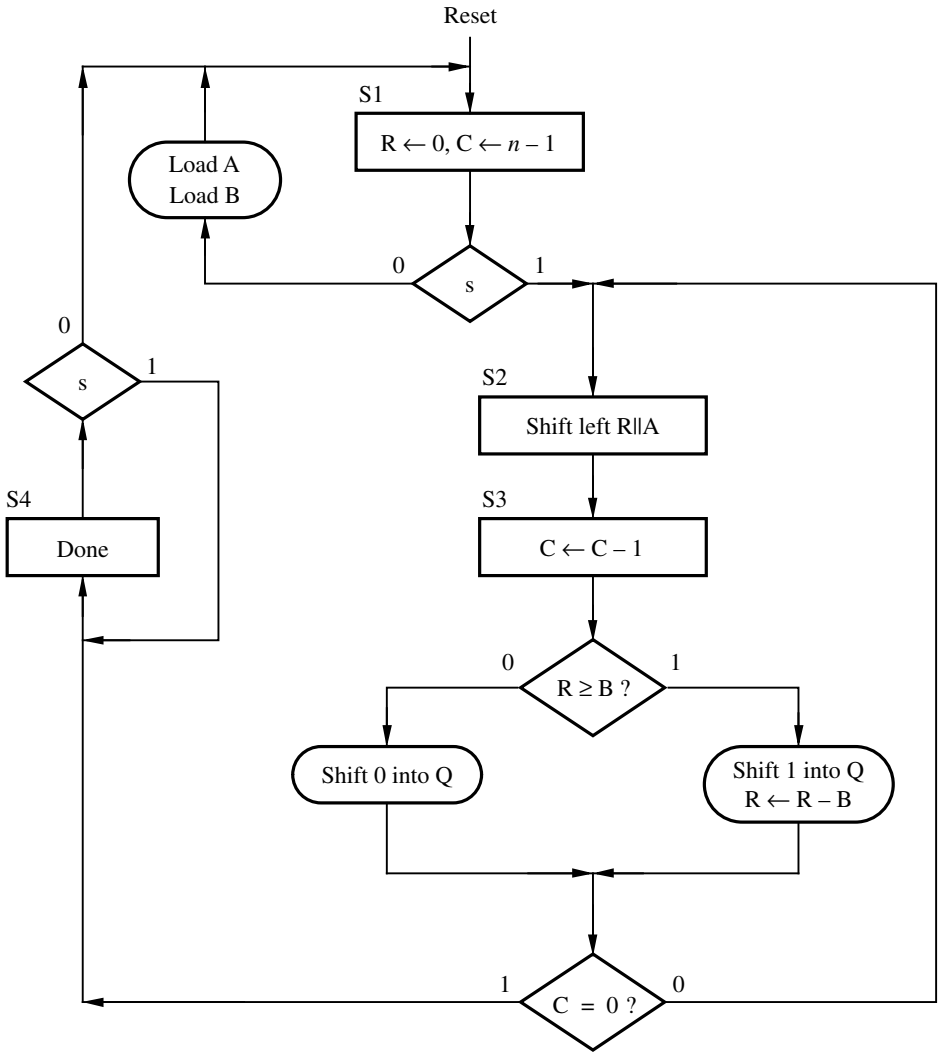


Figure 7.29 ASM chart for the divider.

Note that the down-counter needed to implement C and the NOR gate that outputs a 1 when $C = 0$ are not shown in the figure.

Control Circuit

An ASM chart that shows only the control signals needed for the divider is given in Figure 7.31. In state $S3$ the value of c_{out} determines whether or not the sum output of the adder is loaded into R. The shift enable on Q is asserted in state $S3$. We do not have to specify whether 1 or 0 is loaded into Q, because c_{out} is connected to Q's serial input in

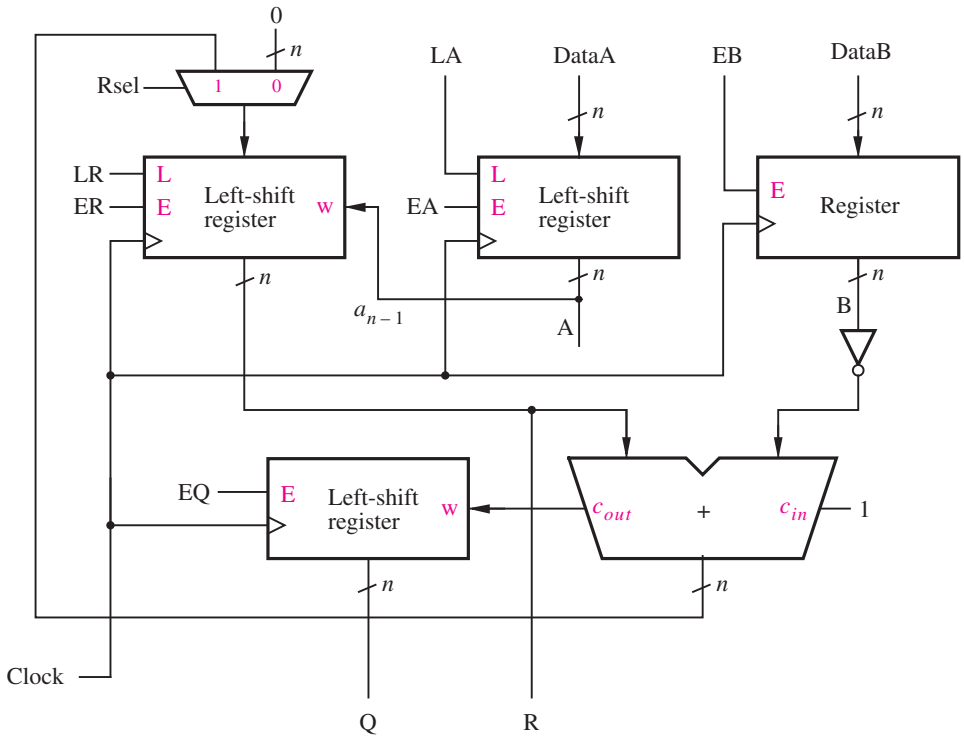


Figure 7.30 Datapath circuit for the divider.

the datapath circuit. We leave it as an exercise for the reader to write Verilog code that represents the ASM chart in Figure 7.31 and the datapath circuit in Figure 7.30.

Enhancements to the Divider Circuit

Using the ASM chart in Figure 7.31 causes the circuit to loop through states $S2$ and $S3$ for $2n$ clock cycles. If these states can be merged into a single state, then the number of clock cycles needed can be reduced to n. In state $S3$, if $c_{out} = 1$, we load the sum output (result of the subtraction) from the adder into R, and (assuming $z = 0$) change to state $S2$. In state $S2$ we then shift R (and A) to the left. To combine $S2$ and $S3$ into a new state, called $S2$, we need to be able to place the sum into the left-most bits of R while at the same time shifting the MSB of A into the LSB of R. This step can be accomplished by using a separate flip-flop for the LSB of R. Let the output of this flip-flop be called rr_0. It is initialized to 0 when $s = 0$ in state $S1$. Otherwise, the flip-flop is loaded from the MSB of A. In state $S2$, if $c_{out} = 0$, R is shifted left and rr_0 is shifted into R. But if $c_{out} = 1$, R is loaded in parallel from the sum outputs of the adder.

Figure 7.32 illustrates how the division example from Figure 7.28*b* can be performed using n clock cycles. The table in the figure shows the values of R, rr_0, A, and Q in each step

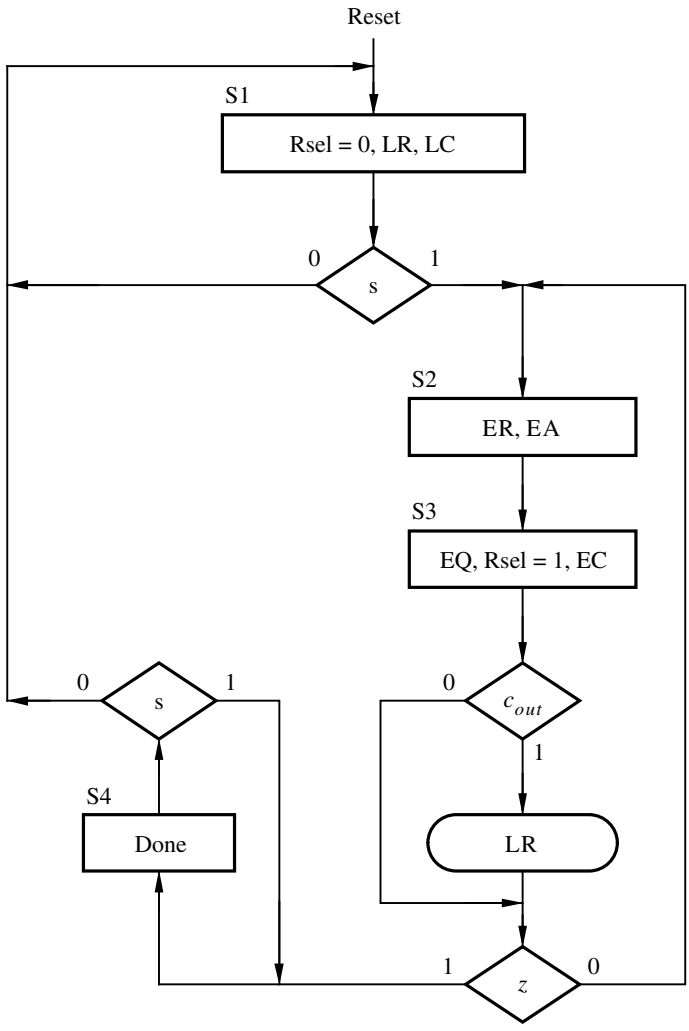


Figure 7.31 ASM chart for the divider control circuit.

of the division. In the datapath circuit in Figure 7.30, we use a separate shift register for Q. This register is not actually needed, because the digits in the quotient can be shifted into the least-significant bit of the register used for A. In Figure 7.32 the digits of Q that are shifted into A are shown in blue. The first row in the table represents loading of initial data into registers A (and B) and clearing R and rr_0 to 0. In the second row of the table, labeled clock cycle 0, the diagonal blue arrow shows that the left-most bit of A (1) is shifted into rr_0. The number in $R||rr_0$ is now 000000001, which is smaller than B (1001). In clock cycle 1, rr_0 is shifted into R, and the MSB of A is shifted into rr_0. Also, as shown in blue, a 0 is shifted into

B → 1001 ⟩ 10001100 ← A

	Clock cycle	R								rr_0	A/Q							
	Load A, B	0	0	0	0	0	0	0	0	0	1	0	0	0	1	1	0	0
0	Shift left	0	0	0	0	0	0	0	0	1	0	0	0	1	1	0	0	0
1	Shift left, $Q_0 \leftarrow 0$	0	0	0	0	0	0	0	1	0	0	0	1	1	0	0	0	0
2	Shift left, $Q_0 \leftarrow 0$	0	0	0	0	0	0	1	0	0	0	1	1	0	0	0	0	0
3	Shift left, $Q_0 \leftarrow 0$	0	0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0
4	Shift left, $Q_0 \leftarrow 0$	0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0	0
5	Subtract, $Q_0 \leftarrow 1$	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	1
6	Subtract, $Q_0 \leftarrow 1$	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1
7	Subtract, $Q_0 \leftarrow 1$	0	0	0	0	0	1	1	1	0	0	0	0	0	0	1	1	1
8	Subtract, $Q_0 \leftarrow 1$	0	0	0	0	0	1	0	1	0	0	0	0	0	1	1	1	1

Figure 7.32 An example of division using $n = 8$ clock cycles.

the LSB of Q (A). The number in $R||rr_0$ is now 000000010, which is still smaller than B. Hence, in clock cycle 2 the same actions are performed as for clock cycle 1. These actions are also performed in clock cycles 3 and 4, at which point $R||rr_0 = 000010001$. Since this is larger than B, in clock cycle 5 the result of the subtraction $000010001 - 1001 = 00001000$ is loaded into R. The MSB of A (1) is still shifted into rr_0, and a 1 is shifted into Q. In clock cycles 6, 7, and 8, the number in $R||rr_0$ is larger than B; hence in each of these cycles the result of the subtraction $R||rr_0 - B$ is loaded into R, and a 1 is loaded into Q. After clock cycle 8 the correct result, Q = 00001111 and $R = 00000101$, is obtained. The bit rr_0 is not a part of the final result.

