National University of Sciences and Technology (NUST)
School of Electrical Engineering and Computer Science

Department of Electrical Engineering and Computer Science

Faculty Member: Dr. Rehan Ahmed Dated: 8/03/2023

Semester: 6th Section: BEE 12C

EE-421: Digital System Design

Lab 8: Finite State Machines

Group Members

		PLO4-CLO3		PLO5 - CLO4	PLO8 - CLO5	PLO9 - CLO6
Name	**Reg. No**	**Viva / Quiz / Lab Performance**	**Analysis of data in Lab Report**	**Modern Tool Usage**	**Ethics and Safety**	**Individual and Teamwork**
		5 Marks	**5 Marks**	**5 Marks**	**5 Marks**	**5 Marks**
Danial Ahmad	331388					
Muhammad Umer	345834					
Tariq Umar	334943					

1 Table of Contents

2 Finite State Machines

2.1 Objectives

This is an exercise in using finite state machines.

2.2 Introduction

Finite state machines (FSMs) are abstract models of systems that have a finite number of states and can transition from one state to another based on some inputs and outputs. FSMs are widely used in computer science, engineering, and mathematics to design and analyze algorithms, circuits, protocols, and languages. The main objective of this lab report is to demonstrate our understanding and skills in using FSMs as a powerful tool for modeling and solving computational problems.

2.3 Software

Quartus Prime is a comprehensive design software developed by Intel Corporation for designing digital circuits using Field-Programmable Gate Arrays (FPGAs). It is a leading software platform in the field of digital design, offering a range of advanced tools and features that enable users to easily create, debug, and verify complex digital circuits. With Quartus Prime, users can benefit from a streamlined design flow that facilitates the creation of digital circuits from concept to implementation. It provides an intuitive graphical user interface that allows users to easily design, test, and debug their circuits. Additionally, Quartus Prime supports a variety of popular programming languages, making it a versatile platform for digital designers of all levels.

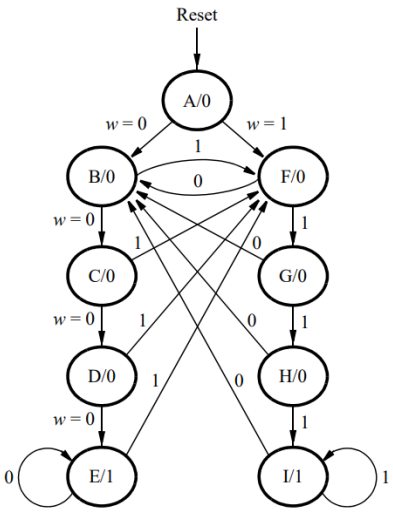
3 Lab Procedure

3.1 Part I

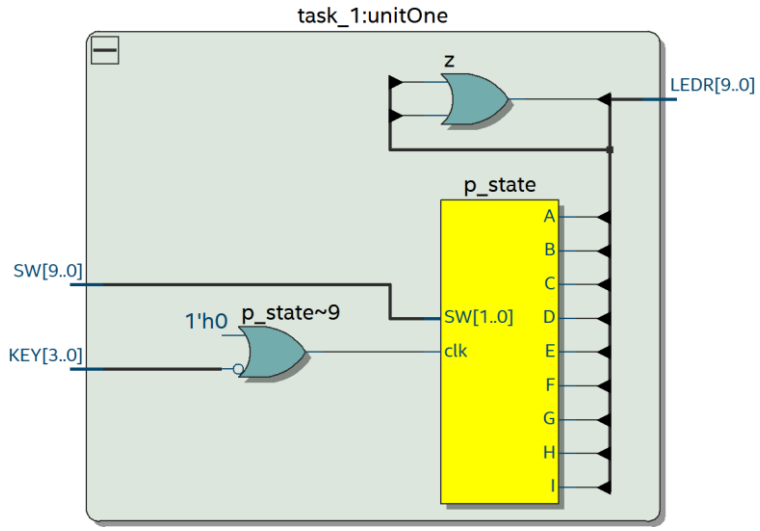
1. Create a new Quartus project for the FSM circuit.
2. Write a Verilog file that instantiates the nine flip-flops in the circuit and which specifies the logic expressions that drive the flip-flop input ports. Use only simple assign statements in your Verilog code to specify the logic feeding the flip-flops. Note that the one-hot code enables you to derive these expressions by inspection. Use the toggle switch SW0 as an active-low synchronous reset input for the FSM, use SW1 as the w input, and the pushbutton KEY0 as the clock input which is applied manually. Use the red light LEDR9 as the output z and assign the state flip-flop outputs to the red lights LEDR8 to LEDR0.
3. Include the Verilog file in your project and assign the pins on the FPGA to connect to the switches and the LEDs.
4. Simulate the behavior of your circuit.
5. Once you are confident that the circuit works properly because of your simulation, download the circuit into the FPGA chip. Test the functionality of your design by applying the input sequences and observing the output LEDs. Make sure that the FSM properly transitions between states as displayed on the red LEDs, and that it produces the correct output values on LEDR9.

