Ellis Chen
09/11/2022

ECPE 174 – Lab 1 Report

Problem Summary:

The lab objective is to design an LED controller that controls different lighting patterns of an LED strip. The controller must control a strip of four LEDs and these LEDs must produce five different patterns:

- All four LEDs are off
- All four LEDs are lit constantly
- All four LEDs flash on and off repeatedly
- Only one LED is on at a time as it travels along the strip (chasing) from right to left
- Alternating between two states
    - State 1: Only two non-adjacent LEDs are active (flashing on) at a time while the other two LEDs are inactive (flashing off)
    - State 2: The controller reverses the pattern. Active LEDs switch to inactive LEDs. Likewise, Inactive LEDs switch to active LEDs

The controller uses three input switches to control the state of the LED strip. The controller uses two input switches to select between the lighting patterns. The third input switch turns the controller on or off. The system runs on a 2 Hz clock, which means the LEDs should respond to current inputs and switch between the appropriate patterns at that frequency.

System Design:

The system utilizes a Moore-based finite state machine (FSM) to control the LED patterns. The system uses 3 bits of input logic, in which each bit is assigned to an input switch. Eight different combinations are possible for the input. The system incorporates an input clock signal at 2 Hz for the synchronous FSM. The FSM requires a 4-bit output signal assigned to the four LED. 16 different combinations are possible for the LEDs, but this design will not use all of them.

The design of the FSM assumes that the off state of an LED is represented by a binary 0 for the output. The LED is in its on state when its output bit is a binary 1. For example, an FSM output of 4’b0000 represents all LEDs are turned off. On the contrary, an output of 4’b1111 represents all LEDs are turned on. More information about the relationship between inputs and outputs of the FSM is tabulated in table 1.

The FSM requires eight states in total for the LED controller to cover all input/output scenarios: One state exists to disable all LEDs, another state exists to enable all LEDs, two states exist to enable the LEDs to alternate, and four states exist to cycle the LED chasing pattern across the strip.

The two states that control the alternating pattern are expected to cycle between the outputs 4’b0101 and 4’b1010 for the LEDs as enabled by the FSM inputs. Each of the four chasing states maps their outputs in the following order: 4’b0001, 4’b0010, 4’b0100, and 4’b1000. As long as the inputs result in a chasing state, these outputs are cycled continuously in this pattern.

The system should respond to any changes in its inputs immediately after each clock cycle as it traverses through the FSM. According to table 1, the FSM should be able to traverse from the state ‘Chase 2’ to ‘ALL OFF’ when the inputs change to 3’b0xx (bits represented by x are do not care values).

Ellis Chen
09/11/2022

Present State	Next State [2:0]abc					output
	0xx	100	101	110	111	z
All OFF	ALL OFF	ALL ON	ALL ON	Chase 1	Alt 1	0000
ALL ON	ALL OFF	ALL ON	ALL OFF	Chase 1	Alt 1	1111
Alt 1	ALL OFF	ALL ON	ALL ON	Chase 1	Alt 2	1010
Alt 2	ALL OFF	ALL ON	ALL ON	Chase 1	Alt 1	0101
Chase 1	ALL OFF	ALL ON	ALL ON	Chase 2	Alt 1	0001
Chase 2	ALL OFF	ALL ON	ALL ON	Chase 3	Alt 1	0010
Chase 3	ALL OFF	ALL ON	ALL ON	Chase 4	Alt 1	0100
Chase 4	ALL OFF	ALL ON	ALL ON	Chase 1	Alt 1	1000

Table 1: The state table for the LED controller

Figure 1 shows the diagram of the FSM. The diagram includes all possible input/output cases for each of the eight states. This design uses logic 'abc' as the label for the 3-bit input and logic 'z' as the label for the 4-bit output. The design uses the most significant bit (third bit) in 'abc' as the on/off bit; the system will turn off all LEDs when this bit is 0, otherwise the system functions depending on the other two input bits.

