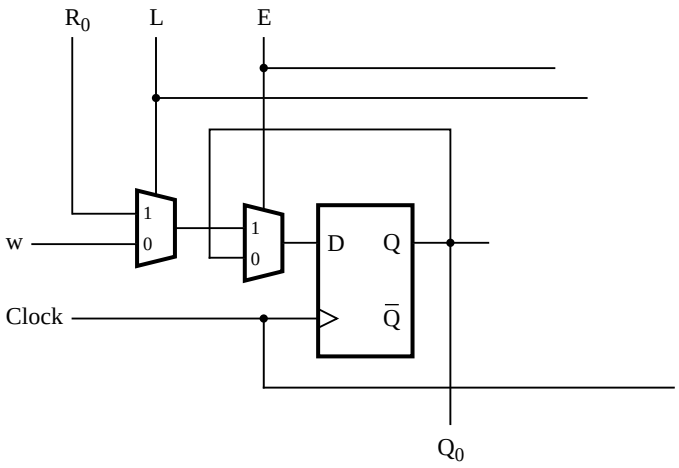
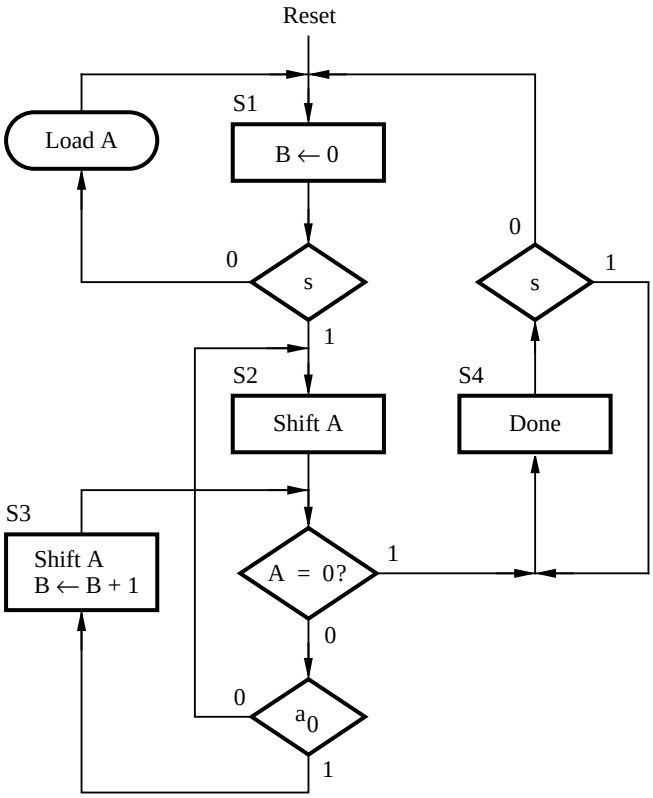
Chapter 10

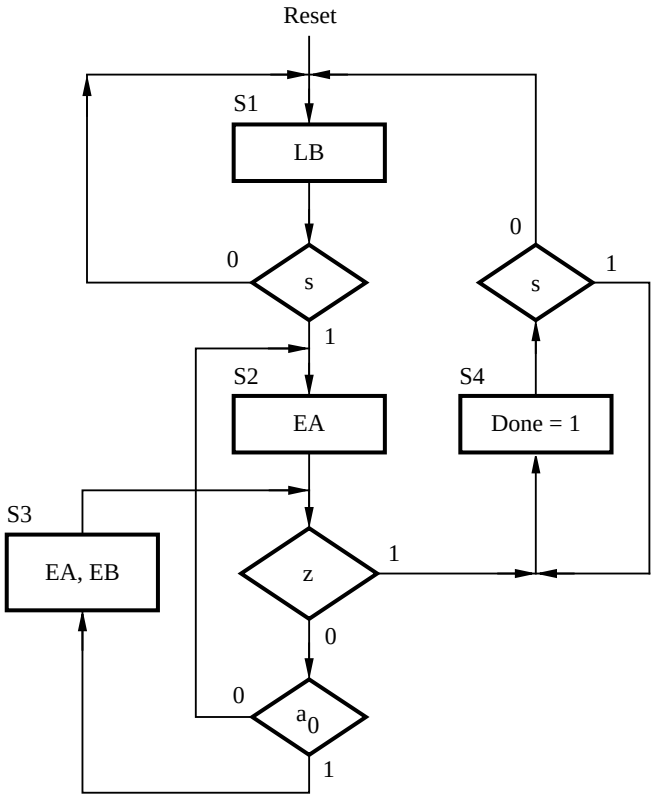
10.1. In the modified shift register the order of the multiplexers that perform the load and enable operations are reversed from the order in Figure 10.4. Bit zero of the modified register is show below.



10.2. (a) A modified ASM chart that has only Moore-type outputs in state S2 is given below.



(*b*)

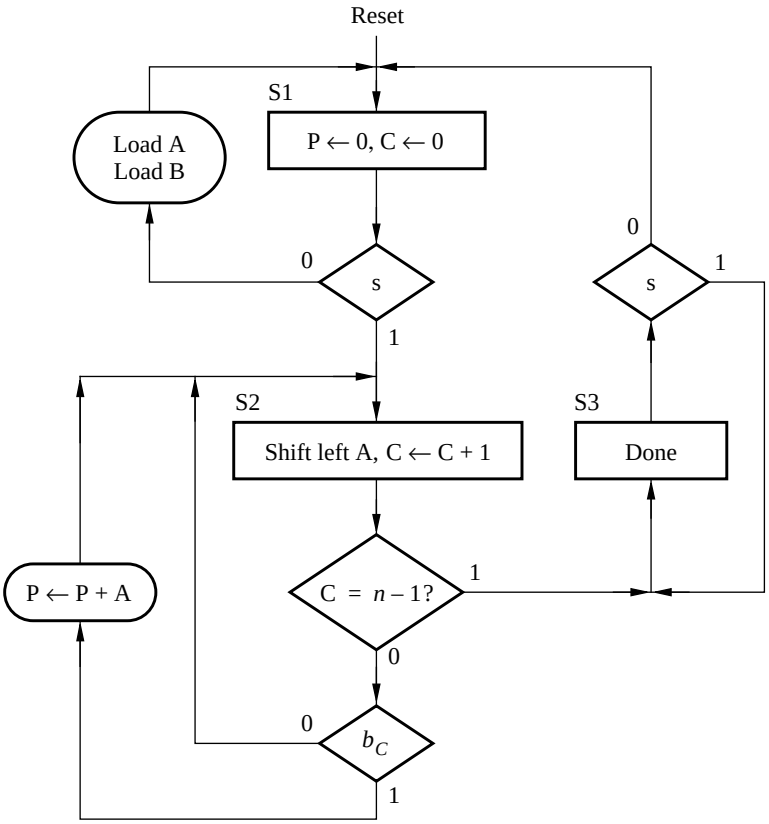


(*c*)

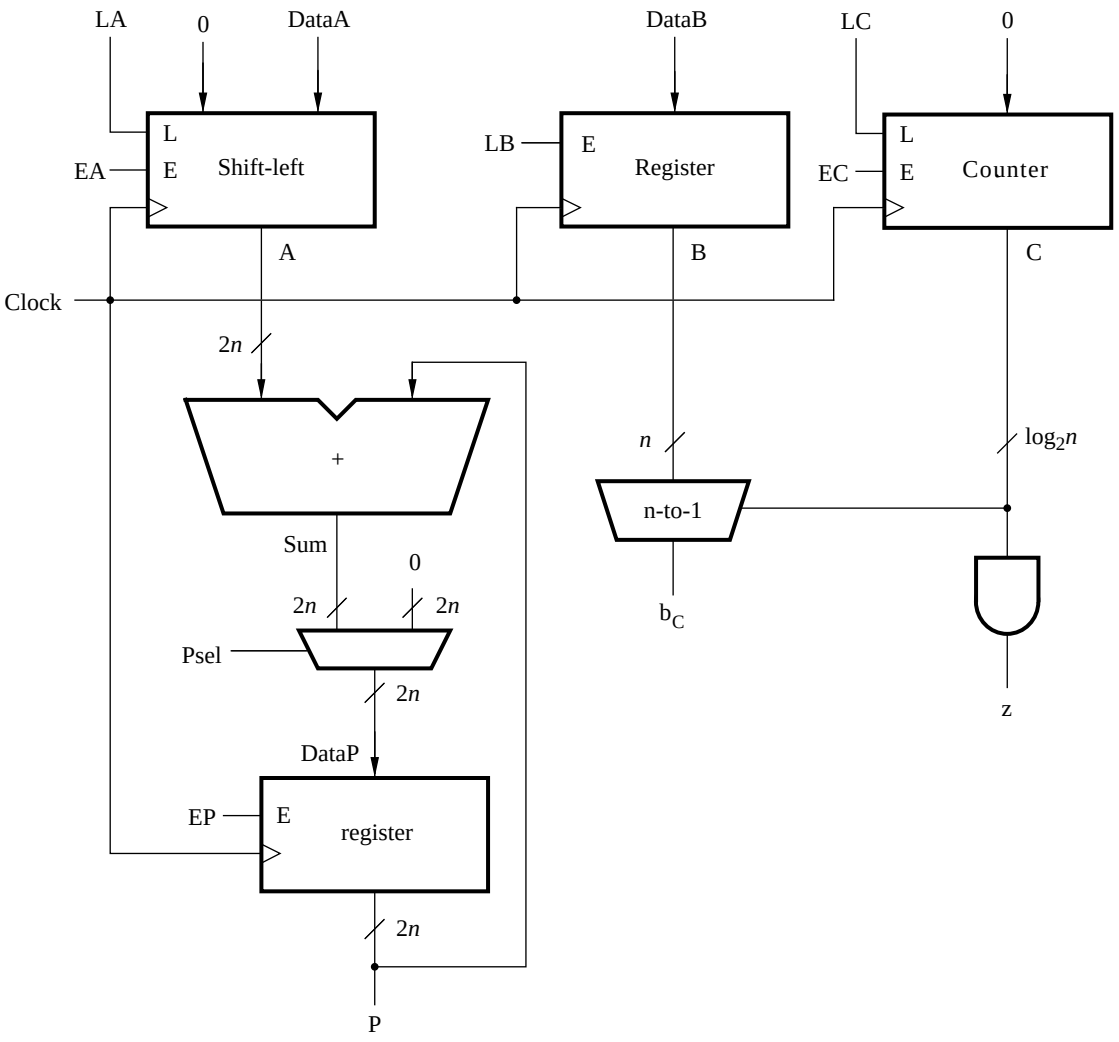
```
module  bitcount (Clock, Resetn, LA, s, Data, B, Done);
    input  Clock, Resetn, LA, s;
    input  [7:0] Data;
    output  [3:0] B;
    output  Done;
    wire  [7:0] A;
    wire  z;
    reg  [1:0] Y, y;
    reg  [3:0] B;
    reg  Done, EA, EB, LB;

// control circuit
    parameter  S1 = 2'b00, S2 = 2'b01, S3 = 2'b10, S4 = 2'b11;

    always @(s or y or z)
    begin: State_table
        case (y)
            S1:     if (s == 0)  Y = S1;
                    else  Y = S2;
            S2,S3:  if (!z && !A[0])  Y = S2;
                    else if (!z && A[0])  Y = S3;
                    else  Y = S4;
            S4:     if (s == 1)  Y = S4;
                    else  Y = S1;
        endcase
    end

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

    always @(y or A[0])
    begin: FSM_outputs
        EA = 0; LB = 0; EB = 0; Done = 0;  // defaults
        case (y)
            S1:     LB = 1;
            S2:     EA = 1;
            S3:     begin
                        EA = 1; EB = 1;
                    end
            S4:     Done = 1;
        endcase
    end
```

```
// datapath circuit

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

    shiftrne ShiftA (Data, LA, EA, 0, Clock, A);
    assign z = ~|A;

endmodule
```

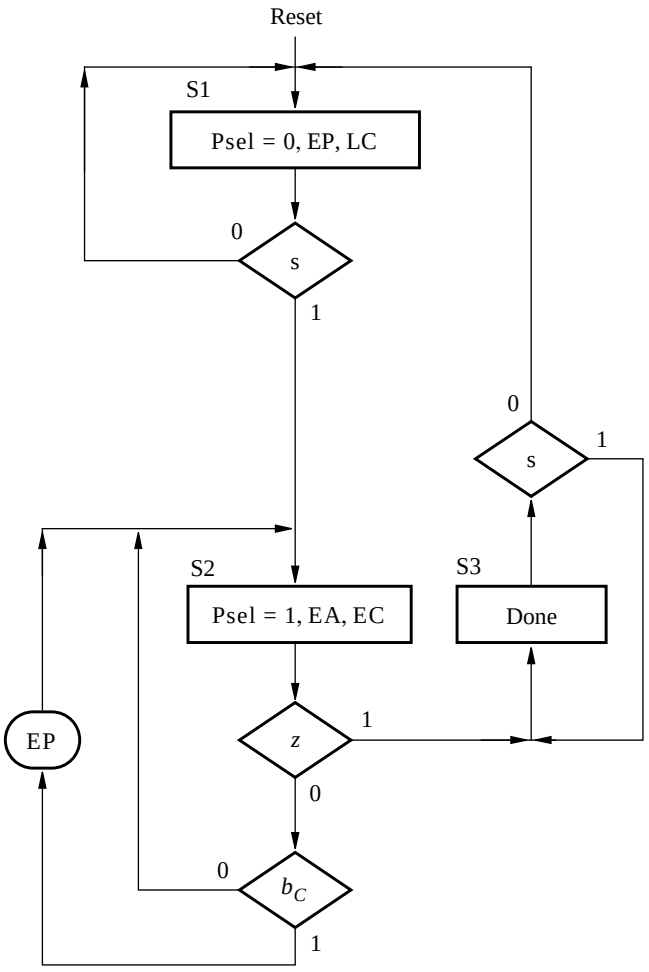
10.3. (*a*)



(*b*)



(*c*) The ASM chart for the control circuit is shown below. Note that we assume the EB signal is controlled by external logic.



