# Lab 1

Cameron Jennings (ID: 2029631), Donovan Clay (ID: 2276005)

October 5, 2023

CSE 371

# Contents

# 1 Design Procedure

In this lab we designed a parking lot system to keep track of the occupied spaces. There are two sensors, implemented with switches in a breadboard, one sensor is scanning the outside of the lot and one is scanning. The outside sensor connected to V_GPIO[28] and the inner V_GPIO[30]. In addition, when a sensor is triggered, its corresponding LED turns on to signify the signal, the outer sensor is connected to V_GPIO[33] and inner V_GPIO[35]. Based on the pattern of the sensors, our system can determine whether a car has entered or exited the lot. Internally, the system then counts the number of cars inside the lot and displays that number on the HEX displays. The reset used is connected to another switch in the breadboard connected to V_GPIO[24].

## 1.1 Car Detection

The module to detect cars receives input from the switches connected to the DE1_SoC board. A finite state machine, implemented in a always_comb block, takes the input and determines what state the system should move to next, with the reset/default setting the system to the state with neither sensor triggered. To ensure metastability, the changing of states is then run through a flip-flop that syncs the change of the state with the positive edge of the clock. Finally, the output of a car entering or exiting is assigned based on the system being in a specific state and not receiving input from one the sensors. The output can either be written to signal a car entering or exiting through the net specified.

