CRN: 22840

CET3136 – Logic Devices Programming

Spring 2025

Experiment 10

Finite State Machines

Performed By:

Anthony Paul Sevarino

Submitted to:

Prof. Ashley Evans

Department of Electrical & Computer Engineering Technology (ECET)

School of Engineering, Technology, and Advanced Manufacturing (ETAM)

Valencia College

Date Submitted

04/03/2025

Introduction

The objective of this experiment tis to design systems using distinct finite state machine approaches, namely the *Moore* and *Mealy* FSMs. A finite state machine is a method of system behavior description which contains unique states that change throughout the course of a program. Both aforementioned finite state machines will be observed and analyzed in this study, and the subtle differences between them which drastically change their efficiency and behavior assessed. The MAX 10 Intel development board will be used to demonstrate this subject, taking advantage of it's on-board switches, LEDs, and push buttons. For further analysis, functional simulations will be generated for each component of the lab in order to visually assess the behavior of each system, for analysis and confirmation of proper function.

LIST OF EQUIPMENT/PARTS/COMPONENTS/SOFTWARE

- DE10-Lite MAX-10 Dev Board
- USB 2.0 Type B Cable
- Windows Desktop
- Quartus FPGA Design Software

PROCEDURE / DISCUSSION

Part A – Finite State Machines

Before constructing either of the two required finite state machines, a fundamental understanding as to their function is necessary for ensuring proper design. First, a finite state machine is a digital sequential system, of which it's outputs are functions of current inputs, and previous inputs. Due to this fundamental nature of finite state machine's requirement of knowledge of previous inputs, the concept of *memory* must be addressed. A state register can be created which allows for the storage of these past inputs, while the system continues running. A schematic of an example of this is shown to the right.

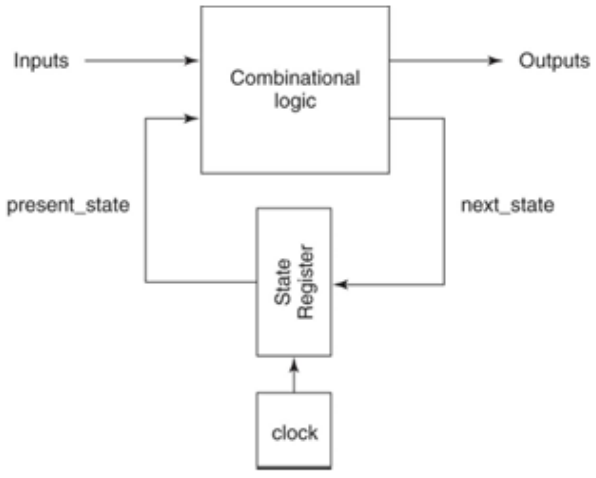
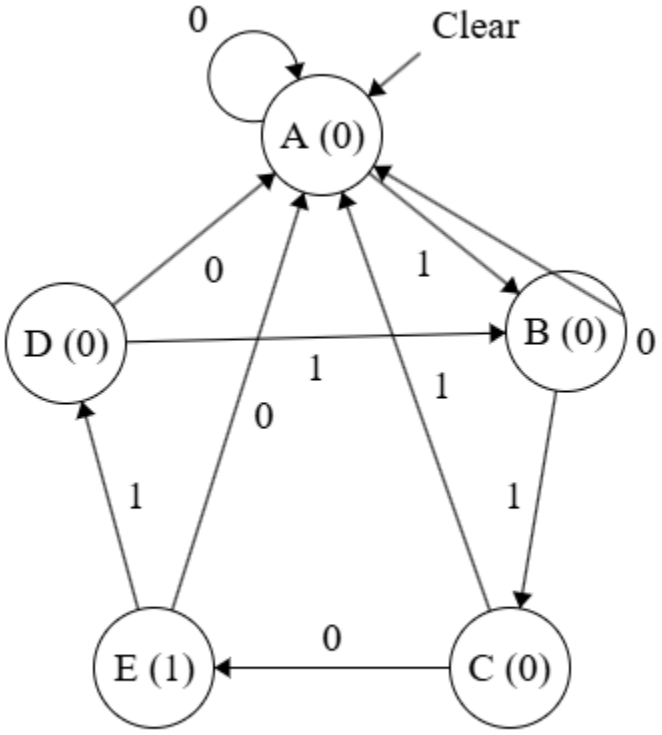
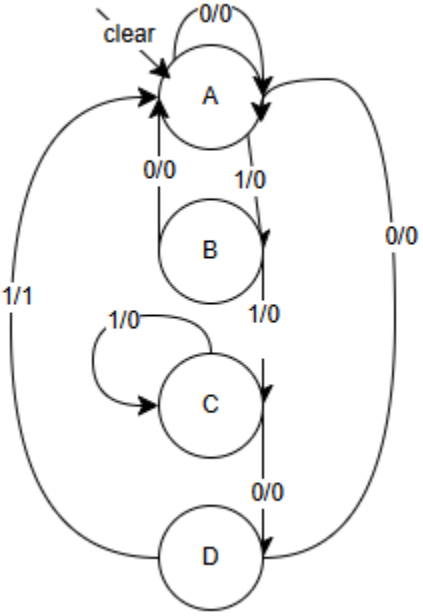
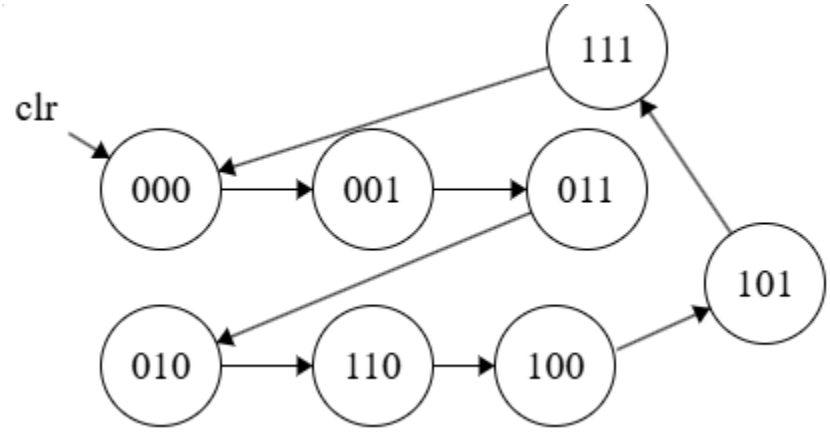