(*d*)

```
module multiply (Clock, Resetn, LA, LB, s, DataA, DataB, P, Done);
    parameter n = 8;
    parameter m = 3;
    input Clock, Resetn, LA, LB, s;
    input [n−1:0] DataA, DataB;
    output [n+n−1:0] P;
    output Done;
    wire bc, z;
    reg [n+n−1:0] DataP;
    wire [n+n−1:0] A, Sum;
    reg [1:0] y, Y;
    wire [n−1:0] B;
    wire [m−1:0] C;
    reg Done, EA, EP, Psel, LC, EC;
    integer k;
```

```
// control circuit
   parameter  S1 = 2'b00, S2 = 2'b01, S3 = 2'b10;

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

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

   always @(y or bc)
   begin: FSM_outputs
      EA = 0; EP = 0; Done = 0; Psel = 0; EC = 0; LC = 0;  // defaults
      case (y)
         S1:   begin
                  EP = 1; EC = 1; LC = 1;
               end
         S2:   begin
                  EA = 1; Psel = 1; EC = 1; LC = 0;
                  if (bc)  EP = 1;
                  else  EP = 0;
               end
         S3:   Done = 1;
      endcase
   end

// datapath circuit
   regne  RegB  (DataB, Clock, Resetn, LB, B);
      defparam  RegB.n = 8;
   shiftlne  ShiftA  ({{n{1'b0}}, DataA}, LA, EA, Clock, A);
      defparam  ShiftA.n = 16;
   upcount  Counter  (LC, Clock, EC, C);
      defparam  Counter.n = m;
```

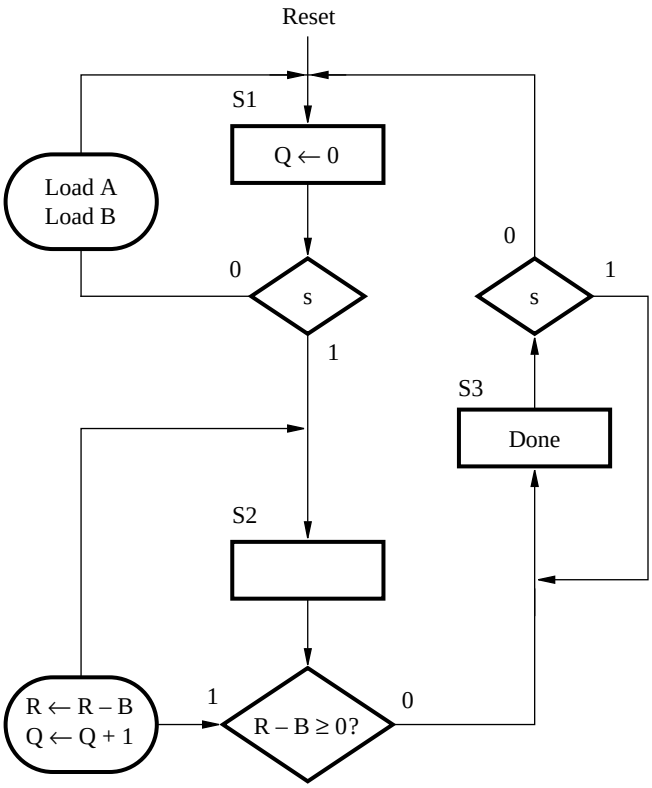
```
    assign  bc = B[C];
    assign  z = &C;
    assign  Sum = A + P;

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

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

endmodule
```

10.4.

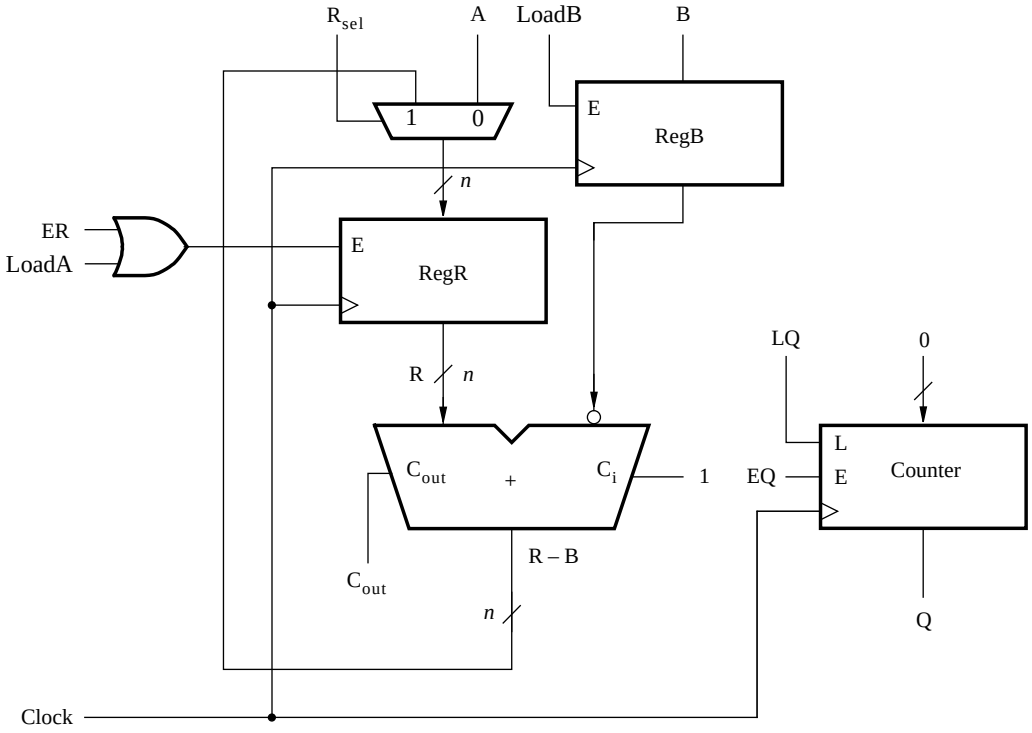
```
module divider (Clock, Resetn, s, LA, EB, DataA, DataB, R, Q, Done);
   parameter  n = 8, logn = 3;
   input  Clock, Resetn, s, LA, EB;
   input  [n−1:0] DataA, DataB;
   output  [n−1:0] R, Q;
   output  Done;
   wire  Cout, z;
   wire  [n−1:0] DataR;
   wire  [n−1:0] Sum;
   reg  [1:0] y, Y;
  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

// control circuit
   parameter  S1 = 2'b00, S2 = 2'b01, S3 = 2'b10, S4 = 2'b11;

   always @(s or y or z)
   begin: State_table
      case (y)
         S1:   if (s == 0)  Y = S1;
               else  Y = S2;
         S2:   Y = S3;
         S3:   if (z == 1)  Y = S4;
               else  Y = S2;
         S4:   if (s == 1)  Y = S4;
               else  Y = S1;
      endcase
   end
```

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

    always @(y or s or Cout or z)
    begin: FSM_outputs
        LR = 0; ER = 0; LC = 0; EC = 0; EA = 0;  // defaults
        EQ = 0; Rsel = 0; Done = 0;  // defaults
        case (y)
            S1:   begin
                      LC = 1; LR = 1; Rsel = 0;
                  end
            S2:   begin
                      ER = 1; EA = 1;
                  end
            S3:   begin
                      Rsel = 1; EQ = 1; EC = 1;
                      if (Cout)  LR = 1;
                      else  LR = 0;
                      if (z == 0) EC = 1;
                      else  EC = 0;
                  end
            S4:   Done = 1;
        endcase
    end

// datapath circuit
    regne  RegB  (DataB, Clock, Resetn, EB, B);
        defparam  RegB.n = n;

    shiftlne  ShiftR  (DataR, LR, ER, A[n−1], Clock, R);
        defparam  ShiftR.n = n;

    shiftlne  ShiftA  (DataA, LA, EA, 0, Clock, A);
        defparam  ShiftA.n = n;

    shiftlne  ShiftQ  (0, 0, EQ, Cout, Clock, Q);
        defparam  ShiftQ.n = n;

    downcount  Counter  (Clock, EC, LC, Count);
        defparam  Counter.n = logn;

    assign  z = (Count == 0);
    assign  {Cout, Sum} = R + {0, ∼B} + 1;

