Charles Tung Fang
EE 371
2019/10/7
Professor Rania Hussein

# Lab 1 Parking Lot Occupancy Counter Report

## Procedure

I approached the problem by understanding everything at the very top level. After reading the specs, I drew the top level block diagram (Figure 1) to make sure I know which modules are connecting together. I began drawing a FSM (Figure 2) and understand the relationship between key press and entry/exit situation. The major components in the system are the parking lot module, userInput module, and counter module. The parking lot module is simply the FSM logic I drew in Figure 2. It gives me two logics, enter and exit, for further determination of whether a car is entering or exiting. The userInput module takes care of metastability and make sure enter and exit status only valid for one clock cycle. The counter method counts how many cars are there and report the logics to display status module, which will display the results on the HEXs. Lastly the LED module turns on green light when A is pressed, and red light when B is pressed.

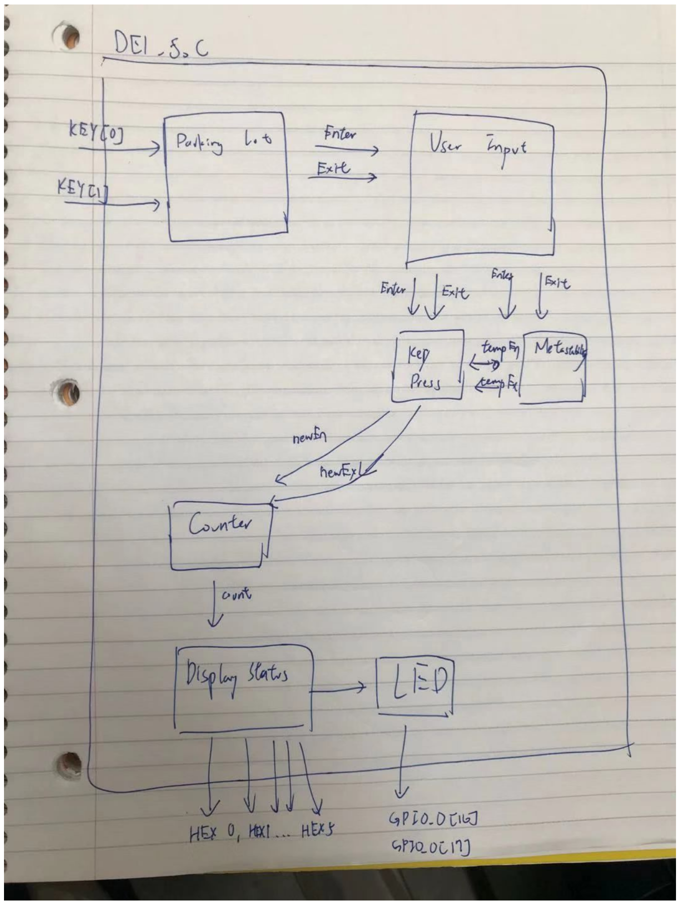


Figure 1. Top-level block diagram of the project

I implemented the way I did because we are dividing our clock based on enter and exit signals and not key presses. In order to achieve this goal, I need to process the key press logic and entering and exiting the parking lot logic first and then pass the modified enter and exit to make them sustain one clock cycle only.

