

PART 1 QUESTION

Resetrn is a active low synchronous reset.

Turn SW[0] to 0 and click KEY[0]

Current State	W	Next	Out
A	0	A	0
A	1	B	0
B	0	A	0
B	1	C	0
C	0	E	0
C	1	D	0
D	0	E	0
D	1	F	0
E	0	A	0
E	1	G	0
F	0	E	1
F	1	F	1
G	0	A	1
G	1	C	1

PART 1 VERILOG CODE

```
// 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 = E;
```

```

        else Y_D = D;
    end
D: begin
    if(!w) Y_D = E;
    else Y_D = F;
end
E: begin
    if(!w) Y_D = A;
    else Y_D = G;
end
F: begin
    if(!w) Y_D = E;
    else Y_D = F;
end
G: begin
    if(!w) Y_D = A;
    else Y_D = C;
end
default: Y_D = A;
endcase
end // state_table

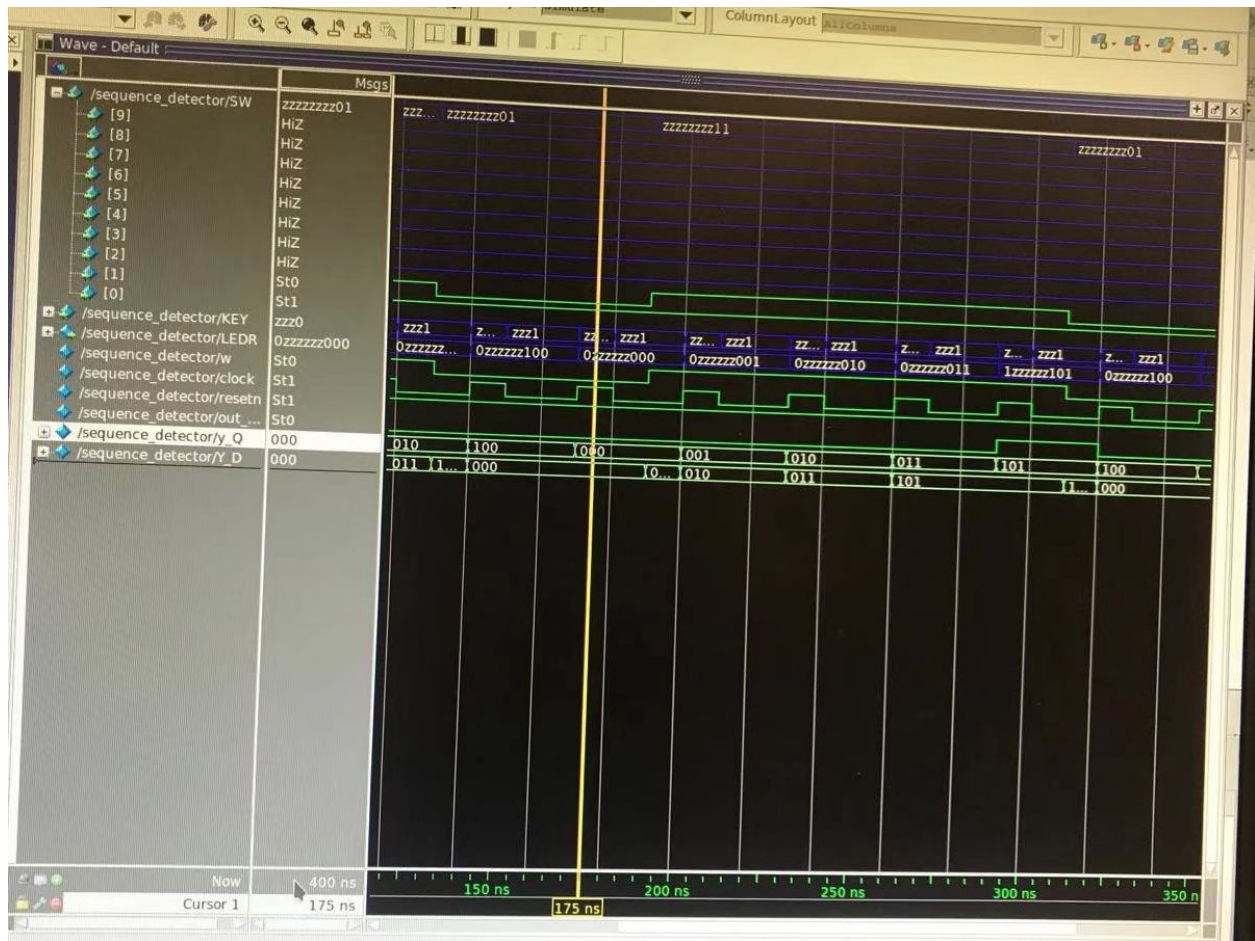
// State Register (i.e., FFs)
always @(posedge clock)
begin: state_FF
    if(resetn == 1'b0)
        y_Q <= A; // Should set reset state to state A
    else
        y_Q <= Y_D;
end // State Register

// Output logic
// Set out_light to 1 to turn on LED when in relevant states
assign out_light = ((y_Q == F) || (y_Q == G));

assign LEDR[9] = out_light;
assign LEDR[2:0] = y_Q;
endmodule