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

endmodule
```

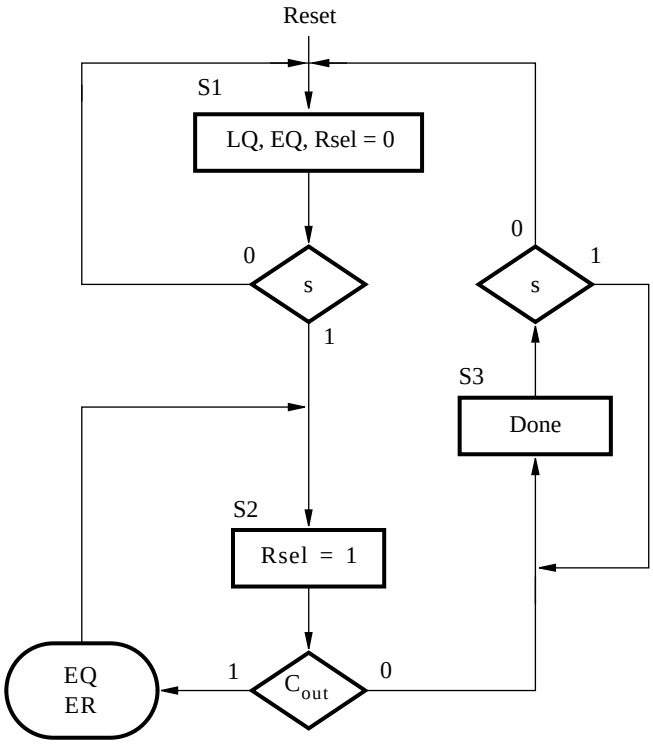
10.5. (*a*)



(*b*)


R_sel
A
LoadB
B
1
0
E
RegB
n
ER
LoadA
E
RegR
R
n
LQ
0
L
E
Counter
EQ
C_out
+
C_i
1
R – B
C_out
n
Q
Clock

(*c*)


Reset
S1
LQ, EQ, Rsel = 0
0
s
1
0
s
1
S3
Done
S2
Rsel = 1
EQ
ER
1
C_out
0

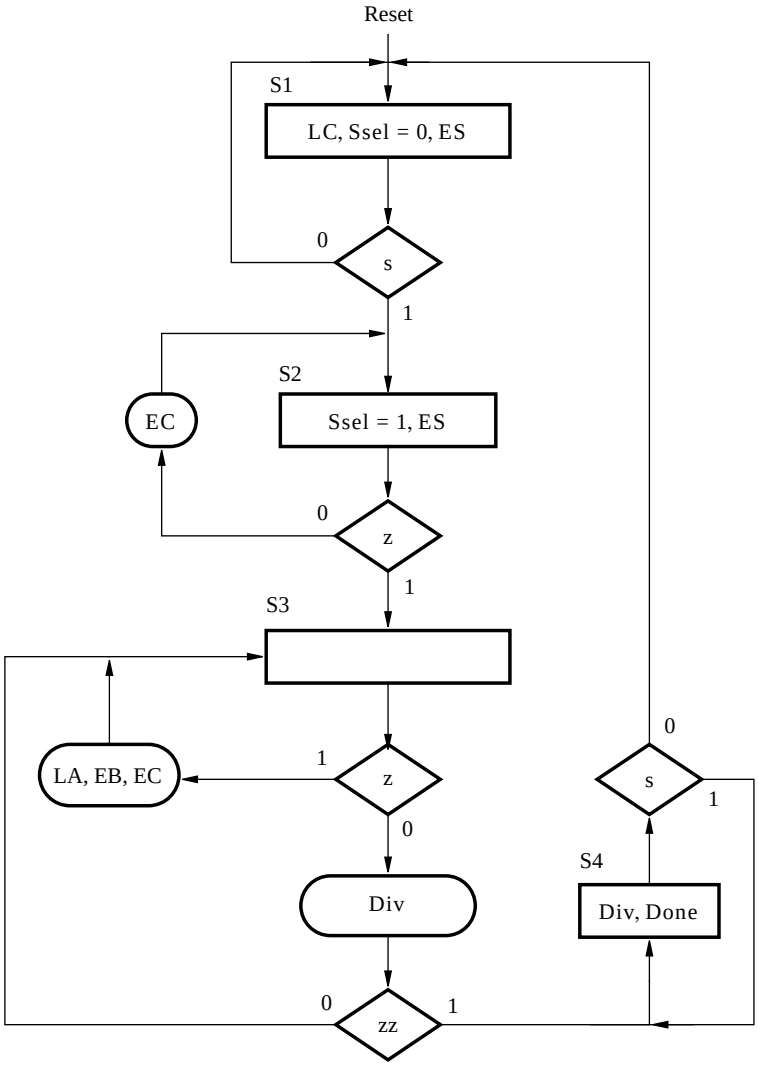
(*d*)

```
module divider (Clock, Resetn, s, EA, EB, DataA, DataB, R, Q, Done);
    parameter n = 8;
    input Clock, Resetn, s, EA, EB;
    input [n−1:0] DataA, DataB;
    output [n−1:0] R, Q;
    output Done;
    wire Cout, ERegR;
    wire [n−1:0] DataR;
    wire [n−1:0] Sum;
    reg [1:0] y, Y;
    wire [n−1:0] A, B, Q;
    reg Done, Rsel, ER, LQ, EQ;

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

    always @(s or y or Cout)
    begin: State_table
        case (y)
            S1:   if (s == 0)  Y = S1;
                  else  Y = S2;
            S2:   if (Cout == 0)  Y = S3;
                  else  Y = S2;
            S3:   if (s == 1)  Y = S3;
                  else  Y = S1;
            default: Y = S1;
        endcase
    end

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

    always @(y or s or Cout)
    begin: FSM_outputs
        ER = 0; LQ = 0; EQ = 0; Rsel = 0; Done = 0;  // defaults
        case (y)
            S1:   begin
                      LQ = 1; EQ = 1; Rsel = 0;
                  end
            S2:   begin
                      Rsel = 1;
                      if (Cout)
                      begin
                          EQ = 1; ER = 1;
                      end
```

```
                    else
                    begin
                        EQ = 0; ER = 0;
                    end
                end
            S3:  Done = 1;
        endcase
    end
// datapath circuit
    regne  RegB  (DataB, Clock, Resetn, EB, B);
        defparam  RegB.n = n;

    regne  RegR  (DataR, Clock, Resetn, ERegR, R);
        defparam  RegR.n = n;

    upcount  Counter  (Clock, EQ, LQ, Q);
        defparam  Counter.n = n;

    assign  ERegR = ER | EA;
    assign  {Cout, Sum} = {1'b0, R} + {1'b0, ~B} + 1;

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

endmodule
```

(e) This implementation of a divider is less efficient in the worst case when compared to the other implementations shown. The efficient algorithms presented are able to perform division in n cycles for n-bit inputs. However, the method of repeated subtraction takes 2^n cycles for the worst case, which is when dividing by 1. On the other hand, if the two numbers A and B are close in size, then repeated subtraction is an efficient approach.

10.6. State S3 is responsible for loading the operands into the divider, while state S4 starts the division operation. These states can be combined into a single state. We can use the z flag to indicate the first time that we've entered the new combined state. When $z = 1$ a mealy output is produced which loads the operands and decrements the counter. Thus, the z flag changes to a 0. The combined state now produces a mealy output which starts the division, on the condition that $z = 0$. This control circuit ASM chart is shown below.



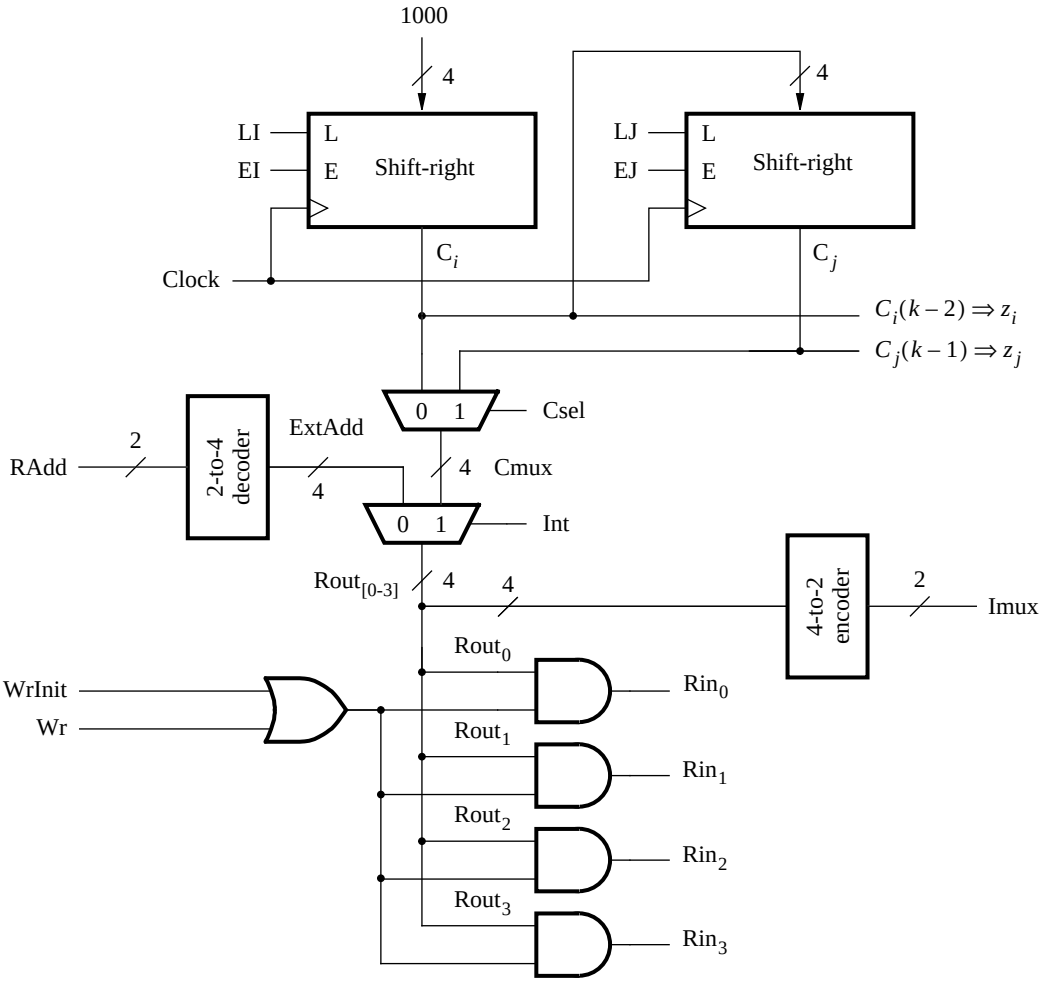
10.7.

```
module  meancntl (Clock, Resetn, s, z, zz, EC, LC, Ssel, ES, LA, EB, Div, Done);
   input  Clock, Resetn, s, z, zz;
   output  EC, LC, Ssel, ES, LA, EB, Div, Done;
   reg  EC, LC, Ssel, ES, LA, EB, Div, Done;
   reg  [1:0] y, Y;
   parameter  S1 = 2'b00, S2 = 2'b01, S3 = 2'b10, S4 = 2'b11;
   always @(s or y or z or zz)
   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 (zz == 1)  Y = S3;
                else  Y = S4;
            S4: if (s == 1)  Y = S4;
                else  Y = S1;
        endcase
    end

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

    always @(s or y or z or zz)
    begin: FSM_outputs
        LC = 0; EC = 0; ES = 0; LA = 0; EB = 0; Div = 0; Done = 0; Ssel = 0;  // defaults
        case (y)
            S1: begin
                    LC = 1; ES = 1;
                end
            S2: begin
                    Ssel = 1; ES = 1;
                    if (z == 0)  EC = 1;
                    else  EC = 0;
                end
            S3: if (z == 0)
                begin
                    Div = 1; LA = 0; EB = 0; EC = 0;
                end
                else
                begin
                    LA = 1; EB = 1; EC = 1;
                end
            S4: begin
                    Div = 1; Done = 1;
                end
        endcase
    end

endmodule
```

10.8. The states $S2$ and $S3$ can be merged into a single state by performing the assignment $C_j = C_i + 1$. The circuit would require an adder to increment C_i by 1 and the outputs of this adder would be loaded in parallel into the counter C_j. If instead of using counters to implement C_i and C_j we used shift registers, then the effect of producing $C_i + 1$ could be efficiently implemented by wiring C_i to the parallel-load data inputs on C_j such that the bits are shifted by one position.

10.9. (a) The part of the datapath circuit that needs to be modified is shown below. The rest of the datapath is the same as the circuit shown in Figure 10.37.



(*b*)

```
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 Done;
    wire [0:3] Ci, Cj, Cmux;
    reg [1:0] Imux;
    wire [n−1:0] R0, R1, R2, R3, A, B;
    wire [n−1:0] RData, ABMux;
    wire BltA, zi, zj;
    reg Int, Csel, Wr, Ain, Bin, Aout, Bout, LI, LJ, EI, EJ, Done;
    reg [3:0] y, Y;
    reg Rin0, Rin1, Rin2, Rin3;
    reg [n−1:0] ABData;
    wire [0:3] Rout, ExtAdd;
    wire [0:3] Addr0; //Parallel load for Ci
```

```
// 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 or BltA or zj or zi or 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

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

   always @(y or zj or zi)
   begin: FSM_outputs
      Int = 1; Done = 0; LI = 0; LJ = 0; EI = 0; EJ = 0;  // defaults
      Csel = 0; Wr = 0; Ain = 0; Bin = 0; Aout = 0; Bout = 0;  // defaults
      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 ouputs asserted in this state
         S6:  begin
                 Csel = 1; Wr = 1; Aout = 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
        S9: Done = 1;
    endcase
end

// 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;
assign  Addr0 = 4'b1000;
shiftrne  Outerloop  (Addr0, LI, EI, 0, Clock, Ci);
shiftrne  Innerloop  (Ci, LJ, EJ, 0, Clock, Cj);
dec2to4  Decoder  (RAdd, ExtAdd);
assign  Rout = Int ? Cmux : ExtAdd;
assign  Cmux = (Csel == 0) ? Ci : Cj;

always @(WrInit or Wr or Rout or R3 or R2 or R1 or R0)
begin
    case (Rout)
        4'b1000: Imux = 0;
        4'b0100: Imux = 1;
        4'b0010: Imux = 2;
        4'b0001: Imux = 3;
        default: Imux = 0;
    endcase
```

```
        if (WrInit || Wr)
            case (Rout)
                4'b1000: {Rin3, Rin2, Rin1, Rin0} = 4'b0001;
                4'b0100: {Rin3, Rin2, Rin1, Rin0} = 4'b0010;
                4'b0010: {Rin3, Rin2, Rin1, Rin0} = 4'b0100;
                4'b0001: {Rin3, Rin2, Rin1, Rin0} = 4'b1000;
                default: {Rin3, Rin2, Rin1, Rin0} = 4'bx;
            endcase
        else {Rin3, Rin2, Rin1, Rin0} = 4'b0000;

        case (Imux)
            0: ABData = R0;
            1: ABData = R1;
            2: ABData = R2;
            3: ABData = R3;
        endcase
    end

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

(c) The major drawback of using shift-registers instead of counters is that the number of flip-flops is increased. Each counter uses $log_2 n$ flip-flops while each shift register contains n flip-flops. However, the shift-register requires no combinational logic to perform tests such as whether the count value $k-2$ has been reached — in the shift register we directly access bit $k-2$ of the register to perform this test. It should also be possible to clock the datapath at a higher maximum clock frequency when using shift-registers, because they are simpler than counters.

