**CS M152B Lab 2: State Machine Design**

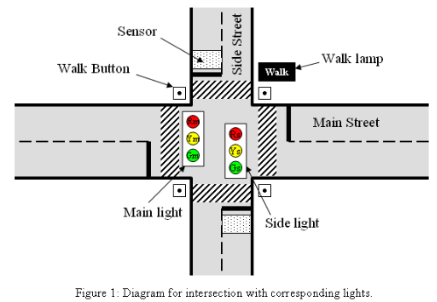
**By: Chris Baker and Rodrigo Gonzalez**

**UID: 105180929, 905161154**

**TA: Ghazal Ershadi**

1. **Introduction**
   1. **Overview**

A Traffic Light Controller is used to handle the timing and street lights for an intersection. There are time intervals involved with how long a light should stay on. There are also dependencies for which adjacent street lights should behave as well as the street light opposite of the first light. For the purposes of our lab we designate one street as the Main Street and another as the Side Street. A diagram for the assignment is shown below.



The traffic light controller for the intersection contains green, yellow, and red lights. There is also a Walk Button that can be pushed to cause the traffic lights to turn red and a Walk Lamp to turn on for a pedestrian to cross the street. Further, there is also a street sensor that can extend the time interval for a light to allow cars to keep driving.

In essence the traffic light controller will behave like a common street lamp that may be familiar to most pedestrians. The goal of this lab is to implement Finite State Machine Design with the Basys3 FPGA board as well as to utilize button presses and switches to trigger events such as state transitions or an LED to light up.

* 1. **Requirements**

Aside from the typical behavior stated above in the overview for how traffic lights at a signal generally behave, we do have specific requirements in this lab. There is a base interval of 6 seconds, an extended interval of 3 seconds, and a yellow light interval of 2 seconds.

The initial Main Green Light Signal is green for 12 seconds, then it turns yellow for 2 seconds, and then turns red while the side light simultaneously turns green. The side light must then remain green for 6 seconds and then stay on yellow for 2 seconds. While a light is green or yellow the other street’s lights must remain red in order to prevent crashes or injuries to pedestrians. These conditions repeat in sequence if no other signals in the sensor or walker are detected.

The Walk signal can trigger a deviation from this loop when a pedestrian submits a walk request. This is triggered through pressing our center button on the FPGA board. When the request is recognized (button press registered), then after the Main Street yellow light interval ends all street lights are set to red and the Walk Button turns on. The Walk Light occurs for 3 seconds where the pedestrian has a chance to run across the street (i.e. quickly play frogger). Because thereafter, the Side Street sequence resumes the lighting sequence by turning green. We also note that the walk signal must be cleared when the walk cycle is finished.

The other deviation from the loop that can occur is the traffic sensor. If the traffic sensor is set to high then at the end of the first 6 second length of the Main Street light going green, the light must be green for an additional 3 seconds, rather than the full 6 seconds.

1. **Design and Implementation**
   1. **Clock Design**

Since the whole project used time intervals of seconds we needed to implement a clock divider that would yield 1 second cycles for which we could track or change events on rising clock edges. We looked up documentation on the FPGA board and were able to find that letting a counter go to 50,000,000-1 and flipping a sec\_clk\_reg register would be able to give us the desired time cycles. We assigned the output seconds\_clk to this sec\_clk\_reg value to feed back into our top module so we could properly time street light functions. A snippet of the code is shown below:

always @(posedge clk)

begin

if (rst)

begin

seconds\_counter <= 0;

sec\_clk\_reg <= 1;

end

else if (seconds\_counter == 50000000-1)

begin

sec\_clk\_reg <= ~sec\_clk\_reg;

seconds\_counter <= 0;