Figure 1 - Finite State Machine Example

Part A of the experiment demands the design of a *Mealy* finite state machine, which is known for it's reliance on both it's present state and it's inputs in order to generate outputs. Part B, however, requires the design of a *Moore* FSM, which is a more linear sequential system and is conceptually simpler than a *Mealy* FMS, due to its restriction wherein outputs are a function only of the present state. Below are constructed state diagrams for Parts A and B (Part A requests Moor and mealy diagrams).







Part B – The Mealy FSM

This part of the lab requires the design of a *Mealy* finite state machine for a system which awaits a unique binary input, in order to enable an LED. The output should not "wrap around", meaning that the LED should be disabled as soon as the next character is entered.

Code Explanation Part B

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3
4   entity sequencedetector is
5   port  (
6         clk, reset, x : in std_logic;
7         y : out std_logic
8         );
9   end sequencedetector;
10
11  architecture mealy of sequencedetector is
12
13  type state is(state_a, state_b, state_c, state_d, state_temp);
14  signal present_state, next_state : state;
15
16  begin
17
18     state_reg : process(clk, reset)     --initializing register
19     begin
20        if rising_edge(clk) then
21           if reset = '1' then           --reset to state A if reset thrown, otherwise continue
22              present_state <= state_a;
23           else
24              present_state <= next_state;
25           end if;
26        end if;
27     end process;
28
29     combinational_logic : process(present_state, x)    --handle combinational logic (mealy is based on present_state and inputs)
30     begin
31        case present_state is
32           when state_a =>
33              y <= '0';          --instantiating y as zero at beginning of state A.
34              if x = '1' then
35                 next_state <= state_b;     --switch states depending on correct code order
36              else
37                 next_state <= state_a;
38              end if;
39           when state_b =>
40              if x = '1' then
41                 next_state <= state_c;
42              else
43                 next_state <= state_a;
44              end if;
45           when state_c =>
46              if x = '1' then
47                 next_state <= state_c;
48              else
49                 next_state <= state_d;
50              end if;
51           when state_d =>
52              if x ='1' then
53                 next_state <= state_temp;     --go to buffer state for output throw
54              else
55                 next_state <= state_a;
56              end if;
57           when state_temp =>
58              y <= '1';
59              next_state <= state_a;
60
61           end case;
62     end process;
63
64  end mealy;
```

Figure 2 - Completed Code for Part B

