Stanislav Pochynok 1001339205

LAB 6

Part I

2) To reset SW[0] has to be switched to 0 when clock hoes high, so reset\_n is synchronous and active low.

3)

// SW[0] reset when 0

// SW[1] input signal (w)

// KEY[0] clock signal

// LEDR[2:0] displays current state

// LEDR[9] displays output

module sequence\_detector(SW, KEY, LEDR);

input [9:0] SW;

input [3:0] KEY;

output [9:0] LEDR;

wire w, clock, resetn, out\_light;

reg [2:0] y\_Q, Y\_D; // y\_Q represents current state, Y\_D represents next state

localparam A = 3'b000, B = 3'b001, C = 3'b010, D = 3'b011, E = 3'b100, F = 3'b101, G = 3'b110;

assign w = SW[1];

assign clock = ~KEY[0];

assign resetn = SW[0];

// State table

// The state table should only contain the logic for state transitions

// Do not mix in any output logic. The output logic should be handled separately.

// This will make it easier to read, modify and debug the code.

always @(\*)

begin: state\_table

case (y\_Q)

A: begin

if (!w) Y\_D = A;

else Y\_D = B;

end

B: begin

if(!w) Y\_D = A;

else Y\_D = C;

end

C: begin

if (!w) Y\_D = D;

else Y\_D = E;

end

D: begin

if (!w) Y\_D = F;

else Y\_D = E;

end

E: begin

if (!w) Y\_D = G;

else Y\_D = A;

end

F: begin

if (!w) Y\_D = F;

else Y\_D = E;

end

G: begin

if (!w) Y\_D = C;

else Y\_D = A;

end

default: Y\_D = A;

endcase

end // state\_table

// State Register (i.e., FFs)

always @(posedge clock)

begin: state\_FFs

if(resetn == 1'b0)

y\_Q <= A; // Should set reset state to state A

else

y\_Q <= Y\_D;

end // State Register

// Output logic

// Set out\_light to 1 to turn on LED when in relevant states

assign out\_light = ((y\_Q == A) || (y\_Q == B) || (y\_Q == C)

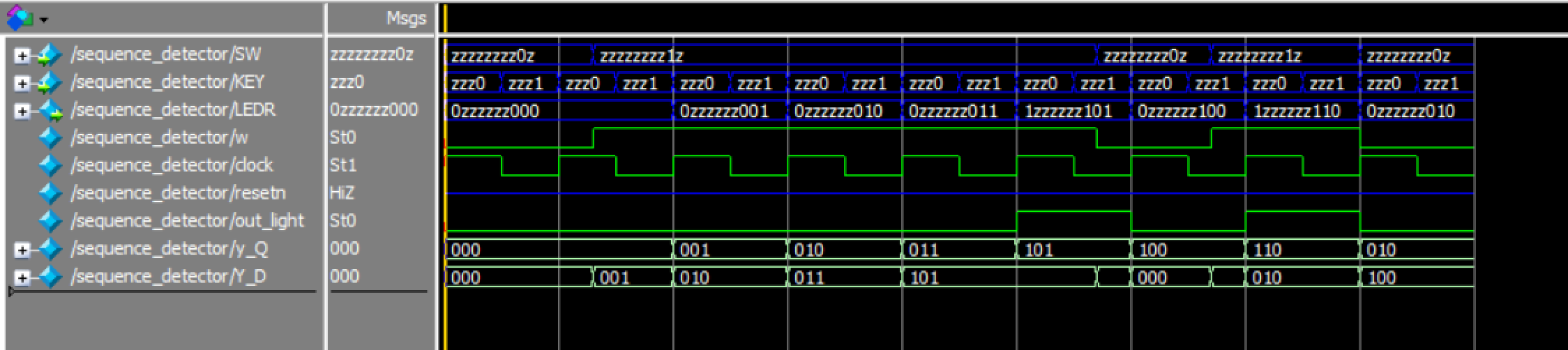
|| (y\_Q == D) || (y\_Q == E) || (y\_Q == F) || (y\_Q == G));

assign LEDR[9] = out\_light;

assign LEDR[2:0] = y\_Q;

endmodule

4)



Part II

2)

Load a, load b, load c, load x

multiply a with x, and store it to register A. A = a \* x

multiply A with x, and store it in register A. A = a \* (x \* x)

multiply b with x, store it in register B. B = (b \* x)

Add A and B, and store it in register A. A = a \* (x \* x) + (b \* x)