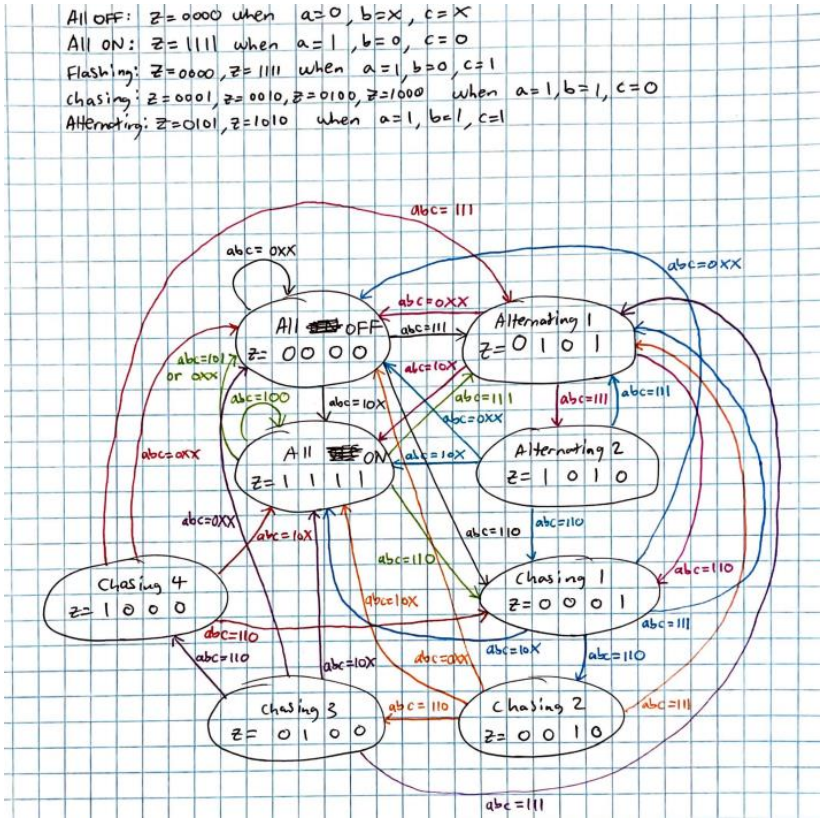

Figure 1: FSM Diagram for the LED controller

Ellis Chen
09/11/2022

Testing Approach:

The system incorporates a synchronous FSM, in which the design requires a clock signal to function. The design specifies the clock frequency to be 2 Hz. In simulation and on-system testing, the presence and integrity of the clock signal needs to be verified. The FSM itself requires testing to ensure all possible input/output combinations are accounted for and correct in navigating the various states.

For simulation, my testbench forces the clock to simulate a 2 Hz signal. I will run the test of the FSM using this clock signal. The test should cover all input combinations and show all possible output combinations. If the test is successful, the simulation will show all five patterns for the LEDs.

For on-system testing, I will test that the Cyclone produces a 2 Hz signal for the FSM. The simplest way to see if the signal is roughly 2 Hz is by looking at the speed at which the LEDs flash their patterns, assuming that the FSM works as intended. If the clock is too fast or two slow, I will change the clock divider value in the code repeatedly until I achieve the desired frequency.

Results:

The FSM waveforms from figure 2 show all the expected output patterns for the LEDs. As expected, the system sets all LEDs low or off when the 3rd input bit was low. The testbench prolonged some test cases more than others, such as the chasing states to ensure the patterns complete properly.

