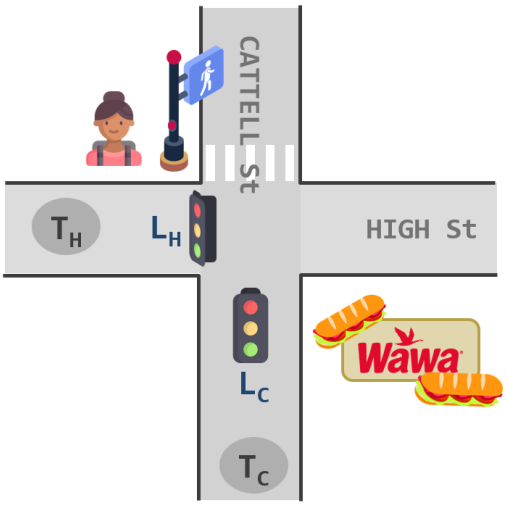
Lab 10: Traffic Light Controller

ECE 211: Digital Circuits I — Spring 2025

Abstract. In this lab, you will design and implement a traffic light controller for a busy intersection on campus: the intersection of Cattell Street and High Street. You are planning to install two traffic lights, L_C for Cattell Street and L_H for High Street. Each light is made up of three individual LEDs that you can turn on and off, $L_{C,Red}$, $L_{C,Green}$, and $L_{C,Yellow}$, for example. In addition, you plan to use two traffic sensors, T_C for Cattell Street and T_H for High Street, to determine when to change the traffic signal.

You prototyped an initial design but found that it did not allow you to easily and safely cross to Wawa. So, in your new design, you will modify the traffic signal so both lights will be red before changing a signal to green. You also added a flashing crosswalk button on Cattell St, called **`button`**. When pressed, this button will make L_C red to give priority to the pedestrian crossing to Wawa.



This lab assignment extends the traffic light controller discussed during the lecture. In addition, this lab assignment will introduce you to handling a button input, which must be remembered until it is used to influence the transition of the finite state machine.

Objectives.

- Gain experience designing sequential circuits using finite state machines
- Handle a button input by storing it until it is used by a synchronous sequential circuit
- Use a clock enable signal to slow the operation of a particular synchronous component
- Test the operation of individual circuit components using testbenches
- Test the operation of the circuit using the input and output components on the Nexys A7

Materials and Equipment.

- Nexys A7-100T FPGA board with USB cable

Deliverables.

- In-person lab demonstration (per team)
- Lab report (per team)

Part 0: Set Up a Vivado Project

To begin the lab, we will set up a new Vivado project for Lab 10 and configure the Nexys A7 board. You can reference Lab 1 for a more detailed description of these steps.

1. Create a Xilinx Vivado project and import the provided constraints file and SystemVerilog modules. This lab will use both simulation and design sources. The file **`lab010_top.sv`** should be uploaded as a design source. The file **`lab010_tb.sv`** should be uploaded as a simulation source. You will use this file as a testbench to write test cases for each module.

2. In this lab, you will primarily use Vivado's simulator to verify your design at each part, and then use the FPGA board at the end to demonstrate the entire circuit. If you would like, you can connect your FPGA board and open the hardware target at this point in the lab.

Part 1: Design the Finite State Machine (FSM)

In the first part of this lab, you will design a finite state machine that implements the behavior of the traffic light controller specified above. Your finite state machine will differ from the lecture example in two ways: *1)* Your FSM will include a state where both traffic lights are red to avoid accidents, and *2)* Your FSM will handle an additional input, **`button`**, that, when pressed, will turn the Cattell St. light to red to allow a pedestrian to cross safely to Wawa.

1. Sketch a state transition diagram that implements the traffic light controller.
   a. Determine the number of unique states *(Hint: If you are implementing a Moore machine, the number of states is synonymous with the number of possible output combinations you have)*
   b. Add these states to the state transition diagram
   c. Draw transitions between each state depending on the input signals

2. Choose a state encoding scheme for your design

3. Create a truth table from your state transition diagram.

4. Depending on your design, your truth table may be ***too large*** to derive Boolean equations directly! There are three inputs to this system. If you have four bits of state, you will have 2^7 = 128 rows in your truth table!
   a. You may opt to derive Boolean equations for each bit of *next state* and each *output*. If you do this, include your Boolean equations and a sketch of the circuit in your lab report.
   b. If you cannot derive Boolean equations directly because the truth table is too large, only include your truth table in the lab report. Implement your FSM using if-statements in SystemVerilog. (This is representative of how FSMs are designed in practice!)

