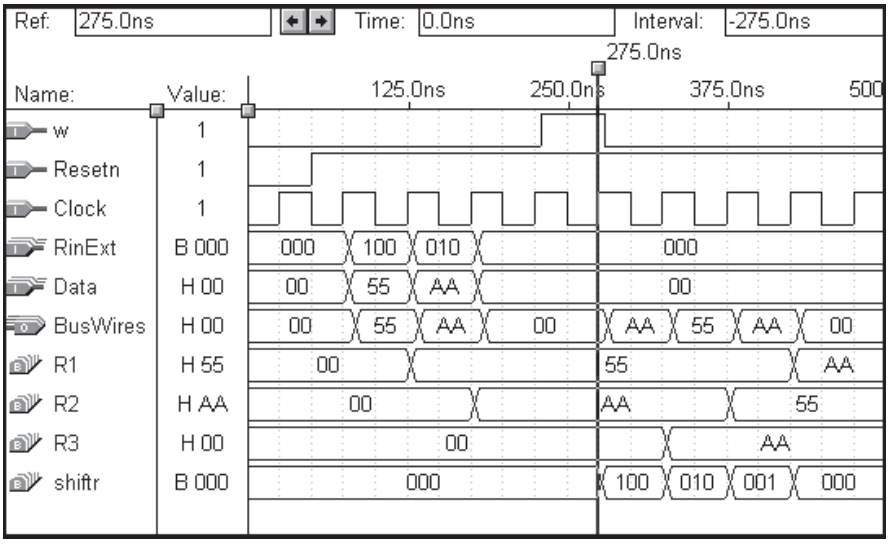

Figure 7.72 Timing simulation for the Verilog code in Figure 7.71.

7.14.2 SIMPLE PROCESSOR

A second example of a digital system like the one in Figure 7.60 is shown in Figure 7.73. It has four *n*-bit registers, $R0, \ldots, R3$, that are connected to the bus with tri-state buffers. External data can be loaded into the registers from the *n*-bit *Data* input, which is connected to the bus using tri-state buffers 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 5.3. 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 7.73 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 7.2. 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 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