end

else

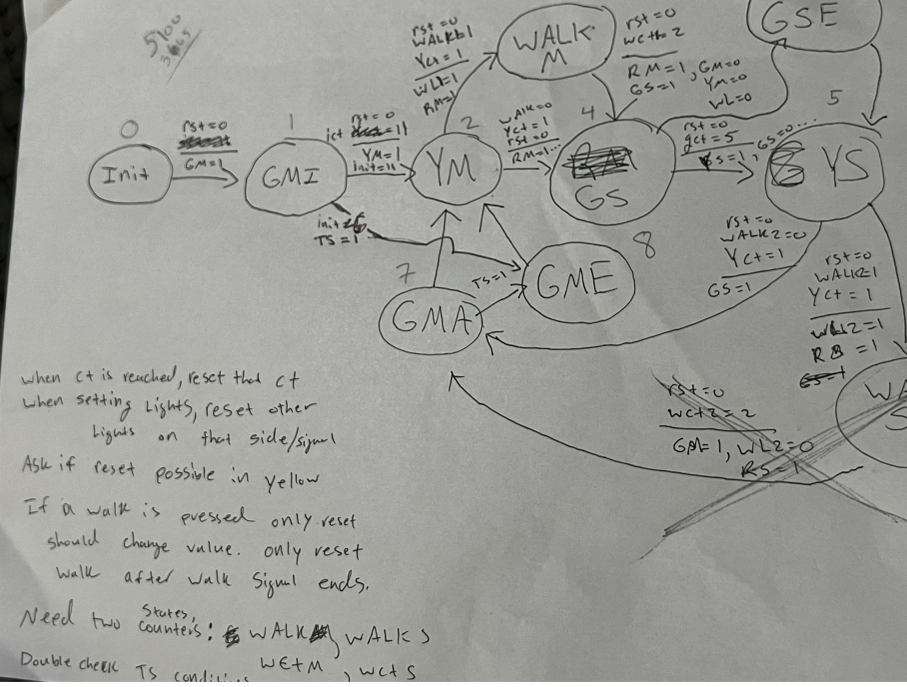
seconds\_counter <= seconds\_counter + 1;

end

* 1. **Initial State Design**

The initial state design for our project is shown below and there have been modifications as needed which led to the professor verifying and approving our demo. Some general notes are discussed here. The initial state is basically a state that is used for reset. Although it is not shown in the diagram, there is a state transition to the initial state from any state via the reset signal going high. States will transition to the next state based on a signal or counter or sometimes both. We check for the next state on a positive clock edge and update current state on the clock edge, so generally there is a 1s delay from determining the next state to moving into that state, however exceptions are made with sensitivity lists including some signal to update the next state nearly instantaneously. Generally a state will stay looping in its state until the count or signal that determines its next state is triggered. The counters are keeping track of seconds which are used with ‘==’ or ‘<’ evaluations to help determine transition logic. This concludes the general notes on the diagram, but finer details and updates were made that are not shown on the initial drawing. Those updates accomplished all design specs which were verified during demonstration to the professor and are discussed throughout this paper.

The **initial state** transitions to the **GMI** state when the reset signal is off. In **GMI** (Green Main Initial) we have two counters. If the initial counter is less than 6 and the traffic sensor goes high, then we go to **GME**. **GME** (Green Main Extended) adds 2 seconds to the previous 6 second green light by checking for extended counter == 2, if so we transition into **YM**. However, If the traffic sensor was not high by 6 seconds, then we stayed in **GMI** instead of moving into **GME**. We remain in **GMI** till the initial counter is 11 (we update current state to next state on clock edges, but change the “always sensitivity list” to update on traffic sensor changes, but still preserve a 1 second delay for state change from initial counter) we transition into **YM**. **YM** (Yellow Main) is supposed to last for 2 seconds, thus on the yellow counter == 1 we can check if the Walk Button is pressed. If it is, we move into the **WALK** State. The **WALK** state checks if the walk counter == 2 and if so it resets the values tracking the walk signal and moves to the **GS** state. If the walk button is not pressed we move into the **GS** state from **YM** bypassing the **WALK** state. In the Green Side state we check for green counter < 6 and if the Sensor is set. If the sensor is set we move into **GSE** (Green Side Extended), if it is not set, then we move into **YS**. **GSE** checks if the extended counter == 2, if so it moves into **YS** (Yellow Side). In the Yellow Side we check if yellow counter == 1 and if so we move into **GMA**. A brief note here, but it was clarified that no walk needs to occur after the yellow side street, only for the main yellow. In **GMA** (Green Main After) we check for green counter < 6 and if the sensor is high we move to **GME**, if it is not high then we move to **YM** which would get us to an earlier position in this loop, where we repeat these state transitions based on these signals. In total we have 9 states, but likely could have removed the Initial State with changes to coding/design.