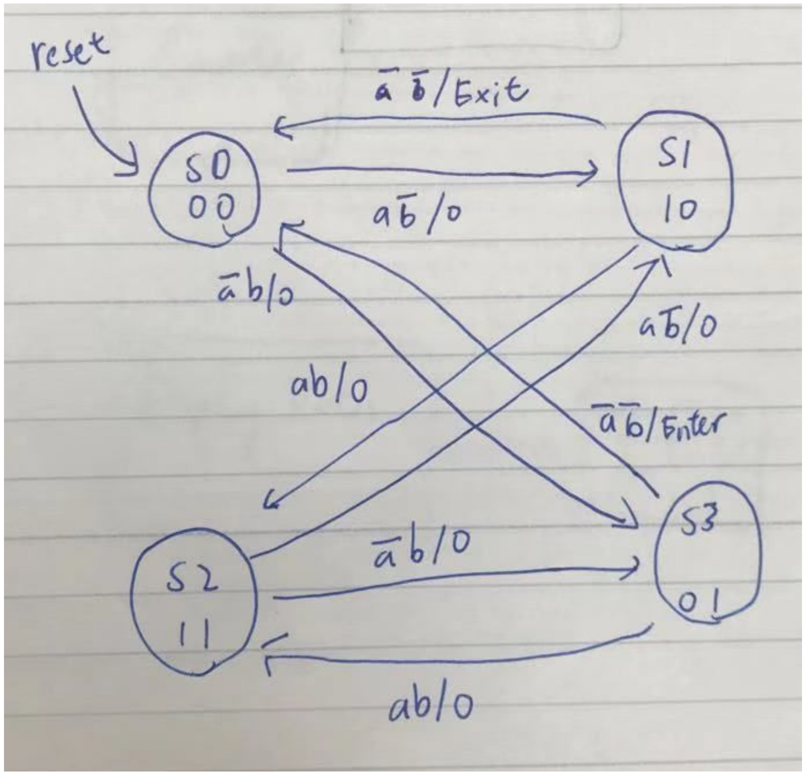


Figure 2. FSM for Counter

Finally, I used two LED lights and two resistors to build the A and B sensors connections. I then implemented two GPIO lines to connect the LED lights in order to conclude the whole design.

**Results**

My result was successfully shown on both ModelSim and the FPGA board. In my counter testbench (Figure 3), one can easily see that the count is incrementing while the enter signal is 1 as well as the count is decrementing while the exit signal is 1.

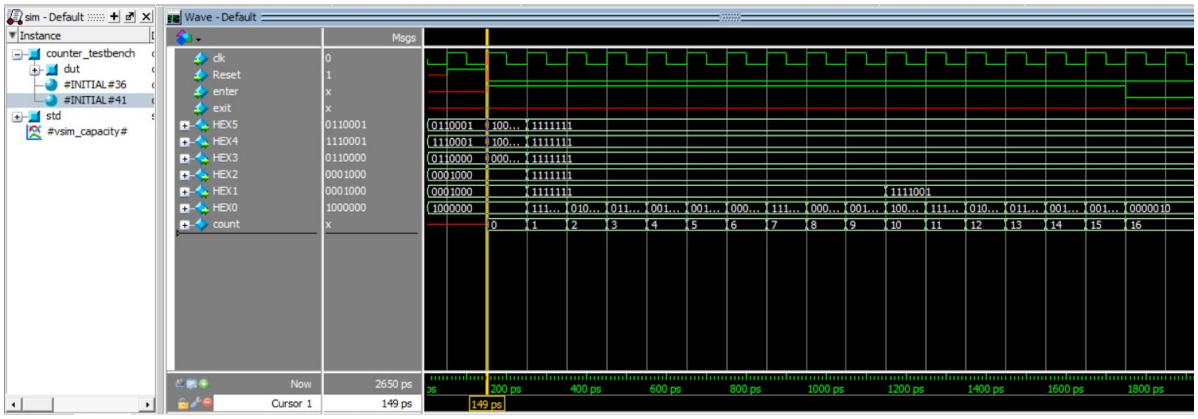


Figure 3. ModelSim for counter

In my Parking Lot testbench (Figure 4), I checked if my module match with my FSM logic. The screenshot shows that the present state and next state change accordingly as different A and B is pressed and the entry and exit signal only valid at enter: (ps == s3 && (~A&~B)) or exit: (ps == s1 && (~A&~B)).