An ASM chart that shows the values of the required control signals for the enhanced divider is depicted in Figure 7.33. The signal ER0 is used in conjunction with the flip-flop that has the output rr_0. When ER0 = 0, the value 0 is loaded into the flip-flop. When ER0 is set to 1, the MSB of shift register A is loaded into the flip-flop. In state $S1$, if $s = 0$, then LR is asserted to initialize R to 0. Registers A and B can be loaded with data from external inputs. When s changes to 1, the machine makes a transition to state $S2$ and at the same time shifts $R||R0||A$ to the left. In state $S2$, if $c_{out} = 1$, then R is loaded in parallel from the sum outputs of the adder. At the same time, $R0||A$ is shifted left (rr_0 is not shifted into R in this case). If $c_{out} = 0$, then $R||R0||A$ is shifted left. The ASM chart shows how the parallel-load and enable inputs on the registers have to be controlled to achieve the desired operation.

The datapath circuit for the enhanced divider is illustrated in Figure 7.34. As discussed for Figure 7.32, the digits of the quotient Q are shifted into register A. Note that one of the n-bit data inputs on the adder module is composed of the $n - 1$ least-significant bits in register R concatenated with bit rr_0 on the right.

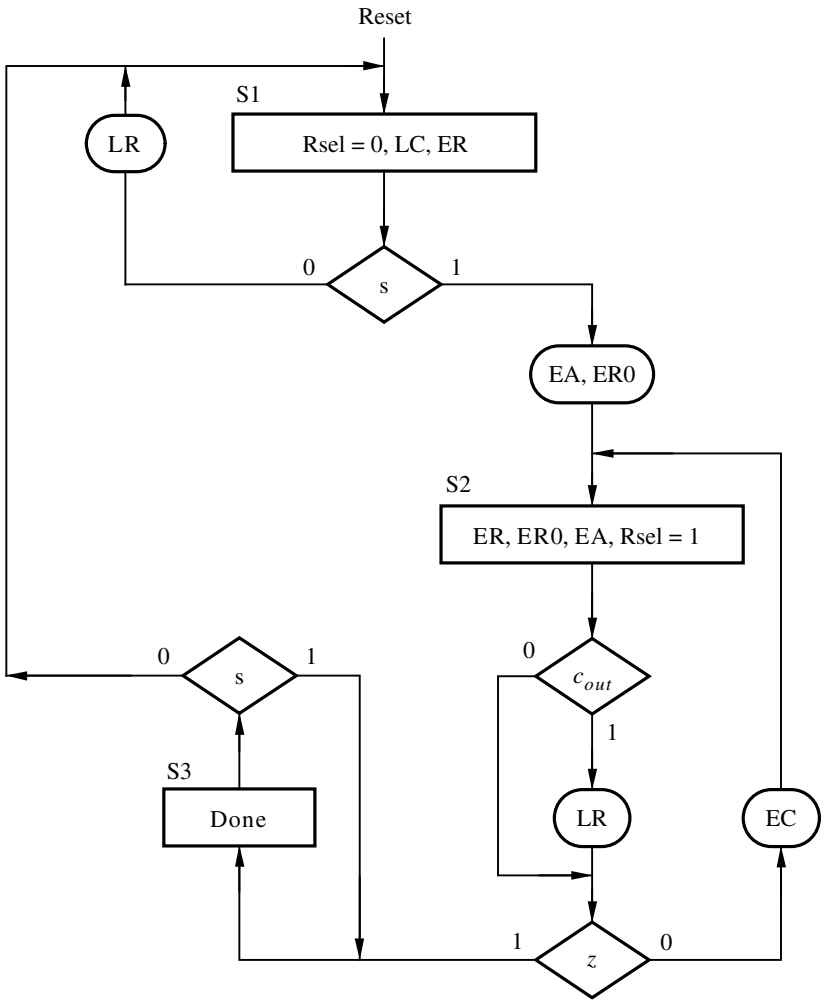


Figure 7.33 ASM chart for the enhanced divider control circuit.

Verilog Code

Figure 7.35 shows Verilog code that represents the enhanced divider. The parameter *n* sets the number of bits in the operands. The *State_table*, *State_flipflops*, and *FSM_outputs* **always** blocks describe the control circuit, as in the previous examples. The shift registers and counters in the datapath circuit are instantiated at the bottom of the code. The signal rr_0 in Figure 7.32 is represented in the code by the signal R0. This signal is implemented as the output of the *muxdff* component; the code for this subcircuit is shown in Figure 5.47. Note that the adder that produces the *Sum* signal has one input defined as the concatenation of R with R0. The multiplexer needed for the input to R is represented by the *DataR* signal. This multiplexer is defined in the last statement of the code.

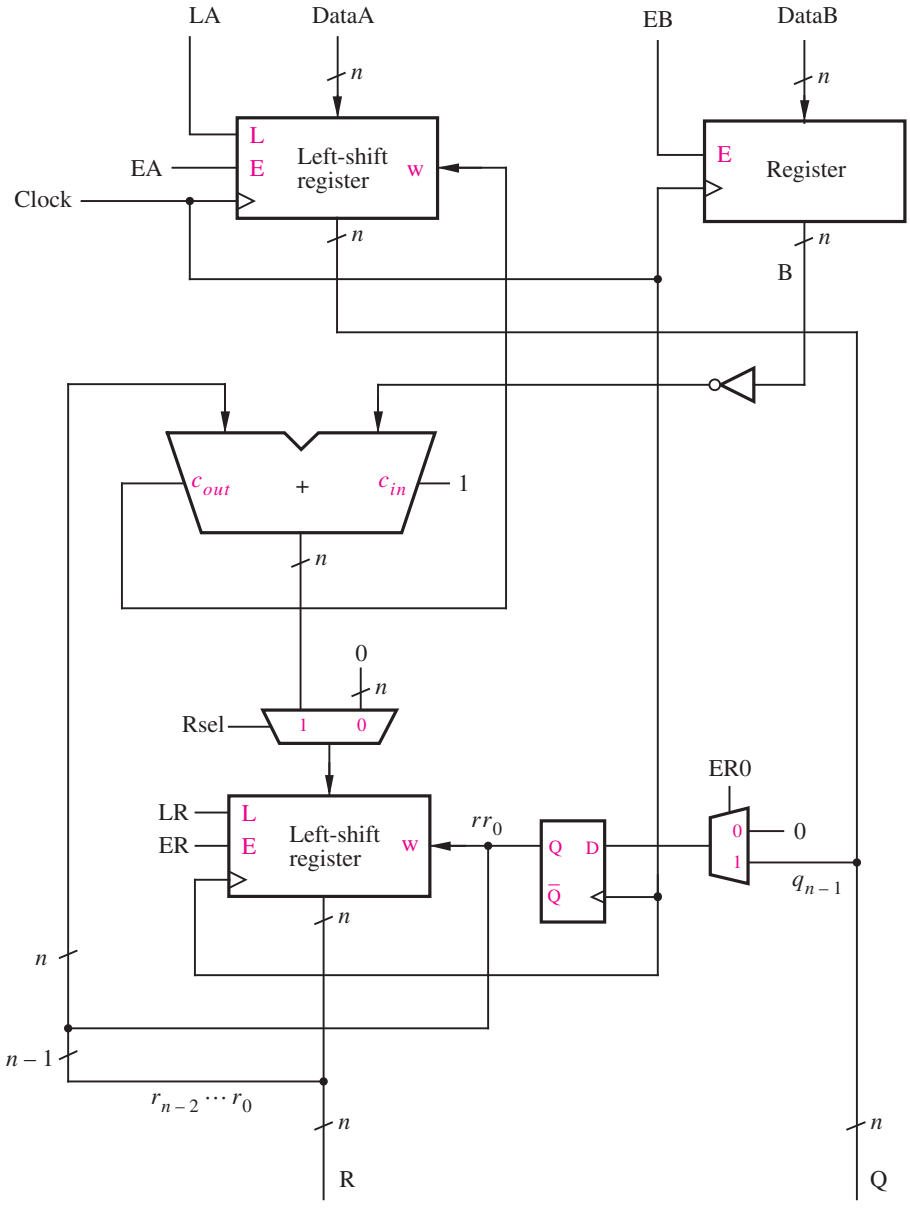


Figure 7.34 Datapath circuit for the enhanced divider.

```
module divider (Clock, Resetn, s, LA, EB, DataA, DataB, R, Q, Done);
    parameter n = 8, logn = 3;
    input Clock, Resetn, s, LA, EB;
    input [n–1:0] DataA, DataB;
    output [n–1:0] R, Q;
    output reg Done;
    wire Cout, z, R0;
    wire [n–1:0] DataR;
    wire [n:0] Sum;
    reg [1:0] y, Y;
    wire [n–1:0] A, B;
    wire [logn–1:0] Count;
    reg EA, Rsel, LR, ER, ER0, LC, EC;
    integer k;

// control circuit

    parameter S1 = 2'b00, S2 = 2'b01, S3 = 2'b10;

    always @(s, y, z)
    begin: State_table
        case (y)
            S1: if (s == 0) Y = S1;
                else Y = S2;
            S2: if (z == 0) Y = S2;
                else Y = S3;
            S3: if (s == 1) Y = S3;
                else Y = S1;
            default: Y = 2'bxx;
        endcase
    end

    always @(posedge Clock, negedge Resetn)
    begin: State_flipflops
        if (Resetn == 0)
            y <= S1;
        else
            y <= Y;
    end
```

. . . continued in Part *b*.

Figure 7.35 Verilog code for the divider circuit (Part *a*).

```
always @(y, s, Cout, z)
begin: FSM_outputs
   // defaults
   LR = 0; ER = 0; ER0 = 0; LC = 0; EC = 0; EA = 0;
   Rsel = 0; Done = 0;
   case (y)
      S1:  begin
              LC = 1; ER = 1;
              if (s == 0)
              begin
                 LR = 1; ER0 = 0;
              end
              else
              begin
                 LR = 0; EA = 1; ER0 = 1;
              end
           end
      S2:  begin
              Rsel = 1; ER = 1; ER0 = 1; EA = 1;
              if (Cout) LR = 1;
              else LR = 0;
              if (z == 0) EC = 1;
              else EC = 0;
           end
      S3:  Done = 1;
   endcase
end
```

. . . continued in Part *c*.

Figure 7.35 Verilog code for the divider circuit (Part *b*).