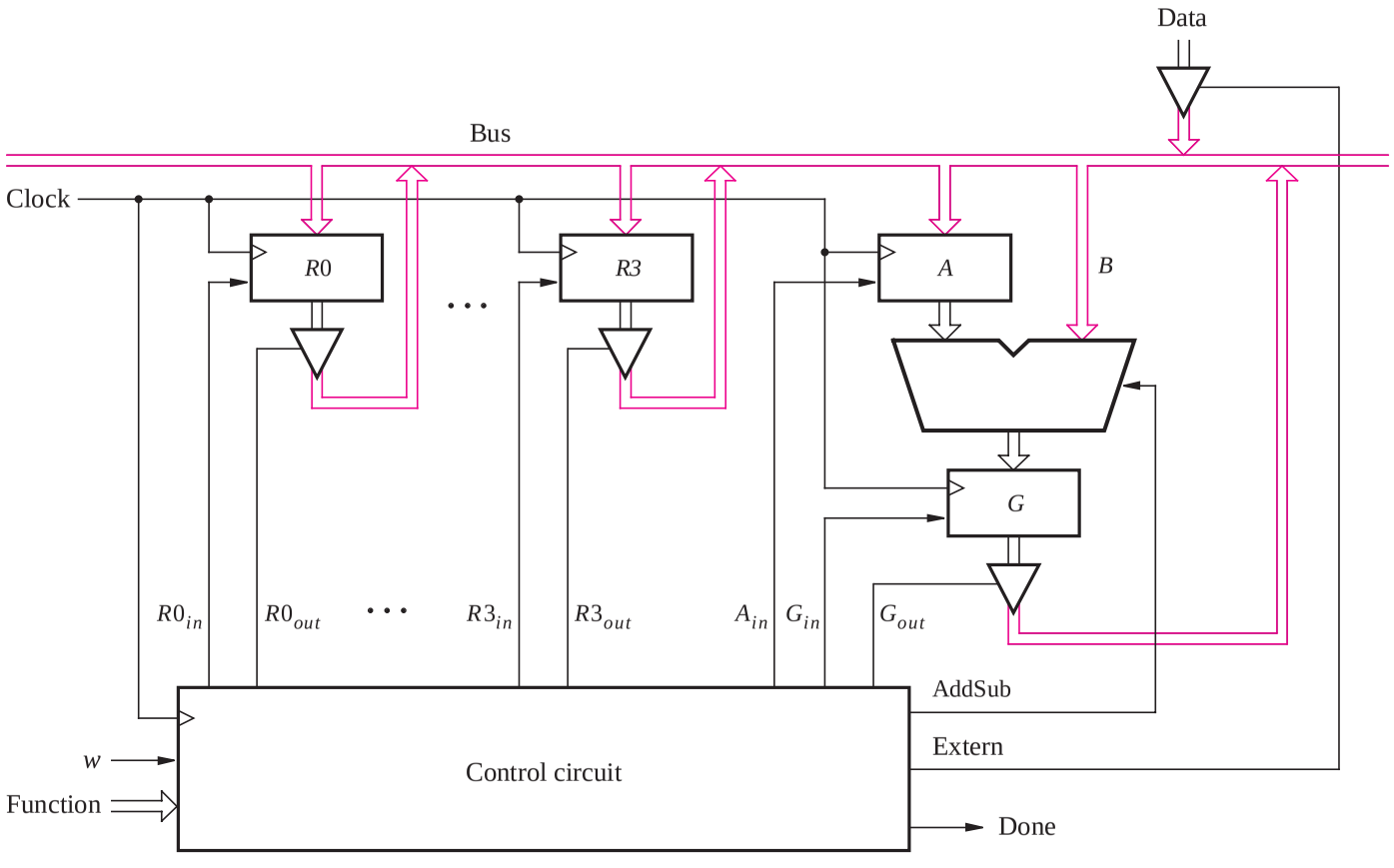

Figure 7.73 A digital system that implements a simple processor.

Table 7.2 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 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 7.2 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 7.60 we used a shift register to implement the control circuit. It is possible to use a similar design for the system in Figure 7.73. To illustrate a different 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 7.74 shows a two-bit up-counter connected to a 2-to-4 decoder. Decoders are discussed in section 6.2. 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

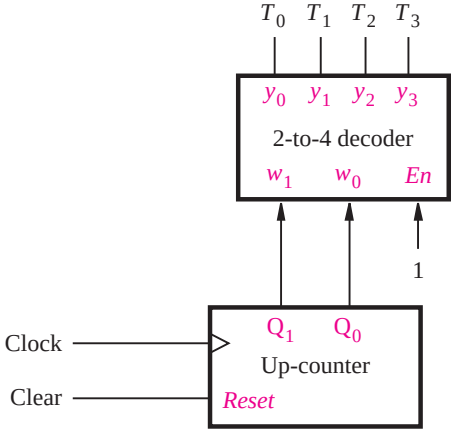

Figure 7.74 A part of the control circuit for the processor.

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 7.75 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_1Rx_0 are a binary number that identifies the *Rx* operand, while Ry_1Ry_0 identifies the *Ry* operand. The *Function* inputs are stored in a six-bit Function Register when the FR_{in} signal is asserted.

Figure 7.75 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 7.74 and 7.75 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 7.74 and 7.75.

