```
`timescale 1ns / 1ns // `timescale time_unit/time_precision

module restoringDivider4bit(input[9:0] SW, input[3:0] KEY, input CLOCK_50, output[9:0] LEDR, output[6:0] HEX0,HEX1,HEX2,HEX3,HEX4,HEX5);
    // Wires to connect datapath and control
    wire ld_m,ld_a,ld_q,ld_a4;
    wire ld_r,ld_Q,ld_dividend;
    wire ld_alu_out,alu_op,enable_shift;
    wire Pload,ld_shift,set_a;

    // Datapath output wires
    wire[3:0] dividend,quotient;
    wire[4:0] remainder,divisor;
    wire a4; // ALSO A CONTROL INPUT WIRE

    // Input wires
    wire reset,go;
    wire[7:0] DATA_IN;

    assign reset = KEY[0];
    assign go = KEY[1];
    assign DATA_IN = SW[7:0];

    // Control module
    control c0(
        .clock(CLOCK_50),.reset(reset),.go(go),
        .a4(a4),
        .ld_m(ld_m),.ld_a(ld_a),.ld_q(ld_q),.ld_a4(ld_a4),
        .ld_r(ld_r),.ld_Q(ld_Q),.ld_dividend(ld_dividend),
        .ld_alu_out(ld_alu_out),.alu_op(alu_op),.enable_shift(enable_shift),
        .Pload(Pload),.ld_shift(ld_shift),.set_a(set_a)
    );

    // Datapath module
    datapath d0(
        .clock(CLOCK_50),.reset(reset),
        .DATA_IN(DATA_IN),
        .ld_m(ld_m),.ld_a(ld_a),.ld_q(ld_q),.ld_a4(ld_a4),
        .ld_r(ld_r),.ld_Q(ld_Q),.ld_dividend(ld_dividend),
        .ld_alu_out(ld_alu_out),.alu_op(alu_op),.enable_shift(enable_shift),
        .Pload(Pload),.ld_shift(ld_shift),.set_a(set_a),
        .m(divisor),.dividend(dividend),.Q(quotient),.r(remainder),
        .a4(a4)
    );

    // Output
    assign LEDR[3:0] = quotient;
    displayHEX h0(.BIN(divisor[3:0]),.HEX(HEX0));
    displayHEX h1(.BIN(4'b0),.HEX(HEX1));
    displayHEX h2(.BIN(dividend),.HEX(HEX2));
    displayHEX h3(.BIN(4'b0),.HEX(HEX3));
    displayHEX h4(.BIN(quotient),.HEX(HEX4));
    displayHEX h5(.BIN(remainder[3:0]),.HEX(HEX5));
endmodule

// Tracks state and datapath control signals depending on state
module control(
    input clock,reset,go,
    input a4,
    output reg ld_m,ld_a,ld_q,ld_a4,
    output reg ld_r,ld_Q,ld_dividend,
    output reg ld_alu_out,alu_op,enable_shift,
    output reg Pload,ld_shift,set_a
    );

    // Keeps track of state
    reg[3:0] current_state,next_state;
    reg[1:0] counter;
    reg reset_counter,increment;
```

```
69
70      // Assigning state values
71      localparam  S_LOAD_VALS         = 4'd0,
72                  S_LOAD_VALS_WAIT    = 4'd1,
73                  S_CYCLE_0           = 4'd2,
74                  S_CYCLE_1           = 4'd3,
75                  S_CYCLE_2           = 4'd4,
76                  S_CYCLE_3           = 4'd5,
77                  S_CYCLE_4           = 4'd6,
78                  S_CYCLE_5           = 4'd7,
79                  S_CYCLE_6           = 4'd8,
80                  S_CYCLE_7           = 4'd9;
81
82      // Counter register for number of cycles done
83      always @(posedge clock)
84      begin
85          if(reset) // Active high reset
86              counter <= 2'b0;
87          else if(reset_counter) // Reset to beginning of cycle
88              counter <= 2'b0;
89          else if(increment) // Increment counter (done at end of a cycle)
90              counter <= counter+1;
91      end
92
93      // State table
94      always @(*)
95      begin
96          case(current_state)
97              S_LOAD_VALS: // Loop in state until value is input
98                  next_state = go ? S_LOAD_VALS_WAIT:S_LOAD_VALS;
99              S_LOAD_VALS_WAIT: // Loop in state until go signal goes low
100                 next_state = go ? S_LOAD_VALS_WAIT:S_CYCLE_0;
101             S_CYCLE_0: // Pload shift register
102                 next_state = S_CYCLE_1;
103             S_CYCLE_1: // Shift shift register
104                 next_state = S_CYCLE_2;
105             S_CYCLE_2: // Load shifted values to a and q
106                 next_state = S_CYCLE_3;
107             S_CYCLE_3: // a <- a-m
108                 next_state = S_CYCLE_4;
109             S_CYCLE_4: // q0 <- !a4
110                 next_state = a4 ? S_CYCLE_5:S_CYCLE_6;
111             S_CYCLE_5: // a <- a+m
112                 next_state = S_CYCLE_6;
113             S_CYCLE_6: // Increment counter
114                 next_state = (counter==3) ? S_CYCLE_7:S_CYCLE_0;
115             S_CYCLE_7: // Load a and q values to output
116                 next_state = S_LOAD_VALS;
117             default: next_state = S_LOAD_VALS;
118         endcase
119     end
120
121     // Changing data control signals
122     always @(*)
123     begin
124         // Initializing signals to 0 to avoid latches
125         ld_m = 0; ld_a = 0; ld_q = 0; ld_a4 = 0;
126         ld_r = 0; ld_Q = 0; ld_dividend = 0;
127         ld_alu_out = 0; alu_op = 0; enable_shift = 0;
128         Pload = 0; ld_shift = 0; set_a = 0;