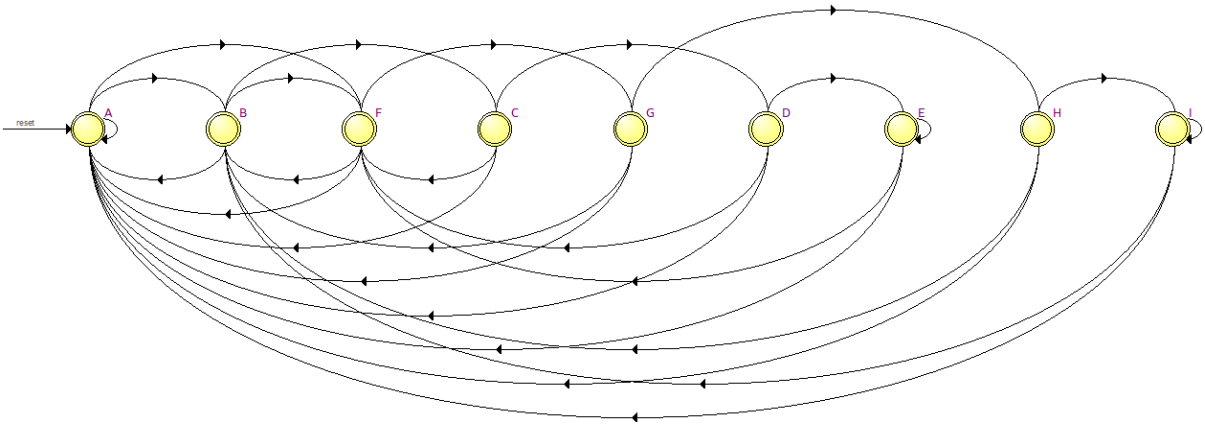
Name	State Code $y_8y_7y_6y_5y_4y_3y_2y_1y_0$
A	000000000
B	000000011
C	000000101
D	000001001
E	000010001
F	000100001
G	001000001
H	010000001
I	100000001