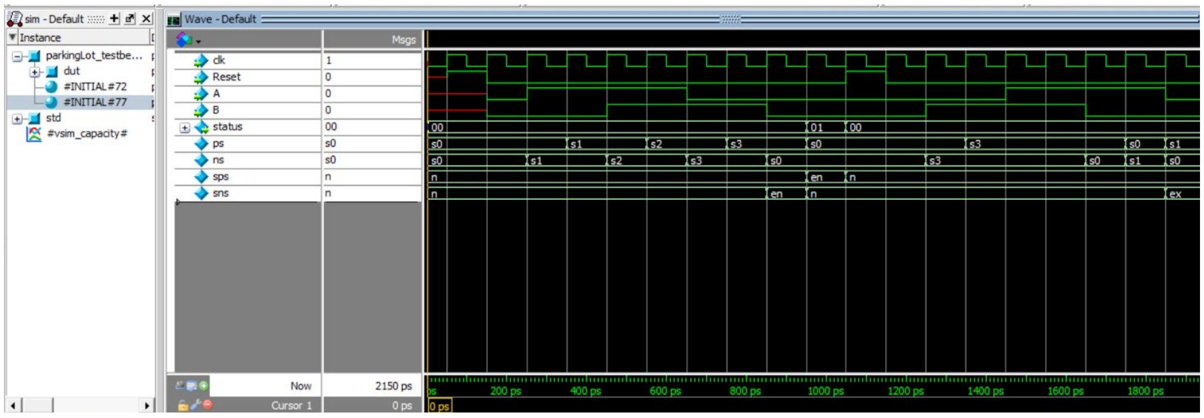


Figure 4. ModelSim for Parking Lot

The overall project offered the user count the car occupancy in the parking lot. Initially when the parking lot is empty, the HEXs shows "CLEAR" and 0 to indicate the situation(Figure 5).

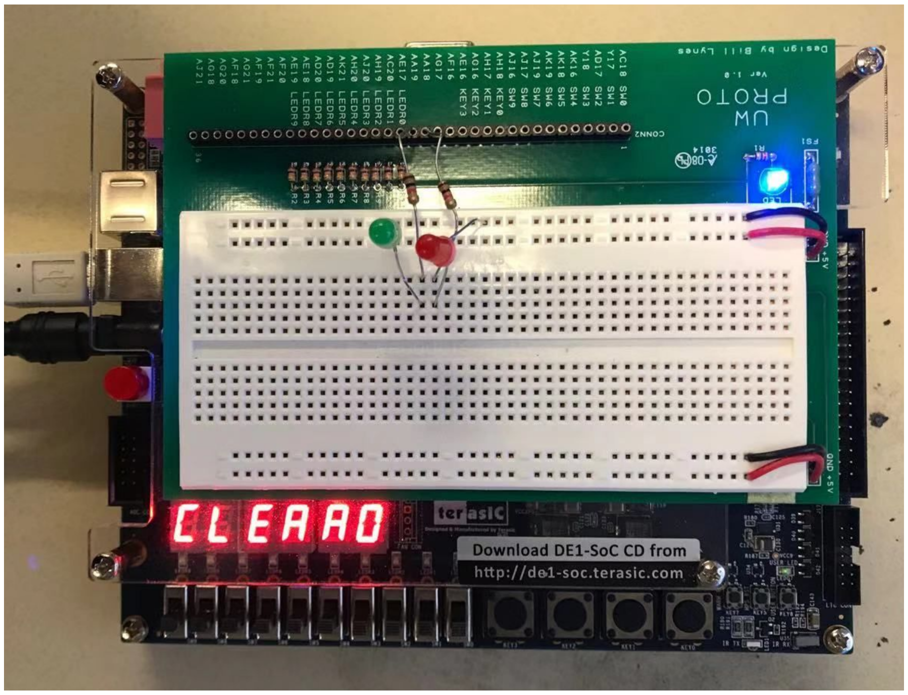


Figure 5. The HEXs show CLEAR and 0 when the parking lot is empty

It turns on red and green LEDs when the sensors (KEY0 & KEY1) detected a car is passing through (Figure 6~9).