A simulation result for the circuit produced from the code is given in Figure 7.36. The data $A = \text{A6}$ and $B = 8$ is loaded, and then s is set to 1. The circuit changes to state $S2$ and concurrently shifts R, $R0$, and A to the left. The output of the shift register that holds A is labeled Q in the simulation results because this shift register contains the quotient when the division operation is complete. On the first three active clock edges in state $S2$, the number represented by $R||R0$ is less than the number in B (8); hence $R||R0||A$ is shifted left on each clock edge, and 0 is shifted into Q. In the fourth consecutive clock cycle for which the FSM has been in state $S2$, the contents of R are $00000101 = (5)_{10}$, and $R0$ is 0; hence $R||R0 = 000001010 = (10)_{10}$. On the next active clock edge, the output of the adder, which is $10 - 8 = 2$, is loaded into R, and 1 is shifted into Q. After n clock cycles in state $S2$, the circuit changes to state $S3$, and the correct result, $\text{Q} = 14 = (20)_{10}$ and $R = 6$, is obtained.

```
//datapath circuit

    regne  RegB (DataB, Clock, Resetn, EB, B);
       defparam RegB.n = n;
    shiftlne  ShiftR (DataR, LR, ER, R0, Clock, R);
       defparam ShiftR.n = n;
    muxdff  FF_R0 (1'b0, A[n–1], ER0, Clock, R0);
    shiftlne  ShiftA (DataA, LA, EA, Cout, Clock, A);
       defparam  ShiftA.n = n;
    assign   Q = A;
    downcount  Counter (Clock, EC, LC, Count);
       defparam Counter.n = logn;

    assign   z = (Count == 0);
    assign   Sum = {1'b0, R[n–2:0], R0} + {1'b0, ~B} + 1;
    assign   Cout = Sum[n];

    // define the n 2-to-1 multiplexers
    assign   DataR = Rsel ? Sum : 0;

endmodule
```

Figure 7.35 Verilog code for the divider circuit (Part *c*).

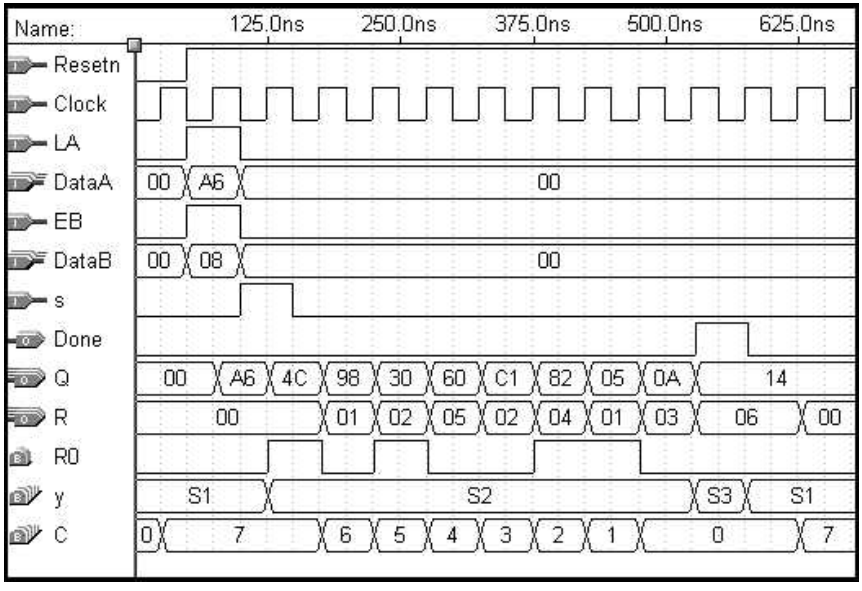


Figure 7.36 Simulation results for the divider circuit.

7.6 ARITHMETIC MEAN

Assume that k n-bit numbers are stored in a set of registers $R_0, \ldots, R_{k-1}$. We wish to design a circuit that computes the mean M of the numbers in the registers. The pseudo-code for a suitable algorithm is shown in Figure 7.37*a*. Each iteration of the loop adds the contents of one of the registers, denoted R_i, to a *Sum* variable. After the sum is computed, M is obtained as Sum/k. We assume that integer division is used, so a remainder R, not shown in the code, is produced as well.

An ASM chart is given in Figure 7.37*b*. While the start input, s, is 0, the registers can be loaded from external inputs. When s becomes 1, the machine changes to state $S2$, where it remains while $C \neq 0$, and computes the summation (C is a counter that represents i in Figure 7.37*a*). When $C = 0$, the machine changes to state $S3$ and computes $M = Sum/k$. From the previous example, we know that the division operation requires multiple clock cycles, but we have chosen not to indicate this in the ASM chart. After computing the division operation, state $S4$ is entered and *Done* is set to 1.

Datapath Circuit

The datapath circuit for this task is more complex than in our previous examples. It is depicted in Figure 7.38. We need a register with an enable input to hold *Sum*. For simplicity, assume that the sum can be represented in n bits without overflowing. A multiplexer is required on the data inputs on the *Sum* register, to select 0 in state $S1$ and the sum outputs of an adder in state $S2$. The *Sum* register provides one of the data inputs to the adder. The other input has to be selected from the data outputs of one of the k registers. One way to select among the registers is to connect them to the data inputs of a k-to-1 multiplexer that is connected to the adder. The select lines on the multiplexer can be controlled by the counter C. To compute the division operation, we can use the divider circuit designed in Section 7.5.

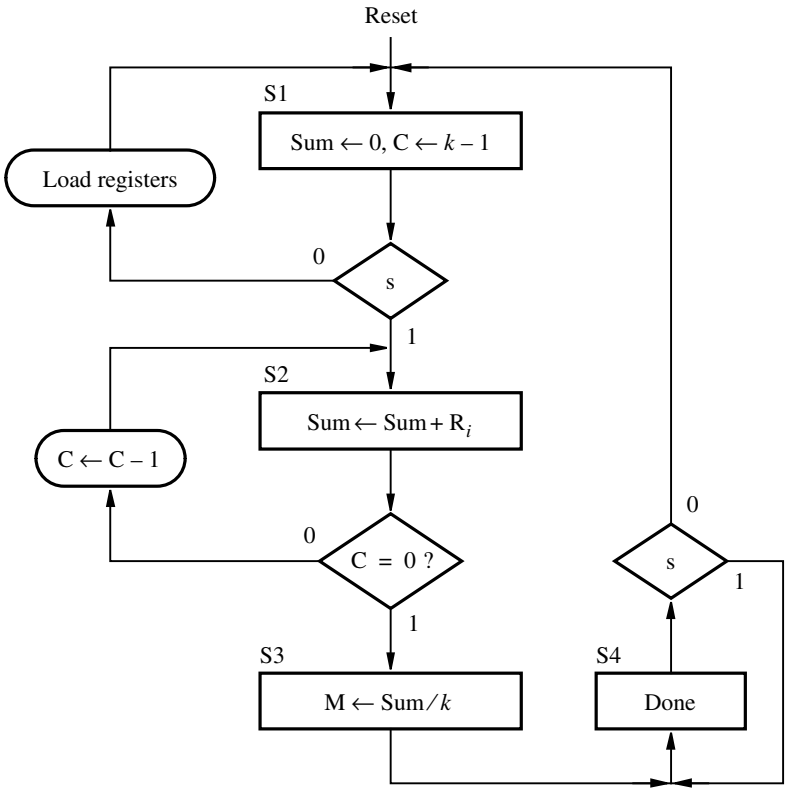
The circuit in Figure 7.38 is based on $k = 4$, but the same circuit structure can be used for larger values of k. Note that the enable inputs on the registers R_0 through R_3 are connected to the outputs of a 2-to-4 decoder that has the two-bit input *RAdd*, which stands for "register address." The decoder enable input is driven by the *ER* signal. All registers are loaded from the same input lines, *Data*. Since $k = 4$, we could perform the division operation simply by shifting *Sum* two bits to the right, which can be done in one clock cycle with a shift register that shifts by two digits. To obtain a more general circuit that works for any value of k, we use the divider circuit designed in Section 7.5.

Control Circuit

Figure 7.39 gives an ASM chart for the FSM needed to control the circuit in Figure 7.38. While in state $S1$, data can be loaded into registers $R_0, \ldots, R_{k-1}$. But no control signals have to be asserted for this purpose, because the registers are loaded under control of the *ER* and *RAdd* inputs, as discussed above. When $s = 1$, the FSM changes to state $S2$, where it asserts the enable *ES* on the *Sum* register and allows C to decrement. When the counter reaches 0 ($z = 1$), the machine enters state $S3$, where it asserts the *LA* and *EB* signals to

$Sum = 0$;
for $i = k - 1$ down to 0 do
    $Sum = Sum + R_i$
end for;
$M = Sum \div k$;

(a) Pseudo-code



(b) ASM chart

Figure 7.37 An algorithm for finding the mean of k numbers.

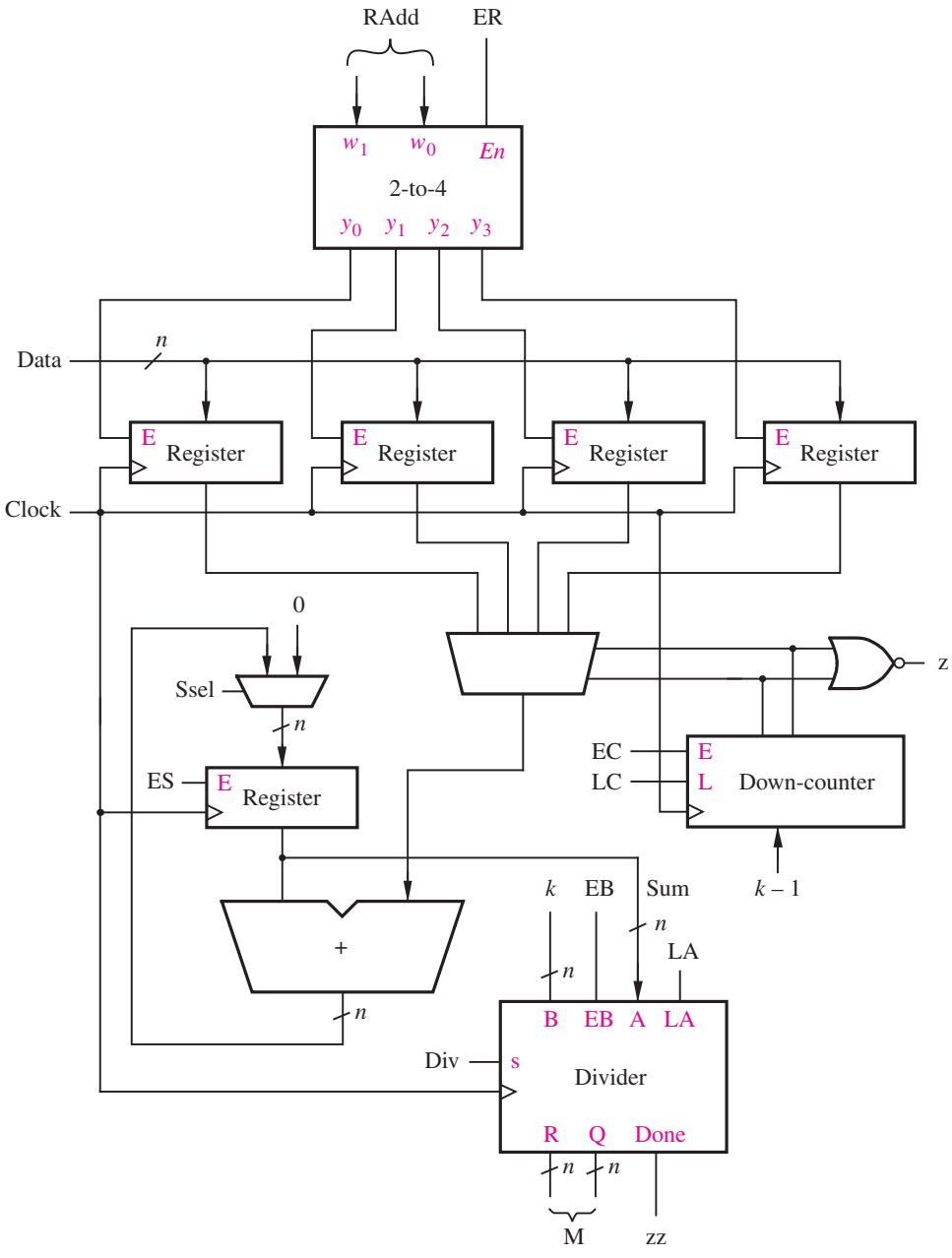


Figure 7.38 Datapath circuit for the mean operation.