As mentioned a couple times above, the WALK side State was removed since it was not a part of the spec requirements, and some other signals for moving may not be shown above, but were discussed above. The code may also clarify some of the state transition triggers.

* 1. **Implementing the Design**
     1. **Parameters and Initializations**

In CS152A I learned how to use parameters to act as macros for certain values, so we implemented parameters for each state used:

parameter INIT = 4'b0000; parameter GMI = 4'b0001;

parameter YM = 4'b0010; parameter GS = 4'b0100;

parameter YS = 4'b0101; parameter GMA = 4'b0111;

parameter GME = 4'b1000; parameter WALK = 4'b1001;

parameter GSE = 4'b1010;

Then we initialized a current state and set it to our Initial State as well as a next state that would help us to track two different state positions and update the current state to the next state on positive clock edges.We also initialized counters to track the length of time within each state.

reg [3:0] cur\_state = INIT; reg [3:0] next\_state;

reg [4:0] init\_counter; reg [3:0] green\_counter;

reg [2:0] yellow\_counter; reg [2:0] walk\_counter;

reg [2:0] extended\_counter;

* + 1. **Always Block Next State Handler**

Next we implemented four different always blocks that ran simultaneously while handling different specific functions. The first always block handles choosing the next state based on the current state. In any of the states if reset is encountered then the next state is set to Initial State. Also if conditions for transitioning are not reached, then a current state’s next state is itself. However, given that every state transition involves some type of counter, then a state will always move eventually in some 2, 3, or 6 second interval. We use a case statement that takes cur\_state as input and we implement the logic required to reach the next states as discussed above. One coding example is shown below for clarity:

always @ (\*)//(clk, cur\_state, rst, Sensor, WalkEn, init\_counter, green\_counter, yellow\_counter, walk\_counter, extended\_counter)

Begin

if (WalkButton)

begin

walk\_on <= 1;

WalkPressed <= 1;

end

case(cur\_state)

… // Other State Logic//…

WALK:

if (rst)

next\_state <= INIT;

else

begin

if (walk\_counter == 2)

begin

next\_state <= GS;

WalkPressed <= 0;

walk\_on <= 0;

end

else

next\_state <= WALK;

end

* + 1. **Always Block State Updater**

Our next always block is the state update implementer. It performs the function of updating the current state to the next state as determined in the previous block:

always @(posedge seconds\_clk)

begin

if (rst)

begin

cur\_state <= INIT;

end

else

cur\_state <= next\_state;

end

* + 1. **Always Block Counter Handler**

This block checks for the reset signal as well as a case statement for all the current states. If reset was encountered, then the walk signals, traffic sensor, and counters would be set to 0. In the switch statements we checked the current state and incremented the value of the counter corresponding to that state which is required for the state transitions based on number of seconds passed. For safety and explicitness we also reset the values of other counters here as we tried debugging our code to make sure we had the complete functionality intended which was achieved. An example of two states is shown below:

//…//

case (cur\_state)

GMI:

begin

init\_counter <= init\_counter + 1;

green\_counter <= 0;

yellow\_counter <= 0;

extended\_counter <= 0;

walk\_counter <= 0;

end

YM:

begin

yellow\_counter <= yellow\_counter + 1;

init\_counter <= 0;

green\_counter <= 0;

extended\_counter <= 0;

walk\_counter <= 0;

End

//… More State Logic…//

* + 1. **Always Block Output Handler**

This block handled the outputs which determined which signal lights were on and when the WalkLamp should be set. We coded this to the spec such that one green light or yellow light on a street should result in a red light on the other street. Also the WALK state would trigger the walk lamp to be on and result in both streets having red lights. An example is shown below:

always @(\*)

begin

case (cur\_state)

