# Verilog Intersection Controller

Kathryn Hite & William Lu

December 15, 2016

## 1 Abstract

This project uses Verilog to simulate the traffic flow and control at a four-way intersection through the structural implementation of an application specific integrated circuit (ASIC) for traffic control.

To achieve this, we built many smaller simulated hardware modules that we wired together to create the intersection controller. These included things like timer modules and multiplexer modules designed to blink lights or choose between red, yellow, and green. Each module was built and tested individually before we integrated it into the the full intersection controller.

Our intersection controller currently works over a 4-way intersection with cars and pedestrians. It passes through timed states based on the settings from binary input switches that would be located on the outside of the physical module for setup. The system is also built such that it can easily be extended to add features like turn lanes, traffic sensors, and signal synchronisation with other intersection controllers.

## 2 Motivation and Background

Intersection control is imperative for safe transportation infrastructure and requires increasingly smart and adaptive control given complicated intersections, heavy traffic flow, and the inclusion of pedestrians, bicycles, and other vehicles. The optimisation problem of how to best control an intersection has been rigorously studied, and the problem itself continues to change as newer technology, like neural networks, comes into play. We wanted to see what a somewhat more "traditional," hardware based solution might look like for intersection control and investigate what we might be able to do to extend the hardware's functionality.

## 3 Structure and Modules

### 3.1 Structural Overview

Our intersection controller is split up into different modules, with each module accomplishing a specific task. For each kind of signal light at our 4-way intersection, we have a module that controls the state of that light. We also have an general up counter that will count to any specified 17 bit integer in order to facilitate light blinking and state switching in the finite state machine that ultimately controls the intersection.

## 3.2 17 bit Up Counter

This module is included in several other modules in the intersection controller to time certain things, like the blinking of an LED. It is designed to take in a 17 bit integer that it will then count to. When the target number is reached, the counter will flip the state of its output, which initially starts pulled low. This effectively allows the use of this up counter as a frequency divider if necessary.

### 3.2.1 Specification

This up counter counts whenever the clock goes high and is capable of counting up to just over 130 seconds with a 1 Khz clock.

### 3.2.2 Inputs

- **clk:** System clock for the up counter to count by.
- **enable_wire:** When high, allows the up counter to increment by 1 on the positive edge of every clock cycle.
- **reset_wire:** When high, resets the up counter back to 0. This will hold the counter at 0, even if enable is high.
- **count_target_wire:** 17 bit integer input that tells the up counter what number to count to.

### 3.2.3 Outputs

- **count_result:** Toggled high or low based on whether or not the up counter has counted to the target number.

## 3.3 Blink Controller

This module is effectively a frequency divider that toggles between high and low at 2Hz to allow lights in the intersection to blink at 2Hz (e.g. the array of orange "don't walk" LEDs in the pedestrian signal controller).

### 3.3.1 Specification

This module toggles an output high and low at a rate of 2Hz when given a 1Khz clock.

### 3.3.2 Inputs

- **clk:** System clock for the blink controller to count by.
- **enable_wire:** When high, allows the blink controller to increment by 1 on the positive edge of every clock cycle.
- **reset_wire:** When high, resets the blink controller back to 0. This will hold the counter in the blink controller at 0, even if enable is high.

### 3.3.3 Outputs

- **blink:** Toggled high and low at a frequency of 2Hz when this module is given a 1Khz clock signal.

## 3.4 Car Signal

This module is used to control the color of a standard red, yellow, and green traffic signal and is instantiated once for every standard traffic signal present. In the case of our 4-way intersection, there are 4 of these traffic signals.

### 3.4.1 Specification

This module is primarily a 1:4 demultiplexer that takes an input that is constantly driven "high." Although there are 4 outputs in the demultiplexer, the fourth output is left disconnected to facilitate an "off" state for the light.

Table 1: Function table for the car signal controller module.

| **in** | **sel** | **r LED** | **y LED** | **g LED** |
|---|---|---|---|---|
| 1 | 2'd0 | 1 | 0 | 0 |
| 1 | 2'd1 | 0 | 1 | 0 |
| 1 | 2'd2 | 0 | 0 | 1 |
| 1 | 2'd3 | 0 | 0 | 0 |

### 3.4.2 Inputs

- **fsmCarControl:** The signal from the main intersection controller that selects what light the car signal turns on.

### 3.4.3 Outputs

- **leds:** Four wires, each corresponding to a different LED (or the off state).