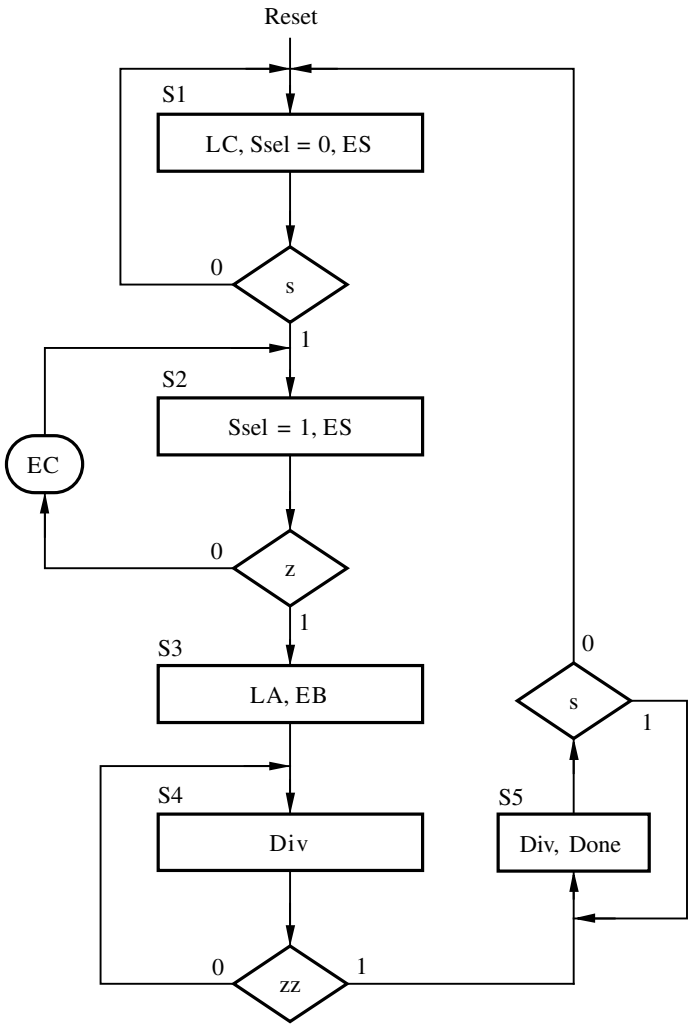


Figure 7.39 ASM chart for the mean operation control circuit.

load the *Sum* and *k* into the *A* and *B* inputs of the divider circuit, respectively. The FSM then enters state $S4$ and asserts the *Div* signal to start the division operation. When it is finished, the divider circuit sets $zz = 1$, and the FSM moves to state $S5$. The mean M appears on the Q and R outputs of the divider circuit. The *Div* signal must still be asserted in state $S5$ to prevent the divider circuit from reinitializing its registers. Note that in the ASM chart in Figure 7.37*b*, only one state is shown for computing $M = Sum/k$, but in Figure 7.39, states $S3$ and $S4$ are used for this purpose. It is possible to combine states $S3$ and $S4$, which we will leave as an exercise for the reader (Problem 7.10).

7.7 SORT OPERATION

Given a list of k unsigned n-bit numbers stored in a set of registers $R_0, \ldots, R_{k-1}$, we wish to design a circuit that can sort the list (contents of the registers) in ascending order. Pseudo-code for a simple sorting algorithm is shown in Figure 7.40. It is based on finding the smallest number in the sublist $R_i, \ldots, R_{k-1}$ and moving that number into R_i, for $i = 1, 2, \ldots, k-2$. Each iteration of the outer loop places the number in R_i into A. Each iteration of the inner loop compares this number to the contents of another register R_j. If the number in R_j is smaller than A, the contents of R_i and R_j are swapped and A is changed to hold the new contents of R_i.

An ASM chart that represents the sorting algorithm is shown in Figure 7.41. In the initial state $S1$, while $s = 0$ the registers are loaded from external data inputs and a counter C_i that represents i in the outer loop is cleared. When the machine changes to state $S2$, A is loaded with the contents of R_i. Also, C_j, which represents j in the inner loop, is initialized to the value of i. State $S3$ is used to initialize j to the value $i + 1$, and state $S4$ loads the value of R_j into B. In state $S5$, A and B are compared, and if $B < A$, the machine moves to state $S6$. States $S6$ and $S7$ swap the values of R_i and R_j. State $S8$ loads A from R_i. Although this step is necessary only for the case where $B < A$, the flow of control is simpler if this operation is performed in both cases. If C_j is not equal to $k - 1$, the machine changes from $S8$ to $S4$, thus remaining in the inner loop. If $C_j = k - 1$ and C_i is not equal to $k - 2$, then the machine stays in the outer loop by changing to state $S2$.

Datapath Circuit

There are many ways to implement a datapath circuit that meets the requirements of the ASM chart in Figure 7.41. One possibility is illustrated in Figures 7.42 and 7.43. Figure 7.42 shows how the registers $R_0, \ldots, R_{k-1}$ can be connected to registers A and B using 4-to-1 multiplexers. We assume the value $k = 4$ for simplicity. Registers A and B are connected to a comparator subcircuit and, through multiplexers, back to the inputs of the

```
for i = 0 to k – 2 do
    A = Ri;
    for j = i C 1 to k – 1 do
        B = Rj;
        if B < A then
            Ri = B;
            Rj = A;
            A = Ri;
        end if;
    end for;
end for;
```

Figure 7.40 Pseudo-code for the sort operation.

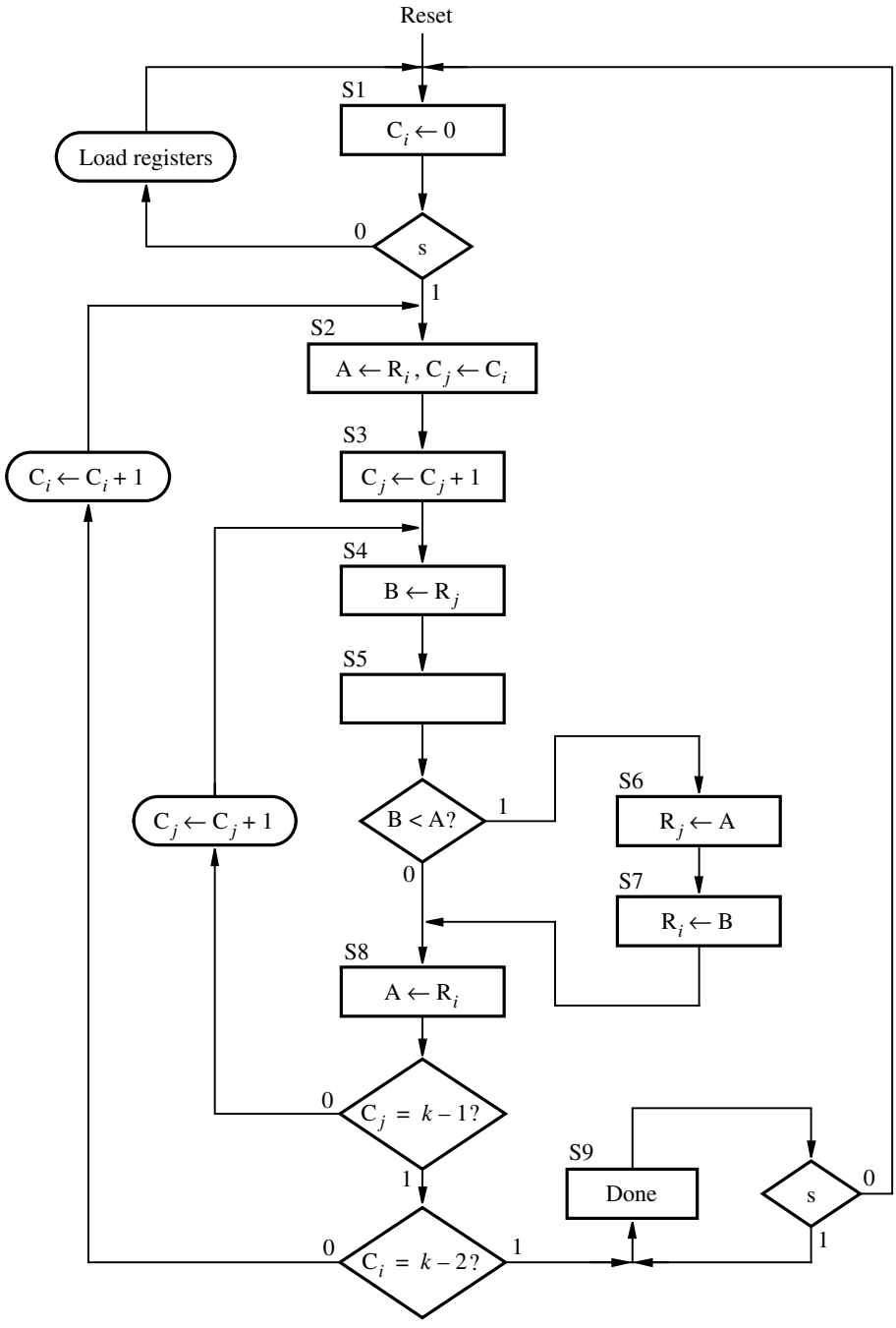


Figure 7.41 ASM chart for the sort operation.

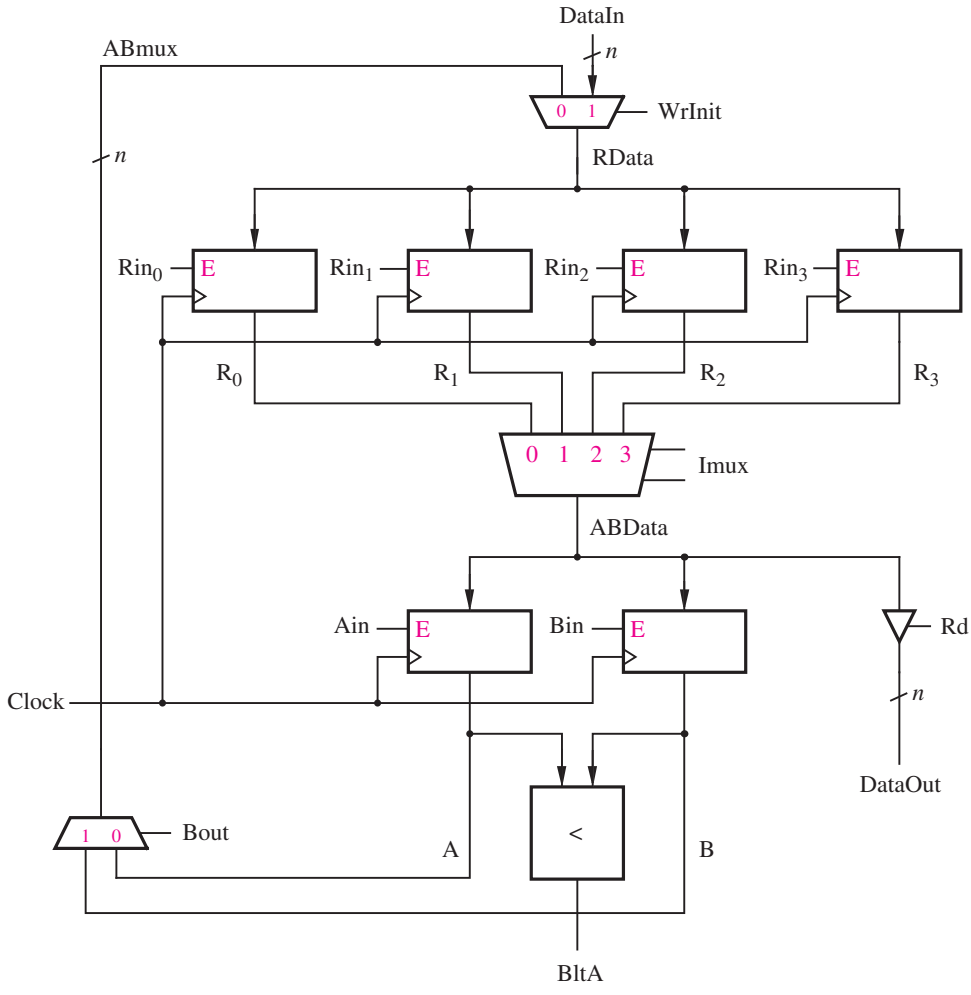


Figure 7.42 A part of the datapath circuit for the sort operation.

registers $R_0, \ldots, R_{k-1}$. The registers can be loaded with initial (unsorted) data using the *DataIn* lines. The data is written (loaded) into each register by asserting the *WrInit* control signal and placing the address of the register on the *RAdd* input. The tri-state driver driven by the *Rd* control signal is used to output the contents of the registers on the *DataOut* output.