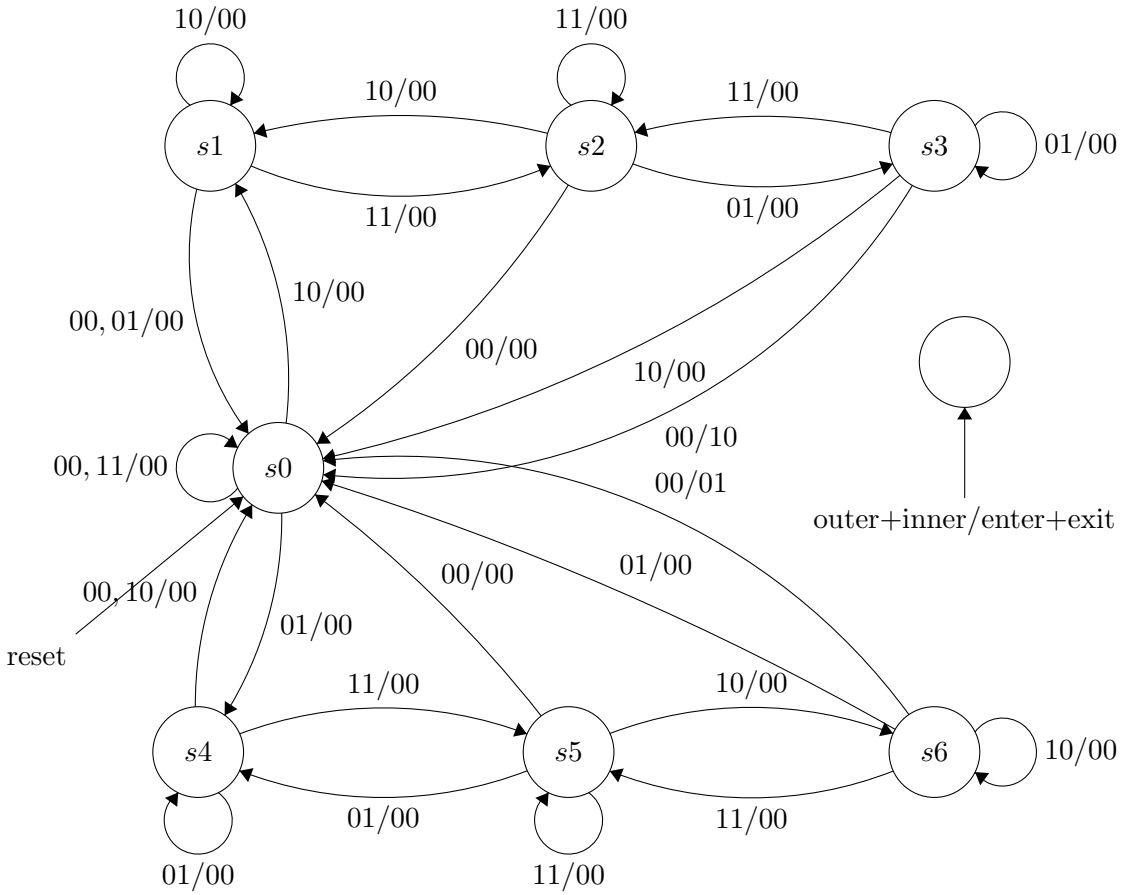


Figure 1: Car Detection FSM diagram

Our state machine has 7 states.

**S0:** Both sensors are not triggered.

**S1:** A car is starting to enter the lot.

**S2:** A car is entering the lot and triggering both sensors.

**S3:** A car is entering the lot and is only triggering the inside sensor.

**S4:** A car is starting to exit the lot.

**S5:** A car is exiting the lot and triggering both sensors.

**S6:** A car is exiting the lot and only triggering the outside sensor.

Our FSM also accounts for pedestrians triggering the sensors. For example, if a car starts to enter the parking lot and triggers the outside sensor. A pedestrian could then trigger the inside sensor, but we don't want the pedestrian to mess up our detector so if the inside sensor then becomes untriggered then we know the car is still only triggering the outside sensor.

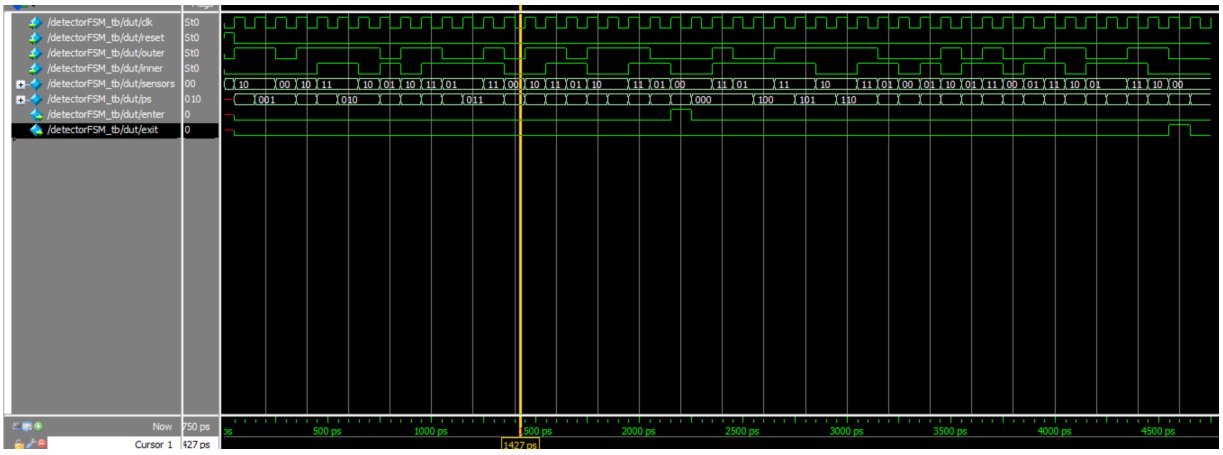


Figure 2: Car Detection Waveform

This testbench test every transition in the Car Detection FSM. We can see from the waveform that the enter signal is only high when the sensors go through the sequence $00 \rightarrow 10 \rightarrow 11 \rightarrow 01 \rightarrow 00$. We can also see the exit signal is only high when the sensors go through the sequence $00 \rightarrow 01 \rightarrow 11 \rightarrow 10 \rightarrow 00$.