10.11. The Verilog code below shows the changes needed in the datapath so that an SRAM block can be used instead of registers. The SRAM block is clocked on the negative edge of the Clock signal, hence changes in the outputs produced by the other datapath elements must be stable before the negative edge; the clock period must be long enough to accommodate this constraint.

```
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 Done;
```

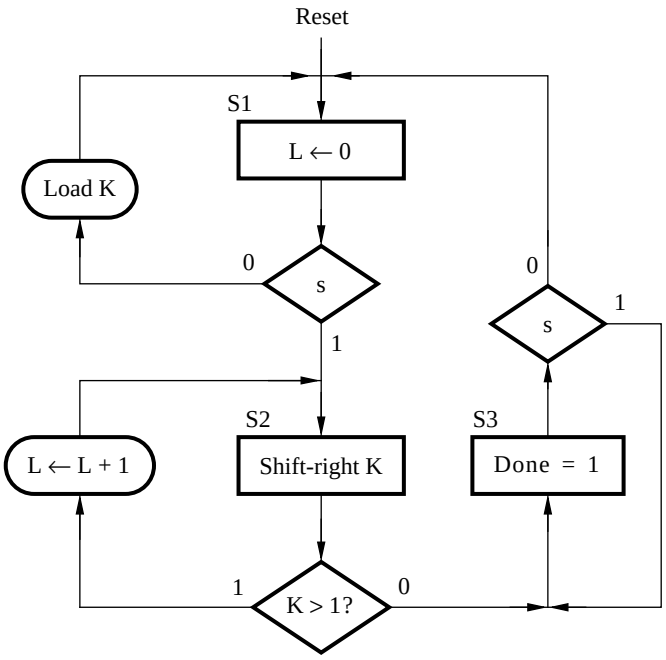
```
    wire [1:0] Ci, Cj, CMux, IMux, MemAdd;
    wire [n−1:0] A, B, RData, ABMux;
    wire BltA, zi, zj, WE, NClock;
    reg Int, Csel, Wr, Ain, Bin, Aout, Bout;
    reg LI, LJ, EI, EJ, Done;
    reg [3:0] y, Y;
    reg [n−1:0] ABData;

// 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 or BltA or zj or zi)
    begin: State_table
        . . . code not shown: see Figure 10.40
    end

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

    always @(y or zj or zi)
    begin: FSM_outputs
        . . . code not shown: see Figure 10.40
    end

    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 (0, 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;

    assign MemAdd = IMux;
    assign WE = WrInit | Wr;
    assign NClock = ∼Clock;
```

```
lpm_ram_dq mem_block (.address(MemAdd), .we(WE), .q(ABData),
    .inclock(NClock), .data(RData));
    defparam mem_block.lpm_width = 4;
    defparam mem_block.lpm_widthad = 2;
    defparam mem_block.lpm_address_control = "registered";
    defparam mem_block.lpm_indata = "registered";
    defparam mem_block.lpm_outdata = "unregistered";

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

endmodule
```

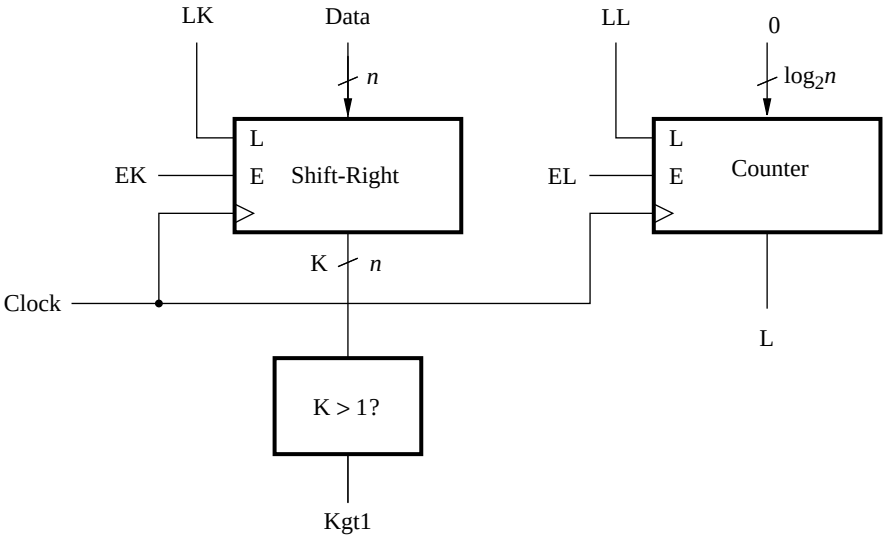
10.12. Pseudo-code that represents the log_2 operation is

```
-- assume that K ≥ 1
L = 0 ;
while (K > 1) do
    K = K ÷ 2 ;
    L = L + 1 ;
end while ;
-- L now has the largest value such that 2^L < K
```

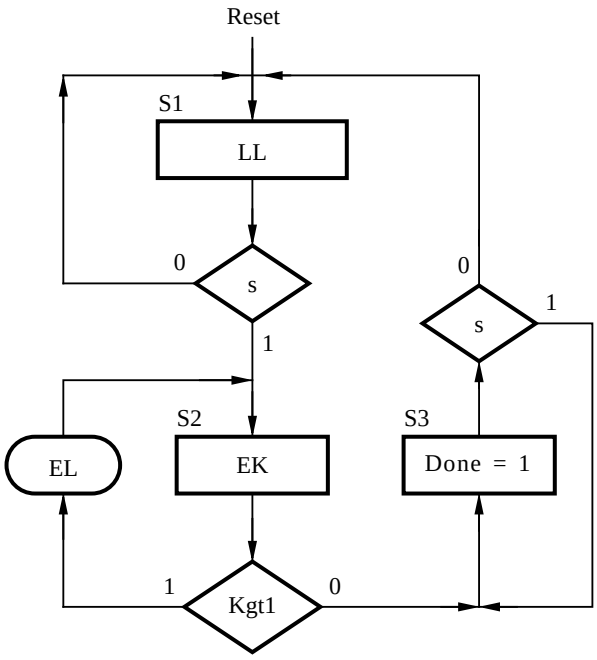
An ASM chart that corresponds to the pseudo-code is



From the ASM chart, a shift-register is needed to divide K by 2, and a counter is needed for L. An appropriate datapath circuit is



An ASM chart for the control circuit is



Complete Verilog code for this circuit is shown below.

```
module log2k (Clock, Resetn, LData, s, Data, L, Done);
    input Clock, Resetn, LData, s;
    input [7:0] Data;
    output [3:0] L;
    output Done;

    wire [7:0] K;
    wire Kgt1;
    reg [1:0] y, Y;
    reg Done, EL, LL, EK;

// control circuit

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

    always @(s or y or Kgt1)
    begin: FSM_transitions
        case (y)
            S1:   if (s == 0) Y = S1;
                  else Y = S2;
            S2:   if (Kgt1 == 1) Y = S2;
                  else Y = S3;
            S3:   if (s == 1) Y = S3;
                  else Y = S1;
            default: Y = 2'bxx;
        endcase
    end

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

    always @(y or s or LData or Kgt1)
    begin: FSM_outputs
        EL = 0; LL = 0; EK = 0; Done = 0; // defaults
        case (y)
            S1:   begin
                      EL = 1; LL = 1;
                      if (LData == 1) EK = 1;
                      else EK = 0;
                  end
            S2:   begin
                      EK = 1;
                      if (Kgt1) EL = 1;
                      else EL = 0;
                  end
            S3:   Done = 1;
        endcase
    end
```

```
        //datapath circuit

            shiftrne ShiftK (Data, LData, EK, 1'b0, Clock, K);
               defparam ShiftK.n = 8;

            // upcount is in Figure 7.57
            upcount CntL (4'b0, Resetn, Clock, EL, LL, L);

            assign Kgt1 = (K > 1) ? 1 : 0;

        endmodule
```

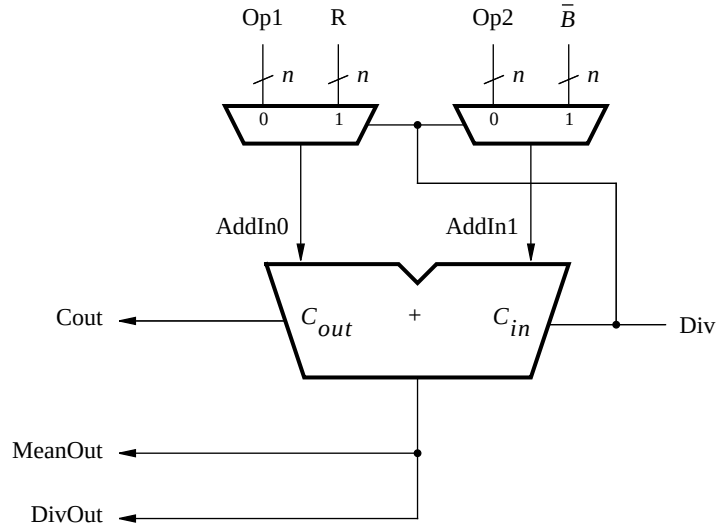
10.13.

```
module mean (Clock, Resetn, Data, RAdd, s, ER, M, Done);
   parameter n = 8;
   input Clock, Resetn;
   input [n−1:0] Data;
   input [1:0] RAdd;
   input s, ER;
   output [n−1:0] M;
   output Done;

   reg LC, EC, Ssel, ES, LA, EB, LB, Div, Done;
   wire z, zz;
   reg [0:3] Dec_RAdd;
   wire [0:3] Rin;
   wire [1:0] C;
   wire [n−1:0] R0, R1, R2, R3, SR, Sin, Sum, Remainder, K;
   reg [n−1:0] Ri;
   reg [2:0] y, Y;
   parameter S1 = 3'b000, S2 = 3'b001, S3 = 3'b010, S4 = 3'b011, S5 = 3'b100;

   always @(s or y or z or zz)
   begin: State_table
      case (y)
         S1:   if (s == 0) Y = S1;
               else Y = S2;
         S2:   if (z == 0) Y = S2;
               else Y = S3;
         S3:   Y = S4;
         S4:   if (zz == 0) Y = S4;
               else Y = S5;
         S5:   if (s == 1) Y = S5;
               else Y = S1;
         default: Y = 3'bxxx;
      endcase
   end
```

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

always @(y or s or z or zz)
begin: FSM_outputs
    LC = 0; EC = 0; ES = 0; LA = 0; EB = 0; // defaults
    Div = 0; Done = 0; Ssel = 0; // defaults
    case (y)
        S1:  begin
                 LC = 1; ES = 1;
             end
        S2:  begin
                 Ssel = 1; ES = 1;
                 if (z == 0) EC = 1;
                 else EC = 0;
             end
        S3:  begin
                 LA = 1; EB = 1;
             end
        S4:  Div = 1; // not really used in this circuit
        S5:  begin
                 Div = 1; Done = 1;
             end
    endcase
end

// Datapath

always @(RAdd)
begin
    case (RAdd)
        0: Dec_RAdd = 4'b1000;
        1: Dec_RAdd = 4'b0100;
        2: Dec_RAdd = 4'b0010;
        3: Dec_RAdd = 4'b0001;
    endcase
end

assign Rin = (ER == 1) ? Dec_RAdd : 4'b0000;

regne Reg0 (Data, Clock, Resetn, Rin[0], R0);
    defparam Reg0.n = n;
regne Reg1 (Data, Clock, Resetn, Rin[1], R1);
    defparam Reg1.n = n;
```

