**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 |  |  |  |  |  |

**Table of Contents**

[2 Finite State Machines 3](#_Toc132133252)

[2.1 Objectives 3](#_Toc132133253)

[2.2 Introduction 3](#_Toc132133254)

[2.3 Software 3](#_Toc132133255)

[3 Lab Procedure 4](#_Toc132133256)

[3.1 Part I 4](#_Toc132133257)

[3.2 Part II 7](#_Toc132133258)

[3.3 Part III 9](#_Toc132133259)

[3.4 Part IV 11](#_Toc132133260)

[4 Conclusion 14](#_Toc132133261)

# Finite State Machines

## Objectives

This is an exercise in using finite state machines.

## 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.

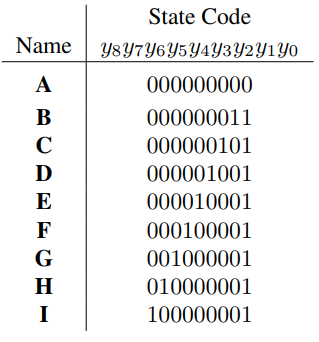
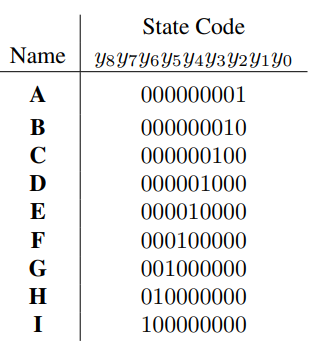
## Software

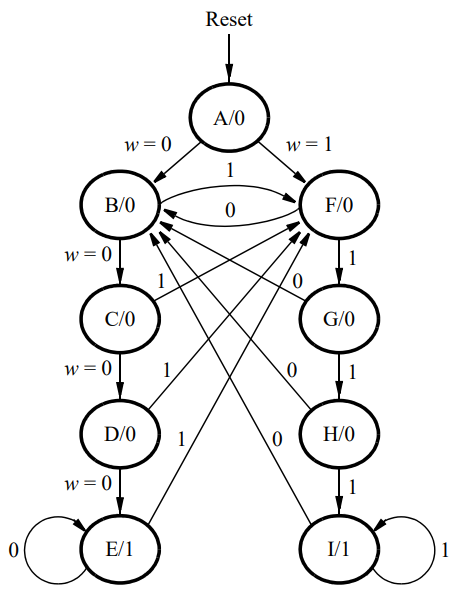
Quartus Prime is a comprehensive design software developed by Intel Corporation for designing digital circuits using Field-Programmable Gate Arrays (FPGAs). It is a leading software platform in the field of digital design, offering a range of advanced tools and features that enable users to easily create, 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.

# Lab Procedure

## 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.



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;