```
library ieee;
use ieee.std_logic_1164.all;

entity sequencedetector is
port      (
                  clk, reset, x : in std_logic;
                  y : out std_logic
                  );
end sequencedetector;

architecture mealy of sequencedetector is

type state is(state_a, state_b, state_c, state_d, state_temp);
signal present_state, next_state : state;

begin

        state_reg : process(clk, reset)              --initializing register
        begin
                  if rising_edge(clk) then
                           if reset = '1' then                        --reset to state A if reset thrown, otherwise
```

```
continue
                        present_state <= state_a;
                    else
                        present_state <= next_state;
                    end if;
                end if;
        end process;

        combinational_logic : process(present_state, x)          --handle combinational logic (mealy is based
on present_state and inputs)
        begin
                case present_state is
                    when state_a =>
                        y <= '0';                    --instantiating y as zero at beginning of state A.
                        if x = '1' then
                            next_state <= state_b;          --switch states depending on correct
code order
                        else
                            next_state <= state_a;
                        end if;
                    when state_b =>
                        if x = '1' then
                            next_state <= state_c;
                        else
                            next_state <= state_a;
                        end if;
                    when state_c =>
                        if x = '1' then
                            next_state <= state_c;
                        else
                            next_state <= state_d;
                        end if;
                    when state_d =>
                        if x ='1' then
                            next_state <= state_temp;       --go to buffer state for output throw
                        else
                            next_state <= state_a;
                        end if;
                    when state_temp =>
                        y <= '1';
                        next_state <= state_a;

                    end case;
        end process;