The signals $Rin_0, \ldots, Rin_{k-1}$ are controlled by the 2-to-4 decoder shown in Figure 7.43. If $Int = 1$, the decoder is driven by one of the counters C_i or C_j. If $Int = 0$, then the decoder is driven by the external input *RAdd*. The signals z_i and z_j are set to 1 if $C_i = k - 2$ and $C_j = k - 1$, respectively. An ASM chart that shows the control signals used in the datapath circuit is given in Figure 7.44.

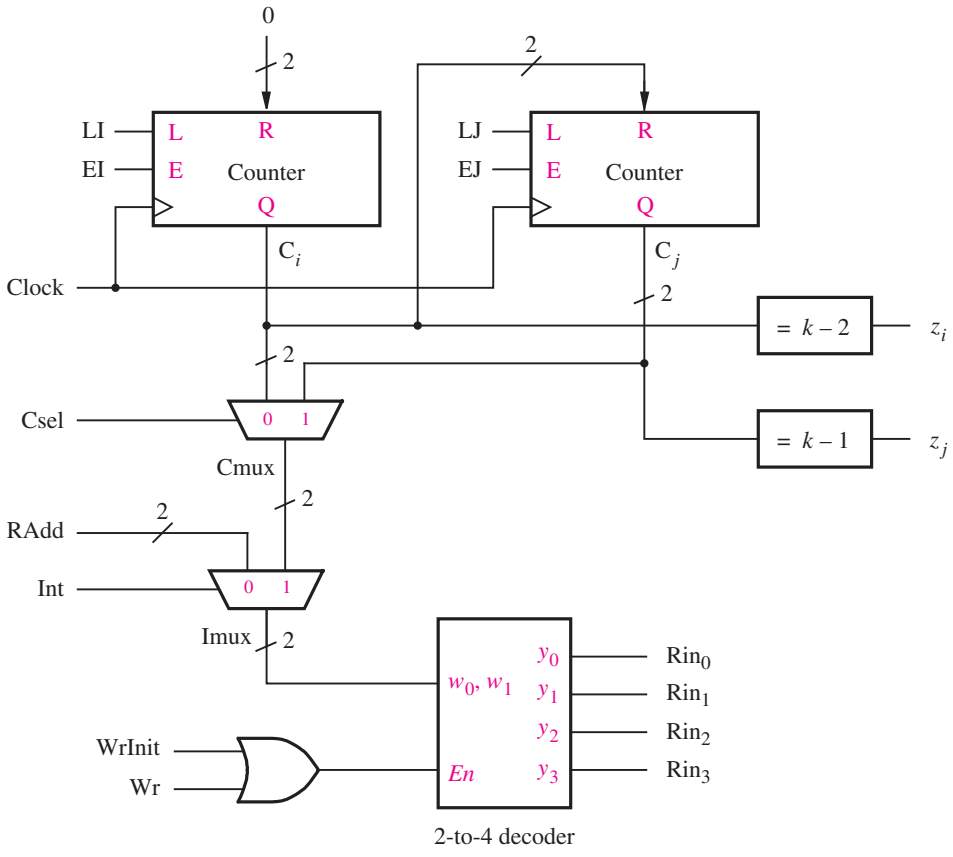


Figure 7.43 A part of the datapath circuit for the sort operation.

Verilog Code

Verilog code for the sorting operation is presented in Figure 7.45. The FSM that controls the sort operation is described in the same way as in previous examples, using the **always** blocks *State_table*, *State_ flipflops*, and *FSM_outputs*. Following these blocks, the code instantiates the registers R_0 to R_3, as well as A and B. The counters C_i and C_j have the instance names *OuterLoop* and *InnerLoop*, respectively. The multiplexers with the outputs *CMux* and *IMux* are specified using the conditional operator. The 4-to-1 multiplexer in Figure 7.42 is defined by the **case** statement that specifies the value of the *ABData* signal for each value of *IMux*. The 2-to-4 decoder in Figure 7.43 with the outputs $Rin_0, \ldots, Rin_3$ is defined by the **case** statement that sets the value of the concatenated signals {Rin3, Rin2, Rin1, Rin0}. Finally, the code specifies the values of the z_i and z_j signals, and defines the tri-state drivers for the *DataOut* output.

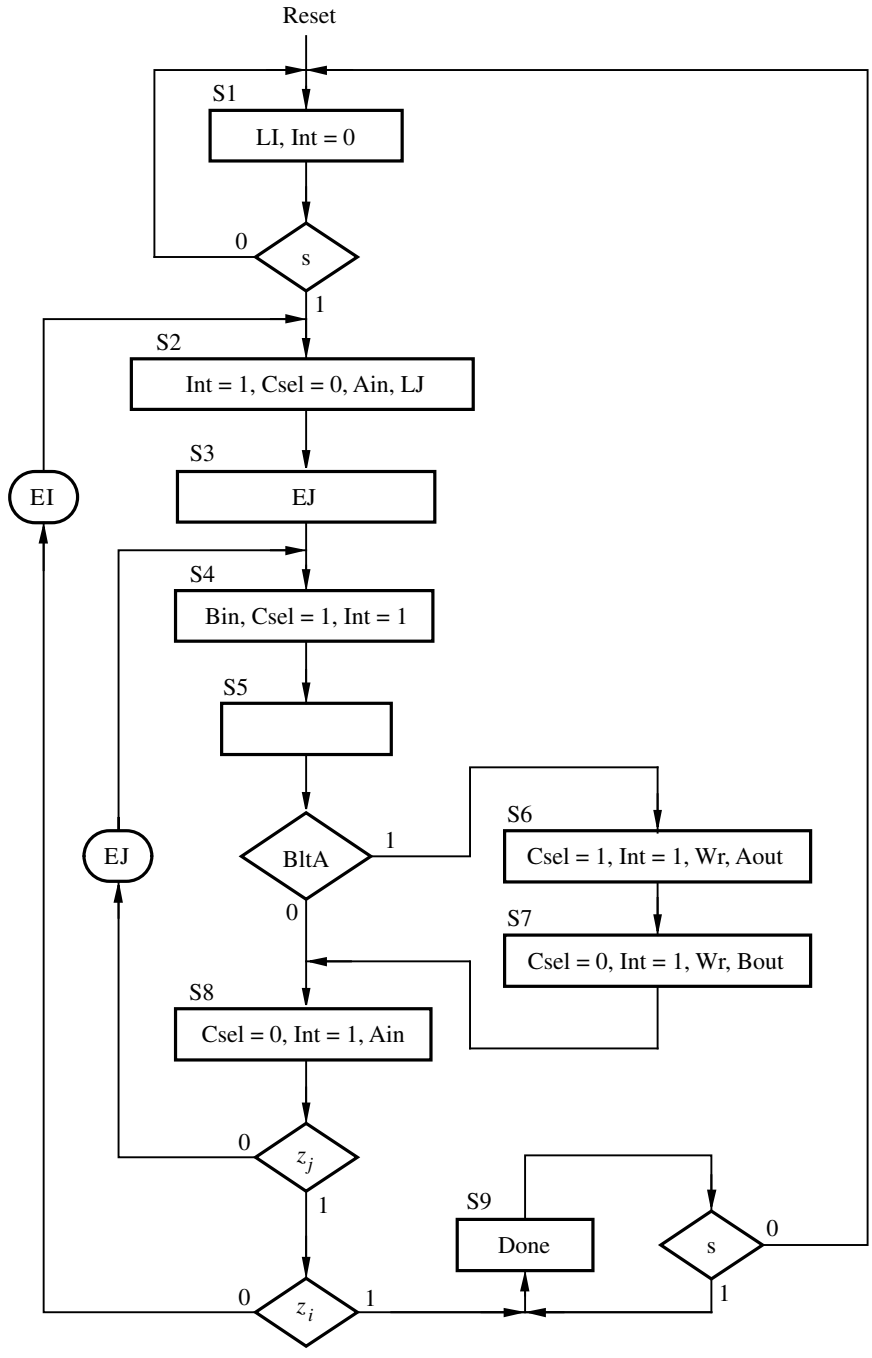


Figure 7.44 ASM chart for the control circuit.

```
module sort (Clock, Resetn, s, WrInit, Rd, DataIn, RAdd, DataOut, Done);
    parameter n = 4;
    input Clock, Resetn, s, WrInit, Rd;
    input [n–1:0] DataIn;
    input [1:0] RAdd;
    output [n–1:0] DataOut;
    output reg Done;
    wire [1:0] Ci, Cj, CMux, IMux;
    wire [n–1:0] R0, R1, R2, R3, A, B, RData, ABMux;
    wire BltA, zi, zj;
    reg Int, Csel, Wr, Ain, Bin, Bout;
    reg LI, LJ, EI, EJ, Rin0, Rin1, Rin2, Rin3;
    reg [3:0] y, Y;
    reg [n–1:0] ABData;

// control circuit
    parameter S1 = 4'b0000, S2 = 4'b0001, S3 = 4'b0010, S4 = 4'b0011;
    parameter S5 = 4'b0100, S6 = 4'b0101, S7 = 4'b0110, S8 = 4'b0111, S9 = 4'b1000;

    always @(s, BltA, zj, zi, y)
    begin: State_table
        case (y)
            S1: if (s == 0) Y = S1;
                else Y = S2;
            S2: Y = S3;
            S3: Y = S4;
            S4: Y = S5;
            S5: if (BltA) Y = S6;
                else Y = S8;
            S6: Y = S7;
            S7: Y = S8;
            S8: if (!zj) Y = S4;
                else if (!zi) Y = S2;
                else Y = S9;
            S9: if (s) Y = S9;
                else Y = S1;
            default: Y = 4'bx;
        endcase
    end
```

. . . continued in Part *b*.

Figure 7.45 Verilog code for the sorting circuit (Part *a*).

```
always @(posedge Clock, negedge Resetn)
begin: State_flipflops
      (Resetn == 0)
       y <= S1;
   else
       y <= Y;
end

always @(y, zj, zi)
begin: FSM_outputs
   // defaults
   Int = 1; Done = 0; LI = 0; LJ = 0; EI = 0; EJ = 0; Csel = 0;
   Wr = 0; Ain = 0; Bin = 0; Bout = 0;
   case (y)
      S1:  begin LI = 1; Int = 0; end
      S2:  begin Ain = 1; LJ = 1; end
      S3:  EJ = 1;
      S4:  begin Bin = 1; Csel = 1; end
      S5:; // no outputs asserted in this state
      S6:  begin Csel = 1; Wr = 1; end
      S7:  begin Wr = 1; Bout = 1; end
      S8:  begin
              Ain = 1;
              if (!zj) EJ = 1;
              else
              begin
                 EJ = 0;
                 if (!zi) EI = 1;
                 else EI = 0;
              end
           end
      S9:  Done = 1;
   endcase
end
```

. . . continued in Part *c*.