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

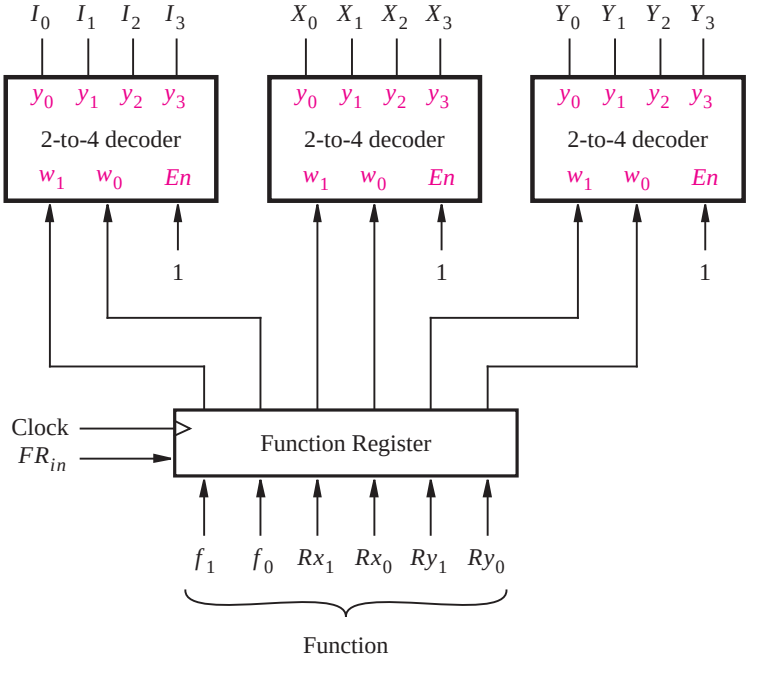

Figure 7.75 The function register and decoders.

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 7.3. 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

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

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 7.3 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 $R_{out} = Y$.

Table 7.3 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*

We will develop the expressions for $R0_{in}$ and $R0_{out}$ by examining Table 7.3 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 7.74 and 7.75, combined with the circuits represented by the above expressions, implement the control circuit in Figure 7.73.

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 7.73. The first style uses tri-state buffers 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 7.3 to describe the outputs of the control circuit.

Verilog code for an up-counter is shown in Figure 7.56. A modified version of this counter, named *upcount*, is shown in the code in Figure 7.76. 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 6.35, 7.66, and 7.67.

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

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

endmodule
```

Figure 7.76 A two-bit up-counter with synchronous reset.

Complete code for the processor is given in Figure 7.77. The instantiated modules *counter* and *decT* represent the subcircuits in Figure 7.74. 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 7.75. The *functionreg* module represents the Function Register with the data inputs *Func* and the

```
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 [7:0] BusWires;
    output Done;
    wire [7:0] BusWires;
    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, 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, I);
    dec2to4 decX (FuncReg[3:4], 1, Xreg);
    dec2to4 decY (FuncReg[5:6], 1, 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 7.77 Code for the prcoessor (Part *a*).

```
    // RegCntl
    always @(I or T or Xreg or 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[1] & Y[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 or A or 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 7.77 Code for the processor (Part *b*).

outputs *FuncReg*. The instantiated modules *decI*, *decX*, and *decY* represent the decoders in Figure 7.75. 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 buffers and registers in the processor are instantiated.

Using Multiplexers and Case Statements

We showed in Figure 7.65 that a bus can be implemented with multiplexers, rather than tri-state buffers. Verilog code that describes the processor using this approach is shown in Figure 7.78. The code illustrates a different way of describing the control circuit in the

```
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 [7:0] BusWires;
    output Done;
    reg [7:0] BusWires, Sum;
    reg [0:3] Rin, Rout;
    reg Extern, Done, 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, Xreg);
    dec2to4 decY (FuncReg[5:6], 1, Y);

    always @(Count or I or Xreg or 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 7.78 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 7.78 Alternative code for the processor (Part *b*).