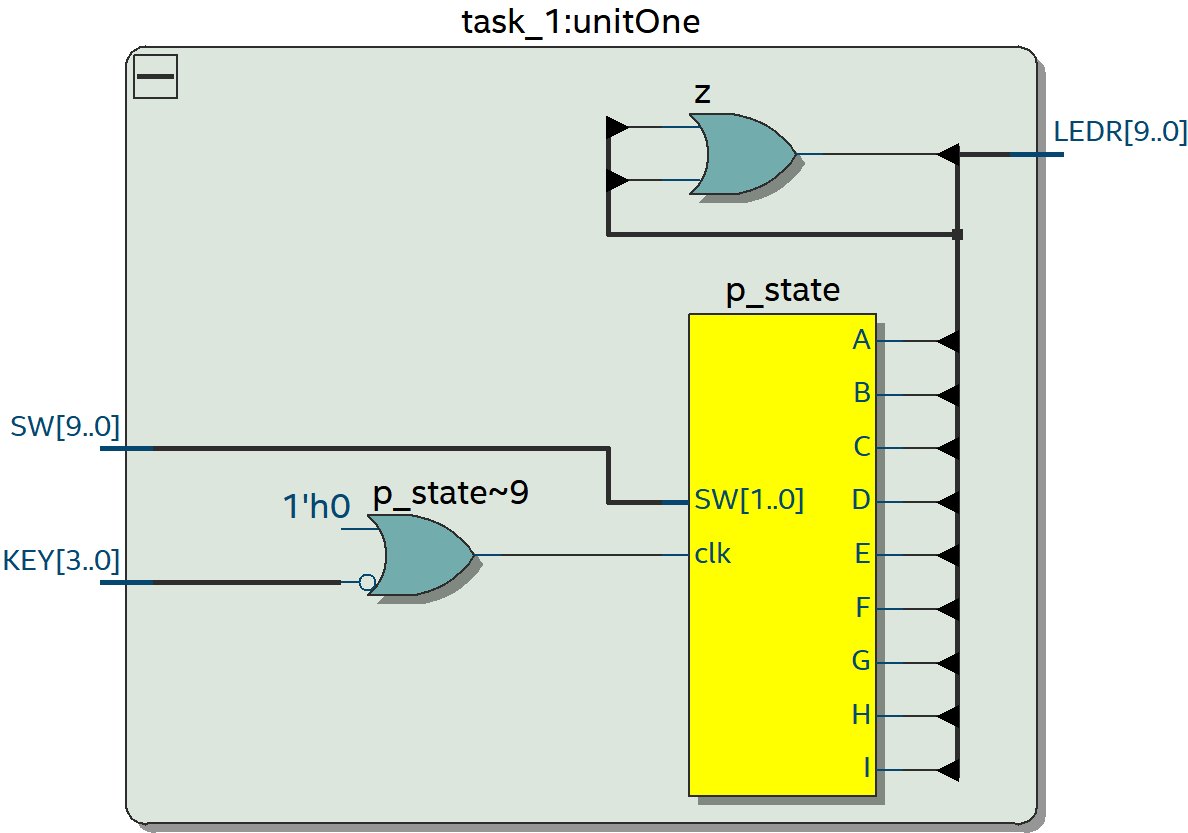
    end

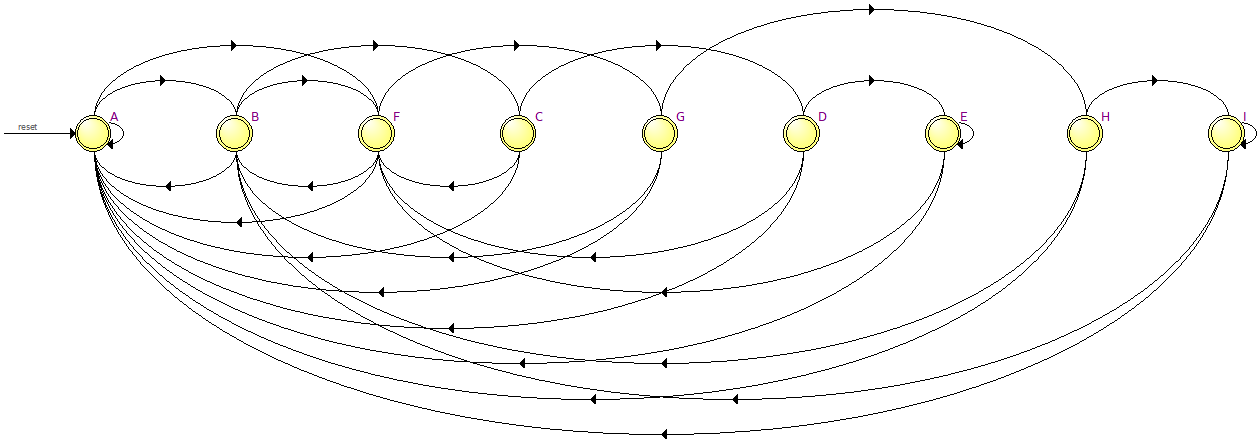
    assign z = (p\_state == E | p\_state == I);