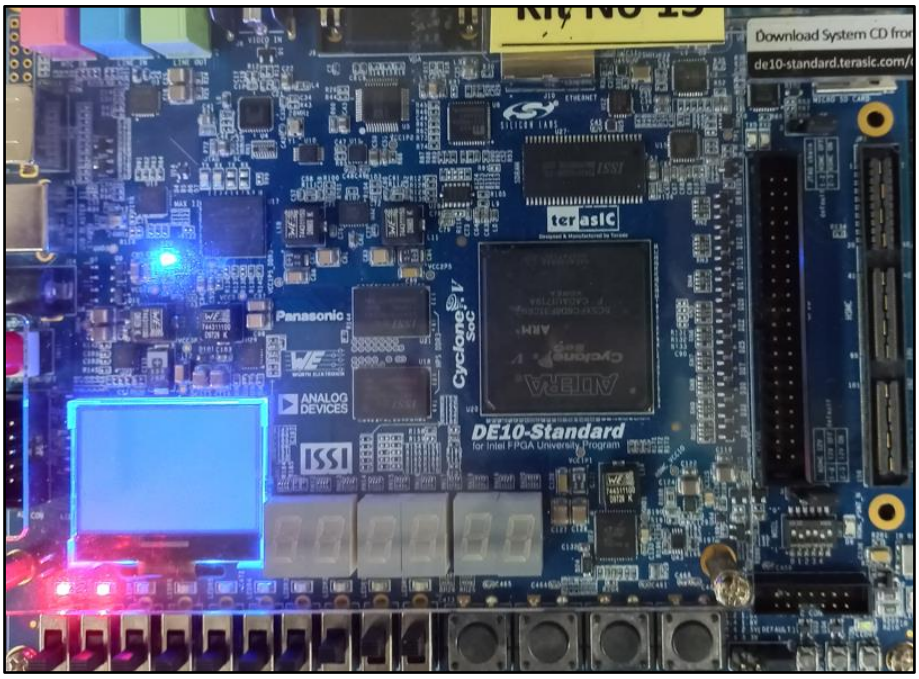
Name	State Code $y_8y_7y_6y_5y_4y_3y_2y_1y_0$
A	000000001
B	000000010
C	000000100
D	000001000
E	000010000
F	000100000
G	001000000
H	010000000
I	100000000



```
module task_1 (
    input [3:0] KEY,
    input [9:0] SW,
    output [9:0] LEDR
);

    wire clk = ~KEY[0];
    wire w = SW[1];
    wire z;
    wire reset = SW[0];

    parameter A = 9'b000000001, B = 9'b000000010, C = 9'b000000100,
              D = 9'b000001000, E = 9'b000010000, F = 9'b000100000,
              G = 9'b001000000, H = 9'b010000000, I = 9'b100000000;

//  for modified state codes
//  parameter A = 9'b000000000, B = 9'b000000011, C = 9'b000000101,
//            D = 9'b000001001, E = 9'b000010001, F = 9'b000100001,
//            G = 9'b001000001, H = 9'b010000001, I = 9'b100000001;

    reg [9:0] p_state;
    wire [9:0] n_state;

    assign n_state = (p_state == A) ? ((w) ? F : B) :
                     (p_state == B) ? ((w) ? F : C) :
                     (p_state == C) ? ((w) ? F : D) :
                     (p_state == D) ? ((w) ? F : E) :
                     (p_state == E) ? ((w) ? F : E) :
                     (p_state == F) ? ((w) ? G : B) :
                     (p_state == G) ? ((w) ? H : B) :
                     (p_state == H) ? ((w) ? I : B) :
                     (p_state == I) ? ((w) ? I : B) : A;

    always @(posedge clk) begin
        if (!reset) p_state <= A;
        else p_state <= n_state;
    end

    assign z = (p_state == E | p_state == I);
    assign LEDR[8:0] = p_state;
    assign LEDR[9] = z;

endmodule
```


task_1:unitOne
z
LEDR[9..0]
p_state
A
B
C
D
E
F
G
H
I
SW[9..0]
1'h0
p_state~9
SW[1..0]
KEY[3..0]
clk


reset
A
B
F
C
G
D
E
H
I


Download System CD fro
de10-standard.terasic.com/
terasIC
Panasonic
ANALOG DEVICES
ISSI
DE10-Standard
for Intel FPGA University Program

3.2 Part II

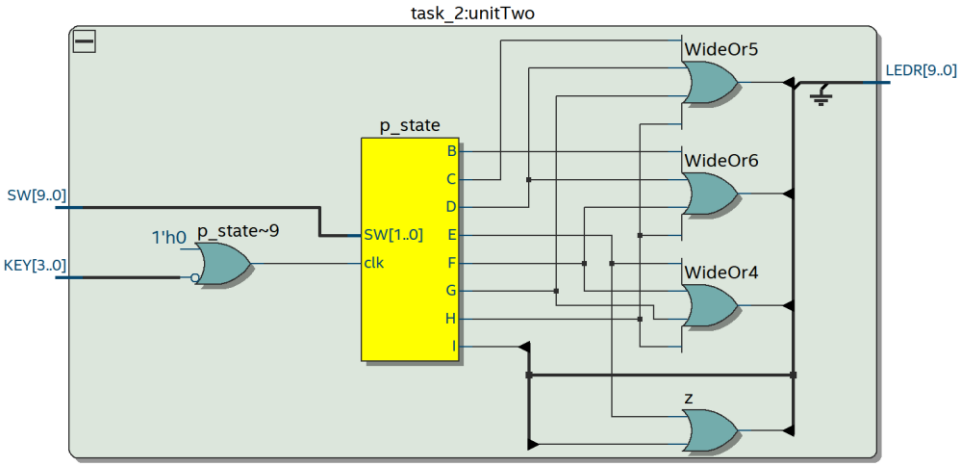
For this part you are to write another style of Verilog code for the FSM in Figure 2. In this version of the code, you should not manually derive the logic expressions needed for each state flip-flop. Instead, describe the state table for the FSM by using a Verilog case statement in an always block, and use another always block to instantiate the state flip-flops. You can use a third always block or simple assignment statements to specify the output z. To implement the FSM, use four state flip-flops y3, . . ., y0 and binary codes, as shown in Table 3.