Figure 7.45 Verilog code for the sorting circuit (Part *b*).

We implemented the code in Figure 7.45 in an FPGA chip. Figure 7.46 gives an example of a simulation result. Part (*a*) of the figure shows the first half of the simulation, from 0 to 1.25 μs, and part (*b*) shows the second half, from 1.25 μs to 2.5 μs. After resetting the circuit, *WrInit* is set to 1 for four clock cycles, and unsorted data is written into the four registers using the *DataIn* and *RAdd* inputs. After *s* is changed to 1, the FSM changes to state *S*2. States *S*2 to *S*4 load *A* with the contents of R_0 (3) and *B* with the contents of

```
//datapath circuit

regne  Reg0 (RData, Clock, Resetn, Rin0, R0);
   defparam Reg0.n = n;
regne  Reg1 (RData, Clock, Resetn, Rin1, R1);
   defparam Reg1.n = n;
regne  Reg2 (RData, Clock, Resetn, Rin2, R2);
   defparam Reg2.n = n;
regne  Reg3 (RData, Clock, Resetn, Rin3, R3);
   defparam Reg3.n = n;

regne  RegA (ABData, Clock, Resetn, Ain, A);
   defparam RegA.n = n;
regne  RegB (ABData, Clock, Resetn, Bin, B);
   defparam RegB.n = n;

assign  BltA = (B < A) ? 1 : 0;
assign  ABMux = (Bout == 0) ? A : B;
assign  RData = (WrInit == 0) ? ABMux : DataIn;

upcount  OuterLoop (2'b00, Resetn, Clock, EI, LI, Ci);
upcount  InnerLoop (Ci, Resetn, Clock, EJ, LJ, Cj);

assign  CMux = (Csel == 0) ? Ci : Cj;
assign  IMux = (Int == 1) ? CMux : RAdd;
```

. . . continued in Part *d*.

Figure 7.45 Verilog code for the sorting circuit (Part *c*).

R_1 (2). State $S5$ compares B with A, and since $B < A$, the FSM uses states $S6$ and $S7$ to swap the contents of registers R_0 and R_1. In state $S8$, A is reloaded from R_0, which now contains 2. Since z_j is not asserted, the FSM increments the counter C_j and changes back to state $S4$. Register B is now loaded with the contents of R_2 (4), and the FSM changes to state $S5$. Since $B = 4$ is not less than $A = 2$, the machine changes to $S8$ and then back to $S4$. Register B is now loaded with the contents of R_3 (1), which is then compared against $A = 2$ in state $S5$. The contents of R_0 and R_3 are swapped, and the machine changes to $S8$. At this point, the register contents are $R_0 = 1$, $R_1 = 3$, $R_2 = 4$, and $R_3 = 2$. Since $z_j = 1$ and $z_i = 0$, the FSM performs the next iteration of the outer loop by changing to state $S2$. Jumping forward in the simulation time, in Figure 7.46*b* the circuit reaches the state in which $C_i = 2$, $C_j = 3$, and the FSM is in state $S8$. The FSM then changes to state $S9$ and sets *Done* to the value 1. The correctly sorted data is read out of the registers by setting the signal $Rd = 1$ and using the *RAdd* inputs to select each of the registers.

```
always @(WrInit, Wr, IMux, R0, R1, R2, R3)
begin
   case (IMux)
      0:  ABData = R0;
      1:  ABData = R1;
      2:  ABData = R2;
      3:  ABData = R3;
   endcase

   if (WrInit || Wr)
      case (IMux)
         0:  {Rin3, Rin2, Rin1, Rin0} = 4'b0001;
         1:  {Rin3, Rin2, Rin1, Rin0} = 4'b0010;
         2:  {Rin3, Rin2, Rin1, Rin0} = 4'b0100;
         3:  {Rin3, Rin2, Rin1, Rin0} = 4'b1000;
      endcase
   else {Rin3, Rin2, Rin1, Rin0} = 4'b0000;
end

assign  zi = (Ci == 2);
assign  zj = (Cj == 3);
assign  DataOut = (Rd == 0) ? 'bz : ABData;

endmodule
```

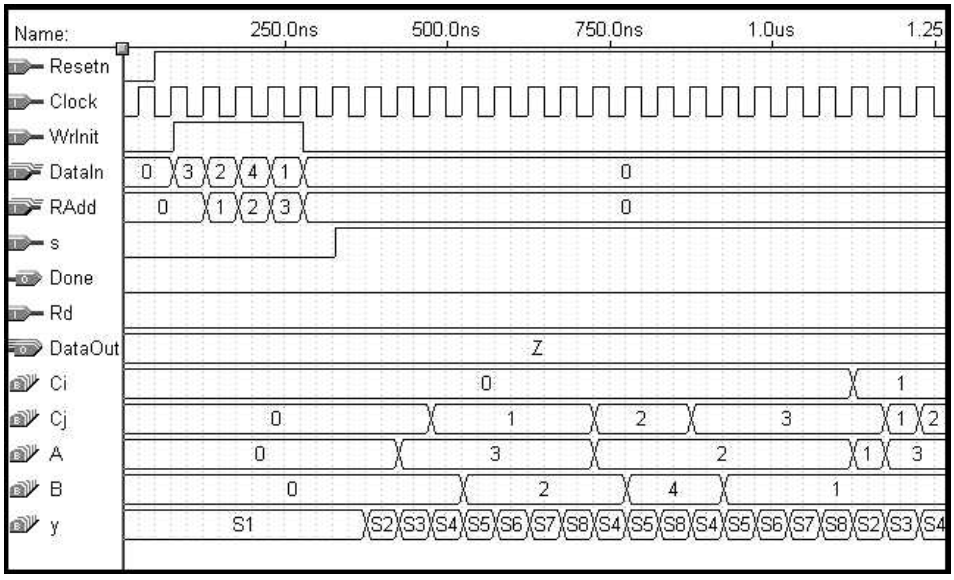
Figure 7.45 Verilog code for the sorting circuit (Part *d*).

7.8 CLOCK SYNCHRONIZATION AND TIMING ISSUES

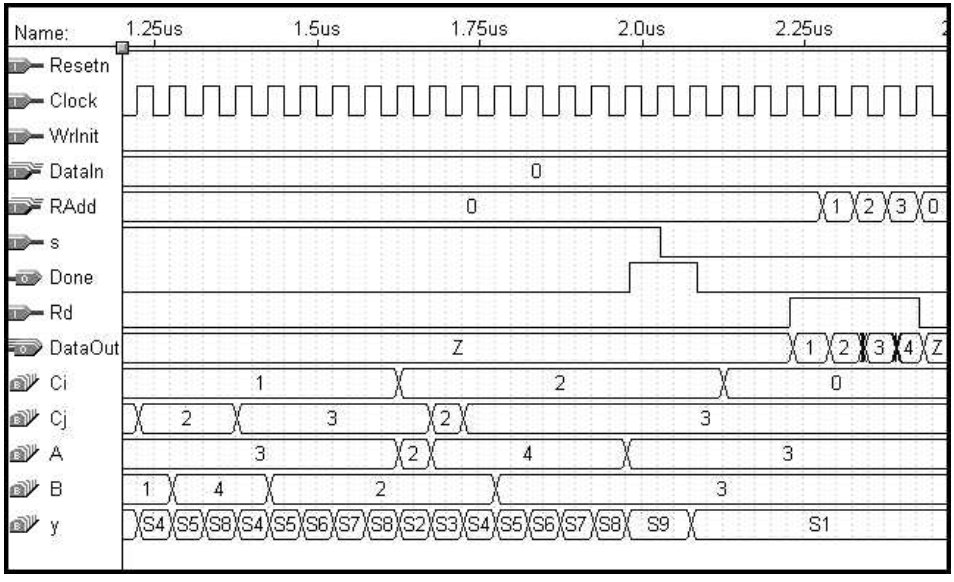
In previous sections we presented several examples of logic circuits that illustrate how a designer may approach the task of designing larger systems. In this section we will examine some timing aspects of such systems.

7.8.1 CLOCK DISTRIBUTION

In Chapter 5 we discussed the need to deal with the clock skew problem. For proper operation of synchronous sequential circuits, it is essential to minimize the clock skew as much as possible. Chips that contain many flip-flops, such as PLDs, use carefully designed networks of wires to distribute the clock signal to the flip-flops. Figure 7.47 gives an example of a clock-distribution network. Each node labeled *ff* represents the clock input



(a) Loading the registers and starting the sort operation



(b) Completing the sort operation and reading the registers

Figure 7.46 Simulation results for the sort operation.

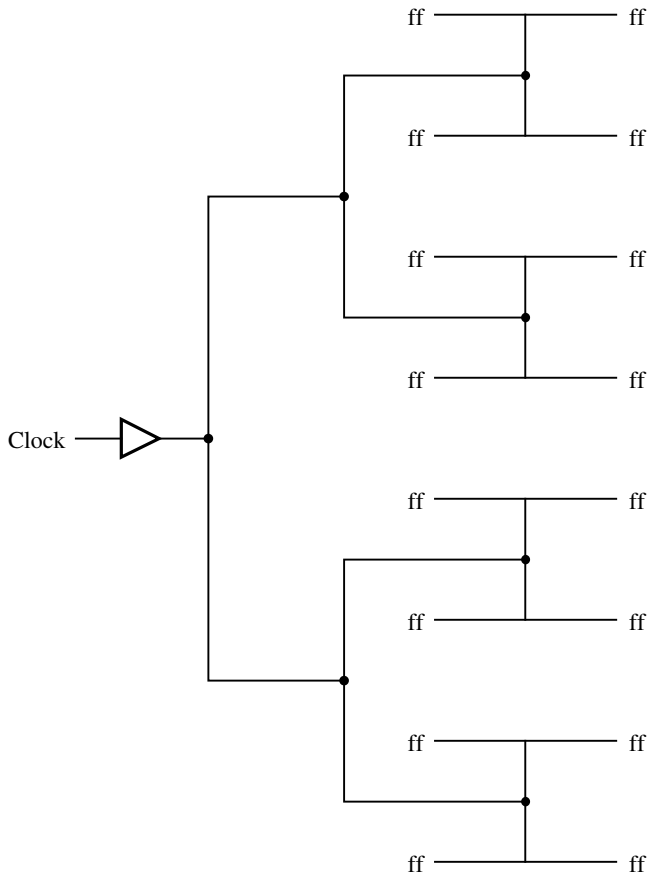


Figure 7.47 An H tree clock distribution network.

of a flip-flop; for clarity, the flip-flops are not shown. The buffer on the left of the figure produces the clock signal. This signal is distributed to the flip-flops such that the length of the wire between each flip-flop and the clock source is the same. Due to the appearance of sections of the wires, which resemble the letter H, the clock distribution network is known as an *H tree*. In PLDs the term *global clock* refers to the clock network. A PLD chip usually provides one or more global clocks that can be connected to all flip-flops. When designing a circuit to be implemented in such a chip, a good design practice is to connect all the flip-flops in the circuit to a single global clock.

It is useful to be able to ensure that a sequential circuit is reset into a known state when power is first applied to the circuit. A good design practice is to connect the asynchronous reset (clear) inputs of all flip-flops to a wiring network that provides a low-skew reset signal. PLDs usually provide a *global reset* wiring network for this purpose.

7.8.2 FLIP-FLOP TIMING PARAMETERS

We discussed the timing parameters for storage elements in Chapter 5. Data to be clocked into a flip-flop must be stable t_{su} before the active clock edge and must remain stable t_h after the clock edge. A change in the value of the output Q appears after the *clock-to-*Q *delay*, t_{cQ}. An *output delay time*, t_{od}, is required for the change in Q to propagate to an output pin on the chip. These timing parameters account for the behavior of an individual flip-flop without considering how the flip-flop is connected to other circuitry in an integrated circuit chip.