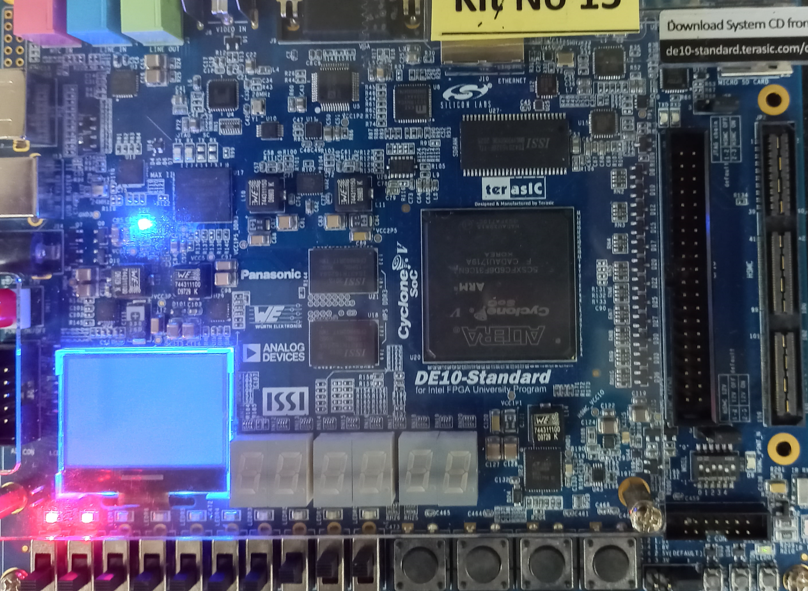
    assign LEDR[8:0] = p\_state;

    assign LEDR[9] = z;

endmodule

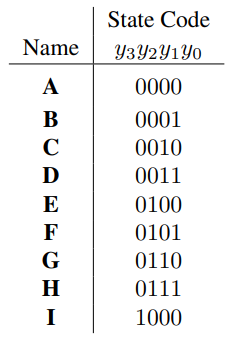






## Part II

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.



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;