```
    regne Reg2 (Data, Clock, Resetn, Rin[2], R2);
       defparam Reg2.n = n;
    regne Reg3 (Data, Clock, Resetn, Rin[3], R3);
       defparam Reg3.n = n;

    downcount Counter (Clock, EC, LC, C);
       defparam Counter.n = 2;
    assign z = (C == 0) ? 1 : 0;
    assign Sin = (Ssel == 1) ? Sum : 0;

    regne RegS (Sin, Clock, Resetn, ES, SR);
       defparam RegS.n = n;

    always @(C)
    begin
       case (C)
          0: Ri = R0;
          1: Ri = R1;
          2: Ri = R2;
          3: Ri = R3;
       endcase
    end

    assign Sum = SR + Ri;
    //divide by 4
    assign M = 2'b00, SR[n-1:2];
    assign zz = 1;

endmodule
```

10.14. From Figures 10.26 and 10.27, we can see that the divider subcircuit does not use its adder while in state S1. Since the control circuit for the divider stays in S1 while $s = 0$, it is possible to lend the adder to another circuit while we are in S1 and $s = 0$. The Figure below shows the changes needed in the divider circuit: a multiplexer is added to each data input on the adder. The multiplexer select line is driven by the divider's s input — this signal is called *Div* in the figure, because *Div* is the signal in the Mean circuit that drives the *s* input on the divider subcircuit. When *Div* $= 1$ the adder is provided with the normal data used in the division operation. But when *Div* $= 0$ the adder is provided with the external data inputs named *Op1* and *Op2*, which come from the Mean circuit. Note that the C_{in} input on the adder is controlled by *Div*. This feature is needed because the divider uses its adder to perform subtraction.



10.15. Verilog code for the modified divider circuit is shown below.

```
module divider (Clock, Resetn, s, LA, EB, DataA, DataB, R, Q, Done, Op1, Op2, Result);
    parameter n = 8, logn = 3;
    input Clock, Resetn, s, LA, EB;
    input [n−1:0] DataA, DataB;
    output [n−1:0] R, Q;
    output Done;
    input [n−1:0] Op1, Op2; // new ports
    output [n−1:0] Result; // new port

    wire Cout, z;
    wire [n−1:0] DataR, AddIn1, AddIn2;
    wire [n−1:0] Sum;
    reg [1:0] y, Y;
    reg [n−1:0] A, B;
    reg [logn−1:0] Count;
    reg Done, EA, Rsel, LR, ER, ER0, LC, EC, R0;
    integer k;

// control circuit

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

    always @(s or y or z)
    begin: State_table
        . . . code not shown: see Figure 10.28
    end
```

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

    always @(y or s or Cout or z)
    begin: FSM_outputs
        . . . code not shown: see Figure 10.28
    end

//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 (0, 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);

    // new code for the divider
    assign AddIn1 = (s == 1) ? {R, R0} : Op1;
    assign AddIn2 = (s == 1) ? ∼B : Op2;
    assign {Cout, Sum} = AddIn1 + AddIn2 + s;

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

endmodule
```

Code for the modified Mean circuit is shown below.

```
module mean (Clock, Resetn, Data, RAdd, s, ER, M, Done);
    parameter n = 8;
    input Clock, Resetn;
    input [n−1:0] Data;
    input [1:0] RAdd;
    input s, ER;
    output [n−1:0] M;
    output Done;

    reg LC, EC, Ssel, ES, LA, EB, LB, zz, Div, Done;
    wire z;
    reg [0:3] Dec_RAdd;
    wire [0:3] Rin;
    wire [1:0] C;
    wire [n−1:0] R0, R1, R2, R3, SR, Sin, Sum, Remainder, K;
    reg [n−1:0] Ri;
    reg [2:0] y, Y;
    parameter S1 = 3'b000, S2 = 3'b001, S3 = 3'b010, S4 = 3'b011, S5 = 3'b100;

    always @(s or y or z or zz)
    begin: State_table
        case (y)
            S1:   if (s == 0) Y = S1;
                  else Y = S2;
            S2:   if (z == 0) Y = S2;
                  else Y = S3;
            S3:   Y = S4;
            S4:   if (zz == 0) Y = S4;
                  else Y = S5;
            S5:   if (s == 1) Y = S5;
                  else Y = S1;
            default: Y = 3'bxxx;
        endcase
    end

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

    always @(y or s or z or zz)
    begin: FSM_outputs
        LC = 0; EC = 0; ES = 0; LA = 0; EB = 0; // defaults
        Div = 0; Done = 0; Ssel = 0; // defaults
        case (y)
            S1:   begin
                      LC = 1; EC = 1; ES = 1;
                  end
```

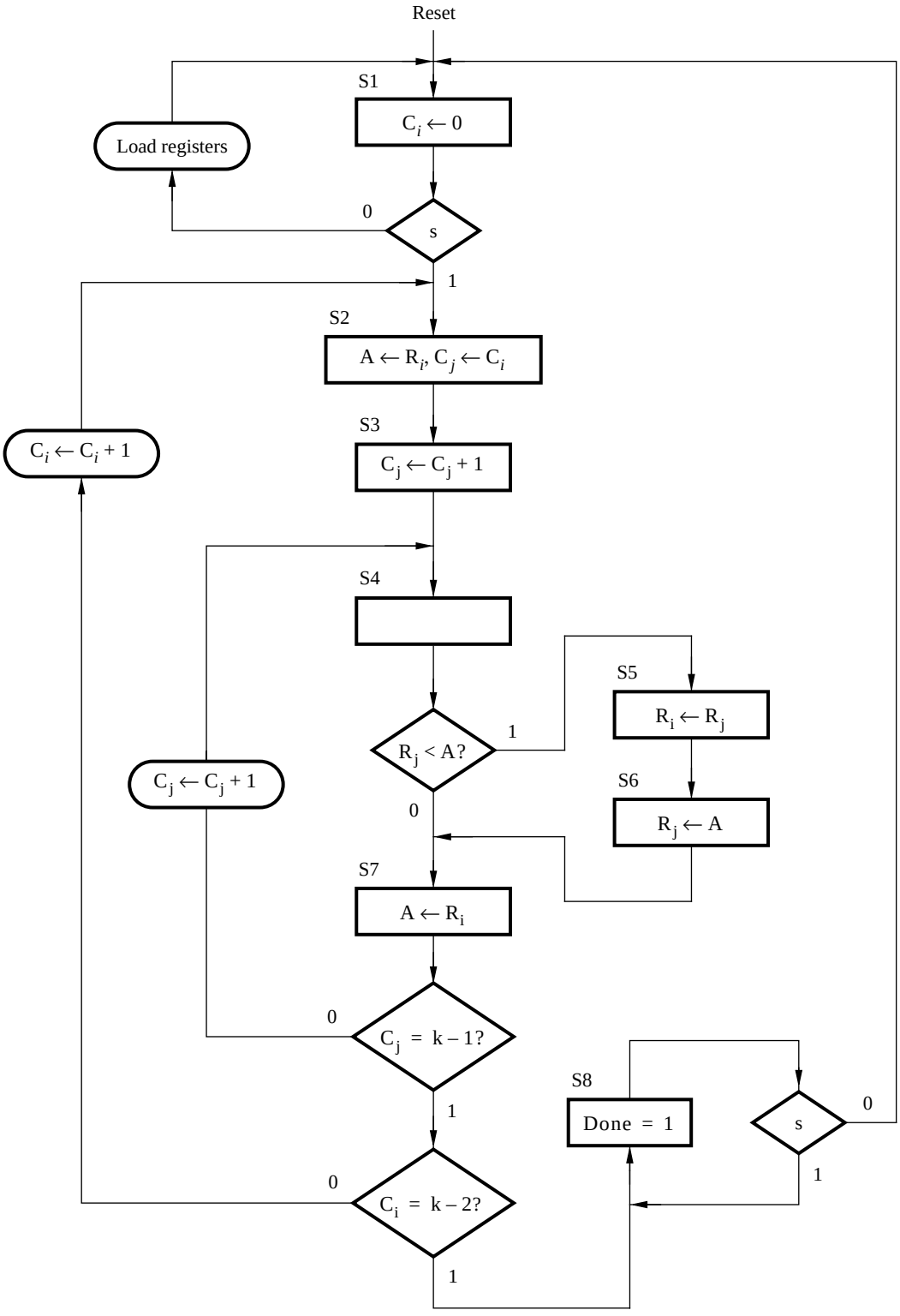
```
            S2:  begin
                     Ssel = 1; ES = 1;
                     if (z == 0) EC = 1;
                     else EC = 0;
                 end
            S3:  begin
                     LA = 1; EB = 1;
                 end
            S4:  Div = 1;
            S5:  begin
                     Div = 1; Done = 1;
                 end
        endcase
    end

// Datapath

    always @(RAdd)
    begin
        case (RAdd)
            0: Dec_RAdd = 4'b1000;
            1: Dec_RAdd = 4'b0100;
            2: Dec_RAdd = 4'b0010;
            3: Dec_RAdd = 4'b0001;
        endcase
    end

    assign Rin = (ER == 1) ? Dec_RAdd : 4'b0000;

    regne Reg0 (Data, Clock, Resetn, Rin[0], R0);
        defparam Reg0.n = n;
    regne Reg1 (Data, Clock, Resetn, Rin[1], R1);
        defparam Reg1.n = n;
    regne Reg2 (Data, Clock, Resetn, Rin[2], R2);
        defparam Reg2.n = n;
    regne Reg3 (Data, Clock, Resetn, Rin[3], R3);
        defparam Reg3.n = n;

    downcount Counter (Clock, EC, LC, C);
        defparam Counter.n = 2;
    assign z = (C == 0) ? 1 : 0;
    assign Sin = (Ssel == 1) ? Sum : 0;

    regne RegS (Sin, Clock, Resetn, ES, SR);
        defparam RegS.n = n;
```

```
    always @(C)
    begin
        case (C)
            0: Ri = R0;
            1: Ri = R1;
            2: Ri = R2;
            3: Ri = R3;
        endcase
    end

    divider DivideBy4 (.Clock(Clock), .Resetn(Resetn), .s(Div), .LA(LA), .EB(EB),
        .DataA(SR), .DataB(4), .R(Remainder), .Q(M), .Done(zz), .Op1(SR),
        .Op2(Ri), .Result(Sum));

endmodule
```