end mealy;
```

The entity of part B takes three inputs, *clk*, *reset*, and *x*. These represent the clock, the reset switch, and the data input for the system. A single output is present, *y*, for the data output of the system.

The architecture for the system begins with defining a type, as well as two signals. The type is defined as *state*, and represents the potential different states of the *Mealy* system. The two signals are placeholders for the present, and next states, defined as type *state*. The first of two processes initializes the state register which will be responsible for storing program memory, and

transitioning from the present state, to the next state. This system operates as a positive-edge triggered synchronous reset circuit, and thus operates on the rising edge of the clock. The second process, called *combinational_logic*, which also determines which state *next_state* holds depending on inputs, as well as eventually sends output values to ds depending on inputs, as well as eventually sends output values to *y*. Recall that in a *Mealy* FSM, the system's outputs are dependent on both the present state, as well as it's inputs.

Below is the block diagram and pin assignments for this part of the experiment:

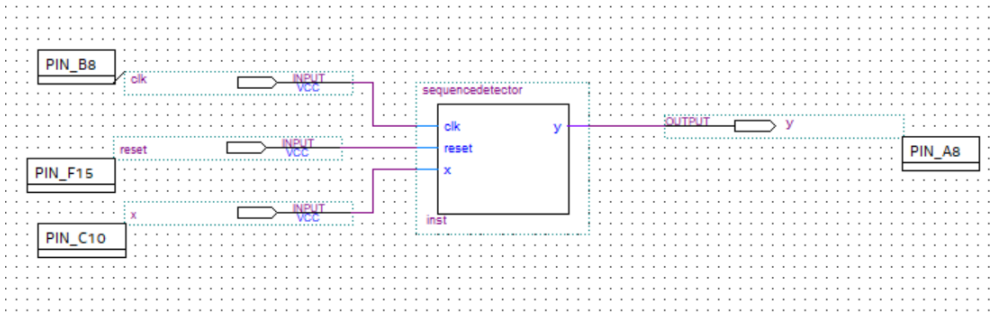


Figure 3 - Part B Block Diagram

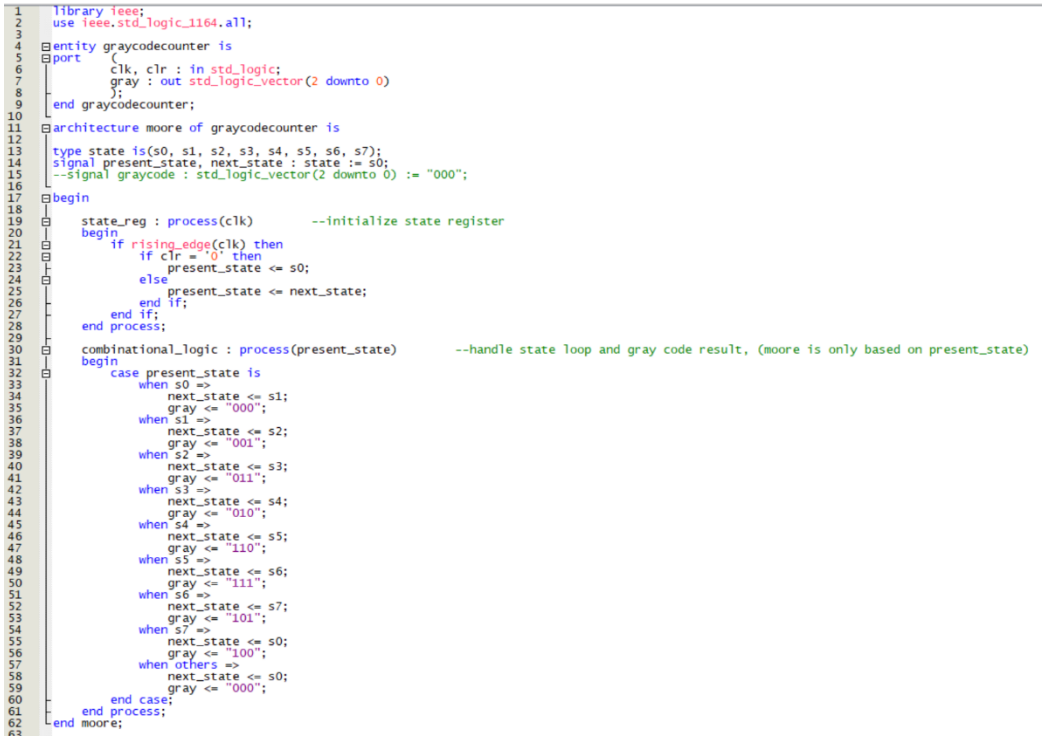
in clk	Input	PIN_B8	7	B7_N0	PIN_B8	3.3-V LVTTL
in reset	Input	PIN_F15	7	B7_N0	PIN_F15	3.3-V LVTTL
in x	Input	PIN_C10	7	B7_N0	PIN_C10	3.3-V LVTTL
out y	Output	PIN_A8	7	B7_N0	PIN_A8	3.3-V LVTTL

Figure 4 - Part B Pin Assignments

Part C – The Moore FSM

This step of the experiment requires the design of a *Moore* finite state machine, which acts as a sequential gray code counter. In a *Moore* FSM, the outputs are only a function of the present state, thus, each transition is unconditional. The LEDs on the board should act as the gray code values, while a logic-low synchronous reset operates along the positive-edge triggered circuit.

Code Explanation Part C



```
library ieee;
use ieee.std_logic_1164.all;

entity graycodecounter is
port    (
        clk, clr : in std_logic;
        gray : out std_logic_vector(2 downto 0)
        );
end graycodecounter;

architecture moore of graycodecounter is

type state is(s0, s1, s2, s3, s4, s5, s6, s7);
signal present_state, next_state : state := s0;
--signal graycode : std_logic_vector(2 downto 0) := "000";

begin

    state_reg : process(clk)        --initialize state register
    begin
        if rising_edge(clk) then
            if clr = '0' then
                present_state <= s0;
            else
                present_state <= next_state;
            end if;
        end if;
    end process;