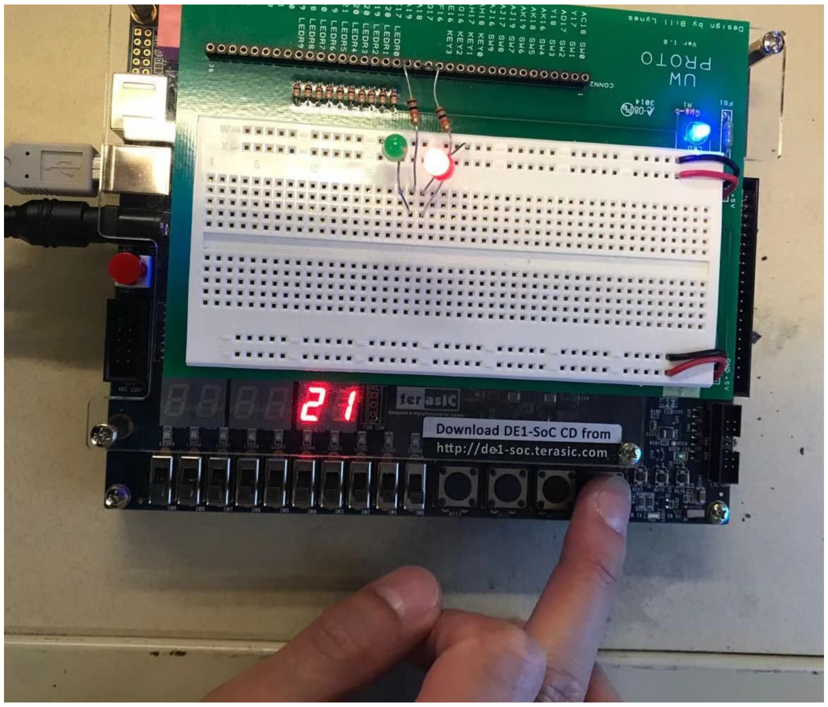


Figure 6 Red LED is turned on when a car passes sensor A

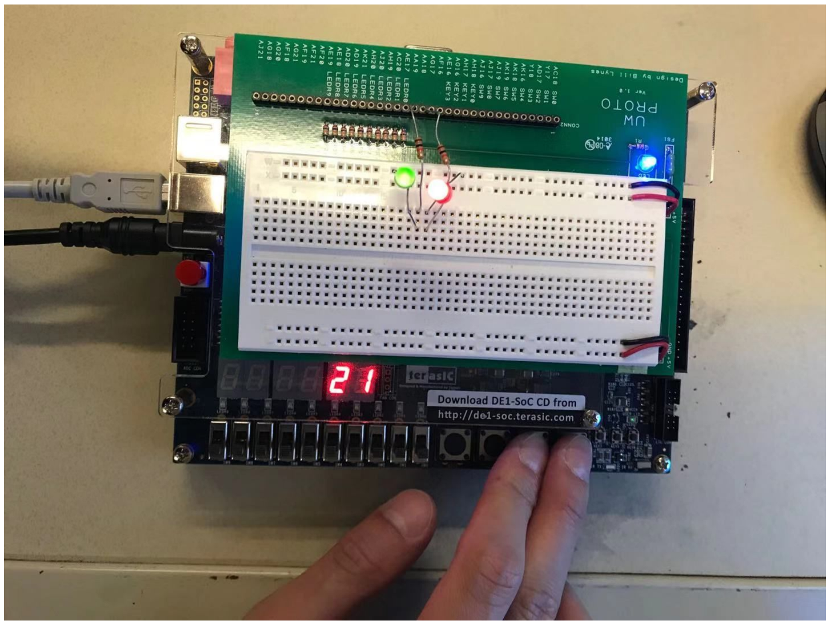


Figure7. Red & Green LED are turned on when a car passes sensor A & B

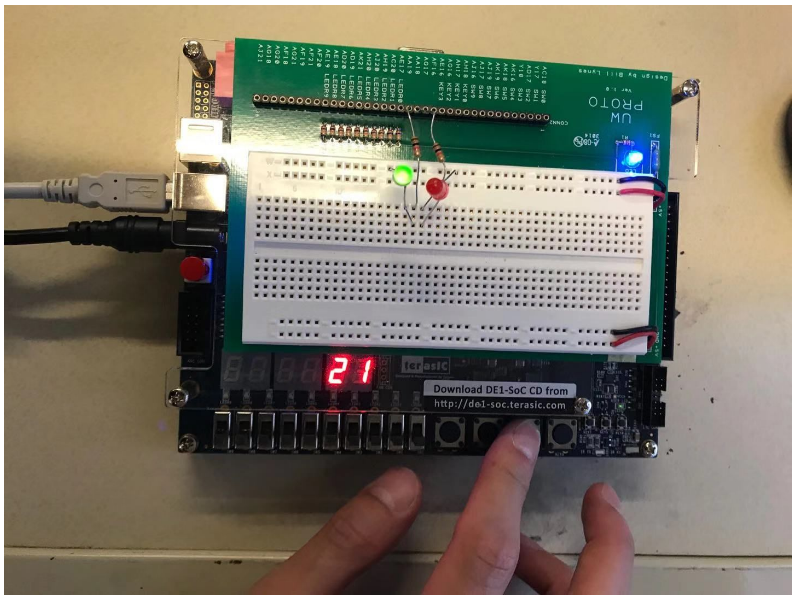


Figure 8. Green LED is turned on when a car passes sensor B

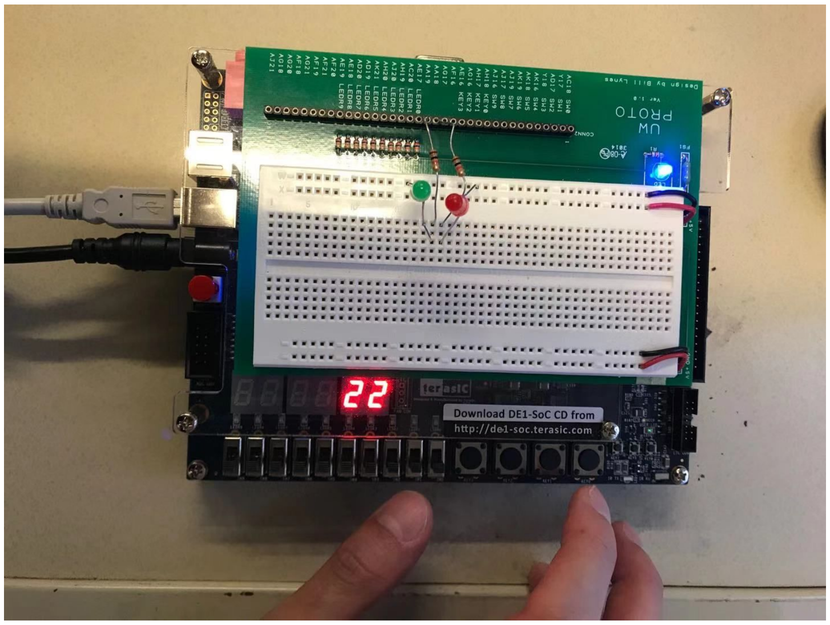