129         increment = 0; reset_counter = 0;
130
131         case(current_state)
132             S_LOAD_VALS: // Load input values
133             begin
134                 ld_m = 1;
135                 set_a = 1;
136                 ld_dividend = 1;
137                 ld_shift = 0; ld_q = 1;
```

```
                reset_counter = 1;
            end
            S_CYCLE_0: // Pload shift register
            begin
                Pload = 1;
            end
            S_CYCLE_1: // Shift shift register
            begin
                enable_shift = 1;
            end
            S_CYCLE_2: // Load shifted values to a and q
            begin
                ld_shift = 1; ld_q = 1;
                ld_alu_out = 0; ld_a = 1;
            end
            S_CYCLE_3: // a <- a-m
            begin
                ld_alu_out = 1; ld_a = 1;
                alu_op = 1; // Subtraction
            end
            S_CYCLE_4: // q0 <- !a4
            begin
                ld_a4 = 1;
            end
            S_CYCLE_5: // a <- a+m
            begin
                ld_alu_out = 1; ld_a = 1;
                alu_op = 0;
            end
            S_CYCLE_6: // Increment counter
            begin
                increment = 1;
            end
            S_CYCLE_7: // Load a and q values to output
            begin
                ld_r = 1;
                ld_Q = 1;
            end
        endcase
    end

    // Register for current state
    always @(posedge clock)
    begin
        if(reset) // Reset to value input state, active high
            current_state <= S_LOAD_VALS;
        else // Load next state
            current_state <= next_state;
    end
endmodule

module datapath(
    input clock,reset,
    input[7:0] DATA_IN,
    input ld_m,ld_a,ld_q,ld_a4,
    input ld_r,ld_Q,ld_dividend,
    input ld_alu_out,alu_op,enable_shift,
    input Pload,ld_shift,set_a,
    output reg[3:0] dividend,Q,
    output reg[4:0] r,m,
    output reg[2:0] counter,
    output a4
    );

    // Internal registers
    reg[4:0] a,alu_out;
    reg[3:0] q;
    reg[8:0] shift_out;

```

```
    // Registers m,a,q,r,Q, and dividend with input logic
    always @(posedge clock)
    begin
        if(reset) // Active high reset
        begin
            m <= 5'b0;
            a <= 5'b0;
            q <= 4'b0;
            r <= 5'b0;
            Q <= 4'b0;
            dividend <= 4'b0;
        end
        else
        begin
            // Divisor register
            if(ld_m)
                m <= {1'b0,DATA_IN[3:0]};
            // Register A
            if(ld_a)
                a <= ld_alu_out ? alu_out:shift_out[8:4];
            else if(set_a)
                a <= 5'b0;
            // Dividend register (NOT OUTPUT)
            if(ld_q)
                q <= ld_shift ? shift_out[3:0]:DATA_IN[7:4];
            else if(ld_a4)
                q[0] <= !a[4];
            // Remainder register
            if(ld_r)
                r <= a;
            // Quotient register
            if(ld_Q)
                Q <= q;
            // Dividend register (OUTPUT)
            if(ld_dividend)
                dividend <= DATA_IN[7:4];
        end
    end

    // ALU
    always @(*)
    begin
        case(alu_op)
            0: // L+R
                alu_out = a+m;
            1: // L-R
                alu_out = a-m;
            default: alu_out = 0;
        endcase
    end

    // Shift register
    always @(posedge clock)
    begin
        if(reset) // Active high reset
            shift_out <= 0;
        else if(Pload) // Parallel load
            shift_out <= {a,q};
        else if(enable_shift) // Shift left, insert complement of MSB to LSB
            shift_out <= {shift_out[7:0],1'b0};
    end

    // Assign a4
    assign a4 = a[4];
endmodule

// Turns decimal to HEX (from lab 2)
module displayHEX(input [3:0] BIN, output [6:0] HEX);
     wire x = BIN[3], y = BIN[2], z = BIN[1], w = BIN[0];
```

```
276         //assign each segment with appropriate formula
277         assign HEX[0] = ~((x|y|z|~w)&(x|~y|z|w)&(~x|y|~z|~w)&(~x|~y|z|~w));
278         assign HEX[1] =
            ~((x|~y|z|~w)&(x|~y|~z|w)&(~x|y|~z|~w)&(~x|~y|z|w)&(~x|~y|~z|w)&(~x|~y|z|~w));
279         assign HEX[2] = ~((x|y|~z|w)&(~x|~y|z|w)&(~x|~y|~z|w)&(~x|~y|~z|~w));
280         assign HEX[3] =
            ~((x|y|z|~w)&(x|~y|z|w)&(x|~y|~z|~w)&(~x|y|z|~w)&(~x|y|~z|w)&(~x|~y|~z|~w));
281         assign HEX[4] =
            ~((x|y|z|~w)&(x|y|~z|~w)&(x|~y|z|w)&(x|~y|z|~w)&(x|~y|~z|~w)&(~x|y|z|~w));
282         assign HEX[5] = ~((x|y|z|~w)&(x|y|~z|w)&(x|y|~z|~w)&(x|~y|~z|~w)&(~x|~y|z|~w));
283         assign HEX[6] = ~((x|y|z|w)&(x|y|z|~w)&(x|~y|~z|~w)&(~x|~y|z|w));
284   endmodule
285
```