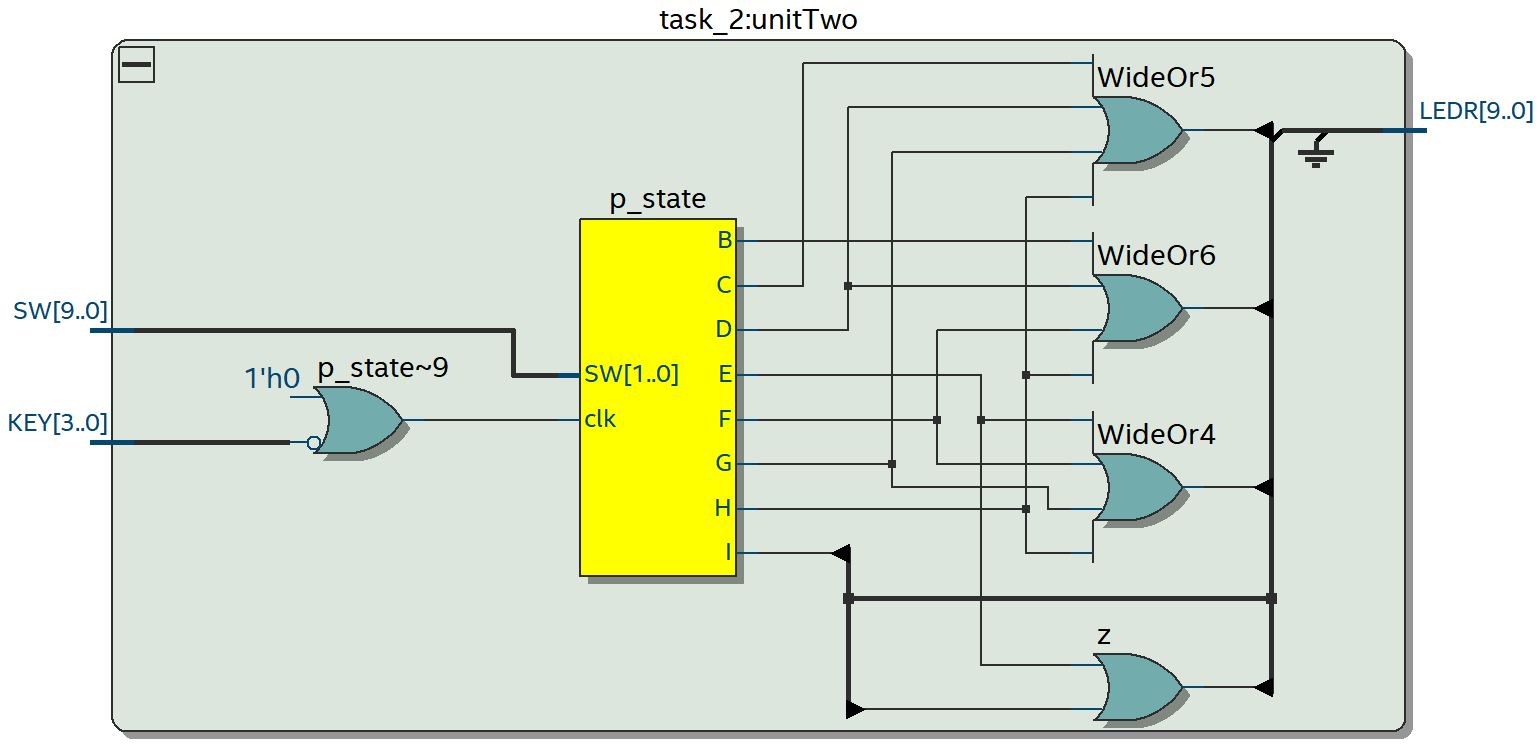
    end

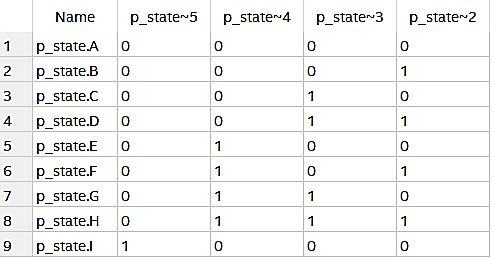
    assign z = (p\_state == E | p\_state == I);

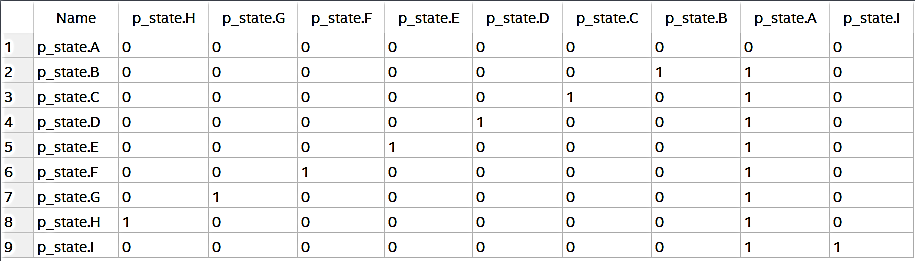
    assign LEDR[8:0] = p\_state;