Figure 7.48 depicts a flip-flop as part of an integrated circuit. Connections are shown from the flip-flop's clock, *D*, and Q terminals to pins on the chip package. There is an input buffer associated with each pin on the chip. Other circuitry may also be connected to the flip-flop; the shaded box represents a combinational circuit connected to *D*. The propagation delays between the pins on the chip package and the flip-flop are labeled in the figure as t_{Data}, t_{Clock}, and t_{od}.

In digital systems the output signals from one chip are used as the input signals to another chip. Often the flip-flops in all chips are driven by a common clock that has low skew. The signals must propagate from the Q outputs of flip-flops in one chip to the *D* inputs of flip-flops in another chip. To ensure that all timing specifications are met, it is necessary to consider the output delays in one chip and the input delays in another.

The t_{co} delay determines how long it takes from when an active clock edge occurs at the clock pin on the chip package until a change in the output of a flip-flop appears at an output pin on the chip. This delay consists of three main parts. The clock signal must first propagate from its input pin on the chip to the flip-flop's *Clock* input. This delay is labeled t_{Clock} in Figure 7.48. After the clock-to-Q delay t_{cQ}, the flip-flop produces a new output, which takes t_{od} to propagate to the output pin. An example of timing parameters taken from a commercial CPLD chip is $t_{Clock} = 1.5$ ns, $t_{cQ} = 1$ ns, and $t_{od} = 2$ ns. These parameters give the delay from the active clock edge to the change on the output pin as $t_{co} = 4.5$ ns.

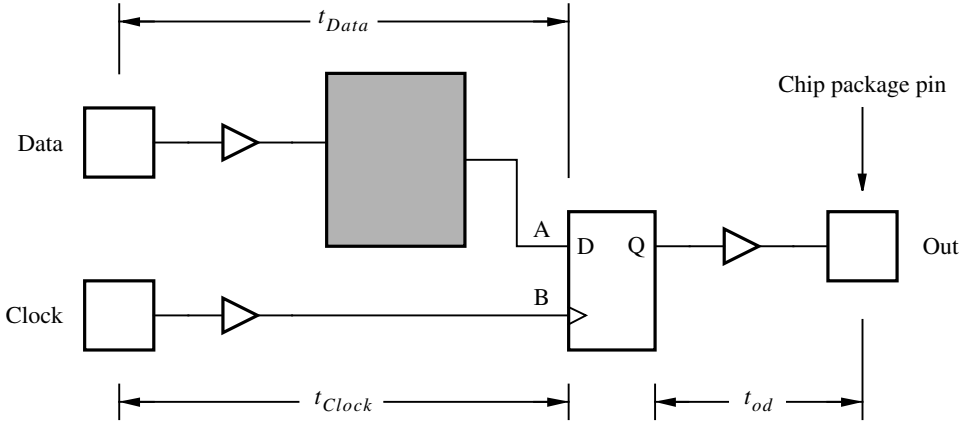


Figure 7.48 A flip-flop in an integrated circuit chip.

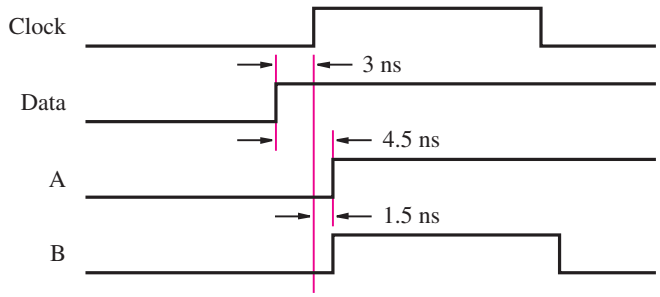


Figure 7.49 Flip-flop timing in a chip.

If chips are separated by a large distance, the propagation delays between them must be taken into consideration. But in most cases the distance between chips is small, and the propagation time of signals between the chips is negligible. Once a signal reaches the input pin on a chip, the relative values of t_{Data} and t_{Clock} (see Figure 7.48) must be considered. For example, in Figure 7.49 we assume that $t_{Data} = 4.5$ ns and $t_{Clock} = 1.5$ ns. The setup time for the flip-flops in the chip is specified as $t_{su} = 3$ ns. In the figure the *Data* signal changes from low to high 3 ns before the positive clock edge, which should meet the setup requirements. The *Data* signal takes 4.5 ns to reach the flip-flop, whereas the *Clock* signal takes only 1.5 ns. The signal labeled *A* and the clock signal labeled *B* reach the flip-flop at the same time. The setup time requirement is violated, and the flip-flop may become unstable. To avoid this condition, it is necessary to increase the setup time as seen from outside the chip.

The hold time for flip-flops is also affected by chip-level delays. The result is usually a reduction in the hold time, rather than an increase. For example, with the timing parameters in Figure 7.49 assume that the hold time is $t_h = 2$ ns. Assume that the signal at the *Data* pin on the chip changes value at exactly the same time that an active edge occurs at the *Clock* pin. The change in the *Clock* signal will reach node *B* $4.5 - 1.5 = 3$ ns before the change in *Data* reaches node *A*. Hence even though the external change in *Data* is coincident with the clock edge, the required hold time of 2 ns is not violated.

For large circuits, ensuring that flip-flop timing parameters are properly adhered to is a challenge. Both the timing parameters of the flip-flops themselves and the relative delays incurred by the clock and data signals must be considered. CAD systems provide tools that can check the setup and hold times at all flip-flops automatically. This task is done using timing simulation, as well as special-purpose timing-analysis tools.

7.8.3 ASYNCHRONOUS INPUTS TO FLIP-FLOPS

In our examples of synchronous sequential circuits, we have assumed that changes in all input signals occur shortly after an active clock edge. The rationale for this assumption is that the inputs to one circuit are produced as the outputs of another circuit, and the same clock signal is used for both circuits. In practice, some of the inputs to a circuit may be

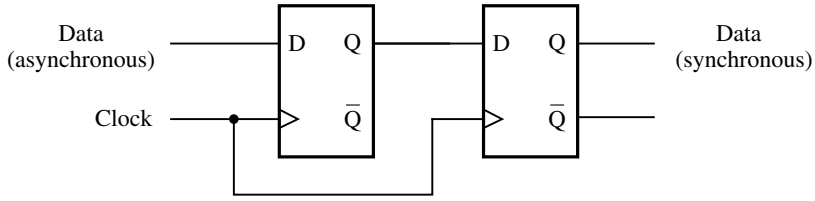


Figure 7.50 Asynchronous inputs.

generated asynchronously with respect to the clock signal. If these signals are connected to the *D* input of a flip-flop, then the setup or hold times may be violated.

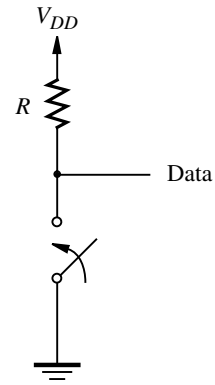
When a flip-flop's setup or hold times are violated, the flip-flop's output may assume a voltage level that does not correspond to either logic value 0 or 1. We say that the flip-flop is in a *metastable* state. The flip-flop eventually settles in one of the stable states, 0 or 1, but the time required to recover from the metastable state is not predictable. A common approach for dealing with asynchronous inputs is illustrated in Figure 7.50. The asynchronous data input is connected to a two-bit shift register. The output of the first flip-flop, labeled *A* in the figure, will sometimes become metastable. But if the clock period is sufficiently long, then *A* will recover to a stable logic value before the next clock pulse occurs. Hence the output of the second flip-flop will not become metastable and can safely be connected to other parts of the circuit. The synchronization circuit introduces a delay of one clock cycle before the signal can be used by the rest of the circuit.

Commercial chips, such as PLDs, specify the minimum allowable clock period that has to be used for the circuit in Figure 7.50 to solve the metastability problem. In practice, it is not possible to *guarantee* that node *A* will always be stable before a clock edge occurs. The data sheets specify a probability of node *A* being stable, as a function of the clock period. We will not pursue this issue further; the interested reader can refer to references [10, 11] for a more detailed discussion.

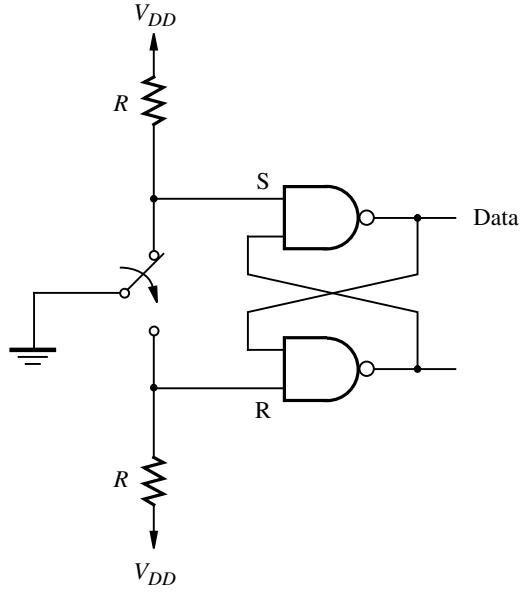
7.8.4 SWITCH DEBOUNCING

Inputs to a logic circuit are sometimes generated by mechanical switches. A problem with such switches is that they bounce away from their contact points when changed from one position to the other. Figure 7.51*a* shows a single-pole single-throw switch that provides an input to a logic circuit. If the switch is open, then the *Data* signal has the value 1. When the switch is thrown to the closed position, *Data* becomes 0, but the switch bounces for some time, causing *Data* to oscillate between 1 and 0. The bouncing typically persists for about 10 ms.

One approach to deal with this problem is to add a capacitor, *C*, between the *Data* node and ground. When the switch is open, *C* is charged to the voltage V_{DD}. When the switch is thrown to the closed position, *C* is quickly discharged to 0 V (ground) through the switch which has a resistance close to zero ohm. Now, when the switch contact bounces away from its closed position, *C* will start to charge toward V_{DD} again. But, this charging will be



(a) Single-pole single-throw switch



(b) Single-pole double-throw switch with a basic SR latch

Figure 7.51 Switch debouncing circuit.

slower due to the resistor R. By choosing the right sizes of R and C, it is possible to prevent the *Data* node from significantly changing its voltage during bouncing.

Another possible solution is to use a circuit, such as a counter, to measure an appropriately long delay to wait for the bouncing to stop (see Problem 7.20).

A good approach for dealing with switch bouncing is depicted in Figure 7.51*b*. It uses a single-pole double-throw switch and a basic SR latch to generate an input to a logic circuit.

When the switch is in the bottom position, the R input on the latch is 0 and $Data = 0$. When the switch is thrown to the top position, the S input on the latch becomes 0, which sets $Data$ to 1. If the switch bounces away from the top position, the inputs to the latch become $R = S = 1$ and the value $Data = 1$ is stored by the latch. When the switch is thrown to the bottom position, $Data$ changes to 0 and this value is stored in the latch if the switch bounces. Note that when a switch bounces, it cannot bounce fully between the S and R terminals; it only bounces slightly away from one of the terminals and then back to it.

7.9 CONCLUDING REMARKS

This chapter has provided several examples of digital systems that include one or more FSMs as well as building blocks like adders, registers, shift registers, and counters. We have shown how ASM charts can be used as an aid for designing a digital system, and we have shown how the circuits can be described using Verilog code. A number of practical issues have been discussed, such as clock distribution, synchronization of asynchronous inputs, and switch debouncing. Some notable books that also deal with the material presented in this chapter include [3–10].

