ECE 281 In Class Exercise 4

Stoplight

Overview

So far in this course you have implemented combinational logic on your FPGA. The ability to implement synchronous logic enables us to make finite state machines and greatly expands your ability to implement useful designs. In order to accomplish this, we will expand our use of processes.

Objectives

The objectives of this in-class exercise are for you to:

- Design the Stoplight FSM in VHDL
- Test the Stoplight FSM in VHDL
- Control a real world miniature stoplight.

Desired End State

You will be able to provide a demo of your simulation and final product to your instructor. You are not required to turn in ICE4 however, you will need to demonstrate functionality and ensure the following files are pushed to your repository:

1. VHDL files used (top_basys3, stoplight, stoplight_tb, clock_divider, clock_divider_tb) in a **code** folder
2. Constraints (.xdc) file in a **code** folder
3. .bit file used to program board

Stoplight FSM

We will design the stoplight to match the description given in Homework 3. The sequential circuit will implement a simplistic traffic light which turns green when a car is present (and stays green while cars are present) and then cycles to red (through yellow) when a car is not present. The system has three outputs (Red, Yellow, and Green) and must be implement with a Moore FSM as seen is Figure 1.

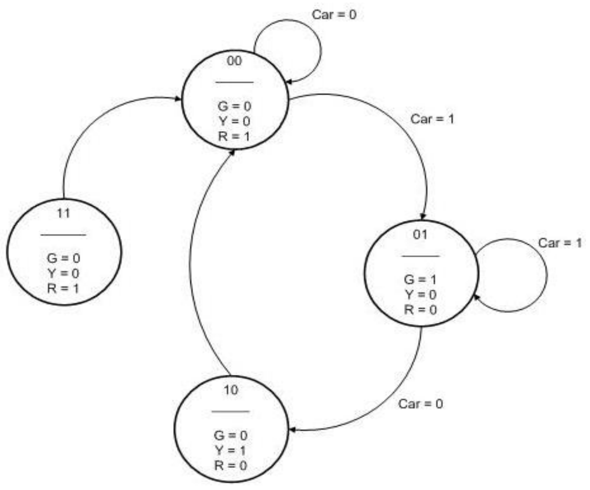


Figure 1. Stoplight state transition diagram

1. Create a Vivado Project

Create a new project titled ICE4. The process to do this was covered in ICE2. Refer to ICE2 instructions if you need help.

2. Create code and images directories and add files to them

Navigate to your ICE4 folder in Windows Explorer (file manager), create a code folder, and add these provided files to it (Found in Teams: ICEs/ICE4/**code** folder):

- Stoplight.vhd and stoplight_tb.vhd
- Clock_divider.vhd and clock_divider_tb.vhd
- top_basys3.vhd
- Constraints file (Basys3_Master.xdc)

Add/stage and commit your changes

- Prepend all commit statements for ICE4 with "ICE4 – ". For instance, your first commit might be:
  - "**ICE4 – created initial project files**"

3. Create Design

Add Sources (Design, Simulation, and Constraints)

Add all of the provided files to your project. Instructions were given in ICE2 to do this. The _tb files will be simulation sources, the .xdc file will be a constraint, and the remaining files will be design sources.

Edit your Stoplight

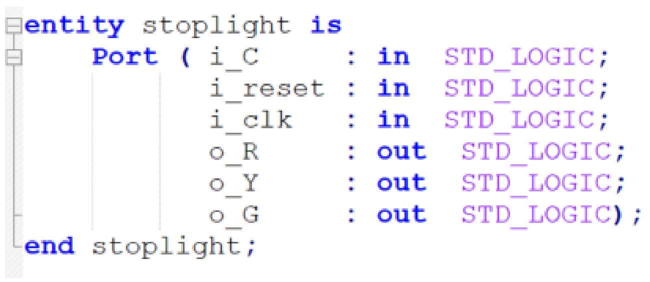
Open up your stoplight.vhd file in a text editor.

File Header

Modify the file header to reflect your work. Do not leave the documentation statement blank. You may refer all documentation statements to top_basys3 header.