GMI:

begin

Gm <= 1;

Ym <= 0;

Rm <= 0;

Gs <= 0;

Ys <= 0;

Rs <= 1;

WalkLamp <= 0;

//…Other Output Logic//

WALK:

begin

Gm <= 0;

Ym <= 0;

Rm <= 1;

Gs <= 0;

Ys <= 0;

Rs <= 1;

WalkLamp <= 1;

//…//

* 1. **Debouncer**

The Debouncer logic was provided to my partner in CS152A a quarter or two ago. Their instructor had advised them to just insert it in their code, so we used the same code here which helped us to catch the button press signal for the Walk Button. Essentially, we know that if a threshold is met then we can pass the result as a button press being recognized, but if it is not met then we can make it that the value was 0. The following code accomplishes the goal of debouncing for our WalkButton input and sends the output for WalkEn:

module debouncer(

input WalkButton,

input clk,

output WalkEn

);

reg [1:0] arst\_ff;

wire arst\_i;

assign arst\_i = WalkButton;

assign WalkEn = arst\_ff;

always @(posedge clk or posedge arst\_i)

begin

if (arst\_i)

arst\_ff <= 2'b11;

else

arst\_ff <= {1'b0, arst\_ff[1]};

end

endmodule

* 1. **Basys3 xdc File**

We set the first 6 leds to hold the signals for the {Green Main, Yellow Main, Red Main, Green Side, Yellow Side, Red Side} where the Green Main was led[5] and Red Side was led[0]. Because of the way we input our signals into the leds they performed the normal light cycle loop by lighting led 5 and led 0 and moved inwards which was pretty cool to see. In the xdc file we renamed these led[] pins to our corresponding output registers:

output reg Gm, output reg Ym, output reg Rm,

output reg Gs, output reg Ys, output reg Rs,

set\_property -dict {PACKAGE\_PIN U16 IOSTANDARD LVCMOS33} [get\_ports {Rs]

set\_property -dict {PACKAGE\_PIN E19 IOSTANDARD LVCMOS33} [get\_ports {Ys]

set\_property -dict {PACKAGE\_PIN U19 IOSTANDARD LVCMOS33} [get\_ports {Gs]

set\_property -dict {PACKAGE\_PIN V19 IOSTANDARD LVCMOS33} [get\_ports {Rm]

set\_property -dict { PACKAGE\_PIN W18 IOSTANDARD LVCMOS33 } [get\_ports {Ym]

set\_property -dict { PACKAGE\_PIN U15 IOSTANDARD LVCMOS33 } [get\_ports {Gm]

There was also a need to set a led to represent the Walk Lamp, however we had some trouble with the button pressing. The professor had given us advice on storing the value in a register until the WALK state occurred which was useful, but we needed to see what was going on explicitly in the code. Thus, we coded two more registers to light up a led when the button was pressed (WalkPressed) and another to stay lit while the walk signal was held in a register (walk\_on). Thus, we coded these three registers to the led[15], led[14], led[13].

set\_property -dict { PACKAGE\_PIN N3 IOSTANDARD LVCMOS33 } [get\_ports WalkLamp]

set\_property -dict { PACKAGE\_PIN P1 IOSTANDARD LVCMOS33 } [get\_ports walk\_on]

set\_property -dict { PACKAGE\_PIN L1 IOSTANDARD LVCMOS33 } [get\_ports WalkPressed]

We also adjusted the clock property to our clk signal, the first switch to reset, and the center button to the traffic sensor.