Load c and add c with A, and output value a \* (x \* x) + (b \* x) + c

|  |  |  |
| --- | --- | --- |
| CURRENT STATE | GO | NEXT\_STATE |
| S\_LOAD\_A | 0 | S\_LOAD\_A |
| S\_LOAD\_A | 1 | S\_LOAD\_A\_WAIT |
| S\_LOAD\_A\_WAIT | 0 | S\_LOAD\_A\_WAIT |
| S\_LOAD\_A\_WAIT | 1 | S\_LOAD\_B |
| S\_LOAD\_B | 0 | S\_LOAD\_B |
| S\_LOAD\_B | 1 | S\_LOAD\_B\_WAIT |
| S\_LOAD\_B\_WAIT | 0 | S\_LOAD\_B\_WAIT |
| S\_LOAD\_B\_WAIT | 1 | S\_LOAD\_C |
| S\_LOAD\_C | 0 | S\_LOAD\_C |
| S\_LOAD\_C | 1 | S\_LOAD\_C\_WAIT |
| S\_LOAD\_C\_WAIT | 0 | S\_LOAD\_C\_WAIT |
| S\_LOAD\_C\_WAIT | 1 | S\_LOAD\_X |
| S\_LOAD\_X | 0 | S\_LOAD\_X |
| S\_LOAD\_X | 1 | S\_LOAD\_X\_WAIT |
| S\_LOAD\_X\_WAIT | 0 | S\_LOAD\_X\_WAIT |
| S\_LOAD\_X\_WAIT | 1 | S\_CYCLE\_0 |
| S\_CYCLE\_0 | 1 | S\_CYCLE\_1 |
| S\_CYCLE\_1 | 1 | S\_CYCLE\_2 |
| S\_CYCLE\_2 | 1 | S\_CYCLE\_3 |
| S\_CYCLE\_3 | 1 | S\_CYCLE\_4 |
| S\_CYCLE\_4 | 1 | S\_LOAD\_A |

4)

//Sw[7:0] data\_in

//KEY[0] synchronous reset when pressed

//KEY[1] go signal

//LEDR displays result

//HEX0 & HEX1 also displays result

module fpga\_top(SW, KEY, CLOCK\_50, LEDR, HEX0, HEX1);

    input [9:0] SW;

    input [3:0] KEY;

    input CLOCK\_50;

    output [9:0] LEDR;

    output [6:0] HEX0, HEX1;

    wire resetn;

    wire go;

    wire [7:0] data\_result;

    assign go = ~KEY[1];

    assign resetn = KEY[0];

    part2 u0(

        .clk(CLOCK\_50),

        .resetn(resetn),

        .go(go),

        .data\_in(SW[7:0]),

        .data\_result(data\_result)

    );

    assign LEDR[9:0] = {2'b00, data\_result};

    hex\_decoder H0(

        .hex\_digit(data\_result[3:0]),

        .segments(HEX0)

        );

    hex\_decoder H1(

        .hex\_digit(data\_result[7:4]),

        .segments(HEX1)

        );

endmodule

module part2(

    input clk,

    input resetn,

    input go,

    input [7:0] data\_in,

    output [7:0] data\_result

    );

    // lots of wires to connect our datapath and control

    wire ld\_a, ld\_b, ld\_c, ld\_x, ld\_r;

    wire ld\_alu\_out;

    wire [1:0]  alu\_select\_a, alu\_select\_b;

    wire alu\_op;

    control C0(

        .clk(clk),

        .resetn(resetn),

        .go(go),

        .ld\_alu\_out(ld\_alu\_out),

        .ld\_x(ld\_x),

        .ld\_a(ld\_a),

        .ld\_b(ld\_b),

        .ld\_c(ld\_c),

        .ld\_r(ld\_r),

        .alu\_select\_a(alu\_select\_a),

        .alu\_select\_b(alu\_select\_b),

        .alu\_op(alu\_op)

    );

    datapath D0(

        .clk(clk),

        .resetn(resetn),

        .ld\_alu\_out(ld\_alu\_out),

        .ld\_x(ld\_x),

        .ld\_a(ld\_a),

        .ld\_b(ld\_b),

        .ld\_c(ld\_c),

        .ld\_r(ld\_r),

        .alu\_select\_a(alu\_select\_a),

        .alu\_select\_b(alu\_select\_b),

        .alu\_op(alu\_op),

        .data\_in(data\_in),

        .data\_result(data\_result)