Inspect the stoplight entity

See Figure 2 for all inputs and outputs to stoplight. C represents the control signal to indicate if a car is present, while R, Y, and G are your outputs to control the colored lights. There are two other signals unique to synchronous circuits. The clk signal drives the flip flops you will be using and provides the rising edge to trigger state updates. Likewise, the reset signal will reset the flip flops to their default state.



```
entity stoplight is
    Port ( i_C      : in  STD_LOGIC;
           i_reset  : in  STD_LOGIC;
           i_clk    : in  STD_LOGIC;
           o_R      : out  STD_LOGIC;
           o_Y      : out  STD_LOGIC;
           o_G      : out  STD_LOGIC);
end stoplight;
```

Figure 2. Entity for stoplight

Modify the Stoplight Architecture

Every finite state machine is going to have the same structure. There are three major sections, highlighted below in Figure 3. The first section is the combinational or next state logic. The second section is the State Register (flip flops). The third section is your output logic. As we develop the stoplight, we will look at each section individually.

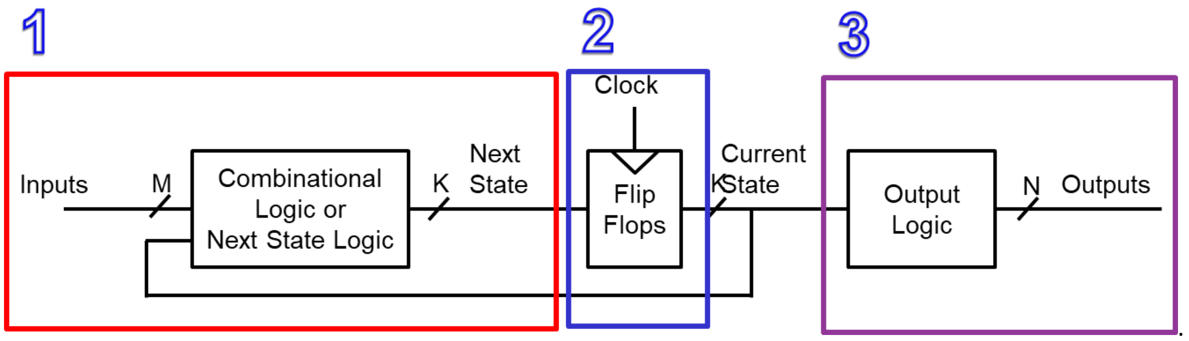


Figure 3. Finite state machine structure

You need to modify the architecture of your stoplight file to reflect the design shown below in Figure 4. Note that the design looks more complicated than it is.

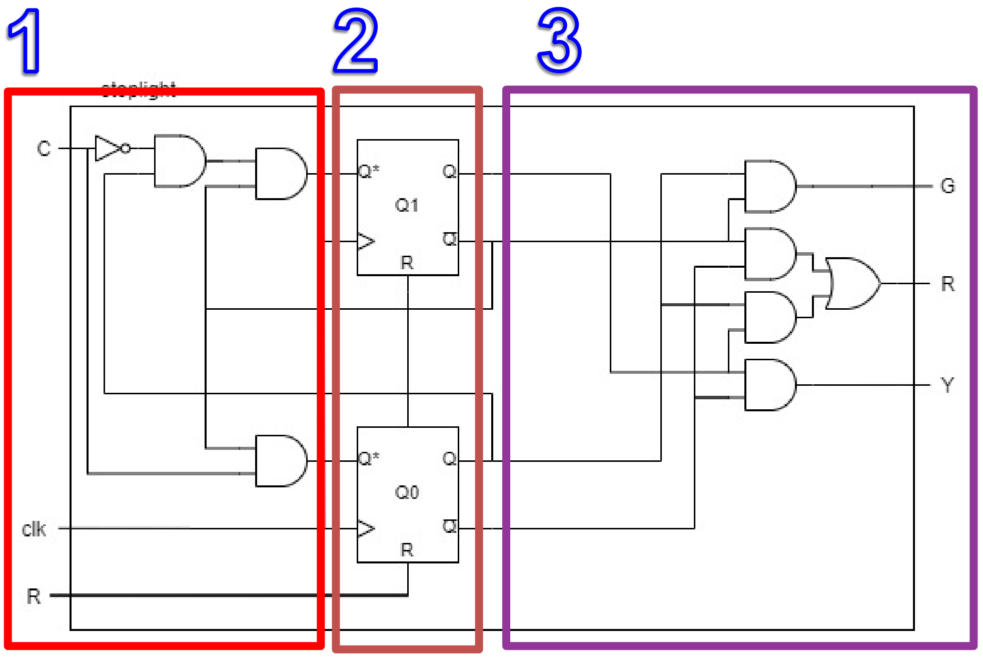
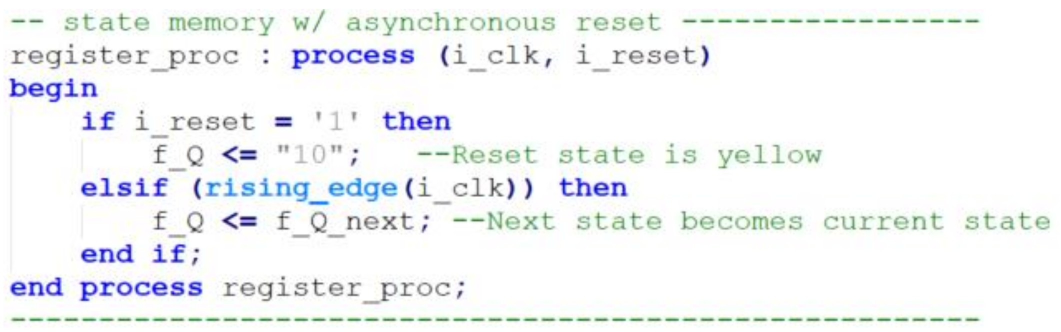