## Clock signal

set\_property -dict {PACKAGE\_PIN W5 IOSTANDARD LVCMOS33} [get\_ports clk]

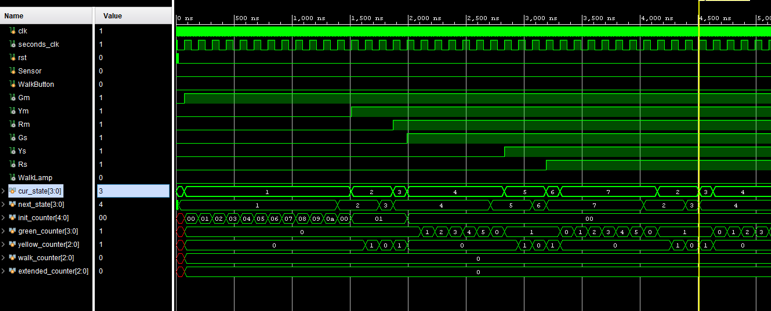
## Switches

set\_property -dict {PACKAGE\_PIN V17 IOSTANDARD LVCMOS33} [get\_ports rst]

##Buttons

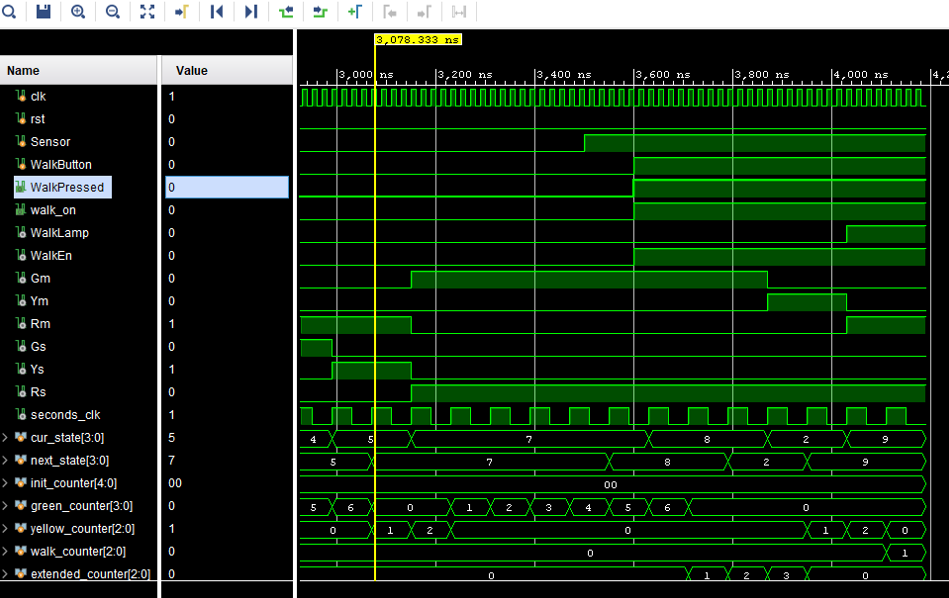
set\_property -dict { PACKAGE\_PIN U18 IOSTANDARD LVCMOS33 } [get\_ports Sensor]

1. **Simulation Waveforms**
   1. **Initial Image, Setting Lights on, but not off, appropriate state transitions**

****

The above image demonstrates our current states changing in 2,4, and 6 second intervals based on the seconds clock, our counters, and other signals. Initially the reset signal is set, but when it goes low, we enter into state 1 which is the Green Main Initial State. Without the Sensor signal we stay here for 12 seconds which is shown by 12 clock cycles from our seconds clock. We also see the initial counter incrementing to the correct value needed to cause this transition at 12 seconds. We then stay in the Yellow Main state for 2 seconds before moving into what used to be a red light state. However, this red light state, which was state 3 was redundant and was removed since it wasted 1 second of a clock cycle through the transition. In either case we also have been setting green light on in the green main state, the yellow light turned on in the yellow main state, and the red light turned on in the red state which would later be turned on when we transition to making the side street green light on or during the walk signal. We continue to move through the normal cycle and state and turn on the appropriate Green Side light, Yellow Side light, and Red side light. This was our initial test to make sure the lights turned on which they did and that nearly every state transition possible happened which occurred except for the WALK state.