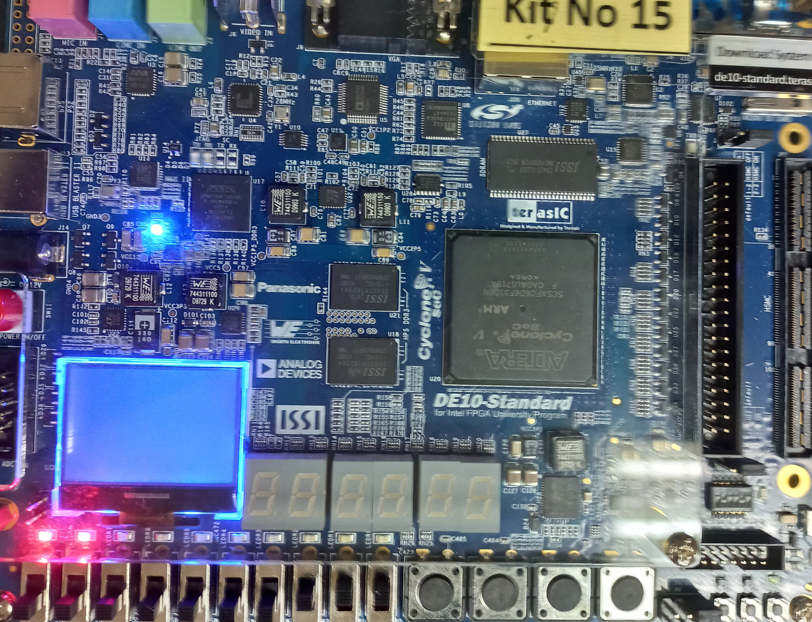
    assign LEDR[9] = z;