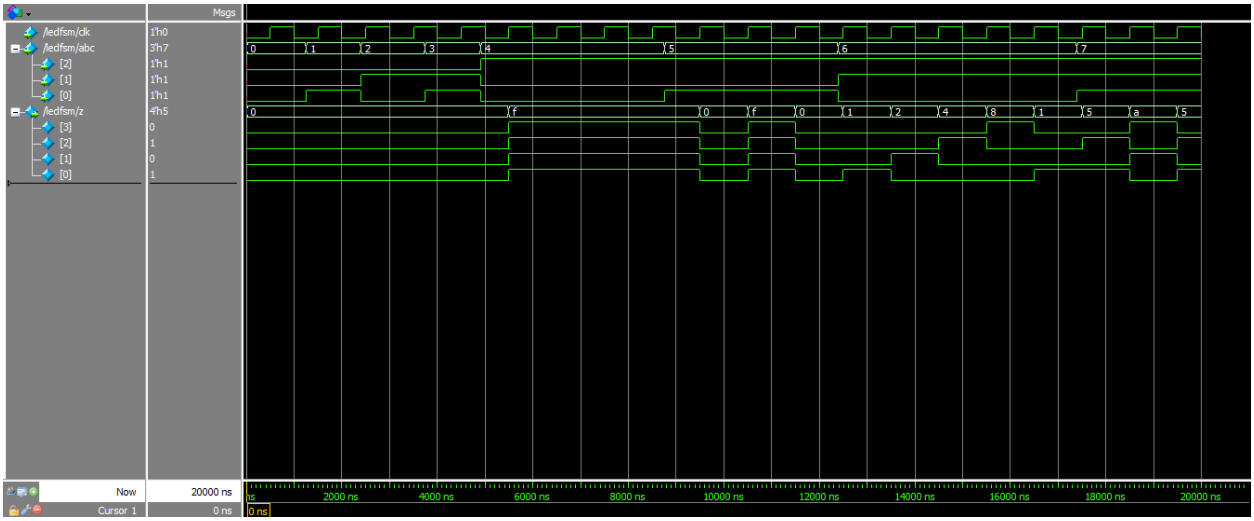

Figure 2: Waveform shows the different test cases for the FSM

The clock divider in figure 3 shows that the input clock signal divides down to produce a slower output clock signal. The clock in the Cyclone runs at 50 MHz. Using the calculated clock divider value to divide down 50MHz to 2 Hz in simulation was not practical. So, the clock divider value used to generate a simulated 25 MHz signal in figure 3 is different from the value used on the Cyclone. The test was successful as the waveforms show that the clock divider was able to reduce the input frequency by half. This test shows the divider is functional enough to implement with the Cyclone.

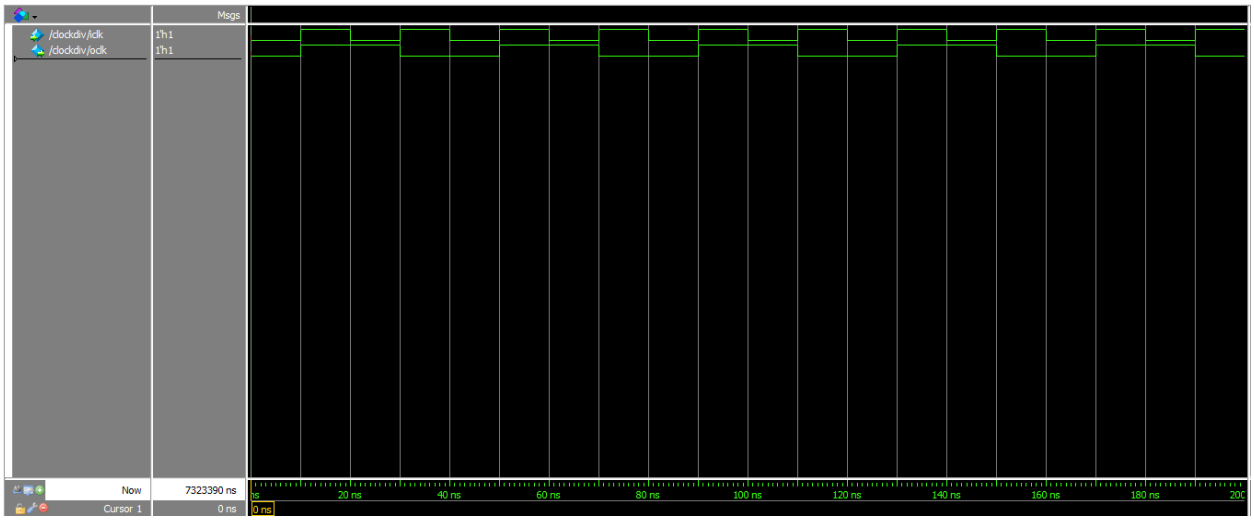

Figure 3: Waveform shows the input and output of the clock divider

The waveforms in figure 4 use the clock signal from the clock divider for the FSM. All input/output combinations are visible and expected. All the LED patterns are clearly seen in figure 4. These tests show that the logic for the LED controller is correct and meets all design specifications.