    combinational_logic : process(present_state)        --handle state loop and gray code result, (moore is only based on present_state)
    begin
        case present_state is
            when s0 =>
                next_state <= s1;
                gray <= "000";
            when s1 =>
                next_state <= s2;
                gray <= "001";
            when s2 =>
                next_state <= s3;
                gray <= "011";
            when s3 =>
                next_state <= s4;
                gray <= "010";
            when s4 =>
                next_state <= s5;
                gray <= "110";
            when s5 =>
                next_state <= s6;
                gray <= "111";
            when s6 =>
                next_state <= s7;
                gray <= "101";
            when s7 =>
                next_state <= s0;
                gray <= "100";
            when others =>
                next_state <= s0;
                gray <= "000";
        end case;
    end process;
end moore;
```

Figure 5 - Completed Architecture Block for Section C of the Experiment

---

```
library ieee;
use ieee.std_logic_1164.all;

entity graycodecounter is
port    (
      clk, clr : in std_logic;
      gray : out std_logic_vector(2 downto 0)
      );
end graycodecounter;

architecture moore of graycodecounter is

type state is(s0, s1, s2, s3, s4, s5, s6, s7);
signal present_state, next_state : state := s0;
--signal graycode : std_logic_vector(2 downto 0) := "000";

begin

   state_reg : process(clk)       --initialize state register
   begin
      if rising_edge(clk) then
         if clr = '0' then
            present_state <= s0;
         else
            present_state <= next_state;
         end if;
      end if;
   end process;
```