Record: Record your state transition diagram and truth table in your Lab Report. *(Optional: Include Boolean equations and a circuit schematic if you have chosen to derive them)*

Part 2: Set Up your Synchronous Traffic Light Controller Module

Organize a SystemVerilog module to implement your traffic light controller. The following steps take you through organizing the module to handle clock updates and button presses, in advance of implementing the state transition logic. Your module will use two new elements: a ***clock enable signal*** and ***storage of a button press***.

For a circuit to be synchronous with a clock, it is best practice to have all flip-flop components driven by the same exact clock signal. In previous labs, we used a clock divider module to slow the FPGA board's 100 MHz clock down to a few hertz. Due to gate delays in the clock divider, the rising edge of this slow clock may arrive *later* than the rising edge of the 100 MHz signal. Because of this slight skew in the rising edge, a flip-flop driven by the 100 MHz clock may activate earlier than a flip-flop driven by the slow clock. Thus, the two flip-flops are no longer perfectly synchronized. In practice, clock dividers are avoided for this reason.

This lab will use flip-flops activated at different frequencies so we will use an ***enabled flip-flop*** instead of a clock divider to update the current state at a lower frequency. The enabled flip-flop is driven by the central 100 MHz clock. However, the flip-flop will only update its state when its **`enable`** input is HIGH. By making the **`enable`** input operate at a low frequency, we can achieve a slow update to the flip-flop without messing with the original clock signal. The steps below describe implementing the clock enable and storing a value of the button press.

3. Create a SystemVerilog module for your traffic light controller.
   a. Add a new source file to your project for `traffic_light.sv`
   b. Your module should take a scalar reset signal and a scalar clock signal, as well the following inputs: $\mathbf{T_C}$, $\mathbf{T_H}$, and **`button`**.
   c. Your module should produce six 1-bit scalar outputs, corresponding to each segment of the traffic signal being on or off: $\mathbf{L_{C,Red}}$, $\mathbf{L_{C,Green}}$, $\mathbf{L_{C,Yellow}}$, $\mathbf{L_{H,Red}}$, $\mathbf{L_{H,Green}}$, and $\mathbf{L_{H,Yellow}}$

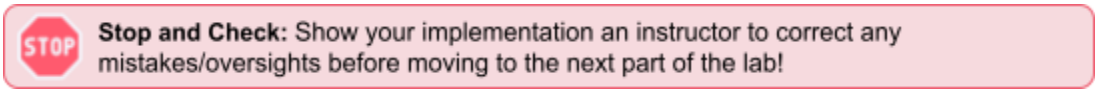
4. Instantiate any internal signals in your traffic light controller module, including **`current_state`**, **`next_state`**, and **`enable`**.

5. Configure your traffic light controller to update its state based on a ***clock enable*** signal.
   a. Your implementation will drive all the flip-flops using the system's centralized 100 MHz clock. However, this frequency is too fast to safely operate a traffic light. Thus, we will use an ***enabled flip-flop*** to update the traffic light's state at a lower frequency. This enabled flip-flop will only update its state when its **`enable`** input is HIGH. If the enabled input is low, the current state will not change.
   b. Configure your `traffic_light` module to correctly use the **`enable`** and **`reset`** signals. Specifically, your SystemVerilog code should do the following:
      - On the rising edge of a clock tick, if **`reset=1`**, the current state should be set to the default state (*e.g.,* State 0)
      - Otherwise, if **`enable=1`**, then the current state should update to the next state
      - Otherwise, the current state should not change (This case occurs if both **`reset=0`** and **`enable=0`**)
      - If implementing this logic using an if-statement, remember to ensure that the current state is assigned on all paths to avoid latch inference!

6. Generate an internal **`enable`** signal to slow the operation of the traffic light.
   a. Given Step 5, the FSM state will update based on the enable signal, rather than the 100 MHz clock. This will allow us to operate our traffic light controller at a slower speed than the system's clock without introducing any clock delays.
   b. We can generate the **`enable`** signal inside of the `traffic_light` module. The provided `clock_enable` module can be parameterized to generate an enable signal that you will use to drive the traffic light.
   c. The period of the enable signal is set via the **`PERIOD_NS`** or **`PERIOD_MS`** parameter. For testing your design in simulation, use a short period like 50 ns, as shown in the following code snippet. When synthesizing your design, you will need to increase the period significantly to match the desired frequency of switching the traffic signal.

```
logic enable;  // Declare enable signal
clock_enable #(.PERIOD_NS(10))    D0(.clk(clk), .reset(reset),
                                     .enb_out(enable));
```