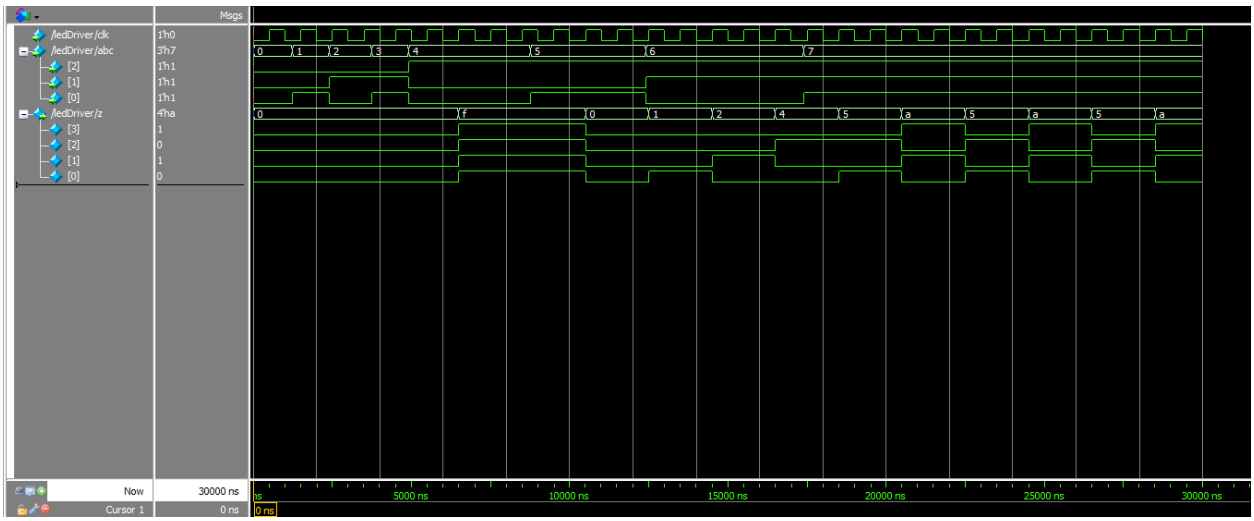

Figure 4: Waveform shows the FSM works properly using the clock divider

The FSM diagram from figure 5 agree with my diagram from figure 1. Both diagrams show most transitions occur when the system rests or turns off. The transitions for the chasing states, flashing, and alternating are clearly shown in figure 5. The direction of these transitions is clearly separated, which provides the viewer with better insight of the FSM.

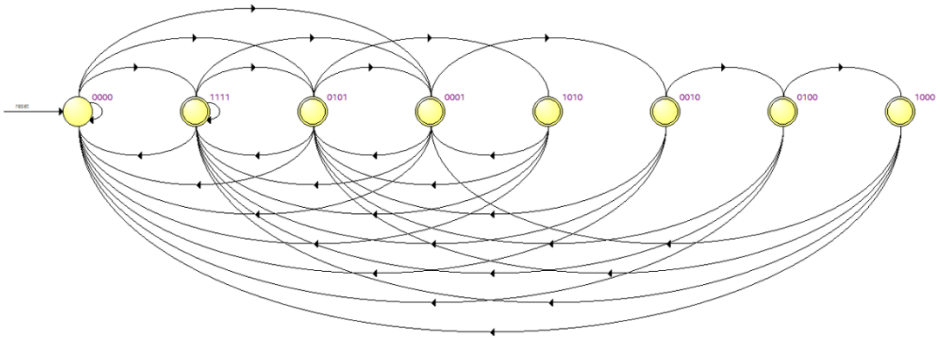

Figure 5: State Machine Viewer Output for the FSM

Table 2 tabulates each present state and the destination state depending on the input conditions. According to table 2, there are 32 possible transitions in the FSM.