Name	State Code $y_3y_2y_1y_0$
A	0000
B	0001
C	0010
D	0011
E	0100
F	0101
G	0110
H	0111
I	1000

1. Create a new project for the FSM.
2. Include in the project your Verilog file that uses the style of code in Figure 3. Use the same switches, pushbuttons, and lights that were used in Part I.
3. Before compiling your code it is necessary to explicitly tell the Synthesis tool in Quartus that you wish to have the finite state machine implemented using the state assignment specified in your Verilog code.
4. Compile your project. To examine the circuit produced by Quartus open the RTL Viewer tool. Double-click on the box shown in the circuit that represents the finite state machine and determine whether the state diagram that it shows properly corresponds to the one in Figure 2. To see the state codes used for your FSM, open the Compilation Report, select the Analysis and Synthesis section of the report, and click on State Machines.
5. Download the circuit into the FPGA chip and test its functionality.

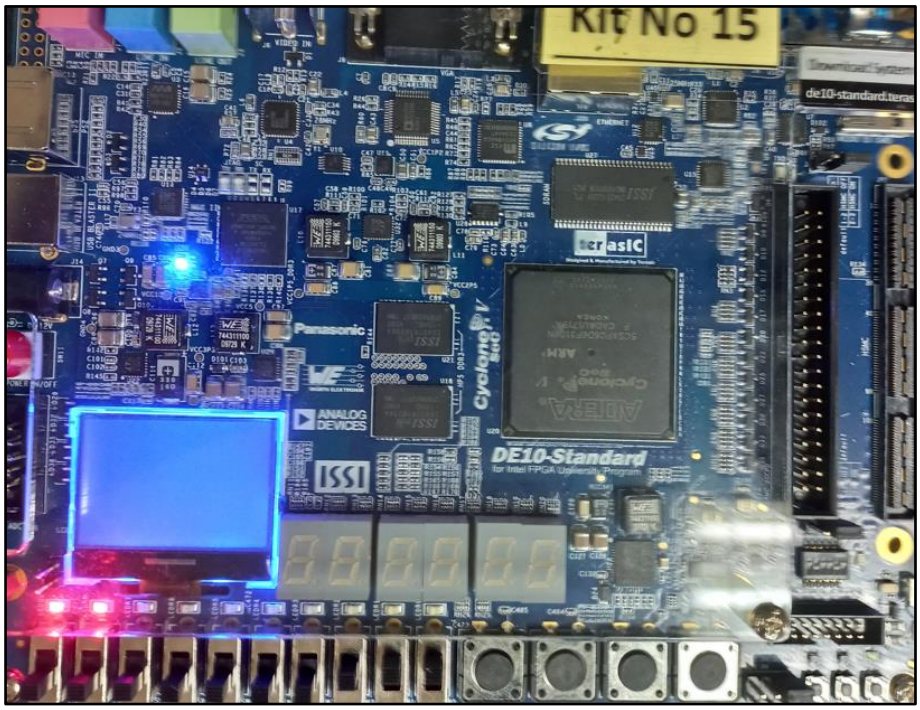
```
module task_2 (
    input [3:0] KEY,
    input [9:0] SW,
    output [9:0] LEDR
);

    wire clk = ~KEY[0];
    wire w = SW[1];
    wire z;
    wire reset = SW[0];

    parameter A = 4'b0000, B = 4'b0001, C = 4'b0010,
              D = 4'b0011, E = 4'b0100, F = 4'b0101,
```

```
            G = 4'b0110, H = 4'b0111, I = 4'b1000;

    reg [3:0] p_state;
    reg [3:0] n_state;

    always @(*) begin
            case (p_state)
            A: if (w) n_state = F; else n_state = B;
            B: if (w) n_state = F; else n_state = C;
            C: if (w) n_state = F; else n_state = D;
            D: if (w) n_state = F; else n_state = E;
            E: if (w) n_state = F; else n_state = E;
            F: if (w) n_state = G; else n_state = B;
            G: if (w) n_state = H; else n_state = B;
            H: if (w) n_state = I; else n_state = B;
            I: if (w) n_state = I; else n_state = B;
            default: n_state = A;
            endcase
    end

    always @(posedge clk) begin
        if (!reset) p_state <= A;
        else p_state <= n_state;
    end

    assign z = (p_state == E | p_state == I);
    assign LEDR[8:0] = p_state;
    assign LEDR[9] = z;

endmodule
```