```

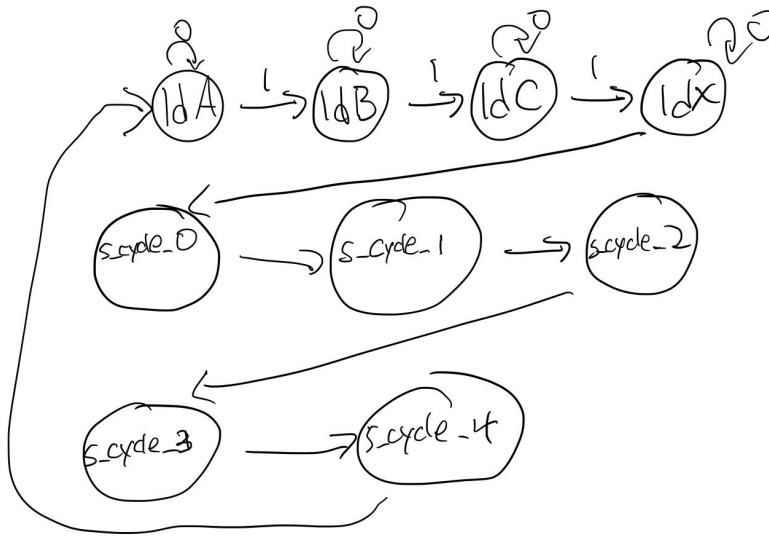
PART1 VSIM SCREENSHOT



PART2 DATAPATH

	alu_select_a	alu_select_b	alu_op	ld_alu_out	ld_a	ld_b	ld_r
b*x	11	01	1	1	0	1	0
bx+a	00	01	0	1	1	0	0
c*x	10	11	1	1	0	1	0
cx*x	11	01	1	1	0	1	0
Cx ² + bx+a	00	01	0	1	0	0	1

PART2 FSM



PART2 VERILOG (modified)

//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),
    .resetsn(resetsn),

    .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 resetsn,
    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;
        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 B <- B*X
```

```
        alu_select_a = 2'b11;
```

```
        alu_select_b = 2'b01;
```

```
        alu_op = 1'b1;
```

```
        ld_alu_out = 1'b1;
```

```
        ld_a = 1'b0;
```

```
        ld_b = 1'b1;
```

```
end
```

```
    S_CYCLE_1: begin
```

```
        alu_select_a = 2'b00;
```

```
        alu_select_b = 2'b01;
```

```
        alu_op = 1'b0;
```

```
        ld_alu_out = 1'b1;
```

```

        ld_a = 1'b1;
        ld_b = 1'b0;
    end
    S_CYCLE_2: begin
        alu_select_a = 2'b10;
        alu_select_b = 2'b01;
        alu_op = 1'b1;
        ld_alu_out = 1'b1;
        ld_a = 1'b0;
        ld_b = 1'b1;
    end
    S_CYCLE_3: begin
        alu_select_a = 2'b11;
        alu_select_b = 2'b01;
        alu_op = 1'b1;
        ld_alu_out = 1'b1;
        ld_a = 1'b0;
        ld_b = 1'b1;
    end
    S_CYCLE_4: begin
        alu_select_a = 2'b00;
        alu_select_b = 2'b01;
        alu_op = 1'b0;
        ld_alu_out = 1'b1;
        ld_a = 1'b0;
        ld_b = 1'b0;
        Ld_r = 1'b1;
    end
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_FF

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
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

```

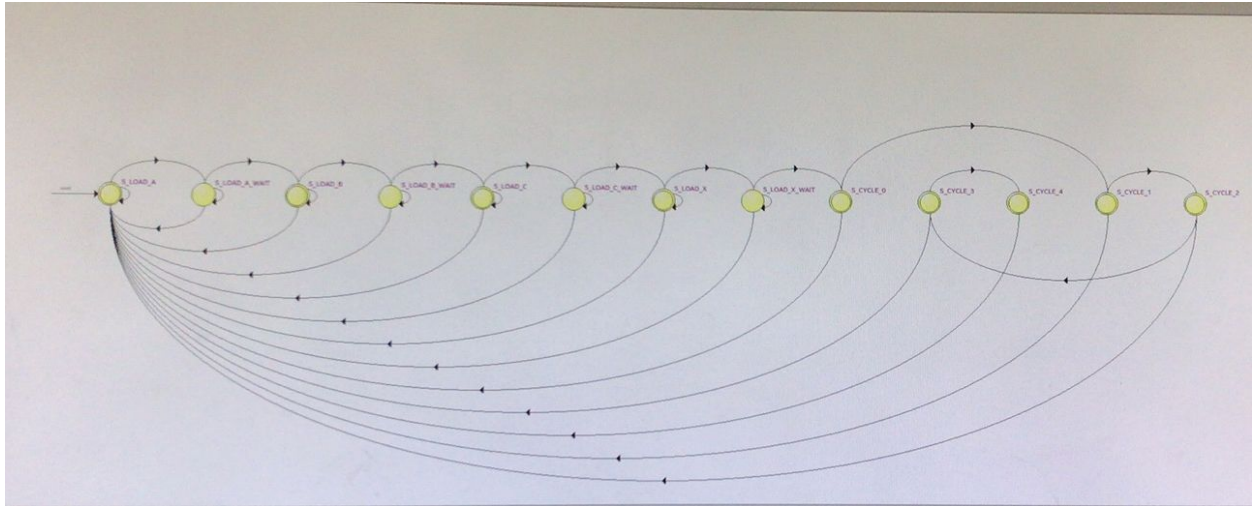
```