Figure 4. Entity and architecture for implementing stoplight

State Register

Our first step is actually going to be implementing section 2, our state register. The state register for our stoplight is comprised of two flip flops so we are going to need to create 2-bit vectors to represent both our current state Q and our next state Q_next. It is critical that you give f_Q and f_Q_next a default value (according to our VHDL naming conventions, the prefix, “f_”, indicates that the signal is used for sequential logic). **Because Q_next is dependent on Q and Q on Q_next, an undefined value in either one could break your system.** Even though the original problem didn’t specify, we are going to make the yellow state our default (and our reset) state. To accomplish this, the vectors need to be initialized with “10”. Remember that this can be accomplished with the :=“10”; syntax following your initial signal declaration.

Once the signals have been created, we can use them in our state register. The basis for a register in VHDL is a process. Declaring a process allows us to create a sub section of our hardware that is only triggered when certain signals change and is the only place we can implement sequential statements. If you remember, we have actually been using processes with every assignment we have done…in our test bench! Figure 5 below shows the code we will use to implement our register.



```
-- state memory w/ asynchronous reset -----------------
register_proc : process (i_clk, i_reset)
begin
    if i_reset = '1' then
        f_Q <= "10";    --Reset state is yellow
    elsif (rising_edge(i_clk)) then
        f_Q <= f_Q_next; --Next state becomes current state
    end if;
end process register_proc;
-------------------------------------------------------
```

Figure 5. Stoplight State Register implementation.

Let's walk through what is happening in this process. The text preceding the key word "process" represents the name you are giving that process. You will notice that this is where the similarity with the test process we have used before ends. Unlike the test process, we have signals inside the parenthesis following the key word "process." These signals indicate which signals are going to trigger the process to run. We put clk and reset in there for a reason. We are ultimately going to be looking for a clk rising edge so we want the process to run every time the clk changes values. Reset is in there to implement an **asynchronous** reset. This means we want the reset to happen immediately without waiting for a clock rising edge. If we waited for the clock edge, it would be a synchronous reset. In order to make this happen, our process will need to be also be triggered when the value for reset changes. Once inside the register, we will be utilizing the if/then statements to do our checks. We are only looking for two different scenarios at this point. Are we resetting or do we have a rising edge? We want to check for the reset first, because the asynchronous reset will happen with or without a rising edge. If reset is indeed active, we want to set f_Q to the default state "10". If reset is not active, we want to look for a rising edge. It is only when we have a rising edge that a register (or flip flops) will read in a new value. Once the rising edge is triggered we want to assign our current state f_Q to become f_Q_next. Because we made both f_Q and f_Q_next vectors, we don't need to worry about individually assigning the bits. We can just set f_Q<=f_Q_next. If there is no rising edge then we simply exit the process.

Next State Logic

Now that we have our register in place we can complete section 1, next state logic. This section is completely made of combinational logic so there isn't anything new we will need to learn. We simply need to implement the Q(0)* and Q(1)* equations we developed in HW 3. Go ahead and implement the equations below in your architecture:

$$Q_{next}(0) = \overline{Q(1)} * C$$
$$Q_{next}(1) = \overline{Q(1)} * Q(0) * \overline{C}$$

Once you have made these two signal assignments, you are done with the next state logic!