	Name	p_state~5	p_state~4	p_state~3	p_state~2
1	p_state.A	0	0	0	0
2	p_state.B	0	0	0	1
3	p_state.C	0	0	1	0
4	p_state.D	0	0	1	1
5	p_state.E	0	1	0	0
6	p_state.F	0	1	0	1
7	p_state.G	0	1	1	0
8	p_state.H	0	1	1	1
9	p_state.I	1	0	0	0

	Name	p_state.H	p_state.G	p_state.F	p_state.E	p_state.D	p_state.C	p_state.B	p_state.A	p_state.I
1	p_state.A	0	0	0	0	0	0	0	0	0
2	p_state.B	0	0	0	0	0	0	1	1	0
3	p_state.C	0	0	0	0	0	1	0	1	0
4	p_state.D	0	0	0	0	1	0	0	1	0
5	p_state.E	0	0	0	1	0	0	0	1	0
6	p_state.F	0	0	1	0	0	0	0	1	0
7	p_state.G	0	1	0	0	0	0	0	1	0
8	p_state.H	1	0	0	0	0	0	0	1	0
9	p_state.I	0	0	0	0	0	0	0	1	1



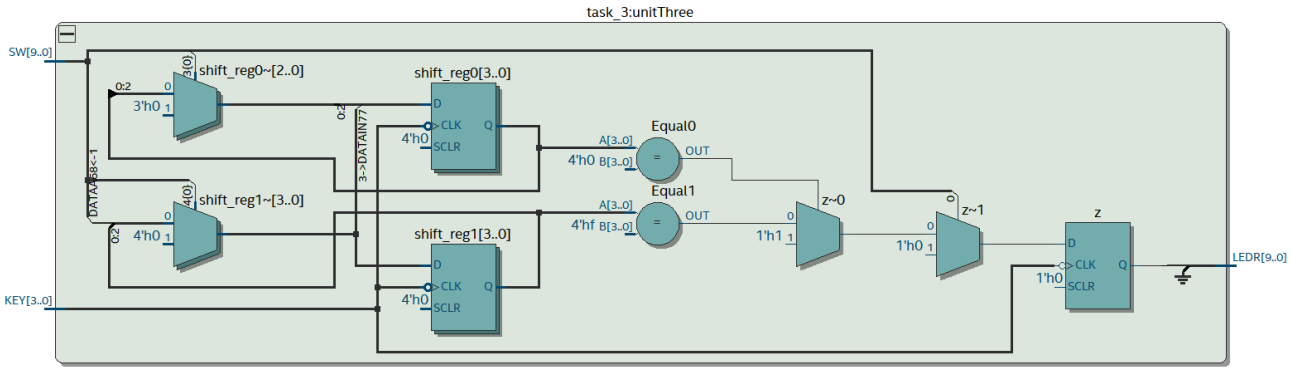
3.3 Part III

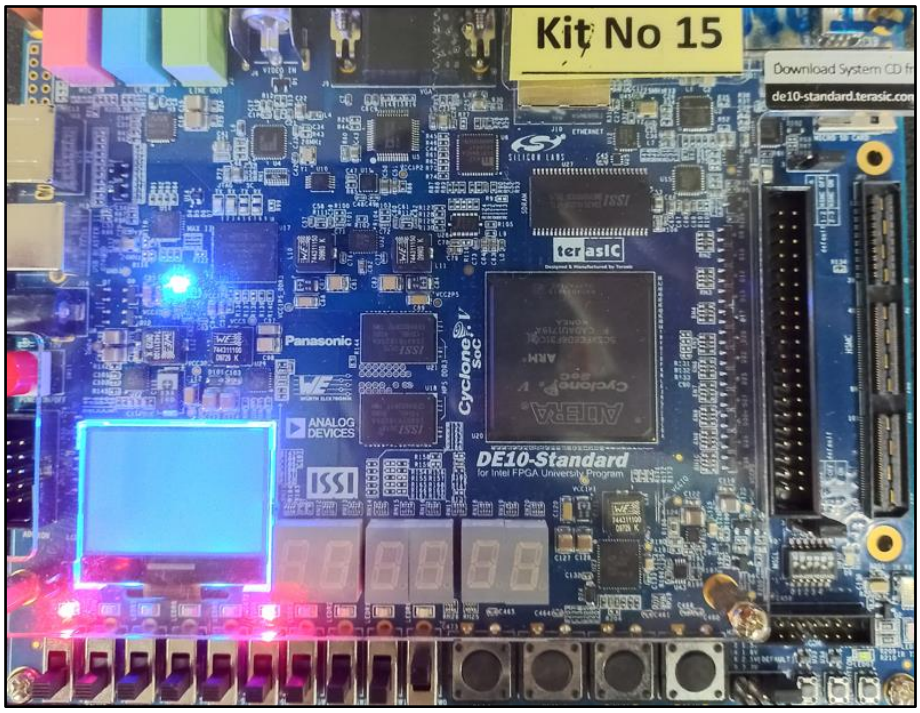
The sequence detector can be implemented in a straightforward manner using shift registers, instead of using the more formal approach described above. Create Verilog code that instantiates two 4-bit shift registers; one is for recognizing a sequence of four 0s, and the other for four 1s. Include the appropriate logic expressions in your design to produce the output z. Make a Quartus project for your design and implement the circuit on your DE series board. Use the switches and LEDs on the board in a similar way as you did for Parts I and II and observe the behavior of your shift registers and the

output z. Answer the following question: could you use just one 4-bit shift register, rather than two? Explain your answer.

```
module task_3 (
    input [3:0] KEY,
    input [9:0] SW,
    output [9:0] LEDR
);

    wire clk = ~KEY[0];
    wire w = SW[1];
    reg z;
    wire reset = SW[0];
    reg [3:0] shift_reg0, shift_reg1;

    always @(posedge clk) begin
        if (reset) begin
            shift_reg0 <= 4'b0;
            shift_reg1 <= 4'b0;
            z <= 1'b0;
        end else begin
            shift_reg0 <= {shift_reg0[2:0], data};
            shift_reg1 <= {shift_reg1[2:0], data};

            if (shift_reg0 == 4'b0000) begin
                z <= 1'b1;
            end else if (shift_reg1 == 4'b1111) begin
                z <= 1'b1;
            end else begin
                z <= 1'b0;
            end
        end
    end

    assign LEDR[9] = z;

endmodule
```

It is not possible to use just one 4-bit shift register to recognize both sequences of four 0s and four 1s because the shift register would need to recognize two different patterns at the same time. It would be difficult to create logic expressions that can differentiate between the two sequences and set the output z appropriately. Therefore, two separate shift registers are necessary for this task.