    );

 endmodule

module control(

    input clk,

    input resetn,

    input go,

    output reg  ld\_a, ld\_b, ld\_c, ld\_x, ld\_r,

    output reg  ld\_alu\_out,

    output reg [1:0]  alu\_select\_a, alu\_select\_b,

    output reg alu\_op

    );

    reg [3:0] current\_state, next\_state;

    localparam  S\_LOAD\_A        = 4'd0,

                S\_LOAD\_A\_WAIT   = 4'd1,

                S\_LOAD\_B        = 4'd2,

                S\_LOAD\_B\_WAIT   = 4'd3,

                S\_LOAD\_C        = 4'd4,

                S\_LOAD\_C\_WAIT   = 4'd5,

                S\_LOAD\_X        = 4'd6,

                S\_LOAD\_X\_WAIT   = 4'd7,

                S\_CYCLE\_0       = 4'd8,

                S\_CYCLE\_1       = 4'd9,

                S\_CYCLE\_2       = 4'd10,

S\_CYCLE\_3  = 4'd11,

S\_CYCLE\_4  = 4'd12;

    // Next state logic aka our state table

    always@(\*)

    begin: state\_table

            case (current\_state)

                S\_LOAD\_A: next\_state = go ? S\_LOAD\_A\_WAIT : S\_LOAD\_A; // Loop in current state until value is input

                S\_LOAD\_A\_WAIT: next\_state = go ? S\_LOAD\_A\_WAIT : S\_LOAD\_B; // Loop in current state until go signal goes low

                S\_LOAD\_B: next\_state = go ? S\_LOAD\_B\_WAIT : S\_LOAD\_B; // Loop in current state until value is input

                S\_LOAD\_B\_WAIT: next\_state = go ? S\_LOAD\_B\_WAIT : S\_LOAD\_C; // Loop in current state until go signal goes low

                S\_LOAD\_C: next\_state = go ? S\_LOAD\_C\_WAIT : S\_LOAD\_C; // Loop in current state until value is input

                S\_LOAD\_C\_WAIT: next\_state = go ? S\_LOAD\_C\_WAIT : S\_LOAD\_X; // Loop in current state until go signal goes low

                S\_LOAD\_X: next\_state = go ? S\_LOAD\_X\_WAIT : S\_LOAD\_X; // Loop in current state until value is input

                S\_LOAD\_X\_WAIT: next\_state = go ? S\_LOAD\_X\_WAIT : S\_CYCLE\_0; // Loop in current state until go signal goes low

                S\_CYCLE\_0: next\_state = S\_CYCLE\_1;

S\_CYCLE\_1: next\_state = S\_CYCLE\_2;

S\_CYCLE\_2: next\_state = S\_CYCLE\_3;

S\_CYCLE\_3: next\_state = S\_CYCLE\_4;

                S\_CYCLE\_4: next\_state = S\_LOAD\_A; // we will be done our two operations, start over after

            default:     next\_state = S\_LOAD\_A;

        endcase

    end // state\_table

    // Output logic aka all of our datapath control signals

    always @(\*)

    begin: enable\_signals

        // By default make all our signals 0

        ld\_alu\_out = 1'b0;

        ld\_a = 1'b0;

        ld\_b = 1'b0;

        ld\_c = 1'b0;

        ld\_x = 1'b0;

        ld\_r = 1'b0;

        alu\_select\_a = 2'b00;

        alu\_select\_b = 2'b00;

        alu\_op       = 1'b0;

        case (current\_state)

            S\_LOAD\_A: begin

                ld\_a = 1'b1;

                end

            S\_LOAD\_B: begin

                ld\_b = 1'b1;

                end

            S\_LOAD\_C: begin

                ld\_c = 1'b1;

                end

            S\_LOAD\_X: begin

                ld\_x = 1'b1;