### 3.4.4 Schematic

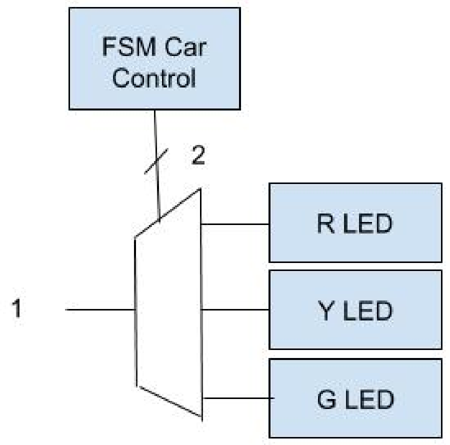

Figure 1: Each car signal is a 1:4 demultiplexer that switches between red, yellow, and green LEDs.

## 3.5 Pedestrian Signal

This module is used to control the color of a standard American white "walk" and orange "don't walk" pedestrian traffic signal and is instantiated once for every pedestrian traffic signal present. In the case of our 4-way intersection, there are 4 of these traffic signals.

### 3.5.1 Specification

This module consists of 2 multiplexers, one that controls the array of white "walk" LEDs (person control) and another that controls the array of orange "don't walk" LEDs (hand control). Similar to the car signal module, the 4:1 multiplexer that controls the array of white "walk" LEDs (person control) has a disconnected wire that isn't shown.

Table 2: Function table for the white "walk" LEDs in the pedestrian signal controller module.

| $I_0$ | $I_1$ | $I_2$ | $I_3$ | **sel** | **hand LED** |
|---|---|---|---|---|---|
| 0 | 1 | blink signal | 0 | 2'd0 | 0 |
| 0 | 1 | blink signal | 0 | 2'd1 | 1 |
| 0 | 1 | blink signal | 0 | 2'd2 | blink signal |
| 0 | 1 | blink signal | 0 | 2'd3 | 0 |

Table 3: Function table for the orange "don't walk" LEDs in the pedestrian signal controller module.

| $I_0$ | $I_1$ | **sel** | **person LED** |
|---|---|---|---|
| 0 | 1 | 1'd0 | 0 |
| 0 | 1 | 1'd1 | 1 |

### 3.5.2 Inputs

- **blink:** Input from the blink controller so that the "don't walk" LEDs can blink.
- **fsmHandControl:** The signal from the main intersection controller that selects what state the "don't walk" LEDs are in.
- **fsmPersonControl:** The signal from the main intersection controller that selects what state the "walk" LEDs are in.

### 3.5.3 Outputs

- **handLED:** A wire for the array of LEDs that makes up the "don't walk" signal.
- **personLED:** A wire for the array of LEDs that makes up the "walk" signal.

### 3.5.4 Schematic

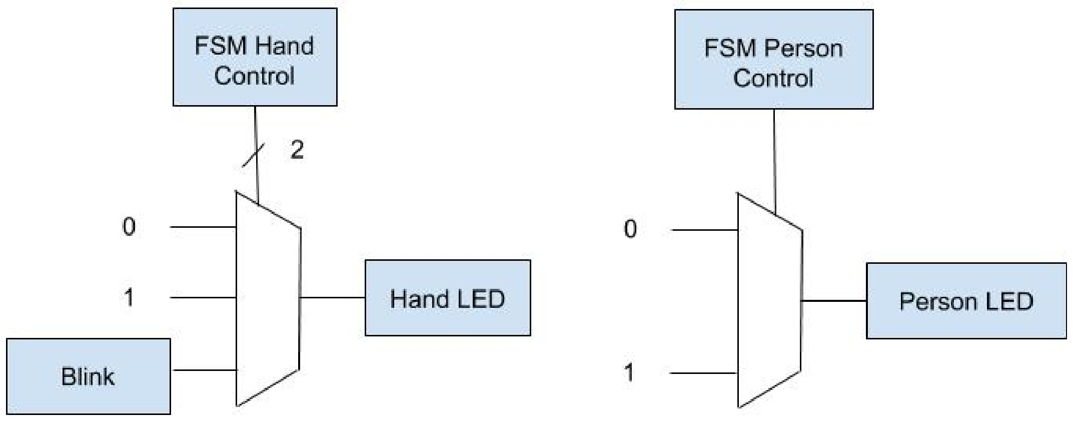

Figure 2: The pedestrian signal includes a walking person LED and a hand LED controlled by an external blink timer.

## 3.6 Intersection Controller

Predefined states with timing inputs control what state each light in our intersection is in at any given time.

The following example in Figure 3 intersection shows the simplest controller model with four car signals.