7. Store the value of **`button`** in a flip-flop until the next **`enable`** pulse.
   a. Now that your traffic light operates at a slow frequency, you will need to *remember* the value of a brief button press until that button is used to update the FSM state.
   b. We can remember if the button was pressed using a flip-flop! For the purpose of this example, let's call the flip-flop's output **`person_waiting`**. Extend your `traffic_light` module to do the following:
      - On the rising edge of a clock tick, if **`button=1`**, then **`person_waiting`** should be also set to **`1`**
      - After the value of **`person_waiting`** is used for an FSM transition (*e.g.*, after the current state is updated to the next state), the value of **`person_waiting`** is cleared.
      - This functionality will ensure that the button press is remembered and used in the next transition of the FSM
8. You may opt to simulate your circuit to verify that an enable signal is generated and a brief button press is held until the next enable pulse.



Part 3: Implement your Finite State Machine by Deciding Next State

In this part of the lab, you will implement your traffic light controller FSM as a SystemVerilog module. The current state of your FSM will be stored using D flip-flops. In SystemVerilog, this functionality is implemented with the **`always_ff`** block. The next state of your FSM can be computed using combinational logic in an **`always_comb`** block. An example of an FSM is shown in Figure 1.

```
module example_fsm(  input  logic rst, clk, output logic light);

   logic [1:0] current_state, next_state;

   // Update state on rising clock edge (no reset or enable)
   always_ff @(posedge clk)
   begin
     current_state <= next_state;
   end

   // Compute next_state with combinational logic
   always_comb
   begin
     if(current_state == 2'd0)
         next_state = 2'd1;
     else if(current_state == 2'd1)
         next_state = 2'd2;
     else
         next_state = 2'd0;
   end

   // Compute the output with combinational logic
   always_comb
   begin
     if(current_state == 2'd0)
         light = 1'd1;
     else
         light= 1'd0;
   end
endmodule
```

Figure 1. Example FSM with Output in SystemVerilog

9. Implement your finite state machine design within the `traffic_light` module.
   a. Implement your finite state machine design. Use either the Boolean equations or if-/case-statements to implement your state transition logic
   b. Implement your output logic ($L_{C,Red}$, $L_{C,Green}$, $L_{C,Yellow}$, $L_{H,Red}$, $L_{H,Green}$, and $L_{H,Yellow}$) using either Boolean equations or if-/case-statements.
10. You will test your implementation in the following section.

Part 4: Test your Traffic Light Controller in Simulation

In this part of the lab assignment, we will simulate our traffic light controller using a testbench. Figure 2 shows a sample organization of the Vivado Wave View to improve its readability. Dividers are used to separate inputs and outputs for each traffic signal. Additionally, the traffic light signals are colored to visually indicate the current color of the traffic signal.

The next page provides a walkthrough of a simulated traffic light controller with the enable signal and button press. Reviewing this walkthrough can help you understand the circuit operation.

Figure 2. Example Organization of Traffic Light Controller Waveform

11. Test your traffic light controller using a testbench.
    a. In `lab10_tb.sv`, instantiate a traffic light module
    b. Generate a clock signal for your module and initialize the module using the reset signal in the first clock period. Refer to Lab 8 for additional information about generating clocks in simulation to test sequential circuits
    c. Configure your simulation so that the traffic light will cycle through most or all of its states

12. Simulate your circuit and verify your circuit's operation using the Wave view or by printing messages to the Tcl Console

13. Verify that your tail light module implements your finite state machine properly before moving to the next part of this lab assignment.

Stop and Check: Using the Vivado Simulator, ensure that your module works correctly and is triggered on the rising edge of the clock pulse.

Record: Screenshot the waveform of your test to include in your Lab Report.

Example Circuit Simulation. The circuit operation is driven by a clock (**clk**) and an enable signal (**enable**). All circuit components are connected to the same clock signal. However, the enable signal dictates when the FSM will transition to the next state. The enable signal allows the FSM to operate more slowly than the rest of the circuit, while still keeping all flip-flop components connected to the same clock. For this example, the clock period is set to 10 ns and the enable signal is triggered once every 50 ns. These periods will change when implementing the final controller.