	Source State	Destination State	Condition
1	0000	0001	(!abc[0]).(abc[1]).(abc[2])
2	0000	0101	(abc[0]).(abc[1]).(abc[2])
3	0000	1111	(!abc[1]).(abc[2])
4	0000	0000	(!abc[2])
5	0001	0010	(!abc[0]).(abc[1]).(abc[2])
6	0001	0101	(abc[0]).(abc[1]).(abc[2])
7	0001	1111	(!abc[1]).(abc[2])
8	0001	0000	(!abc[2])
9	0010	0100	(!abc[0]).(abc[1]).(abc[2])
10	0010	0101	(abc[0]).(abc[1]).(abc[2])
11	0010	1111	(!abc[1]).(abc[2])
12	0010	0000	(!abc[2])
13	0100	0101	(abc[0]).(abc[1]).(abc[2])
14	0100	1000	(!abc[0]).(abc[1]).(abc[2])
15	0100	1111	(!abc[1]).(abc[2])
16	0100	0000	(!abc[2])
17	0101	0001	(!abc[0]).(abc[1]).(abc[2])
18	0101	1010	(abc[0]).(abc[1]).(abc[2])
19	0101	1111	(!abc[1]).(abc[2])
20	0101	0000	(!abc[2])
21	1000	0001	(!abc[0]).(abc[1]).(abc[2])
22	1000	0101	(abc[0]).(abc[1]).(abc[2])
23	1000	1111	(!abc[1]).(abc[2])
24	1000	0000	(!abc[2])
25	1010	0001	(!abc[0]).(abc[1]).(abc[2])
26	1010	0101	(abc[0]).(abc[1]).(abc[2])
27	1010	1111	(!abc[1]).(abc[2])
28	1010	0000	(!abc[2])
29	1111	0001	(!abc[0]).(abc[1]).(abc[2])
30	1111	0101	(abc[0]).(abc[1]).(abc[2])
31	1111	1111	(!abc[0]).(!abc[1]).(abc[2])
32	1111	0000	(!abc[0]).(!abc[2]) + (abc[0]).(!abc[1]) + (abc[0]).(abc[1]).(!abc[2])

Table 2: Transitions between all the states in the FSM

Ellis Chen
09/11/2022

Analysis:

The design of the FSM for the LED controller was successful and intuitive enough to test with simulations and on-board. An improvement for this design and future designs is to instantiate every state in system Verilog by using typedef enum. In my implementation, each state was instantiated individually using static assignments. My implementation would not be easily scalable for larger projects in the future, so using enumeration to declare my states would be more ideal.

The state machine viewer from figure 5 shows transitions in a more organized manner than my diagram. However, each transition does not have any labels, so there is ambiguity in which input conditions results in which states. My diagram from figure 1 may seem difficult to follow as there are multiple crossings in the transitions, but each transition is labeled to cover all possible input combinations. Functionally, both diagrams from figure 1 and 5 are the same, to which these results were to be expected. The results from tables 1 and table 2 are expected as well. Both tables show the destinations states associated with its present states and its current inputs. The only difference between the tables is the display of total number of possible transitions in the FSM from table 2, which count up to 32 transitions.

After attempting to modify the code to change the system to a Mealy FSM, I discovered that removing one state from my Moore-based FSM is not possible without changing the entire structure of the code. My system assigns the present state as the LED outputs, if Mealy requires removing one state, my system would be missing one pattern specified by the lab procedure. My attempt at implementing a Mealy-based FSM first started by using enumeration to declare my states. The state that I dropped to implement Mealy was the last chasing state. Figure 6 shows the mealy-based FSM diagram from the state viewer. Table 3 shows a total of 28 transitions, which is 4 transitions less than the Moore-based FSM. Although the Mealy-based FSM should be more simplified and cost-efficient than the Moore-based FSM, the Moore-based FSM works the best for this problem because each state can be intuitively represented by the LED patterns.