* 1. **Demonstrates State Transitions, Correct Time Intervals, Traffic Sensor, WALK, Correct Street Lights on/off**

****

The waveform above shows from the left edge that we have appropriately set Red Main Light on whenever Green Side or Yellow Side lights are on and in their respective states. We also see that Yellow Street Light turning on is timed to happen exactly on the seconds clock rising edge along with the current state transition from 4 to 5 demonstrating correct state update timing. Further, the yellow light states on for two seconds where we then see the state transition from 5 to 7 as well as the red signal turning on. This demonstrates that our states are following the appropriate 2,3,6 second intervals for changing states as well as the appropriate signals turning on or off in conjunction with those transitions.

To show the correctness of our Traffic Sensor Signal and our Green Extended State we catch the Signal before the 6 seconds mark of the Green Main State and set the next state to state 8 which is GME when 6 seconds of GMI have passed. We stay in the GME state for 3 seconds before setting the next state to state 2 and transitioning into state 2. State 2 is our Yellow Main state and we again see that our green main light has now shut off and the yellow main light has turned on.

Now we get to see the functionality of the Walk signal again. Notice that Walk Button being pressed (going high) instantly triggers Walk Pressed and walk\_on which were the two extra leds we set to make sure we saw the lights turned on in our project since we had debouncing issues earlier in the project. We utilized these lights and this testbench to show that the signal was being detected and the professor even made a comment about this when she was staying after class with Chris one day trying to help him figure it out. Because of these signals being set we also see the Walk Enable get set which updates the next state to the WALK state. When we enter state 9 we are in the WALK state which lasts for 3 seconds while we change both Red Main and Red Side lights on. After the three seconds we resume changing the to green side street light on and we handle the resetting of walking signals going back to zero in the code.

* 1. **Testbench Code Snippet**

//…module instantiation, port declarations, uut instantiation…//

initial begin

clk = 0; rst = 0; Sensor = 0; WalkButton = 0;

#15

rst = 1;

#20

rst = 0;

#3465

Sensor = 1;

#100

WalkButton = 1;

#100

#1435

rst = 1;

#500

rst = 0;

#200

$finish;

end

always begin

#10 clk = ~clk;

end

Notice in our test bench that we begin with all signals initialized to 0. Then we set reset as shown above and then set it to 0 to start the state transitions. We made it through all of the cycles at least once using the wait signal of #3465, thereafter we set the sensor on and walk button to trigger the state transitions into Extended Green light States and the Walking State after the Yellow Light. We waited another #1535 to continue moving through states ensuring accuracy before resetting to 1. When reset was set to 1 we moved right back to the initial state and all signals were sent to 0 again. We then turned reset back to 0 after #500 delay where we waited for another #200 delay to make sure the program continued running state transitions. After ensuring all of the correct behavior we used $finish to end the waveform runtime length.

* 1. **Overview of Tests and Discussion of Professors Demo Testing**

The first waveform demonstrated that we were able to run through the different states of the code and in each state we turned on the correct corresponding signal lights, however we did not turn them off. We also were able to achieve correct timings on the intervals of each state in time intervals of 2,3,6 seconds. The initial Green State could do the 12 seconds initially if uninterrupted as shown. We also showed how the reset signal will reset the board since the don’t cares turned into green values when reset was turned off.

The second waveform then demonstrated that we appropriately turned on and off the light signals as we changed states. The correct time intervals were still shown as well with the states. We further showed the ability of the Green Main State to access the 9 second long green light by entering the Green Main Extended State after 6 seconds. Whenever one street green light or yellow light is on, the other street should be red. When we enter the WALK state we also turn both lights red and stay there for three seconds, after which we clear the signals of walk.

Lastly, we demonstrated the full capabilities of our program to the professor in class. She ran through different cycles of the FPGA board and had Chris time each cycle. One test was to run through the normal cycle of 12 seconds Green Main, 2 Seconds Yellow Main, 6 Seconds Green Side, 2 Seconds Yellow Side. Another test included the walk signal which triggered after the yellow light. The final test was to have the initial state trigger the 9 second interval based on catching the sensor which we applied a fix to do that thinking that the very beginning of the traffic controller started with 12 seconds instead of being able to do 9. This was verified again by the professor who signed off on our results.