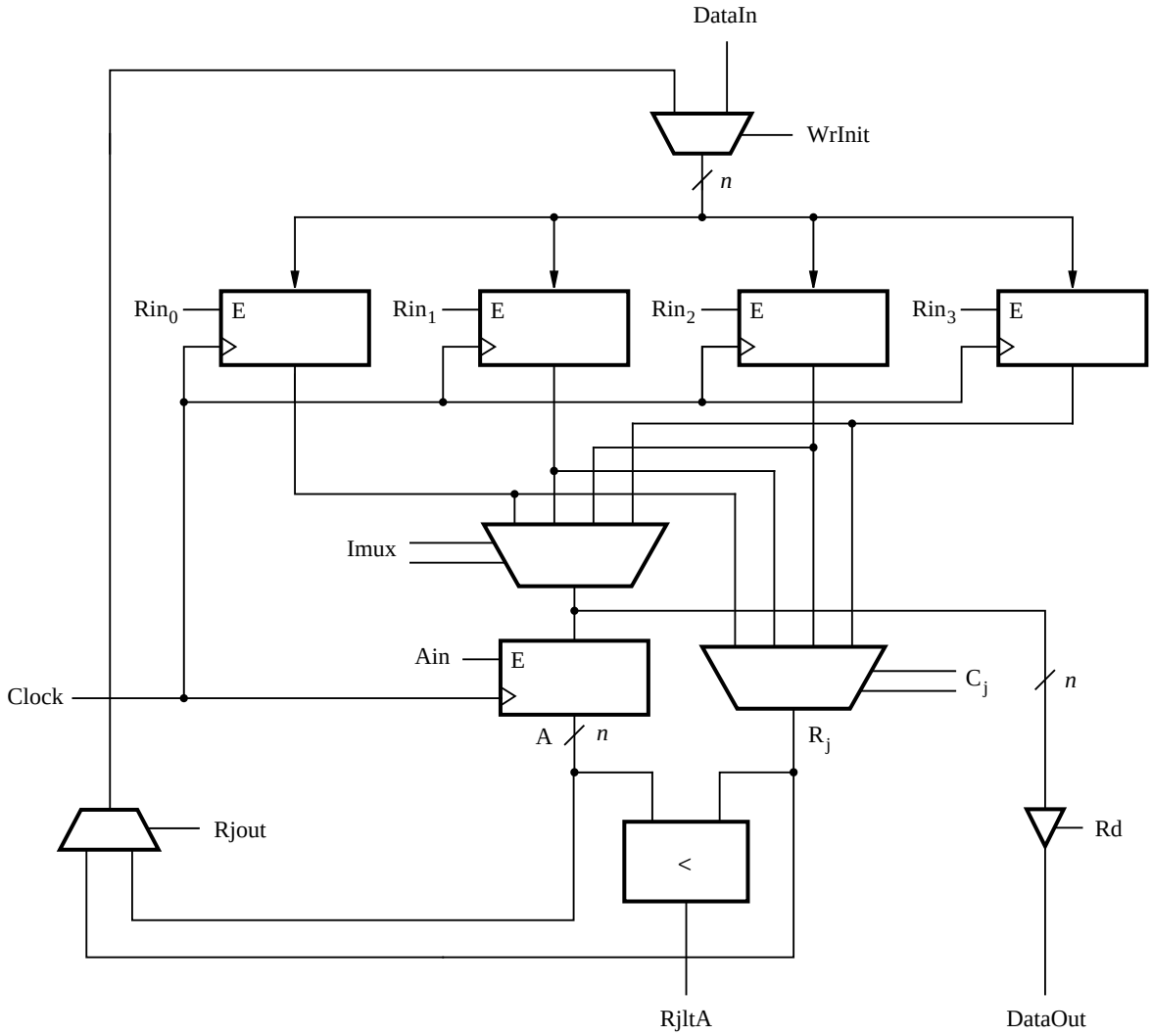
10.16. The modified pseudo-code is

for $i = 0$ to $k - 2$ do
    $A = R_i$;
    for $j = i + 1$ to $k - 1$ do
        if $R_j < A$ then
            $R_i = R_j$;
            $R_j = A$;
        end if ;
        $A = R_i$;
    end for ;
end for ;

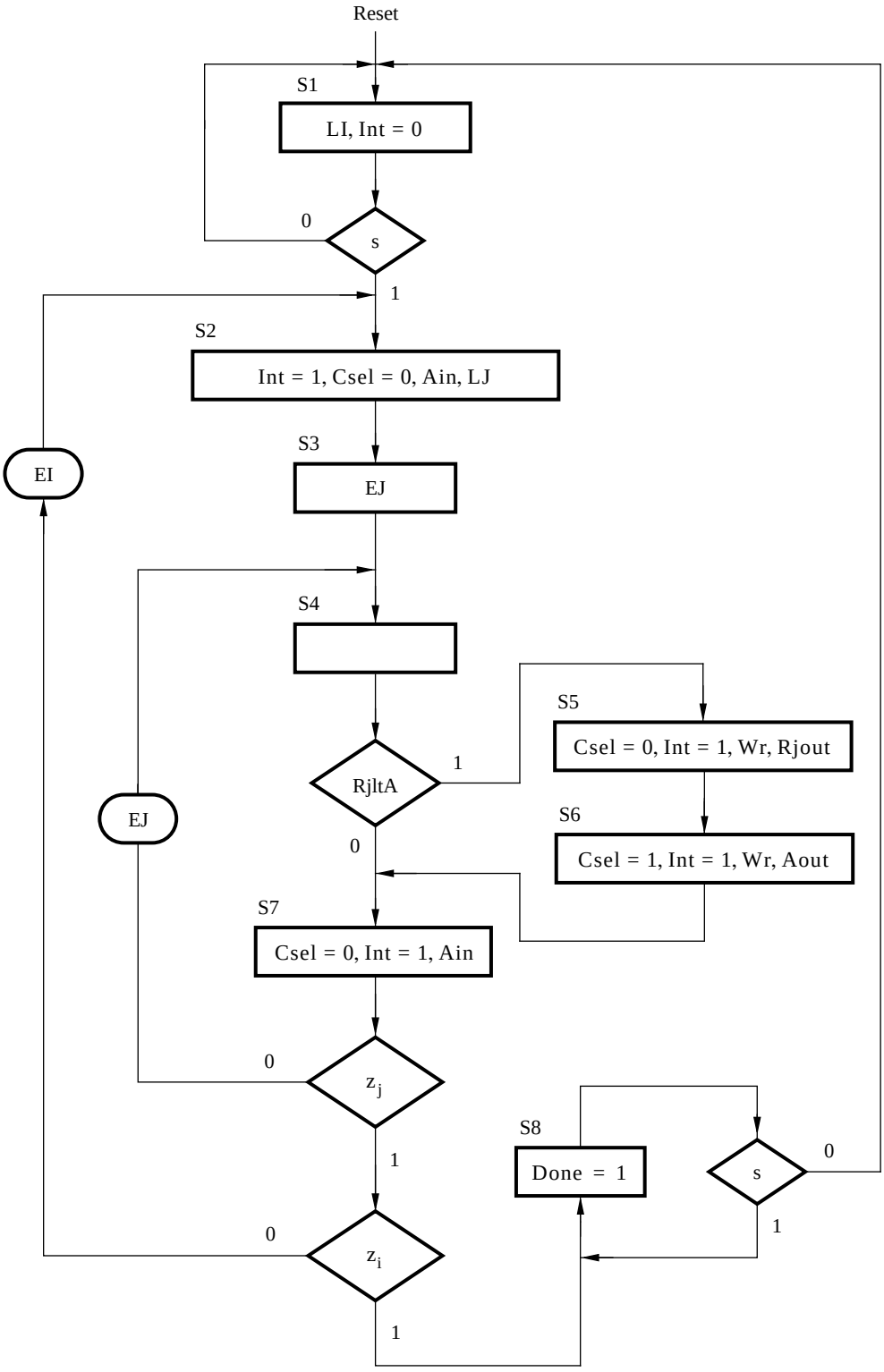
An ASM chart that corresponds to the pseudo-code is


Reset
S1
$C_i \leftarrow 0$
Load registers
0
s
1
S2
$A \leftarrow R_i, C_j \leftarrow C_i$
S3
$C_j \leftarrow C_j + 1$
$C_i \leftarrow C_i + 1$
S4
S5
$R_i \leftarrow R_j$
$R_j < A?$
1
$C_j \leftarrow C_j + 1$
S6
$R_j \leftarrow A$
0
S7
$A \leftarrow R_i$
0
$C_j = k - 1?$
S8
Done = 1
s
0
1
0
$C_i = k - 2?$
1

From the ASM chart, we can see that the datapath circuit needs a multiplexer to allow the operation $R_i \leftarrow R_j$. An appropriate datapath is shown below.



An ASM chart for the control circuit is


Reset
S1
LI, Int = 0
0
s
1
S2
Int = 1, Csel = 0, Ain, LJ
S3
EJ
EI
S4
S5
Csel = 0, Int = 1, Wr, Rjout
RjltA
1
EJ
S6
Csel = 1, Int = 1, Wr, Aout
0
S7
Csel = 0, Int = 1, Ain
0
z_j
1
S8
Done = 1
s
0
1
0
z_i
1

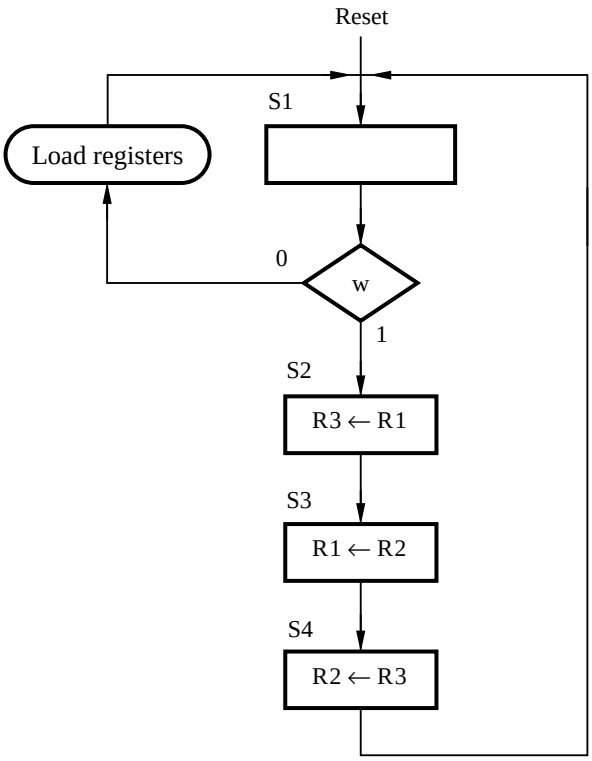
10.17.

```
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 Done;

    wire [1:0] Ci, Cj, CMux, IMux;
    wire [n−1:0] R0, R1, R2, R3, A;
    wire [n−1:0] RData, ARjMux;
    wire RjltA;
    wire zi, zj;
    reg Int, Csel, Wr, Ain;
    reg LI, LJ, EI, EJ, Done, RjOut;
    reg [n−1:0] Rj;
    reg [2:0] y, Y;
    reg Rin0, Rin1, Rin2, Rin3;
    reg [n−1:0] AData;

// control circuit

    parameter S1 = 3'b000, S2 = 3'b001, S3 = 3'b010, S4 = 3'b011;
    parameter S5 = 3'b100, S6 = 3'b101, S7 = 3'b110, S8 = 3'b111;

    always @(s or RjltA or zj or zi)
    begin: State_table
        case (y)
            S1:   if (s == 0) Y = S1;
                  else Y = S2;
            S2:   Y = S3;
            S3:   Y = S4;
            S4:   if (RjltA == 1) Y = S5;
                  else Y = S7;
            S5:   Y = S6;
            S6:   Y = S7;
            S7:   if (!zj) Y = S4;
                  else if (!zi) Y = S2;
                  else Y = S8;
            S8:   if (s) Y = S8;
                  else Y = S1;
            default: Y = 4'bx;
        endcase
    end
```

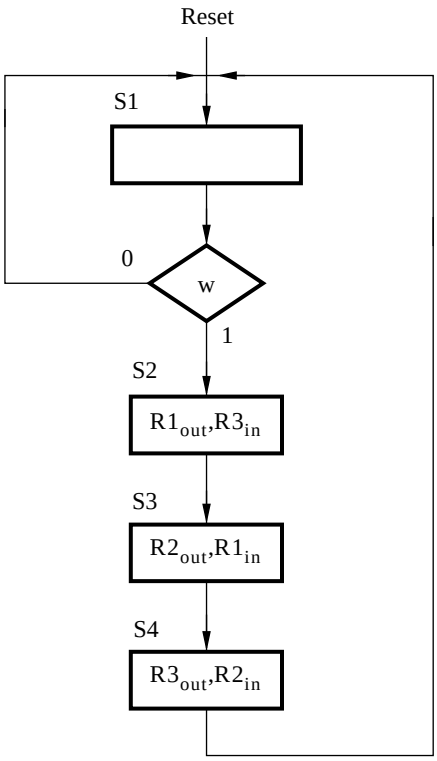
```
    always @(posedge Clock or negedge Resetn)
    begin: State_flipflops
        if (Resetn == 0)
            y <= S1;
        else
            y <= Y;
    end

    assign Int = (y != S1);
    assign Done = (y == S8);

    always @(y or zj or zi)
    begin: FSM_outputs
        LI = 0; LJ = 0; EI = 0; EJ = 0; Csel = 0; // defaults
        Wr = 0; Ain = 0; RjOut = 0; // defaults
        case (y)
            S1:  begin
                    LI = 1;
                 end
            S2:  begin
                    Ain = 1; LJ = 1;
                 end
            S3:  EJ = 1;
            S4:  ;
            S5:  begin
                    RjOut = 1; Wr = 1;
                 end
            S6:  begin
                    Wr = 1; Csel = 1;
                 end
            S7:  begin
                    Ain = 1;
                    if (!zj) EJ = 1;
                    else
                    begin
                        EJ = 0;
                        if (!zi) EI = 1;
                        else EI = 0;
                    end
                 end
            S8:  ;
        endcase
    end

//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 (AData, Clock, Resetn, Ain, A);
        defparam RegA.n = n;

    assign RjltA = (Rj < A) ? 1 : 0;
    assign ARjMux = (RjOut == 0) ? A : Rj;
    assign RData = (WrInit == 0) ? ARjMux : DataIn;

    upcount OuterLoop (0, 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;

    always @(WrInit or Wr or IMux or Cj)
    begin
        case (IMux)
            0: AData = R0;
            1: AData = R1;
            2: AData = R2;
            3: AData = R3;
        endcase

        case (Cj)
            0: Rj = R0;
            1: Rj = R1;
            2: Rj = R2;
            3: Rj = 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 : AData;

endmodule
```

10.18.