```
   combinational_logic : process(present_state)       --handle state loop and gray code result, (moore is only based
on present_state)
   begin
      case present_state is
         when s0 =>
            next_state <= s1;
            gray <= "000";
         when s1 =>
            next_state <= s2;
            gray <= "001";
         when s2 =>
            next_state <= s3;
            gray <= "011";
         when s3 =>
            next_state <= s4;
            gray <= "010";
         when s4 =>
            next_state <= s5;
            gray <= "110";
         when s5 =>
            next_state <= s6;
            gray <= "111";
         when s6 =>
            next_state <= s7;
            gray <= "101";
         when s7 =>
            next_state <= s0;
            gray <= "100";
         when others =>
            next_state <= s0;
            gray <= "000";
      end case;
   end process;
end moore;
```

The entity portion of the code takes two inputs, *clk*, and *clr*, representing the logic-low synchronous reset, and the clock for the circuit. A single output called *gray* acts as the three digit standard logic vector which stores the value of the gray code.

The architecture of the system begins with defining a type called *state*, of which there are 8 (s0 – s7). Then, two signals are created to represent both the present and the current state. There are two processes, one responsible for handling basic state change and initialization of the state register, and the other for handling the combinational logic of the circuit. The first process, titled *state_reg* defines the positive-edge triggered, synchronous reset behavior of the circuit. The second, called *combinational_logic*, uses a case statement to set a value to *gray* based on the current state. This case statement also tells the program which state *next_state* should adopt.

Below is the block diagram for the experiment.

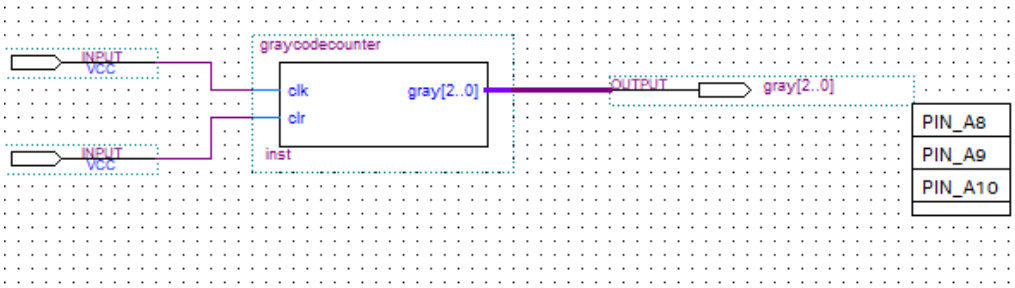


Figure 6 - Part C Block Diagram

in	clk	Input	PIN_A7	7	B7_N0	PIN_A7	3.3 V Sc... Trigger
in	clr	Input	PIN_F15	7	B7_N0	PIN_F15	3.3-V LVTTL
out	gray[2]	Output	PIN_A10	7	B7_N0	PIN_A10	3.3-V LVTTL
out	gray[1]	Output	PIN_A9	7	B7_N0	PIN_A9	3.3-V LVTTL
out	gray[0]	Output	PIN_A8	7	B7_N0	PIN_A8	3.3-V LVTTL

Figure 7 - Pin Assignments for Part C

There were many issues with the behavior of the LED, as it was both inconsistent and would get stuck between cases. The root cause for this was eventually determined to be an incorrectly stated I/O standard on the push button case. 3.3-V LVTTL was given, however that particular pin operates on the 3.3V Schmitt Trigger standard, and thus this change was required.

VALIDATION OF DATA

Functional Simulation Analysis

Below is the functional simulation analysis results for Parts B and C, labelled accordingly (note that the states are not present within the functional simulations, just the inputs and outputs of the system.)

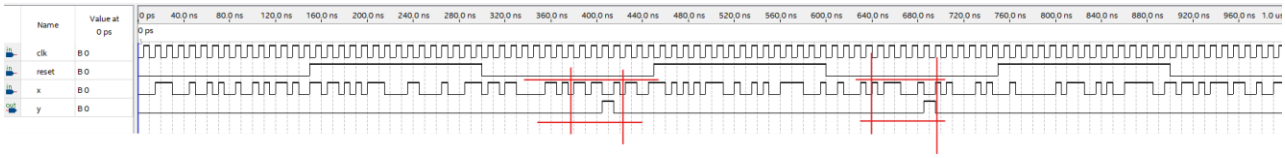


Figure 8 - Part B Functional Simulation

Note here the red boxes which designate the two states where y was determined to be 1. In order to simulate this, the *random values* modifier was applied to the *x* input, and this led to two cases in the visible simulation where there was a “1101” combination, while reset was not thrown.

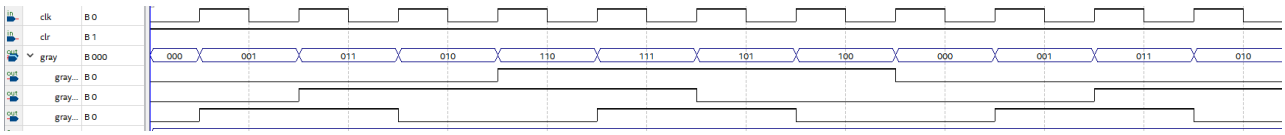


Figure 9 - Part C Functional Simulation (normal program flow)

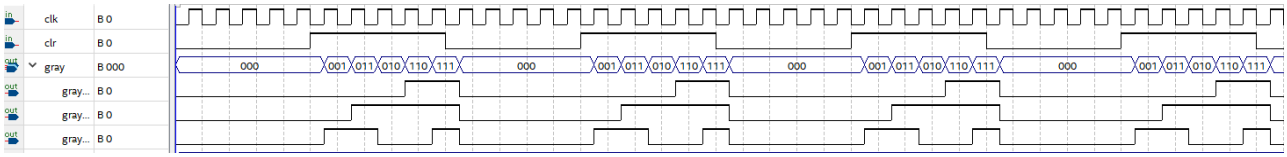


Figure 10 - Part C Functional Simulation (reset throws)

CONCLUSION

This experiment was conducted successfully and efficiently demonstrated two different examples of finite state machines within practical implementations. The first of which followed the design of a *Mealy* FSM, which is known for it's reliance on both it's system's present state and inputs in order to generate outputs. The second program followed a *Moore* FSM approach, which is a conceptually simpler design, more linearly sequential, and relies solely on its present state to derive outputs. Fundamental knowledge of finite state machines, the operations of the Max10DE-Lite Intel development boards, and functional simulations were paramount in the conduct of this lab. The functional simulations created not only served as a method of visualizing behavior of the circuits, but also allowing for confirmation of proper operation of the circuits. If this lab were to be conducted again, it would prove beneficial to analyze other types of finite state machines, such as the Medvedev FSM where the output is related directly to the states, rather than inputs.

REFERENCES

[1] Professor Ashley Evans, *Logic Devices Programming Lab Manual*, 1st ed. Orlando, FL. Valencia College, 2025.