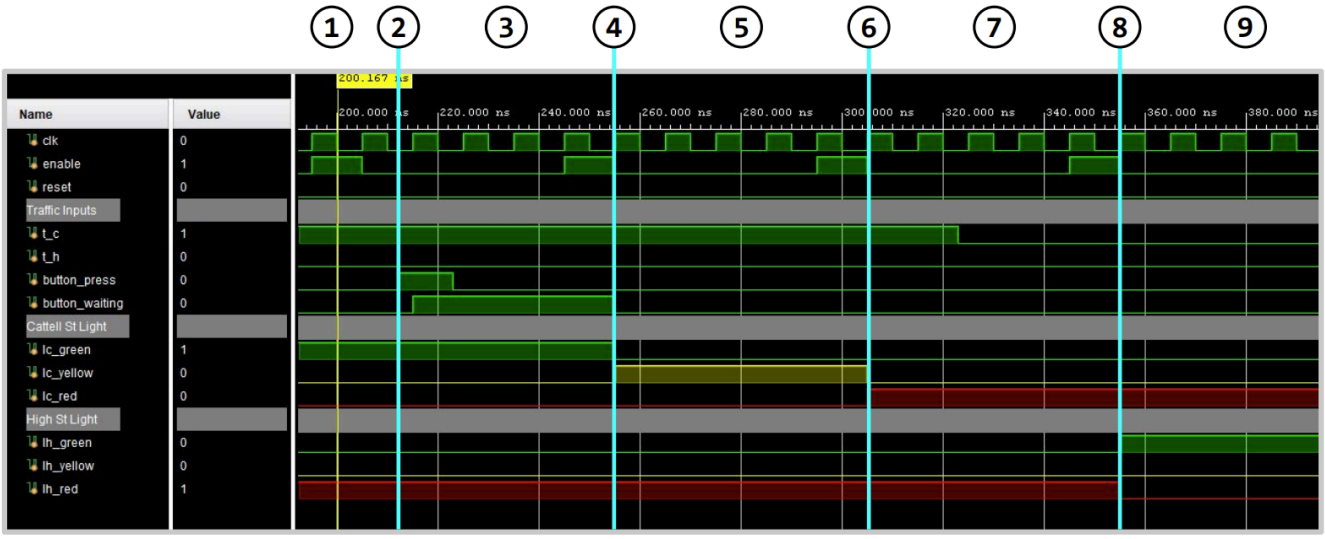


Figure 3. Operation of Traffic Light Controller

(1) Initially, Cattell St. has traffic (**t_c=1**) and the Cattell St. light is green.

(2) At 212 ns, the Cattell St. crosswalk button is pressed (**button=1**) and held for about 11 ns.

(3) The button press is brief and must be remembered until the next update to the FSM! The button press is stored in a flip-flop called **button_waiting**. The value of **button_waiting** will remain at **1** until the next FSM transition.

(4) At 245 ns, **clk** rises from **0** to **1** and **enable=1**. This combination causes the FSM to transition to its next state. The transition is based on the current state, the current traffic pattern (**t_c=1** and **t_h=0**), and whether the crosswalk button has been pressed (**button_waiting=1**). Since the Cattell St. light is green and someone is trying to cross Cattell St., the FSM will transition to the state where the Cattell St. light is yellow.

(5) The Cattell St. light is yellow (**lc_yellow=1**). The light remains yellow until the next **enable** pulse.

(6) At 305 ns, **clk** rises from **0** to **1** and **enable=1**. This combination causes the FSM to transition to its next state. Since the Cattell St. light is yellow, the next state is when the Cattell St. light is red. This transition is automatic and does not depend on the other input values at the time.

(7) The Cattell St. and High St. lights are both red (**lc_red=1**, **lh_red=1**). The lights remain red until the next **enable** pulse.

(8) At 345 ns, **clk** rises from **0** to **1** and **enable=1**. This combination causes the FSM to transition to its next state. The High St. light will become green.

(9) The High St. light is green.

Part 5: Connect your Traffic Light Controller to the Nexys A7 Board

In this part of this lab assignment, you will connect your traffic light controller to the Nexys A7 board. The input signals will come from the switches and buttons onboard the Nexys A7. Switches **SW[1]** and **SW[0]** will be used for T_C for and T_H, respectively. Switch **SW[2]** should be used for reset. The center pushbutton, **BTNC**, will be used as the crosswalk button