PROBLEMS

7.1 In Section 7.1 we showed a digital system with three registers, $R1$ to $R3$, and we designed a control circuit that can be used to swap the contents of registers $R1$ and $R2$. Give an ASM chart that represents this digital system and the swap operation.

7.2 (a) For the ASM chart derived in Problem 7.1, show another ASM chart that specifies the required control signals to control the datapath circuit. Assume that multiplexers are used to implement the bus that connects the registers, as shown in Figure 7.4.
(b) Write complete Verilog code for the system in Problem 7.1, including the control circuit described in part (*a*).
(c) Synthesize a circuit from the Verilog code written in part (*b*) and show a timing simulation that illustrates correct functionality of the circuit.

7.3 In Section 7.2 we designed a processor that performs the operations listed in Table 7.1. Design a modified circuit that performs an additional operation Swap Rx, Ry. This operation swaps the contents of registers Rx and Ry. Use three bits $f_2 f_1 f_0$ to represent the input F shown in Figure 7.11 because there are now five operations, rather than four. Add a new register, named *Tmp*, into the system, to be used for temporary storage during the swap operation. Show logic expressions for the outputs of the control circuit, as was done in Section 7.2.

7.4 In Section 7.2 we gave the design for a circuit that works as a processor. Give an ASM chart that describes the functionality of this processor.

7.5 (a) For the ASM chart derived in Problem 7.4, show another ASM chart that specifies the required control signals to control the datapath circuit in the processor. Assume that multiplexers are used to implement the bus that connects the registers, $R0$ to $R3$, in the processor.
(b) Write complete Verilog code for the system in Problem 7.4, including the control circuit described in part (a).
(c) Synthesize a circuit from the Verilog code written in part (b) and show a timing simulation that illustrates correct functionality of the circuit.

7.6 The ASM chart in Figure 7.17, which describes the bit-counting circuit, includes Moore-type outputs in states $S1$, $S2$, and $S3$, and it has a Mealy-type output in state $S2$.
(a) Show how the ASM chart can be modified such that it has only Moore-type outputs in state $S2$.
(b) Give the ASM chart for the control circuit corresponding to part (a).
(c) Give Verilog code that represents the modified control circuit.

7.7 Figure 7.24 shows the datapath circuit for the shift-and-add multiplier. It uses a shift register for B so that b_0 can be used to decide whether or not A should be added to P. A different approach is to use a normal register to hold operand B and to use a counter and multiplexer to select bit b_i in each stage of the multiplication operation.
(a) Show the ASM chart that uses a normal register for B, instead of a shift register.
(b) Show the datapath circuit corresponding to part (a).
(c) Give the ASM chart for the control circuit corresponding to part (b).
(d) Give Verilog code that represents the multiplier circuit.

7.8 Write Verilog code for the divider circuit that has the datapath in Figure 7.30 and the control circuit represented by the ASM chart in Figure 7.31.

7.9 Section 7.5 shows how to implement the traditional long division that is done by "hand." A different approach for implementing integer division is to perform repeated subtraction as indicated in the pseudo-code in Figure P7.1.
(a) Give an ASM chart that represents the pseudo-code in Figure P7.1.
(b) Show the datapath circuit corresponding to part (a).
(c) Give the ASM chart for the control circuit corresponding to part (b).
(d) Give Verilog code that represents the divider circuit.
(e) Discuss the relative merits and drawbacks of your circuit in comparison with the circuit designed in Section 7.5.

```
Q = 0;
R = A;
while ((R − B) > 0) do
      R = R − B;
      Q = Q + 1;
end while;
```

Figure P7.1 Pseudo-code for integer division.

7.10 In the ASM chart in Figure 7.39, the two states $S3$ and $S4$ are used to compute the mean $M = Sum/k$. Show a modified ASM chart that combines states $S3$ and $S4$ into a single state, called $S3$.

7.11 Write Verilog code for the FSM represented by your ASM chart defined in Problem 7.10.

7.12 In the ASM chart in Figure 7.41, we specify the assignment $C_j \leftarrow C_i$ in state $S2$, and then in state $S3$ we increment C_j by 1. Is it possible to eliminate state $S3$ if the assignment $C_j \leftarrow C_i + 1$ is performed in $S2$? Explain any implications that this change has on the control and datapath circuits.

7.13 Figure 7.40 gives pseudo-code for the sorting operation in which the registers being sorted are indexed using variables i and j. In the ASM chart in Figure 7.41, variables i and j are implemented using the counters C_i and C_j. A different approach is to implement i and j using two shift registers.
(a) Redesign the circuit for the sorting operation using the shift registers instead of the counters to index registers $R_0, \ldots, R_3$.
(b) Give Verilog code for the circuit designed in part (a).
(c) Discuss the relative merits and drawbacks of your circuit in comparison with the circuit that uses the counters C_i and C_j.

7.14 Design a circuit that finds the log_2 of an operand that is stored in an n-bit register. Show all steps in the design process and state any assumptions made. Give Verilog code that describes your circuit.

7.15 The circuit designed in Section 7.6 uses an adder to compute the sum of the contents of the registers. The divider subcircuit used to compute $M = Sum/k$ also includes an adder. Show how the circuit can be redesigned so that it contains only a single adder subcircuit that is used both for the summation operation and the division operation. Show only the extra circuitry needed to connect to the adder; and explain its operation.

7.16 Give Verilog code for the circuit designed in Problem 7.15, including both the datapath and control circuits.

7.17 The pseudo-code for the sorting operation given in Figure 7.40 uses registers A and B to hold the contents of the registers being sorted. Show pseudo-code for the sorting operation that uses only register A to hold temporary data during the sorting operation. Give a corresponding ASM chart that represents the datapath and control circuits needed. Use multiplexers to interconnect the registers, in the style shown in Figure 7.42. Give a separate ASM chart that represents the control circuit.

7.18 Give Verilog code for the sorting circuit designed in Problem 7.17.

7.19 Consider the design of a circuit that controls the traffic lights at the intersection of two roads. The circuit generates the outputs $G1$, $Y1$, $R1$ and $G2$, $Y2$, $R2$. These outputs represent the states of the green, yellow, and red lights, respectively, on each road. A light is turned on if the corresponding output signal has the value 1. The lights have to be controlled in the following manner: when $G1$ is turned on it must remain on for a time period called t_1 and then be turned off. Turning off $G1$ must result in $Y1$ being immediately turned on; it should remain on for a time period called t_2 and then be turned off. When either $G1$ or $Y1$ is on, $R2$ must be on and $G2$ and $Y2$ must be off. Turning off $Y1$ must result in $G2$ being immediately

turned on for the t_1 time period. When $G2$ is turned off, $Y2$ is turned on for the t_2 time period. Of course, when either $G2$ or $Y2$ is turned on, $R1$ must be turned on and $G1$ and $Y1$ must be off.

(a) Give an ASM chart that describes the traffic-light controller. Assume that two down-counters exist, one that is used to measure the t_1 delay and another that is used to measure t_2. Each counter has parallel-load and enable inputs. These inputs are used to load an appropriate value representing either the t_1 or t_2 delay and then allow the counter to count down to 0.

(b) Give an ASM chart for the control circuit for the traffic-light controller.

(c) Write complete Verilog code for the traffic-light controller, including the control circuit from part (*a*) and counters to represent t_1 and t_2. Use any convenient clock frequency to clock the circuit and assume convenient count values to represent t_1 and t_2. Give simulation results that illustrate the operation of your circuit.

7.20 Assume that you need to use a single-pole single-throw switch as shown in Figure 7.51*a*. Show how a counter can be used as a means of debouncing the *Data* signal produced by the switch. (*Hint:* Design an FSM that has *Data* as an input and produces the output z, which is the debounced version of *Data*. Assume that you have access to a *Clock* input signal with the frequency 102.4 kHz, which can be used as needed.)

7.21 Clock signals can be generated using special purpose chips. One example of such a chip is the 555 programmable timer, which is depicted in Figure P7.2. By choosing particular values for the resistors R_a and R_b and the capacitor C_1, the 555 timer can be used to produce

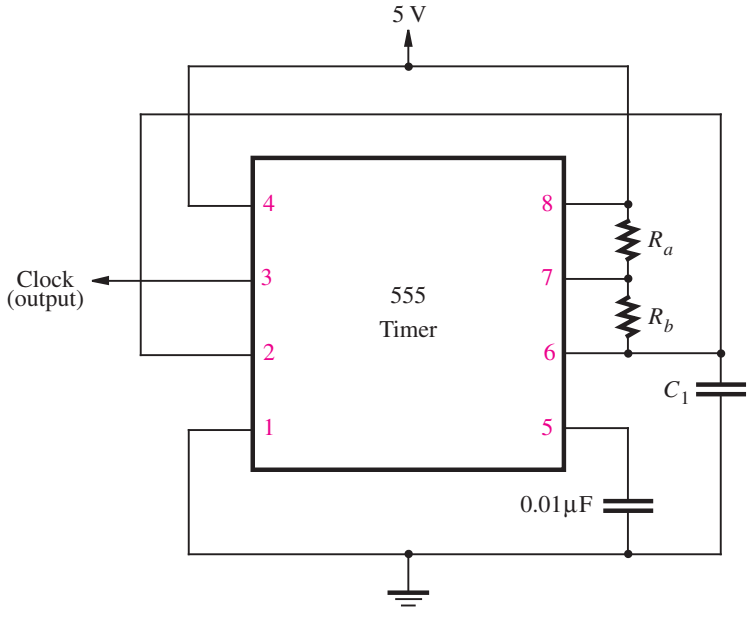


Figure P7.2 The 555 programmable timer chip.

a desired clock signal. It is possible to choose both the period of the clock signal and its duty cycle. The term *duty cycle* refers to the percentage of the clock period for which the signal is high. The following equations define the clock signal produced by the chip

$$\text{Clock period} = 0.7(R_a + 2R_b)C_1$$

$$\text{Duty cycle} = \frac{R_a + R_b}{R_a + 2R_b}$$

(a) Determine the values of R_a, R_b, and C_1 needed to produce a clock signal with a 50 percent duty cycle and a frequency of about 500 kHz.
(b) Repeat part (*a*) for a duty cycle of 75 percent.

REFERENCES

1. V. C. Hamacher, Z. G. Vranesic, and S. G. Zaky, *Computer Organization*, 5th ed. (McGraw-Hill: New York, 2002).
2. D. A. Patterson and J. L. Hennessy, *Computer Organization and Design—The Hardware/Software Interface*, 3rd ed. (Morgan Kaufmann: San Francisco, CA, 2004).
3. D. D. Gajski, *Principles of Digital Design* (Prentice-Hall: Upper Saddle River, NJ, 1997).
4. M. M. Mano and C. R. Kime, *Logic and Computer Design Fundamentals* (Prentice-Hall: Upper Saddle River, NJ, 1997).
5. J. P. Daniels, *Digital Design from Zero to One* (Wiley: New York, 1996).
6. V. P. Nelson, H. T. Nagle, B. D. Carroll, and J. D. Irwin, *Digital Logic Circuit Analysis and Design* (Prentice-Hall: Englewood Cliffs, NJ, 1995).
7. R. H. Katz and G. Borriello, *Contemporary Logic Design*, 2nd ed., (Pearson Prentice-Hall: Upper Saddle River, N.J., 2005).
8. J. P. Hayes, *Introduction to Logic Design* (Addison-Wesley: Reading, MA, 1993).
9. C. H. Roth Jr., *Fundamentals of Logic Design*, 5th ed., (Thomson/Brooks/Cole: Belmont, Ca., 2004).
10. J. F. Wakerly, *Digital Design Principles and Practices*, 4th ed. (Prentice-Hall: Englewood Cliffs, N.J., 2005).
11. C. J. Myers, *Asynchronous Circuit Design*, (Wiley: New York, 2001).

This page intentionally left blank