Output Logic

Our final step in creating our stoplight component is section 3, output logic. Like the next state logic section, there is nothing new we are learning here. We simply take the equations from HW3 and implement them. You can see the equations here:

$$G = \overline{Q(1)} * Q(0)$$

$$Y = Q(1) * \overline{Q(0)}$$
$$R = \overline{Q(1)} * \overline{Q(0)} + Q(1) * Q(0)$$

3. Test Design

At this point your testbench VHDL file (stoplight_tb.vhd) should already be added. If not, go back, add it, and set it as top.

Edit your testbench

Open your testbench file in a text editor. Note the header has stoplight.vhd as a REQUIRED FILE. A fully completed test bench has been provided for this exercise. The provided test bench has two new aspects to it. The first is the declaration of a constant.

```
constant k clk period : time := 10 ns;
```

As you can see from the name, we intend to use this constant to create an artificial clock to drive our FSM. When implementing our design on the board, we will use the clk we get from the board, but inside the test benc we must make a virtual clock. Once the constant has been created, we can make a process to implement our clock as seen in Figure 6.

```
-- Clock process definitions
clk_proc : process
begin
     i_clk <= '0';
     wait for k_clk_period/2;
     i_clk <= '1';
     wait for k_clk_period/2;
end process;
```

Figure 6. Clock process for test bench

Since a process runs in parallel with the rest of the circuit, this will run continually in the background. Essentially, this process sets the clock value low for exactly half of the period and then sets it high for half of the period creating an oscillating signal we can use as our clock.

Now look through the sim_proc to see what the test bench is going to do before running your simulation

Check syntax and simulate your project

See Figure 7 below to see what your simulation should look like. Note the additional signals we have not seen in past simulations. Not only can we check that the outputs are happening as expected, but we can verify that the state (Q) and next state (Q_next) are behaving as they should. Note that these signals do not show up in your waveform by default. The waveform will only show the signals in your test bench itself. However, you can pull any signal from any component and have it added to your test bench. To do this click “Window” and then “Scope.” You will see your stoplight_fsm_tb and the subcomponent stoplight fsm. If you click on stoplight fsm, you will see all of the signals present inside the component. Highlight Q and Q_next, right click, and select” Add to Wave Window.” You will need to run the sim again. You may also remove the constant since it’s presence doesn’t tell you anything useful. The ability to add signals in sub components will be a useful tool when you are debugging your FSM designs. It can help you figure out where your error is. For this design, if the state is correct but the output is not then the issues is in your output equations. If f_Q_next is correct but f_Q is not then maybe you are resetting on accident or your process is incorrect.

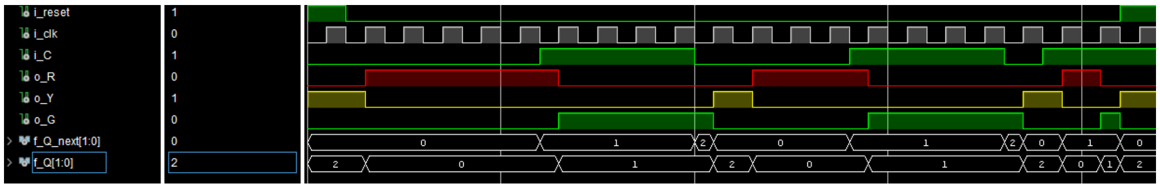


Figure 7. Stoplight FSM test bench waveform

CREATE TOP_BASYS3 DESIGN

Now that we have the stoplight FSM up and running, we need to create the top_basys3 design to connect it to the board's hardware. We will do this with the provided top_basys3.vhd file as well is the schematic seen in Figure 8.