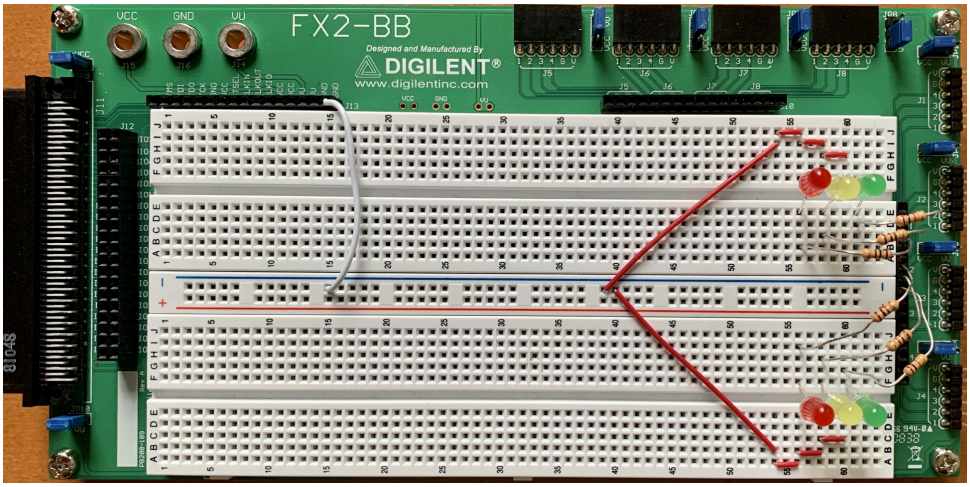


Figure 4. Breadboard Configuration for Lab 10

The outputs will be run on a separate breadboard to mimic two traffic lights, as shown in Figure 4. This breadboard will be wired to the PMOD connectors on the FX2 breadboard. The connections between the Nexys A7 pins, the FX2 pins, and each LED are as follows:

Nexys A7 Constraints Name	FX2 Breadboard Pin Name	LED (In Fig. 3)
JB[2]	J1[3]	Top Green
JB[1]	J1[2]	Top Yellow
JB[0]	J1[0]	Top Red
JA[0]	J2[0]	Bottom Green
JA[1]	J2[1]	Bottom Yellow
JA[2]	J2[2]	Bottom Red

14. In the top-level module (`lab10_top`), instantiate a `traffic_light` module. Connect the module to the board's clock, inputs, and outputs, as described by the problem specification.

15. Verify that the circuit schematic matches your proposed design. You can view the circuit schematic under "RTL Analysis/Open Elaborated Design/Schematic" in the Flow Navigator.

16. Generate a bitstream and program the FPGA. Test that your circuit functions correctly and demonstrate it to an instructor.

Record: Screenshot a schematic for each module for inclusion in your lab report.

Part 6: Wrap-Up

Before leaving the lab, make sure to complete the following steps!

17. ***Demonstrate the operation of your circuit to the instructor.*** A portion of your lab grade will come from the lab report in addition to this demonstration.

18. Turn the Nexys board OFF and disconnect it from the computer. Make sure to save your Vivado project and sign out of your machine!

Lab Report *(Due at beginning of next lab session)*

Lab Report. In collaboration with your lab partner, submit one lab report on Moodle that describes the lab activity and your team's circuit design. Your written report is worth 60% of your grade for each lab assignment. Your lab report should include the following sections:

- **Title Page:** Your lab report should begin with a title block that includes the course number, the title of the lab assignment, the names of each team member, and the date of the lab assignment. The title page should also include a "Statement of Collaboration" (detailed below) and an estimate of the total amount of time spent on the lab assignment. The inclusion of these components are graded, but the content of each statement is not used in determining your grade.

- **Statement of Collaboration:** Describe the contributions of each team member to the lab assignment, including writing of the lab report. What is specified in this section will not affect your grade, however it is expected that you periodically rotate roles within your lab group if you do divide work, including writing the report.

- **Introduction:** Include a brief paragraph summarizing the objectives of the lab and what your team achieved. This section should be written in your own words; it is not acceptable to copy text for this from the lab handout.

- **Design:** Include a concise yet descriptive explanation of the circuit design, following all steps in the process—from the problem specification and initial truth table, to equations and circuit schematics. Tables and equations should be typeset and clearly labeled. Technical information should be accompanied by some text narrating each step or aspect of the design.

- **Testing & Results:** Describe how you tested your circuit, including rationale for why you chose specific test cases if the circuit is not tested exhaustively. If you tested your circuit at multiple points (*i.e.,* using a testbench and then on the Nexys board), this section should include all test cases at each point. Test cases can be included in the form of tables or simulation waveforms. Results should be neatly formatted and labeled, indicating what step is being tested. This section should also include any recorded measurements, tests, diagrams, or images specified by the lab document.

- **Discussion/Conclusion:** Your discussion/conclusion section should comment on any observations about the technique you employed to get your results. Briefly describe any difficulties that you encountered and how you resolved them. Then, summarize what you concluded from your work. (It may also be helpful to think negatively, *i.e.* what your results did not show. Remember that a negative result is often as valuable as a positive result!)

  A number of technical questions may be posed in the assignment sheet. If questions are given, include your answers to them in this section.

Your lab report should be typed and submitted as a .pdf document. Only one member per team needs to submit the lab report.