Figure 9. Green LED is turned off when a car leaves sensor B. It was an enter case so the car count + 1 and become 22

Lastly, the HEX will display "FULL" and the car count "25" when the parking lot is full. No more car is allowed no enter (Figure 10).

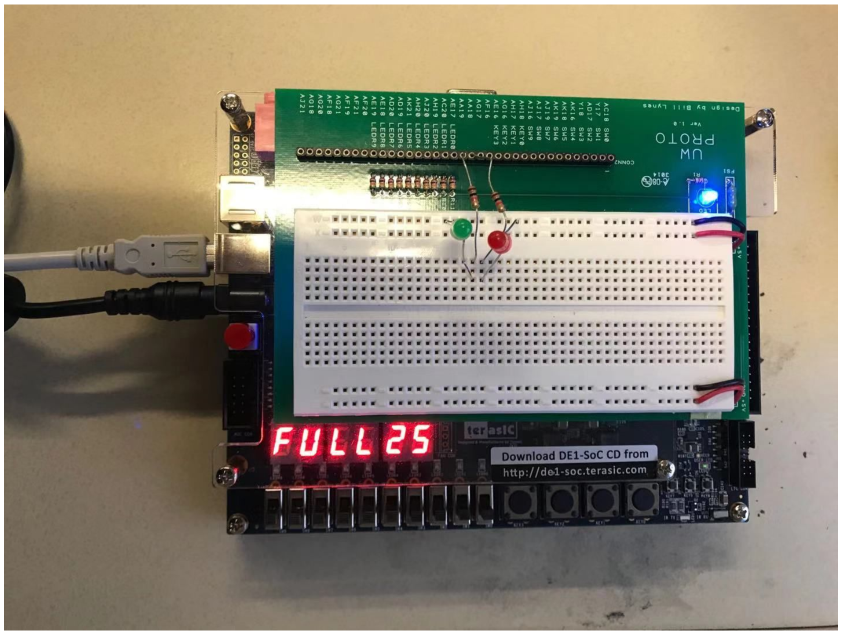
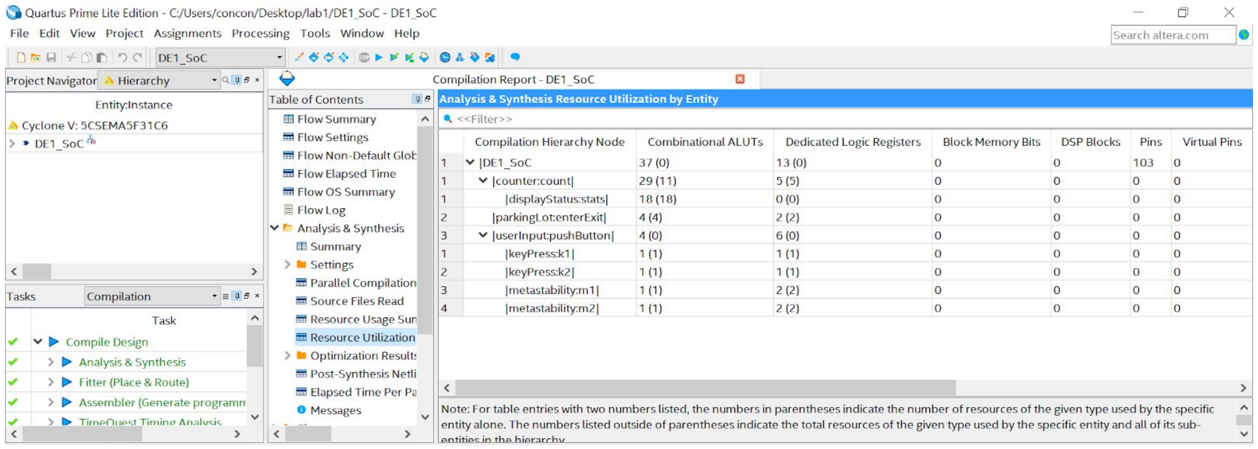