endmodule









## 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

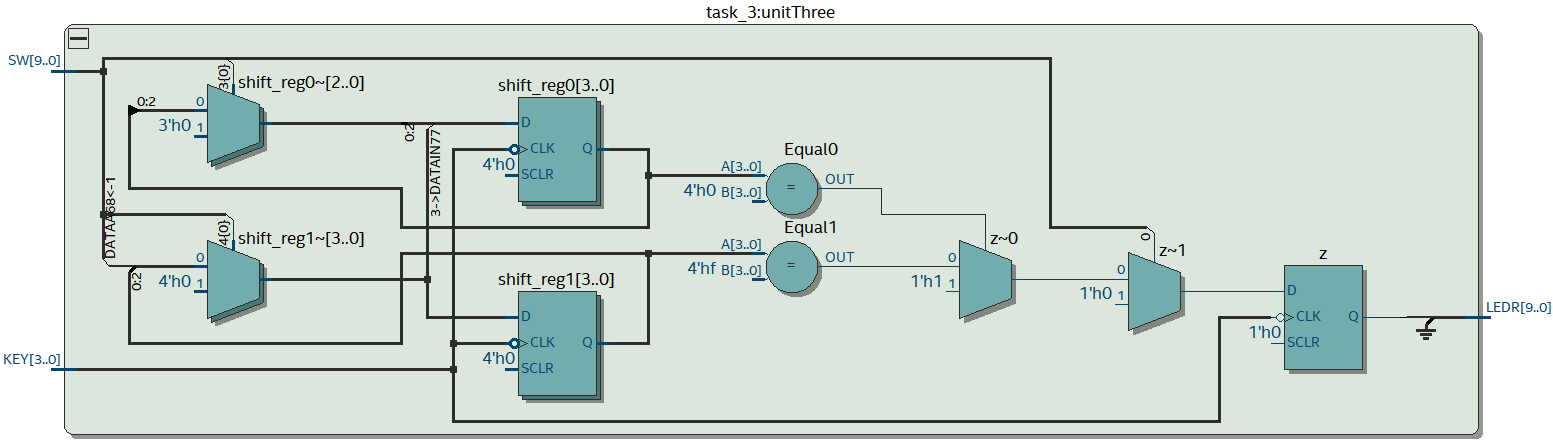
        end

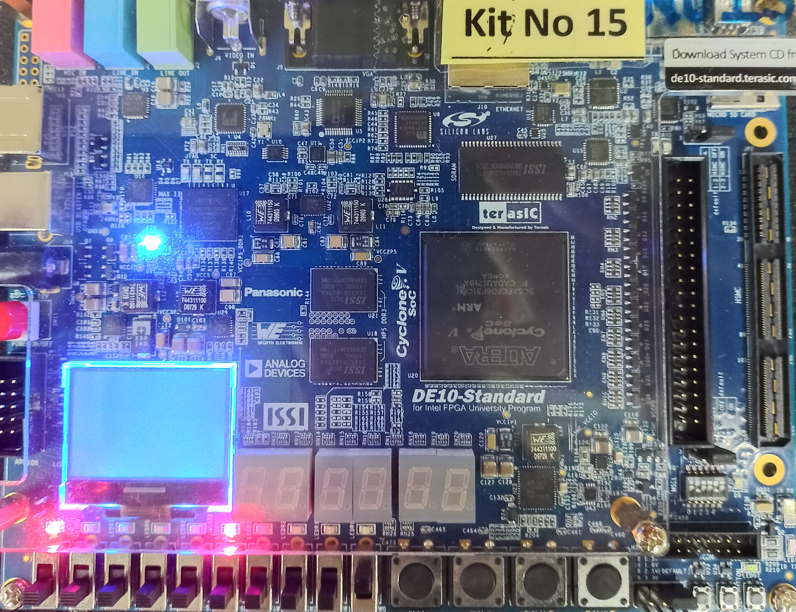
    end

    assign LEDR[9] = z;

endmodule

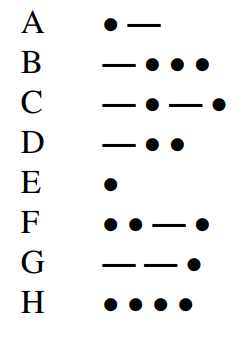
It is not possible to use just one 4-bit shift register to recognize both sequences of four 0s and four 1s because the shift register would need to recognize two different patterns at the same time. It would be difficult to create logic expressions that can differentiate between the two sequences and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



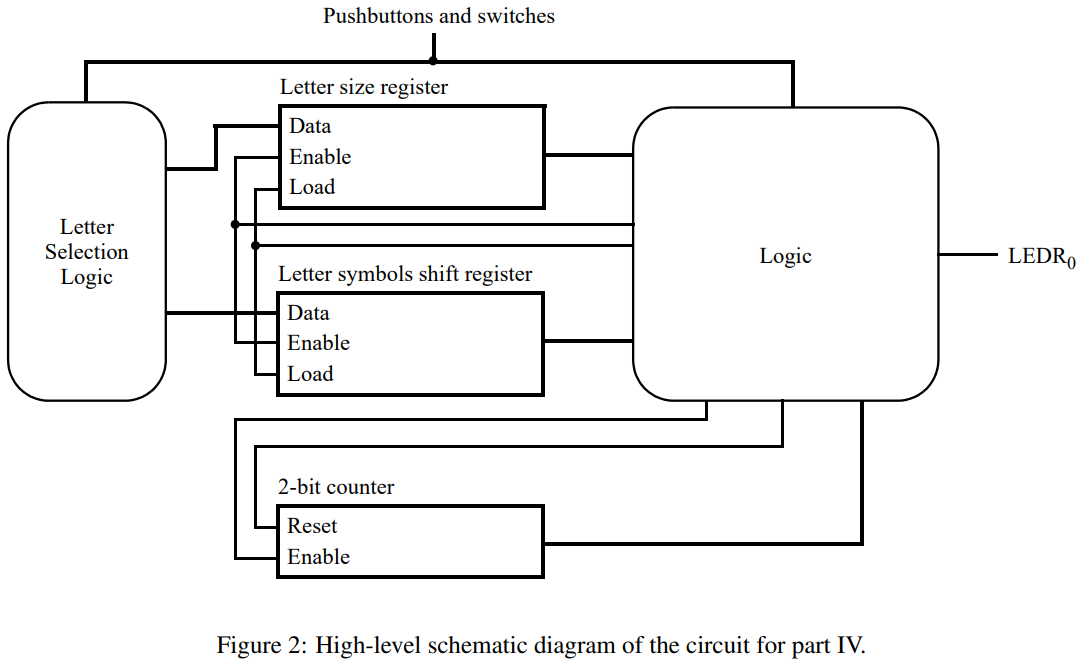


## Part IV

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:



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.