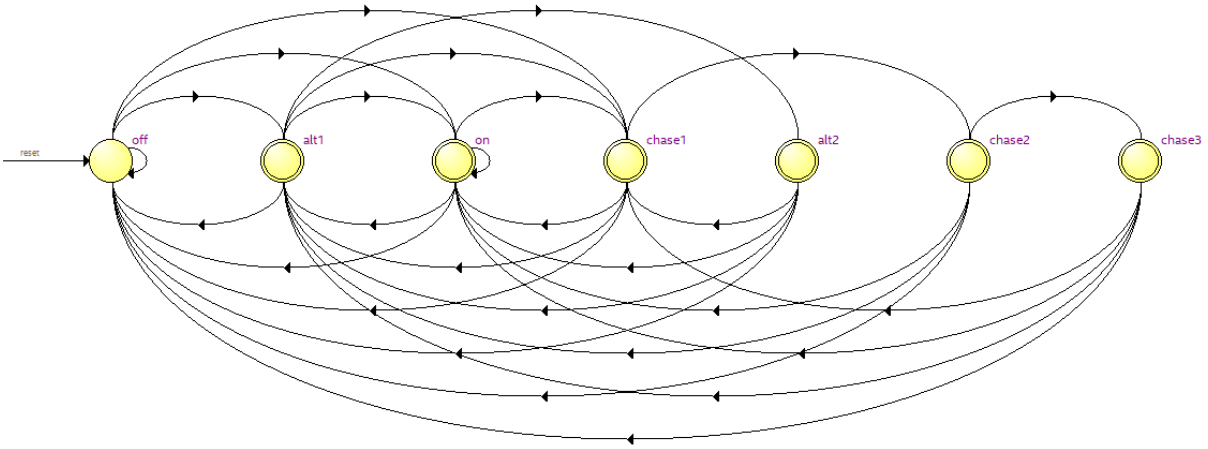

Figure 6: Mealy-based FSM diagram

	Source State	Destination State	Condition
1	alt1	off	(!abc[2])
2	alt1	alt2	(abc[0]).(abc[1]).(abc[2])
3	alt1	on	(!abc[1]).(abc[2])
4	alt1	chase1	(!abc[0]).(abc[1]).(abc[2])
5	alt2	off	(!abc[2])
6	alt2	alt1	(abc[0]).(abc[1]).(abc[2])
7	alt2	on	(!abc[1]).(abc[2])
8	alt2	chase1	(!abc[0]).(abc[1]).(abc[2])
9	chase1	off	(!abc[2])
10	chase1	chase2	(!abc[0]).(abc[1]).(abc[2])
11	chase1	alt1	(abc[0]).(abc[1]).(abc[2])
12	chase1	on	(!abc[1]).(abc[2])
13	chase2	off	(!abc[2])
14	chase2	chase3	(!abc[0]).(abc[1]).(abc[2])
15	chase2	alt1	(abc[0]).(abc[1]).(abc[2])
16	chase2	on	(!abc[1]).(abc[2])
17	chase3	off	(!abc[2])
18	chase3	alt1	(abc[0]).(abc[1]).(abc[2])
19	chase3	on	(!abc[1]).(abc[2])
20	chase3	chase1	(!abc[0]).(abc[1]).(abc[2])
21	off	off	(!abc[2])
22	off	alt1	(abc[0]).(abc[1]).(abc[2])
23	off	on	(!abc[1]).(abc[2])
24	off	chase1	(!abc[0]).(abc[1]).(abc[2])
25	on	off	(!abc[0]).(!abc[2]) + (abc[0]).(!abc[1]) + (abc[0]).(abc[1]).(!abc[2])
26	on	alt1	(abc[0]).(abc[1]).(abc[2])
27	on	on	(!abc[0]).(!abc[1]).(abc[2])
28	on	chase1	(!abc[0]).(abc[1]).(abc[2])

Table 3: Mealy-based FSM transitions