3.4 Part IV

An early method of telegraph communication was based on Morse code. This code uses patterns of short and long pulses to represent a message. Each letter is represented as a sequence of dots (a short pulse), and dashes (a long pulse). For example, the first eight letters of the alphabet have the following representation:

A	• —
B	— • • •
C	— • — •
D	— • •
E	•
F	• • — •
G	— — •
H	• • • •

Design and implement a circuit that takes as input one of the first eight letters of the alphabet and displays the Morse code for it on a red LED. Your circuit should use switches SW2−0 and pushbuttons KEY1−0 as inputs. When a user presses KEY1, the circuit should display the Morse code for a letter specified by SW2−0 (000 for A, 001 for B, etc.), using 0.5-second pulses to represent dots, and 1.5-second pulses to represent dashes. Pushbutton KEY0 should function as an asynchronous reset. A high-level schematic diagram of the circuit is shown in Figure 2.

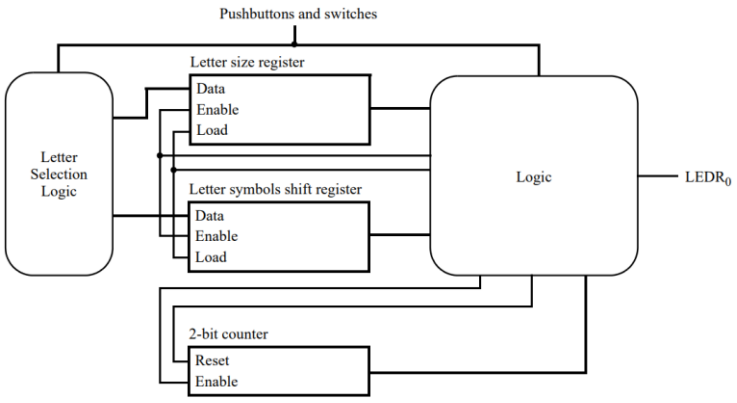
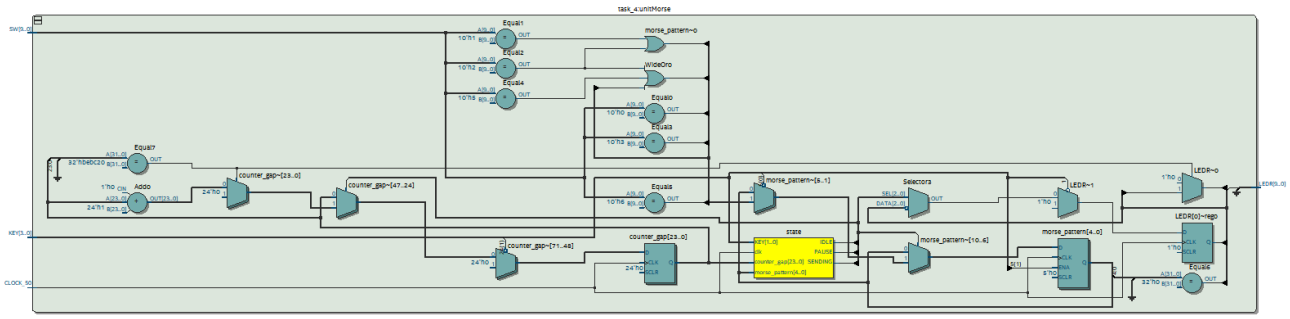