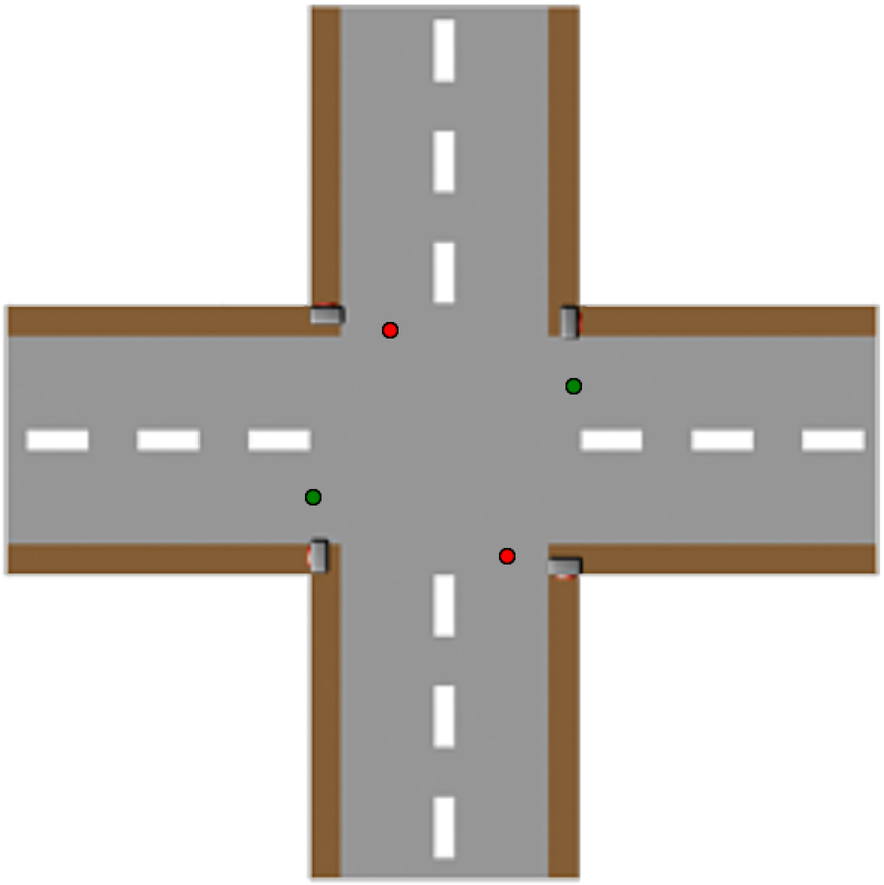

Figure 3: Simple example intersection

From this basic model, we can plug in other signal modules, button inputs, and smart sensors to control more complex intersections that adapt to traffic in real time.