10.19. (*a*) An ASM chart for the control circuit is

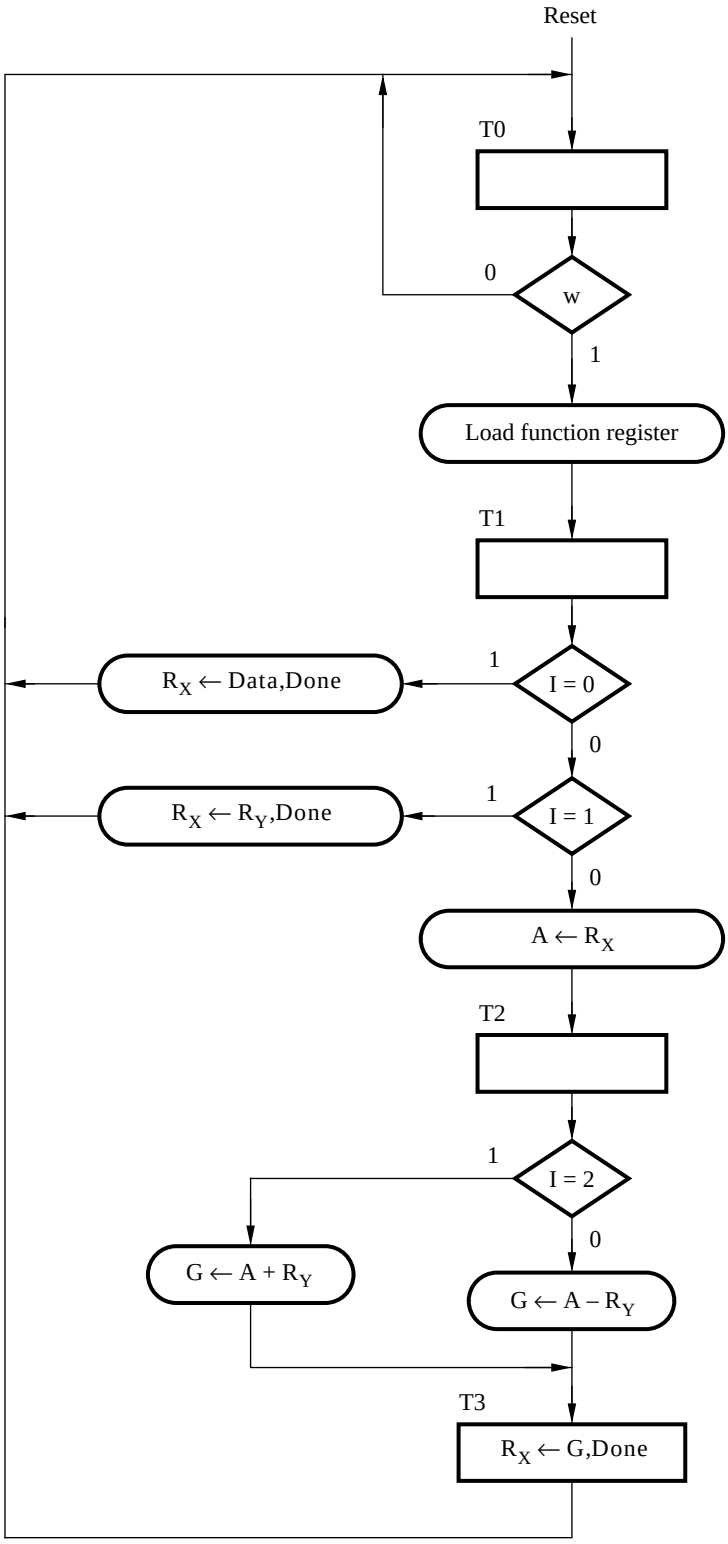


(*b*)

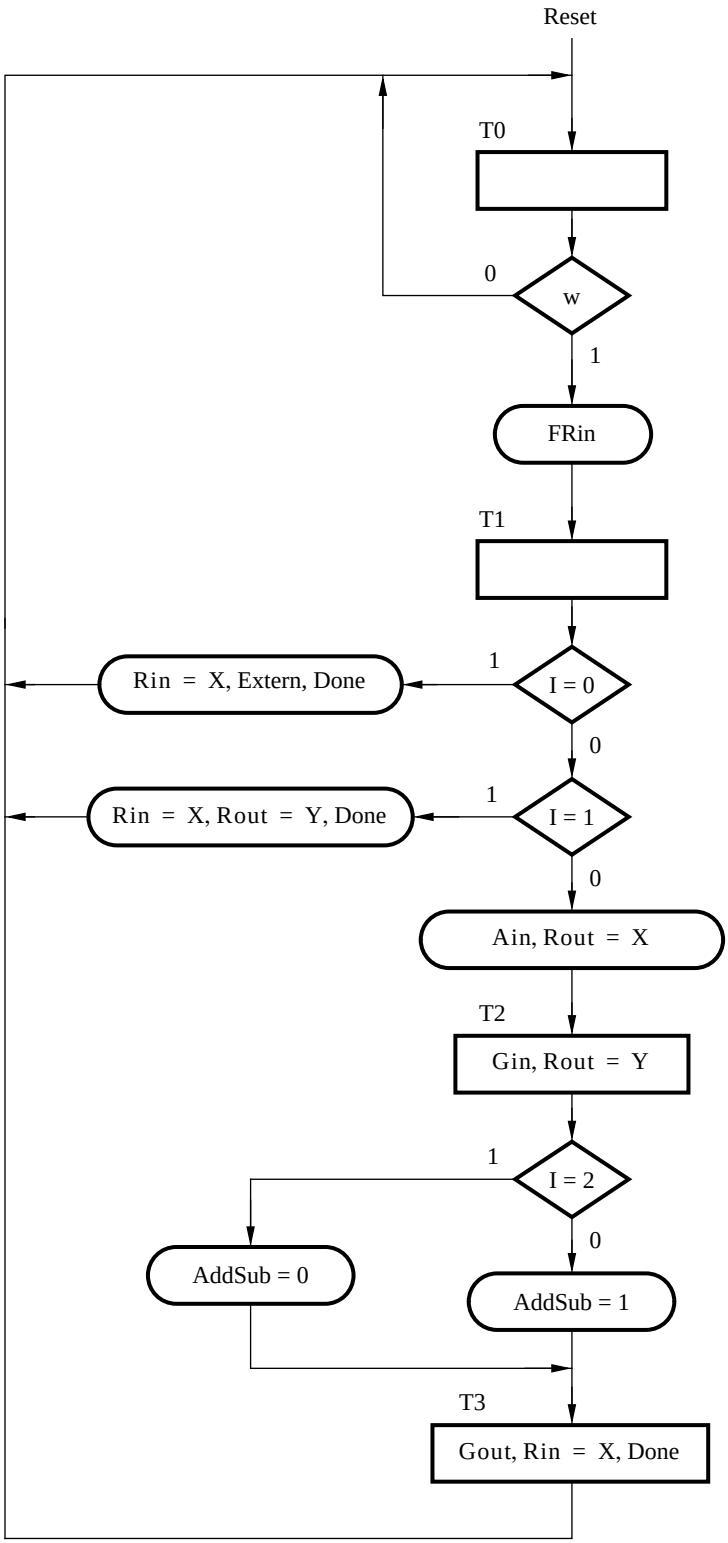
```
module swapmux (Data, Resetn, w, Clock, RinExt, BusWires);
    input   [7:0] Data;
    input   Resetn, w, Clock;
    input   [1:3] RinExt;
    output  [7:0] BusWires;
    reg   [7:0] BusWires;
    wire   [1:3] Rin;
    reg   [1:3] RinCntl, Rout;
    wire   [7:0] R1, R2, R3;
    reg   [1:0] y, Y;

// control circuit

    parameter S1 = 2'b00, S2 = 2'b01, S3 = 2'b10, S4 = 2'b11;

    always @(y or w)
    begin: State_table
        case (y)
            S1:    if (w == 0) Y = S1;
                   else Y = S2;
            S2:    Y = S3;
```

```
            S3:  Y = S4;
            S4:  Y = S1;
            default: Y = 3'bxxx;
        endcase
    end

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

    always @(y)
    begin: FSM_outputs
        RinCntl = 3'b000; Rout = 3'b000; // defaults
        case (y)
            S1:  ;
            S2:  begin
                    Rout[1] = 1; RinCntl[3] = 1;
                 end
            S3:  begin
                    Rout[2] = 1; RinCntl[1] = 1;
                 end
            S4:  begin
                    Rout[3] = 1; RinCntl[2] = 1;
                 end
        endcase
    end

// datapath circuit
    assign Rin = RinExt | RinExt;
    regn reg_1 (BusWires, Rin[1], Clock, R1);
    regn reg_2 (BusWires, Rin[2], Clock, R2);
    regn reg_3 (BusWires, Rin[3], Clock, R3);

    always @(Rout or Data or R1 or R2 or R3)
    begin
        if (Rout == 3'b100)
            BusWires = R1;
        else if (Rout == 3'b010)
            BusWires = R2;
        else if (Rout == 3'b001)
            BusWires = R3;
        else
            BusWires = Data;
    end

endmodule
```

10.20. An ASM chart for the processor is



10.21. (*a*) An ASM chart for the control circuit is



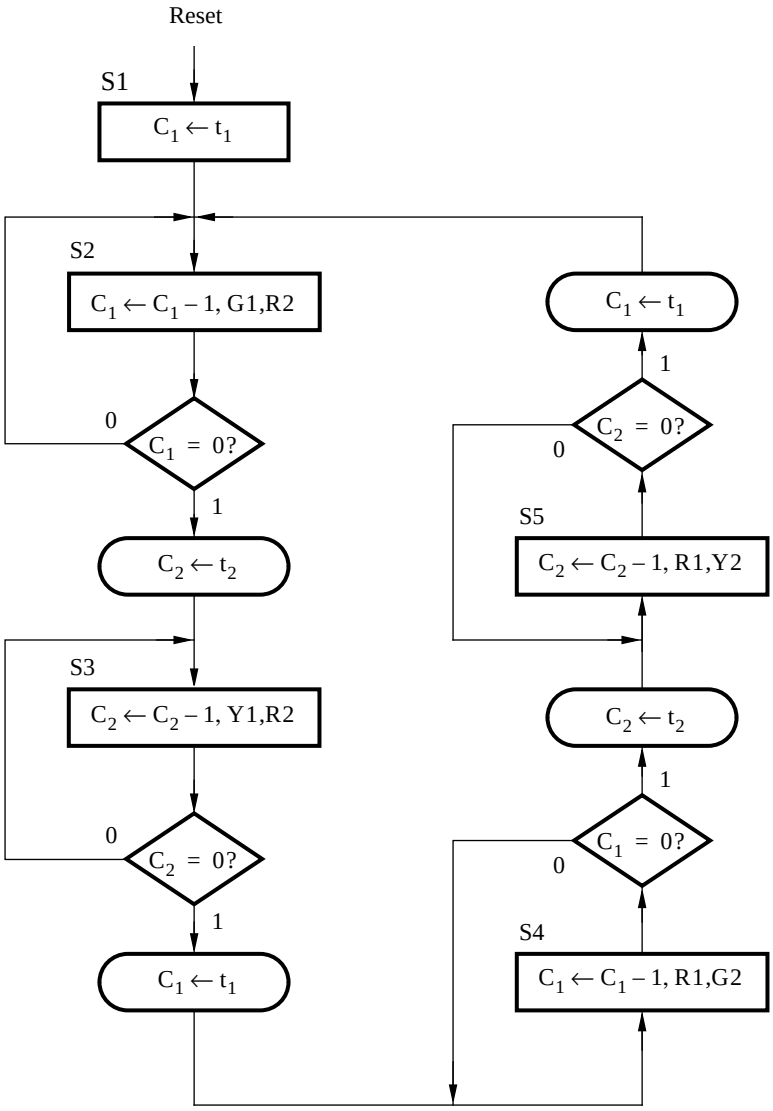
(*b*)

```
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;
    reg [0:3] Rin, Rout;
    reg [7:0] Sum;
    reg Extern, Done, Ain, FRin, 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;

    reg [1:0] t, T;

// control circuit

    parameter T0 = 2'b00, T1 = 2'b01, T2 = 2'b10, T3 = 2'b11;

    always @(t or w or I)
    begin: State_table
        case (t)
            T0:   if (w == 0) T = T0;
                  else T = T1;
            T1:   if (I == 2'b00 || I == 2'b01) T = T0;
                  else T = T2;
            T2:   T = T3;
            T3:   T = T0;
            default: T = 2'bxx;
        endcase
    end

    always @(posedge Clock or posedge Reset)
    begin: State_flipflops
        if (Reset == 1)
            t <= T0;
        else
            t <= T;
    end
```