                end

            S\_CYCLE\_0: begin // Do A <- A \* X

                ld\_alu\_out = 1'b1; ld\_a = 1'b1; // store result back into A

                alu\_select\_a = 2'b00; // Select register A

                alu\_select\_b = 2'b11; // Select register X

                alu\_op = 1'b1; // Do multiply operation

            end

S\_CYCLE\_1: begin // Do A <- AX \* X

                ld\_alu\_out = 1'b1; ld\_a = 1'b1; // store result back into A

                alu\_select\_a = 2'b00; // Select register A

                alu\_select\_b = 2'b11; // Select register X

                alu\_op = 1'b1; // Do multiply operation

            end

S\_CYCLE\_2: begin // Do B <- B \* X

                ld\_alu\_out = 1'b1; ld\_b = 1'b1; // store result back into B

                alu\_select\_a = 2'b01; // Select register B

                alu\_select\_b = 2'b11; // Select register X

                alu\_op = 1'b1; // Do multiply operation

            end

S\_CYCLE\_3: begin // Do A <- AX^2 \* BX

                ld\_alu\_out = 1'b1; ld\_a = 1'b1; // store result into A

                alu\_select\_a = 2'b00; // Select register A

                alu\_select\_b = 2'b01; // Select register B

                alu\_op = 1'b0; // Do multiply operation

            end

            S\_CYCLE\_4: begin

                ld\_r = 1'b1; // store result in result register

                alu\_select\_a = 2'b00; // Select register A

                alu\_select\_b = 2'b10; // Select register C

                alu\_op = 1'b0; // Do Add operation

            end

        // default:    // don't need default since we already made sure all of our outputs were assigned a value at the start of the always block

        endcase

    end // enable\_signals

    // current\_state registers

    always@(posedge clk)

    begin: state\_FFs

        if(!resetn)

            current\_state <= S\_LOAD\_A;

        else

            current\_state <= next\_state;

    end // state\_FFS

endmodule

module datapath(

    input clk,

    input resetn,

    input [7:0] data\_in,

    input ld\_alu\_out,

    input ld\_x, ld\_a, ld\_b, ld\_c,

    input ld\_r,

    input alu\_op,

    input [1:0] alu\_select\_a, alu\_select\_b,

    output reg [7:0] data\_result

    );

    // input registers

    reg [7:0] a, b, c, x;

    // output of the alu

    reg [7:0] alu\_out;

    // alu input muxes

    reg [7:0] alu\_a, alu\_b;

    // Registers a, b, c, x with respective input logic

    always @ (posedge clk) begin

        if (!resetn) begin

            a <= 8'd0;

            b <= 8'd0;

            c <= 8'd0;

            x <= 8'd0;

        end

        else begin

            if (ld\_a)

                a <= ld\_alu\_out ? alu\_out : data\_in; // load alu\_out if load\_alu\_out signal is high, otherwise load from data\_in

            if (ld\_b)

                b <= ld\_alu\_out ? alu\_out : data\_in; // load alu\_out if load\_alu\_out signal is high, otherwise load from data\_in

            if (ld\_x)

                x <= data\_in;

            if (ld\_c)

                c <= data\_in;

        end

    end

    // Output result register

    always @ (posedge clk) begin

        if (!resetn) begin

            data\_result <= 8'd0;

        end

        else

            if(ld\_r)

                data\_result <= alu\_out;

    end

    // The ALU input multiplexers

    always @(\*)

    begin

        case (alu\_select\_a)

            2'd0:

                alu\_a = a;

            2'd1:

                alu\_a = b;

            2'd2:

                alu\_a = c;

            2'd3:

                alu\_a = x;

            default: alu\_a = 8'd0;

        endcase

        case (alu\_select\_b)

            2'd0:

                alu\_b = a;

            2'd1:

                alu\_b = b;

            2'd2:

                alu\_b = c;

            2'd3:

                alu\_b = x;

            default: alu\_b = 8'd0;

        endcase

    end

    // The ALU

    always @(\*)

    begin : ALU

        // alu

        case (alu\_op)

            0: begin

                   alu\_out = alu\_a + alu\_b; //performs addition

               end

            1: begin

                   alu\_out = alu\_a \* alu\_b; //performs multiplication

               end

            default: alu\_out = 8'd0;

        endcase

    end

endmodule

module hex\_decoder(hex\_digit, segments);

    input [3:0] hex\_digit;

    output reg [6:0] segments;

    always @(\*)

        case (hex\_digit)

            4'h0: segments = 7'b100\_0000;

            4'h1: segments = 7'b111\_1001;

            4'h2: segments = 7'b010\_0100;

            4'h3: segments = 7'b011\_0000;

            4'h4: segments = 7'b001\_1001;

            4'h5: segments = 7'b001\_0010;

            4'h6: segments = 7'b000\_0010;

            4'h7: segments = 7'b111\_1000;

            4'h8: segments = 7'b000\_0000;

            4'h9: segments = 7'b001\_1000;

            4'hA: segments = 7'b000\_1000;

            4'hB: segments = 7'b000\_0011;

            4'hC: segments = 7'b100\_0110;

            4'hD: segments = 7'b010\_0001;

            4'hE: segments = 7'b000\_0110;

            4'hF: segments = 7'b000\_1110;

            default: segments = 7'h7f;

        endcase

endmodule