## 1.2 Car Counter

The module to track the number of cars present in the garage receives inputs from the car detection module. It is flagged when a car has entered or exited the lot and then the current state of the counter is increased or decreased. The lot has a maximum of 16 spots so the counter stops tracking any signals after this. This applies to 0 as well as there cannot be negative cars in the lot. The output value is assigned the same value as the state.

This testbench bench demonstrates the expected behavior of the car counter module. The count increments whever incr is high and decrements whenever decr is high. It also tests that the count stays the same when both signals are high or low. It also tests the bounds of the counter, 0 and 16.

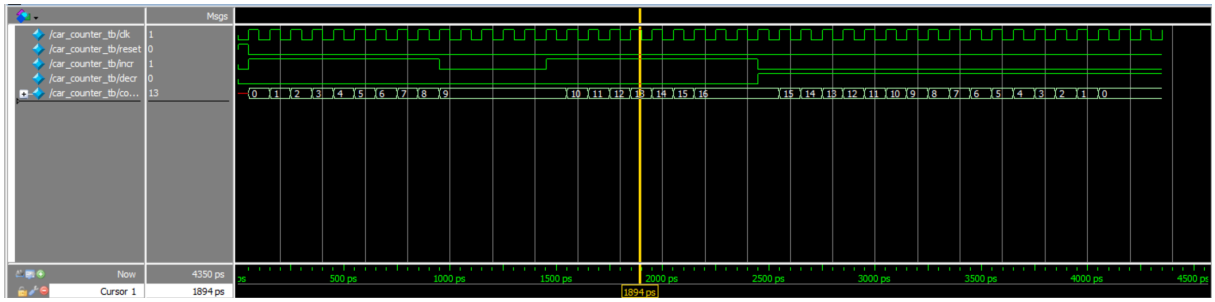


Figure 3: Car Counter Waveform

## 1.3 Seven Segment Display

A module is dedicated to output the current count to the HEX display, it receives the count from the car counter module. Based on the count, it categorizes which state the system should be in, zero through sixteen, with a certain output designated for each state. When there are 16 cars in the lot, it outputs "FULL", with zero cars in the lot it outputs "CLEAR0", otherwise it displays the current number. This module is only combinational logic so there is no diagrams to provide.

## 1.4 DE1_SoC

This is the top-level module in the system. It begins with initializing all the inputs and outputs that will be used in the parking lot occupancy counter. Then it assigns inputs for the sensors and reset and assigns outputs for the LEDs. Next, the system creates nets that will allow information and signals to be transmitted from the different devices in the system.

