Alberto Diaz - UID: 604967268

Evan He - UID: 705619333

Brandon Truong - UID: 705326387

Lab 2 Report

### Introduction

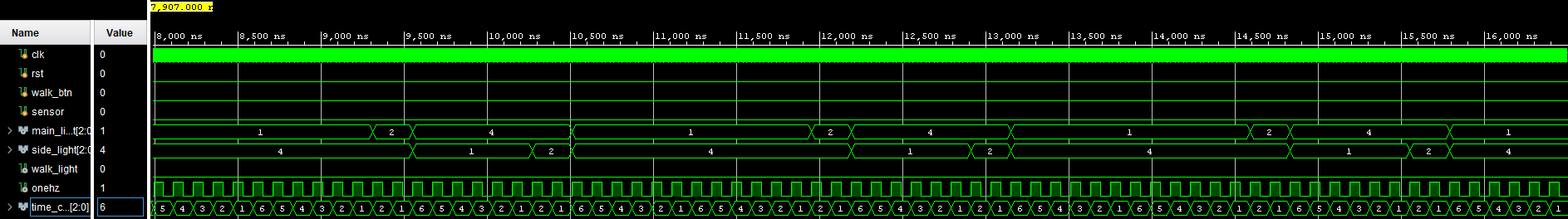
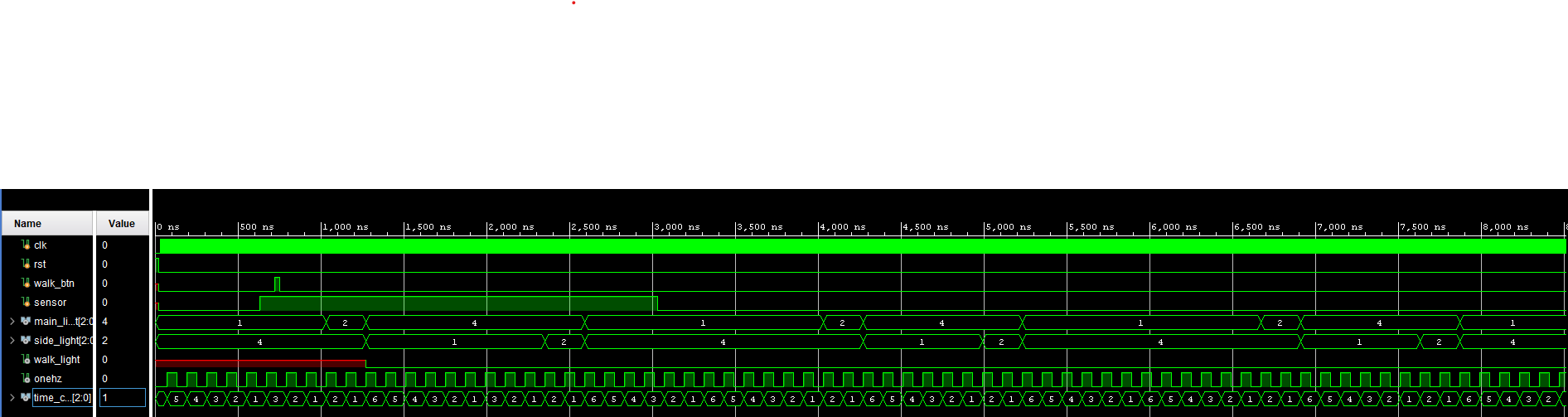
In this project, we were asked to simulate and program a traffic light finite state machine design onto the Basys3 board.   
  
Since this involved the fpga board as well, we utilized these board components:  
  
**clock:** internal 100 Mhz clock in order to make time-based functionality

**switches:** we used the two rightmost switches in order to add sensor and reset functionality  
**leds:** we used three leds on the left to signify red, yellow, green for the main light, and another 3 on the right for the side light. Another 3 were bounded to the time\_counter for easier time measurement, as well as walk\_light and the one hz clock.  
**button:** we utilized one button for the walk functionality, making sure to debounce it with our code.

### Design

**clockdiv.v:** For our one hz clock divider, we created an always block that runs on the posedge of the internal clock. This block increments a counter by one for every iteration, until it hits 49999999, in which the state flips from either low to high, or high to low. The math is . It also takes in a reset input, which is prioritized over all functionality and turns everything back to the initial.   
  
**debouncer.v:** For our debouncer, we implemented a debouncer using two d flip flops, allowing the two d flip flops to make a single pulse. This was done by creating another clock div running on a state flip of 124999, which is equal to . This time allows us to have a long enough pulse that it can be registered as a single flip, ignoring the initial button press where chattering can happen. We then set up the d flip flops, where one runs based on the input of the button, and the other runs based on the output of the first d flip flop. We then negate the second d flip flop, and get the and of the first with the negated second. This allows us to make a single pulse that lasts for 0.0025 Hz, which only has one positive and one negative edge. **debouncer\_test.v:** For our debouncer test, we tested to see if the button being pressed would create a single pulse regardless of how long the button was held for. **traffic\_light.v:** For our state machine, we implemented all the previous modules, and created an always block that runs based on the positive edges of the one hz clock, debounced walk button, and reset. We always check for reset first, changing every value back to the initial if so. Otherwise, we check for the walk button and set the register to indicate a queued walk. We implemented the changing of states by using a time counter that decrements by 1 every second, and gets reset to the next state’s value. Afterwards, we have three main sections:

1. Next\_state calculator: For this section, we check the current state, as well as walk and sensor inputs to determine what the next state is. We did this by using a switch statement, and manually set the conditions of when each state changes.  
  
In between this section is where we change to the next state  
  
2. Time\_counter calculator: We then calculated what the duration of the state is here, changing time counter to whatever the value should be for that state through a switch statement.  
  
3. Light calculator: For this part, I set the lights based on the state with a switch statement, grouping up the ones that are able to be grouped.

**testbench.v:** For our testbench, we linked the modules together with a simulated clock, and tested the sensor and walk buttons.  
  


[source image here:](https://drive.google.com/drive/folders/1J3Jd49pTHUMSyYSmM0z5S9jLnooQgsxG?usp=sharing)

### Problems and Solutions

One of the main problems we ran into was the board not being able to handle our initial design, despite the simulation running as expected. This occurred as a result of improper design, where we changed a variable in two always block, resulting in undefined board behavior when we tried to program it.  
Our solution for this was to arrange the code into a single always block, which works, but is a lot more illegible than our original solution. We could further improve this by reorganizing where things get defined, and then separating them to their own always block, but we lacked time in modifying it.  
  
Another problem we ran into was double state input, which we found out was being caused by non-blocking vs blocking assignments. Since we had set the next\_state in parallel with state, this resulted in incorrect behavior since next\_state is reliant on state, and vice versa.  
We fixed this by making all next\_state and state assignments blocking, fixing our issue.

### Contributions

Alberto - Worked together to design the FSM and supporting modules.

Evan - Worked together to design the FSM and supporting modules.

Brandon - Worked together to design the FSM and supporting modules.