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

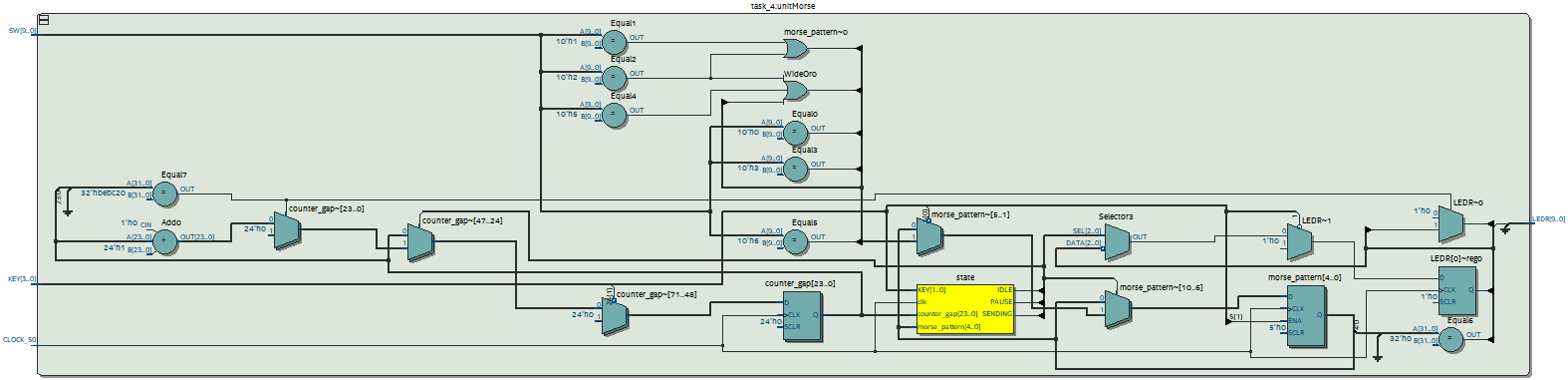
                end

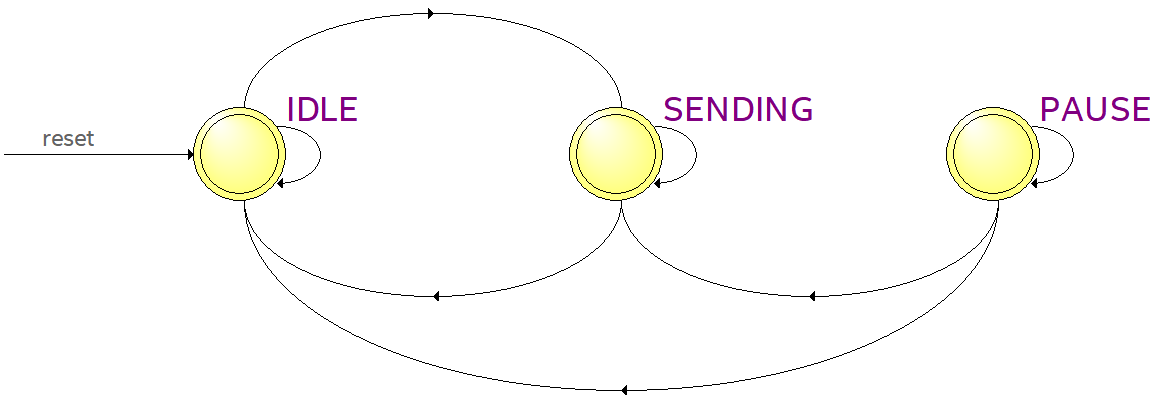
            endcase

        end

    end

endmodule





# 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.