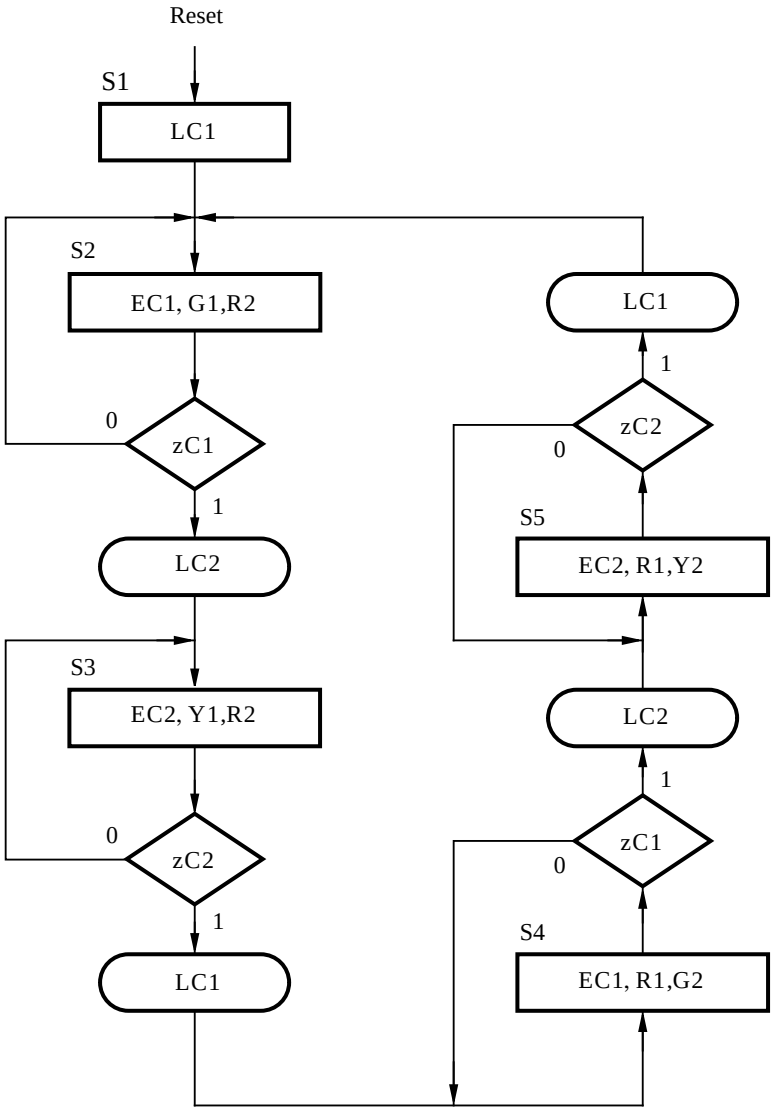
```
    always @(t or w or I)
    begin: FSM_outputs
        FRin = 0; Rin = 4'b0000; Rout = 4'b0000; Done = 0; // defaults
        Gin = 0; Gout = 0; Extern = 0; Ain = 0; AddSub = 0; // defaults
        case (t)
            T0: if (w == 1) FRin = 1;
                else FRin = 0;
            T1: begin
                    Ain = 1; // doesn't matter if we load A when not needed
                    if (I == 2'b00)
                    begin
                        Done = 1; Rin = Xreg; Rout = 4'b0000; Extern = 1;
                    end
                    else if (I == 2'b01)
                    begin
                        Done = 1; Rin = Xreg; Rout = Y; Extern = 0;
                    end
                    else
                    begin
                        Done = 0; Rin = 4'b0000; Rout = Xreg; Extern = 0;
                    end
                end
            T2: begin
                    Gin = 1; Rout = Y;
                    if (I == 2'b10) AddSub = 0;
                    else AddSub = 1;
                end
            T3: begin
                    Gout = 1; Rin = Xreg; Done = 1;
                end
        endcase
    end

//datapath circuit
    assign Func = {F, Rx, Ry};
    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);

    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);
```

```
    // 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
```

10.22. (a) An ASM chart for the traffic controller is



(b). The two counters, C_1 and C_2, each require clock enable and parallel-load inputs. Assuming that the clock enables signals are called *EC1* and *EC2* and the parallel-load inputs are called *LC1* and *LC2*, an ASM chart for the control circuit is



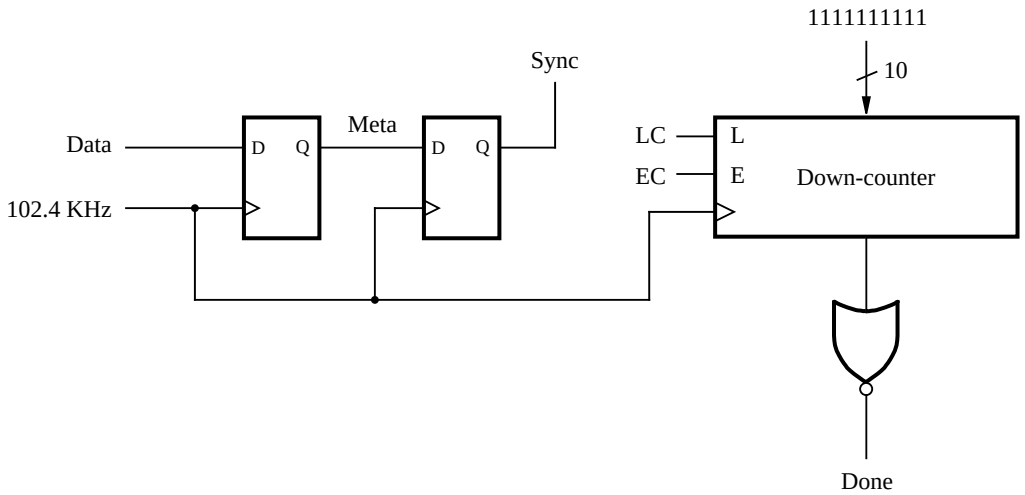
(*c*)

```
module traffic (Clock, Resetn, G1, Y1, R1, G2, Y2, R2);
    input Clock, Resetn;
    output G1, Y1, R1, G2, Y2, R2;
    reg G1, Y1, R1, G2, Y2, R2;

    reg [2:0] y, Y;
    reg EC1, EC2, LC1, LC2;
    reg [3:0] C1, C2;
    wire zC1, zC2;
    parameter Ticks1 = 4'b0011; // 4 ticks for C1
    parameter Ticks2 = 4'b0001; // 2 ticks for C2
```

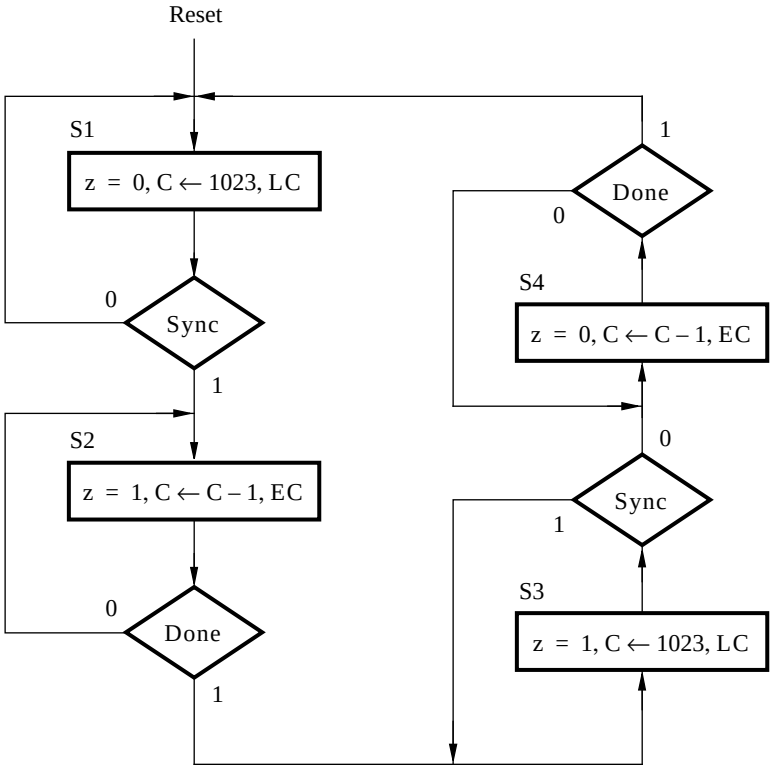
```
// control circuit

    parameter S1 = 3'b000, S2 = 3'b001, S3 = 3'b010, S4 = 3'b011, S5 = 3'b100;

    always @(y or zC1 or zC2)
    begin: State_table
        case (y)
            S1:  Y = S2;
            S2:  if (zC1 == 0) Y = S2;
                 else Y = S3;
            S3:  if (zC2 == 0) Y = S3;
                 else Y = S4;
            S4:  if (zC1 == 0) Y = S4;
                 else Y = S5;
            S5:  if (zC2 == 0) Y = S5;
                 else Y = S2;
            default: Y = 3'bxxx;
        endcase
    end

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

    always @(y or zC1 or zC2)
    begin: FSM_outputs
        G1 = 0; Y1 = 0; R1 = 0; G2 = 0; Y2 = 0; R2 = 0; // defaults
        LC1 = 0; EC1 = 0; LC2 = 0; EC2 = 0; // defaults
        case (y)
            S1:  LC1 = 1;
            S2:  begin
                     EC1 = 1; G1 = 1; R2 = 1;
                     if (zC1) LC2 = 1;
                     else LC2 = 0;
                 end
            S3:  begin
                     EC2 = 1; Y1 = 1; R2 = 1;
                     if (zC2) LC1 = 1;
                     else LC2 = 0;
                 end
            S4:  begin
                     EC1 = 1; R1 = 1; G2 = 1;
                     if (zC1) LC2 = 1;
                     else LC2 = 0;
                 end
```

```
            S5:  begin
                     EC2 = 1; R1 = 1; Y2 = 1;
                     if (zC2) LC1 = 1;
                     else LC2 = 0;
                 end
        endcase
    end

//datapath circuit

    assign zC1 = (C1 == 0);
    assign zC2 = (C2 == 0);

    always @(posedge Clock)
        if (LC1)
            C1 <= Ticks1;
        else if (EC1)
            C1 <= C1 − 1;

    always @(posedge Clock)
        if (LC2)
            C2 <= Ticks2;
        else if (EC2)
            C2 <= C2 − 1;

endmodule
```

10.23. The debounce circuit has three parts, as shown below. The *Data* signal from the switch has to be synchronized to the 102.4 KHz signal using two flip-flops. The synchronized signal called *Sync* is fed to an FSM. The FSM also uses the counter shown, which counts for 1024 cycles of the 102.4 KHz signal, providing a 10 msec delay.



An ASM chart for the FSM is given below. The FSM provides the *z* output, which is the debounced version of the *Data* signal.



10.24. (a) If we set $C_1 = 1$ pF, then $R_a = 0$ and $R_b = 1.43$ kΩ
(b) If we set $C_1 = 1$ pF, then $R_a = 1.42$ kΩ and $R_b = 0.71$ kΩ