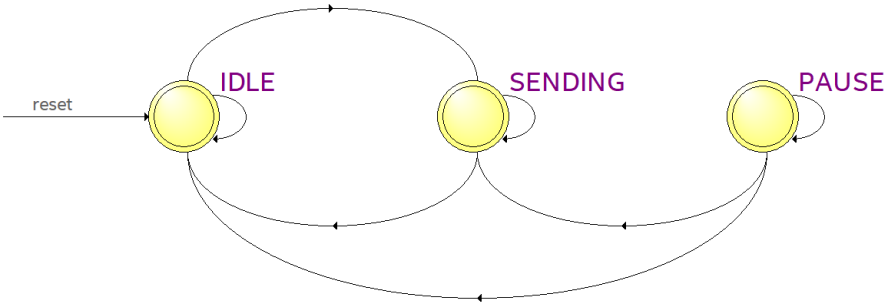
Figure 2: High-level schematic diagram of the circuit for part IV.

```
module task_4 (
    input CLOCK_50,
    input [9:0] SW,
    input [3:0] KEY,
    output reg [9:0] LEDR
);

    // Define the Morse code patterns for each letter from A to H.
    parameter [4:0] A = 5'b00001;
    parameter [4:0] B = 5'b10000;
    parameter [4:0] C = 5'b10100;
    parameter [4:0] D = 5'b1000;
    parameter [4:0] E = 5'b0;
    parameter [4:0] F = 5'b00100;
    parameter [4:0] G = 5'b110;
    parameter [4:0] H = 5'b00000;

    // Define the counter parameters for the dot, dash, and gap durations.
    parameter CNT_05S = 25000000;  // 0.5 second at 50 MHz
    parameter CNT_15S = 75000000;  // 1.5 seconds at 50 MHz
    parameter CNT_05S_GAP = 12500000;  // 0.5 second gap at 50 MHz

    // Define the state variables for the state machine.
    parameter IDLE = 2'b00;
    parameter SENDING = 2'b01;
    parameter PAUSE = 2'b10;
    reg [ 1:0] state = IDLE;

    // Define the variables for the counters and the Morse code pattern.
    reg [23:0] counter_dot = 0;
    reg [23:0] counter_dash = 0;
    reg [23:0] counter_gap = 0;
    reg [ 4:0] morse_pattern = 0;
```

```
always @(posedge CLOCK_50) begin
    // Check for reset button and reset the state machine and counters.
    if (KEY[1] == 1'b0) begin
        state <= IDLE;
        counter_dot <= 0;
        counter_dash <= 0;
        counter_gap <= 0;
        LEDR[0] <= 1'b0;
    end else begin
        case (state)
            IDLE: begin
                if (KEY[0] == 1'b0) begin
                    case (SW)
                        3'b000:  morse_pattern <= A;
                        3'b001:  morse_pattern <= B;
                        3'b010:  morse_pattern <= C;
                        3'b011:  morse_pattern <= D;
                        3'b100:  morse_pattern <= E;
                        3'b101:  morse_pattern <= F;
                        3'b110:  morse_pattern <= G;
                        3'b111:  morse_pattern <= H;
                        default: morse_pattern <= 5'b0;
                    endcase
                    state <= SENDING;
                end
            end
            SENDING: begin
                if (morse_pattern == 0) begin
                    state   <= IDLE;
                    LEDR[0] <= 1'b0;
                end else begin
                    case (morse_pattern[0])
                        1'b0: begin  // dot
                            if (counter_dot == CNT_05S) begin
                                counter_dot <= 0;
                                LEDR[0] <= 1'b0;
                                counter_gap <= counter_gap + 1;
                                if (counter_gap == CNT_05S_GAP) begin
                                    counter_gap   <= 0;
                                    morse_pattern <= morse_pattern >> 1;
                                end
                            end else begin
                                counter_dot <= counter_dot + 1;
                                LEDR[0] <= 1'b1;
                            end
                        end
                        1'b1: begin  // dash
                            if (counter_dash == CNT_15S) begin
                                counter_dash <= 0;
                                LEDR[0] <= 1'b0;
                                counter_gap <= counter_gap + 1;
                                if (counter_gap == CNT_05S_GAP) begin
                                    counter_gap   <= 0;
                                    morse_pattern <= morse_pattern >> 1;
                                end
                            end else begin
```

```
                                    counter_dash <= counter_dash + 1;
                                    LEDR[0] <= 1'b1;
                                end
                            end
                        endcase
                    end
                end
                PAUSE: begin
                    // Pause for the gap duration.
                    if (counter_gap == CNT_05S_GAP) begin
                        counter_gap <= 0;
                        state <= SENDING;
                    end else begin
                        counter_gap <= counter_gap + 1;
                        LEDR[0] <= 1'b0;
                    end
                end
            endcase
        end
    end

endmodule
```





4 Conclusion

In this lab, we have learned how to design and analyze FSMs to solve a variety of problems in computer science, engineering, and mathematics. We have seen how FSMs can be used to model and analyze algorithms, circuits, protocols, and languages. We have also learned how to implement FSMs using Verilog code and simulate them using software tools like Quartus. Through this process, we have gained a deeper understanding of the underlying principles of FSMs and how they can be used to solve real-world problems.