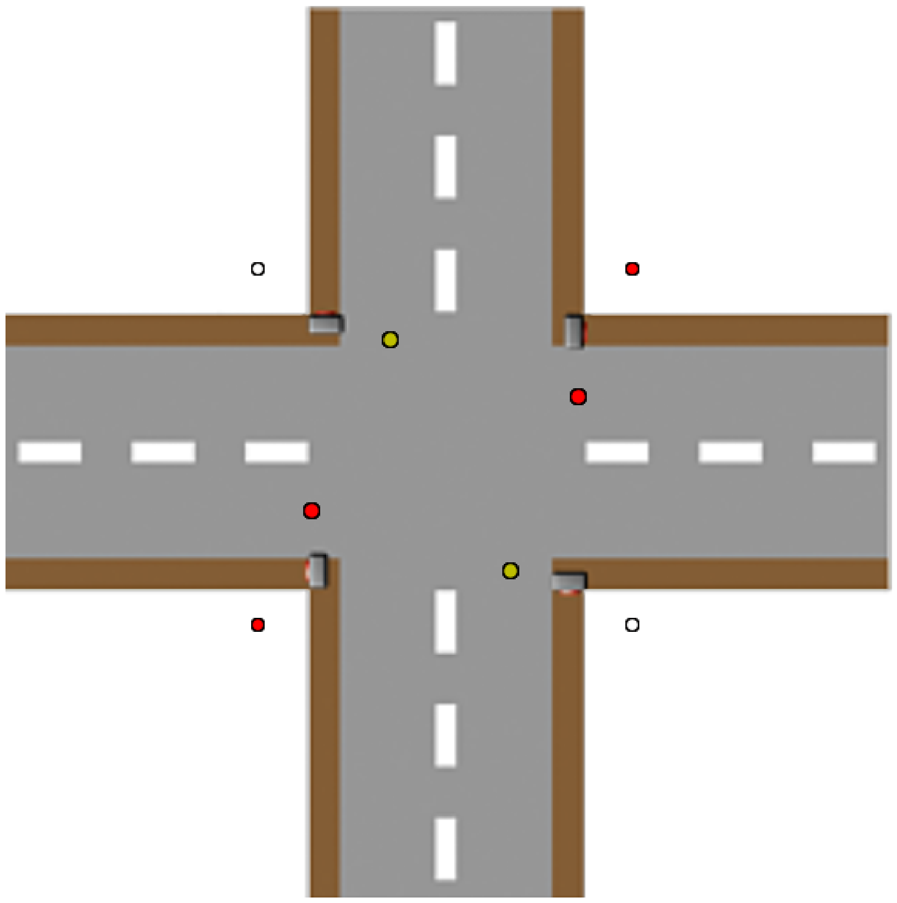

Figure 4: A more complex example of intersection control includes pedestrian crosswalks with pavement sensors for the cars and buttons to signal that a pedestrian wants to cross.

### 3.6.1 Specification

The finite state machine that controls the lights in the example intersection is outlined in the table below. Example times are given though the intersection implementation uses binary switch inputs to allow the timing to be set physically on the controller. The states are followed sequentially, with D looping back to A once it is complete to restart the state procedure.

Table 4: Simple Intersection Controller Example

| **Light** | **A** | **B** | **C** | **D** |
|---|---|---|---|---|
| | 30 sec | 10 sec | 30 sec | 10 sec |
| L1 | Green | Yellow | Red | Red |
| L2 | Green | Yellow | Red | Red |
| L3 | Red | Red | Green | Yellow |
| L4 | Red | Red | Green | Yellow |

### 3.6.2 Inputs

- **personTime:** Binary timer input for the person LED on pedestrian signals.
- **handTime:** Binary timer input for the hand LED on pedestrian signals.
- **redTime:** Binary timer input for the red LED on car signals.
- **yellowTime:** Binary timer input for the yellow LED on car signals.
- **greenTime:** Binary timer input for the green LED on car signals.

### 3.6.3 Outputs

- **leds:** Each set of LEDs comes from one of the instantiated signal modules in the particular intersection.

### 3.6.4 Schematic

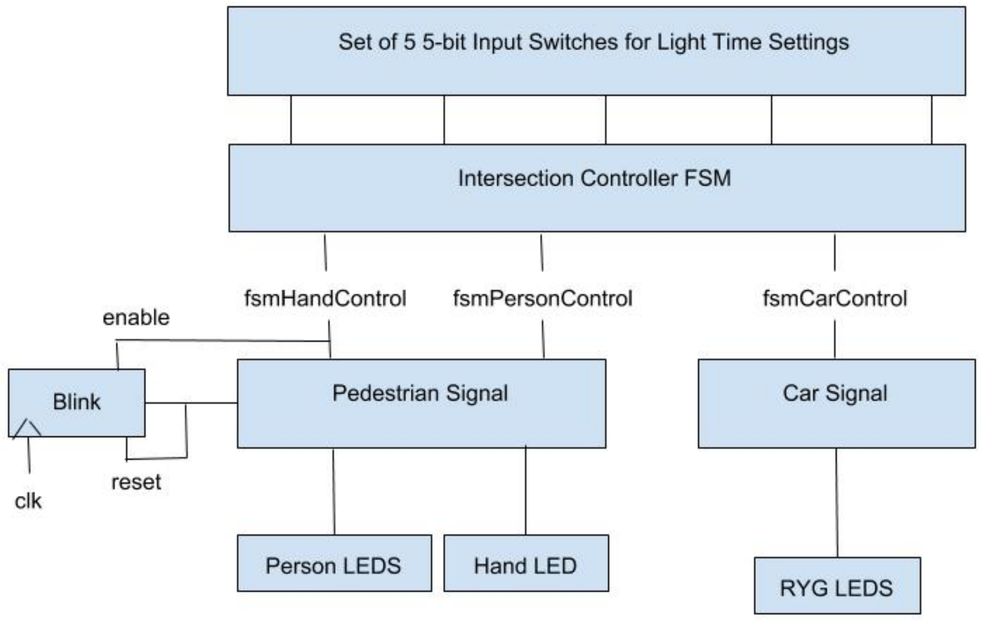

Figure 5: This sample intersection controls one car signal and one pedestrian signal, but the controller module can be used with as many signals as necessary for a given intersection.

## 4 Future Work

The structural decisions in each module support creating almost any combination of signal and sensor types. To build off of this project, these sensor outputs could be used in the python visualization to run a real time traffic controller. This would update the signal timing at each clock cycle to ensure that the intersection fits the current traffic rate.

Another potential addition would be to use Python to easily generate hundreds of instances of the intersection controller. This would allow full city modelling and could be used to verify or improve current traffic control settings.