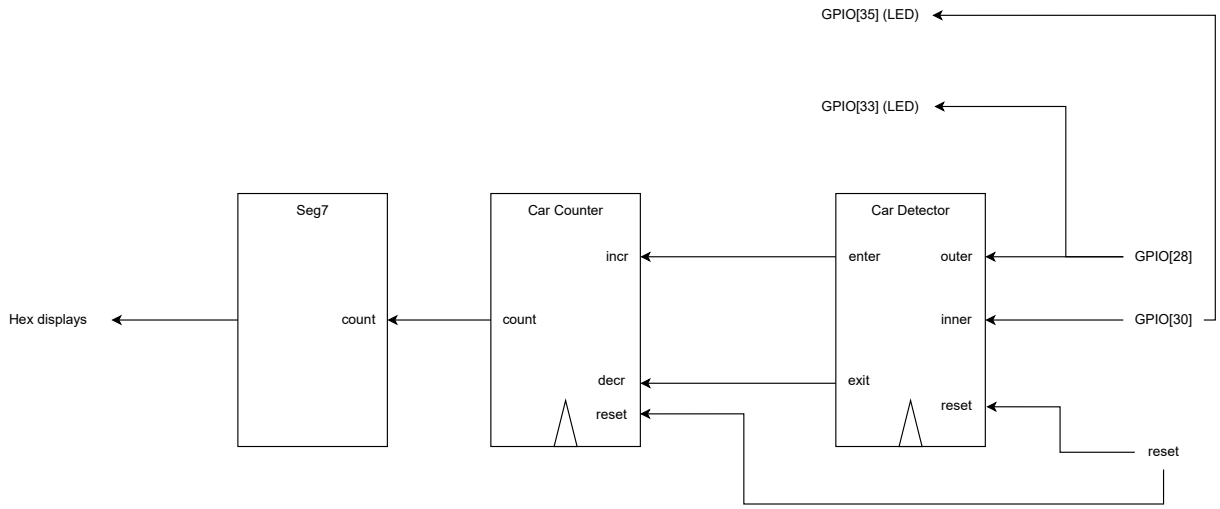


Figure 4: Top-level Module Block Diagram

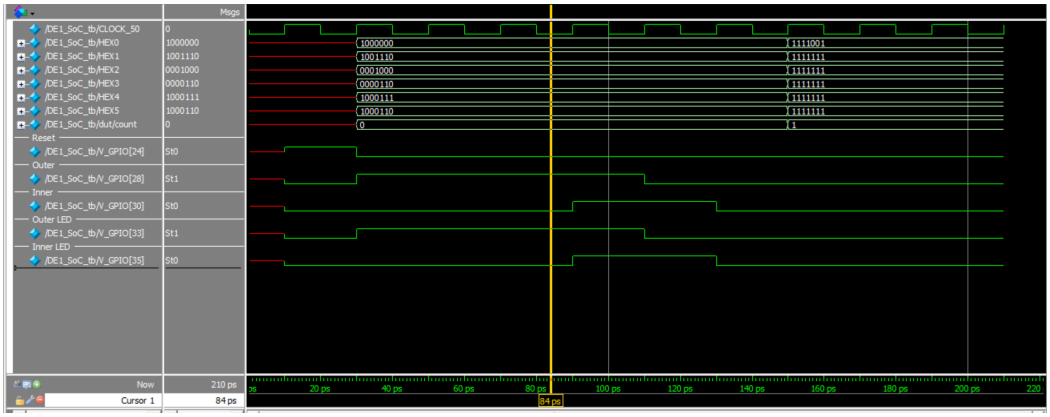


Figure 5: Top-level Module Waveform

This waveform demonstrates the top level module has all the correct connections. When the switches are changed through the GPIO ports, the associated LEDs turn on. When the detector sees cars entering and exiting the counter changes accordingly. And when the count changes, so do the hex displays.

# 2 Results

## 2.1 Flow Summary

| | Compilation Hierarchy Node | Combinational ALUTs | Dedicated Logic Registers | Block Memory Bits | DSP Blocks | Pins | Virtual Pins |
|---|---|---|---|---|---|---|---|
| 1 | ⌄ \|DE1_SoC | 33 (0) | 12 (0) | 0 | 0 | 89 | 0 |
| 1 | \|car_counter:counter\| | 12 (12) | 5 (5) | 0 | 0 | 0 | 0 |
| 2 | \|detectorFSM:detector\| | 10 (10) | 7 (7) | 0 | 0 | 0 | 0 |
| 3 | \|seg7:seg7_0\| | 11 (11) | 0 (0) | 0 | 0 | 0 | 0 |

Figure 6: Resource Utilization

# 3 Experience Report

We found this lab to overall not be too challenging, but a great precursor to get us familiar with SystemVerilog after a break from using it. It was almost all encompassing with general functions and syntax that we will need to use for the rest of the quarter. The most difficult part for us was to ensure that the finite state machine in the car detection module could identify the differences between a car and person, then testing all the edge cases and inputs for the machine. Besides the tedious time spent figuring that out, there were not many other issues encountered.
Although intuitive, we found it very useful to test each of the modules before moving on to implementing the next one. This allowed for us to pinpoint the module, and almost block of code, where an error occurs because the other modules have been tested and work as expected.
This lab took us approximately 12 hours, broken down as follows:

**Reading:** 30 minutes

**Planning:** 30 minutes

**Coding:** 5 hours

**Testing:** 3 hours

**Write up** 3 hours