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

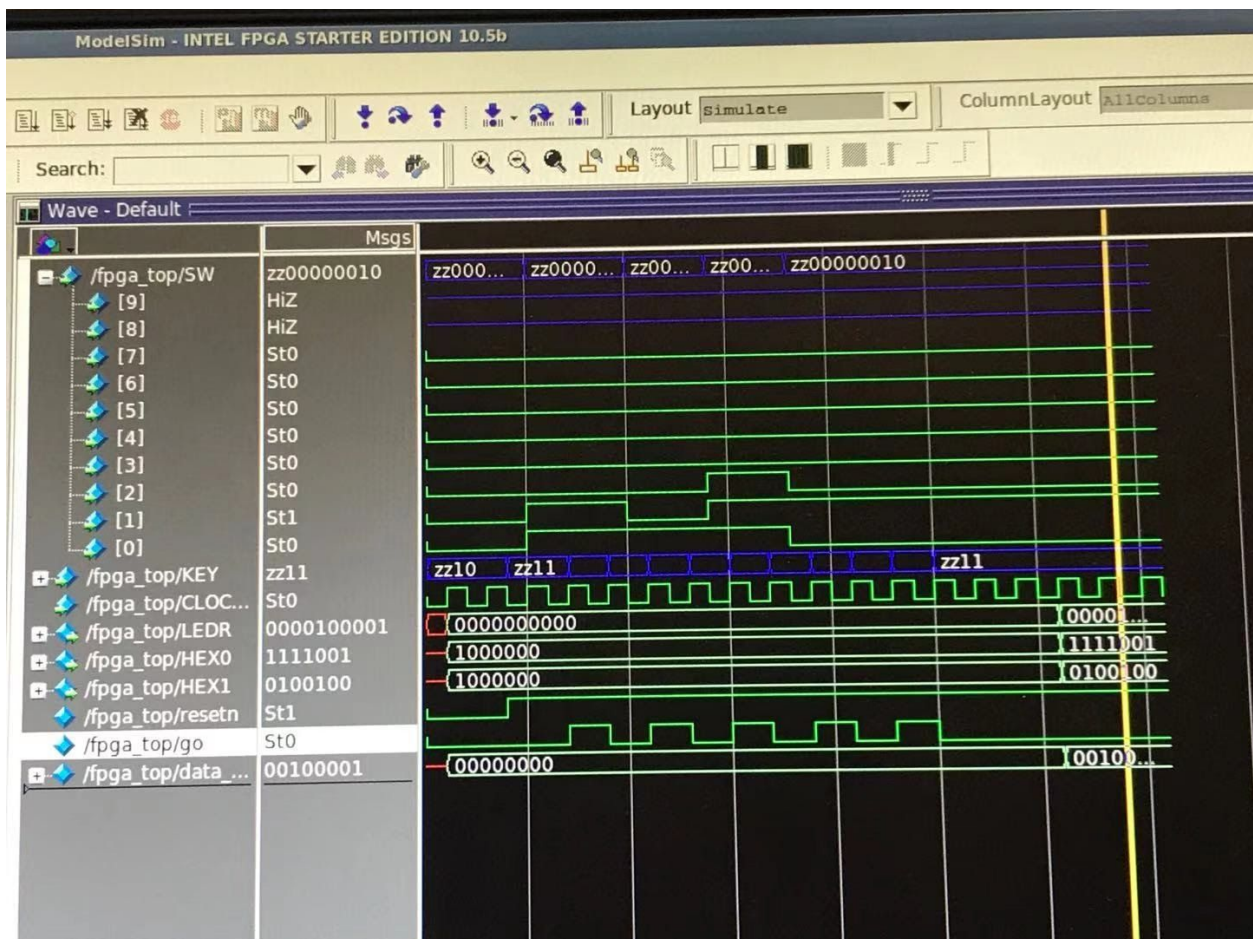
```

PART2 GENERATED FSM



3172

PART2 VSIM



1,2,3,4

$$3*(4)(4) + 2(4) + 1 = 73$$

$$3 \cdot 3^2 + 1(3) +$$