processor. It does not give logic expressions for the signals *Extern*, *Done*, and so on, as in Figure 7.77. Instead, **case** statements are used to represent the information shown in Table 7.3. 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 7.3. As explained in section 7.12.2, when the value of a signal is not specified, the signal retains its current value. This implied memory results in a feedback connection 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 7.77 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 7.78 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 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.

```
// alu
always @(AddSub or A or 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 or R0 or R1 or R2 or R3 or G or 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 7.78 Alternative code for the processor (Part c).

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 7.3, there is a nested **case** statement that enumerates the four values of I. As indicated by the comments in the code, the nested **case** statements correspond exactly to the information given in Table 7.3.

At the end of Figure 7.78, the bus 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*.

The circuits synthesized from the code in Figures 7.77 and 7.78 are functionally equivalent. The style of code in Figure 7.78 has the advantage that it does not require the manual effort of analyzing Table 7.3 to generate the logic expressions for the control signals in Figure 7.77. By using the style of code in Figure 7.78, these expressions are produced automatically by the Verilog compiler as a result of analyzing the **case** statements. The style of code in Figure 7.78 is less prone to careless errors. Also, using this style of code it

would be straightforward to provide additional capabilities in the processor, such as adding other operations.

We synthesized a circuit to implement the code in Figure 7.78 in a chip. Figure 7.79 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 in the simulation time, the values of both inputs F and Rx are 00. Hence the operation corresponds to "*Load* $R0$, *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* $R1$, $R0$." 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* $R3$, $R1$") copies the contents of $R1$ (7F) into $R3$. Finally, the operation starting at 1450 ns ("*Sub* $R3$, $R2$") subtracts the contents of $R2$ (22) from the contents of $R3$ (7F), producing the correct result, $7F - 22 = 5D$.

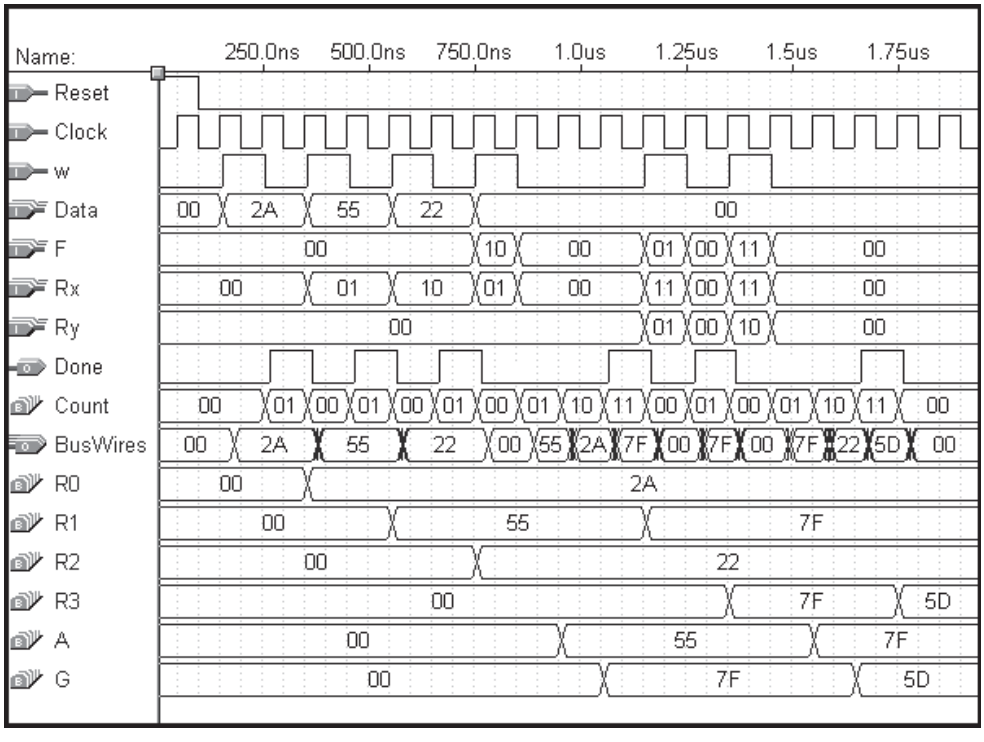

Figure 7.79 Timing simulation for the Verilog code in Figure 7.78.