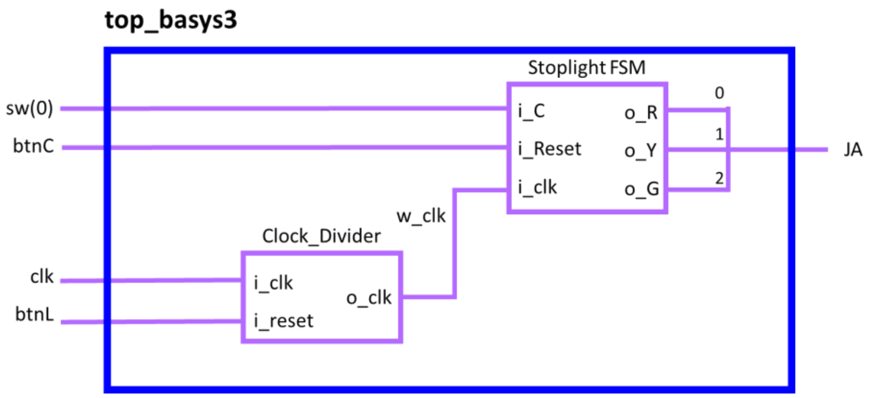


Figure 8. top_basys3 design for stoplight ICE

In order to fully implement this design, we have to introduce a new component that has been made for you. The clock divider's job is to take in the clock from the board, which operates at 100 MHz, and slow it down to a speed we can observe. For this ICE we will slow the clk down to 1 Hz. The module uses a generic natural constant called k_DIV to allow you to dynamically set how much you want to slow the clock down. This is done at instantiation using a generic map statement similar to port mapping and has been done for you in the top_basys3.vhd file. The i_reset on the clock will effectively pause the clock when a high signal is received. I recommend you explore the clock_divider to see if you can understand how it works, since we will continue to use it. A test bench has also been provided for the clock_divider. If you decide to run it, set the clock_divider_tb file as top and simulate.

Much of the top_basys3.vhd file has been done for you, however, you need to declare the stoplight component and instantiated it. You also need to fill out the port map for the clk divider already declared and instantiated.

4. IMPLEMENT DESIGN

Edit the constraints file to ensure that all signals from the top_Basys3 entity are uncommented.

IMPLEMENT AND GENERATE BITSTREAM

5. Connect Stoplight, Download onto FPGA and test functionality

Before we can test whether or not the stoplight works, we need to first connect the stoplight to the FPGA. We will be using the Pmod connection on the side of the Basys3. To help you with this, refer to Figure 9 below. We will be using the top row of the Pmod connection JA. Looking head on, the pins starting from the right and going to the left are red, yellow, green, empty, ground.

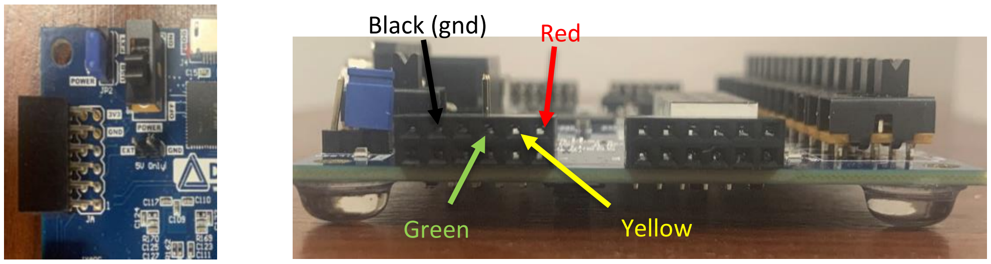


Figure 9. Pmod from above and from the side with connection labels

Luckily, the wires of the stoplight are color-coded! As you can see in Figure 10, black will go to black, red to red, yellow to yellow, and green to green.

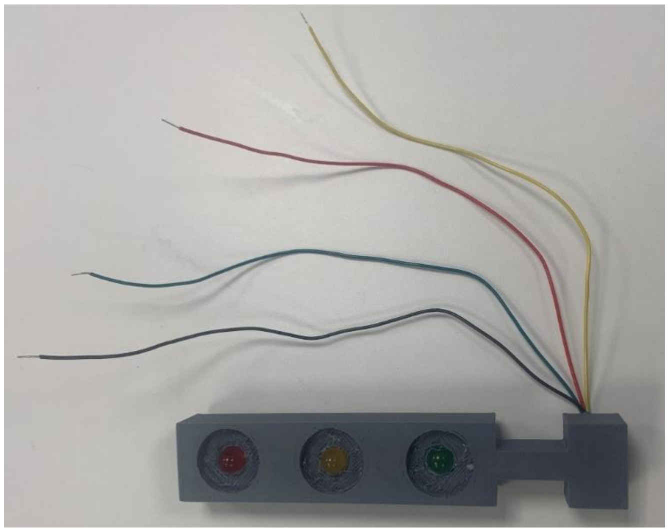


Figure 10. Picture of stoplight model we will be using

Once you have successfully connected the board, push your bit file to the board and show an instructor that your design works! If you flip the car input on, the system should cycle from red to green. If you flip the car input off, the system should cycle to yellow, and then red. Pressing the center button should immediately turn on the yellow light and pressing the left button should pause the whole system.

7. Wrapping up…

- Double check that you have everything **committed** to your git repo.
- Then **push** them to your Bitbucket repo.
- Make sure all your work appears in Bitbucket as you expect.
- Show your simulation waveform to an instructor
- Demo your working board to an instructor

Congratulations, you have completed In Class Exercise 4!