Figure 10. Situation when the parking lot is full

On the other hand, the Resource Utilization by Entity page (Figure 11) shows that the size of the program is 37+13 = 50



Quartus Prime Lite Edition - C:/Users/concon/Desktop/lab1/DE1_SoC - DE1_SoC

Compilation Report - DE1_SoC

Analysis & Synthesis Resource Utilization by Entity

| | Compilation Hierarchy Node | Combinational ALUTs | Dedicated Logic Registers | Block Memory Bits | DSP Blocks | Pins | Virtual Pins |
|---|---|---|---|---|---|---|---|
| 1 | \|DE1_SoC | 37 (0) | 13 (0) | 0 | 0 | 103 | 0 |
| 1 | \|counter:count\| | 29 (11) | 5 (5) | 0 | 0 | 0 | 0 |
| 1 | \|displayStatus:stats\| | 18 (18) | 0 (0) | 0 | 0 | 0 | 0 |
| 2 | \|parkingLot:enterExit\| | 4 (4) | 2 (2) | 0 | 0 | 0 | 0 |
| 3 | \|userInput:pushButton\| | 4 (0) | 6 (0) | 0 | 0 | 0 | 0 |
| 1 | \|keyPress:k1\| | 1 (1) | 1 (1) | 0 | 0 | 0 | 0 |
| 2 | \|keyPress:k2\| | 1 (1) | 1 (1) | 0 | 0 | 0 | 0 |
| 3 | \|metastability:m1\| | 1 (1) | 2 (2) | 0 | 0 | 0 | 0 |
| 4 | \|metastability:m2\| | 1 (1) | 2 (2) | 0 | 0 | 0 | 0 |

Note: For table entries with two numbers listed, the numbers in parentheses indicate the number of resources of the given type used by the specific entity alone. The numbers listed outside of parentheses indicate the total resources of the given type used by the specific entity and all of its sub-entities in the hierarchy

Figure 11. Resource Utilization by Entity page

**Problems Faced and Feedback**

I think the most difficult part of this lab is that HEX1 shares both a digit and a text. Initially I separated the display module into two: displayCount and displayStatus. displayCount takes care of only number while displayStatus takes care of only text. However, I encountered an error saying that I can’t assign two value to a HEX at a time. Therefore, I had to redesign my display module to combine those two together. Fortunately I was able to get it to compile and work. The tip I got out of this lab is to do one thing in a module and don’t over complicate a simple task into separate modules.

I feel pretty comfortable for this lab because it is very similar to what I have learned in 271. It took me around 6 hours to finish it. The project is clear and straightforward and I really enjoyed it.