1. **Conclusion**
   1. **Challenges**

Our challenges in this lab first came in the form of the design, however because of working through the shifter problem in Lab 1 Chris felt confident he could write out a state transition diagram. Chris and Rodrigo discussed the lab document together as they asked questions about the lab's specs and reasoned out most of their concerns. A walk state transition for the side street was in question, but was tossed out as a result of this. Later on we also figured out that it was redundant to make a state for red signals since we immediately switch from a yellow main state to a green side state or a yellow side state to a green main state. When switching we would change the red light anyways and it was resource consuming and redundant to account for the time delay and other overhead costs of switching into an extra unneeded red light state. After fixing these, the general state transition diagram we made was sufficient to complete the lab correctly, but we believe that the initial state could be removed to be more efficient with proper coding adjustments.

Another challenge was getting the counters or state transitions to move. We made a ton of progress in the first class session we worked on in this lab by thinking through the problem and creating a state transition diagram before coding the state transitions with a switch statement. However, when we ran the testbench on the second lab session for this project we noticed it was not working. Both partners stayed after class for a bit to try and figure it out, just shortly after Rodrigo left Chris realized there was a race condition. This is when we were able to get our first test bench waveform for the project.

Another challenge was handling the debouncing logic. Chris did not take the recent M152A class, rather he took it several quarters ago where they had a vending machine state transition project, but did not have debouncing logic since the class was online during COVID. So, we relied heavily on Rodrigo and the previous M152A class he recently took to provide and explain the debouncing logic. It took a few class sessions, but the professor offered good advice on setting a register to hold the value. On a side note the professor saw Chris staying after class one day trying to figure this problem out for a while and came up to ask questions and offered more advice and insight. She also verified that the testbench was correctly showing results based on walk signals and that the testbench entered into the walk state even when the FPGA did not. It took another session or two, but Rodrigo was eventually able to properly code and connect different registers in a way that saw our three debugging leds turn on via button press, walk signal saved, and the walk lamp turning on.

The last challenge was more of a missed detail or unclear part of the assignment specs. The professor when doing the demo with us mentioned that on the FPGA startup we should be able to detect the 9 second green light instead of the 12 second green light if the traffic sensor switch was on. We talked to her about it and she cleared up the confusion, so we implemented that feature and she verified the correct functionality and approved our design.

* 1. **Contribution**

Both members were very involved, helpful in completing this project, and happy to work together. We always distribute the workload evenly and try to utilize our strengths while taking both team members' inputs to create great results. Rodrigo has always been punctual and has offered many useful contributions in coding the projects. Chris offered to stay after class several times to try and solve problems which he was able to do for the state transitions and race conditions, but Rodrigo was crucial to figuring out the debouncing issues and fixing them. Although Chris designed the initial state diagram, both Rodrigo and Chris discussed each aspect and decided to remove an unnecessary state and apply light switching signals upon state transitions to save on resources used in coding. Rodrigo and Chris both made edits to the Basys3 xdc file when trying to debug the Walk signal and setting various leds to communicate to the programmers when a button was pressed, the signal for walking is held, and when we enter the walking state via turning the walk lamp on. Rodrigo created the clock divider for our seconds clock and also created the test bench for it. Chris created the test bench for the top module of the traffic controller, but both Rodrigo and Chris edited this file and discussed different configurations on a good way to set up the waveform to see the state transitions cycle once and reach the WALK state and the Extended Green Light State. Both members had also communicated outside of class about modifications and additions they thought about making to the project throughout the process. Lastly, both team members have been rotating working on lab reports including this document.

This was a cool project that helped revisit important verilog coding aspects as well as important state machine design and functionality while not feeling as lengthy as project1, but sometimes difficult to debug or solve an issue. We really want to mention how appreciative we are of the professor who on several occasions came up to us during class or after class to ask us what we needed help with when she saw we had a problem or other times where she gave advice for things she wanted us to take away or understand in the project.