Lesson 27 – ICE 5 Introduction and Tips

ECE 281

Lesson 27 Notes

Objectives:

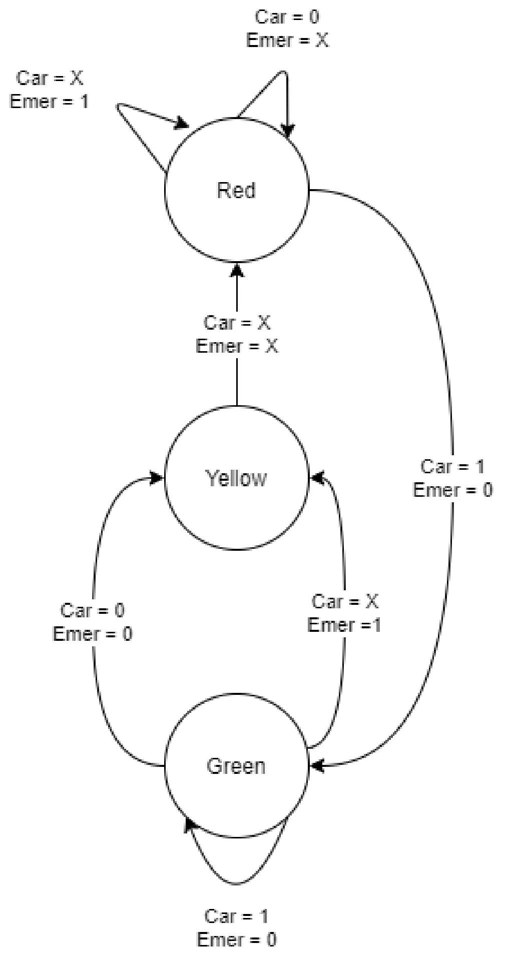
1. Create and simulate synchronous sequential logic system
2. Practice using VHDL to build a provided state diagram
3. Practice using the concept of enumerated types

Scenario

We want to modify the previous built and analyzed stop-light controller to accept an additional input. In addition to monitoring for a car, we want the stop-light to transition back to RED to allow the emergency vehicle to pass safely:

Two Inputs:
Car: 1(car is present) 0(no car is present)
Emer: 1(Emergency vehicle coming) 0(No emergency vehicle coming)

Three States:
Red, Yellow, Green
Must always transition from red to green, green to yellow and yellow to red



Enumerated Types: For this example, we want to create a new type called sm_light, that can take on various values (s_red, s_yellow, s_green).

Additionally, we will create two additional signals (current_state and next_state) that can take on any of the values of sm_light.

By using enumerated types, in this fashion we can make our vhdl a bit more compact, as Vivado will essentially handle much of the logic for us. We simply need to explain in vhdl what conditions would lead us to remain or leave a particular state.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity EmerStopLightController is
    Port ( clk      : in  STD_LOGIC;
           reset    : in  STD_LOGIC;
           car    : in  STD_LOGIC;
           emer : in  STD_LOGIC;
           light    : out STD_LOGIC_VECTOR (2 downto 0)
         );
end EmerStopLightController;


-- Write the code in a similar style as the Lesson 19 ICE (stoplight FSM version 2)
architecture Behavioral of EmerStopLightController is

    -- Below you create a new variable type! You also define what values that
    -- variable type can take on. Now you can assign a signal as
    -- "sm_light" the same way you'd assign a signal as std_logic
    type sm_light is (                              );

    -- Here you create variables that can take on the values
    -- defined above. Neat!
    signal                              : sm_light;
```

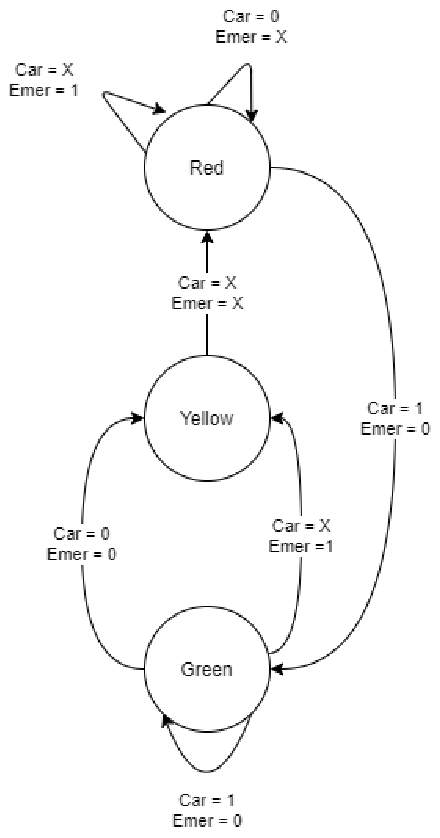
Next State Logic:

Now that we have our enumerated type defined, we can create our next state logic. In this case, we are going to make use of the enumerated values that the signals (current_state and next_state) can take on.

```
begin

    -- Next state logic ---------------------------------
    -- You may also use case statements here if you would prefer that implementation.
    next_state <=  s_red when
                    or
                    or
                   s_yellow
                    or
                   s_green
                    or
```

With the code above, we are defining what conditions are responsible for each state transition. Notice the correlation to the FSM diagram below:



Dealing with State Transitions:

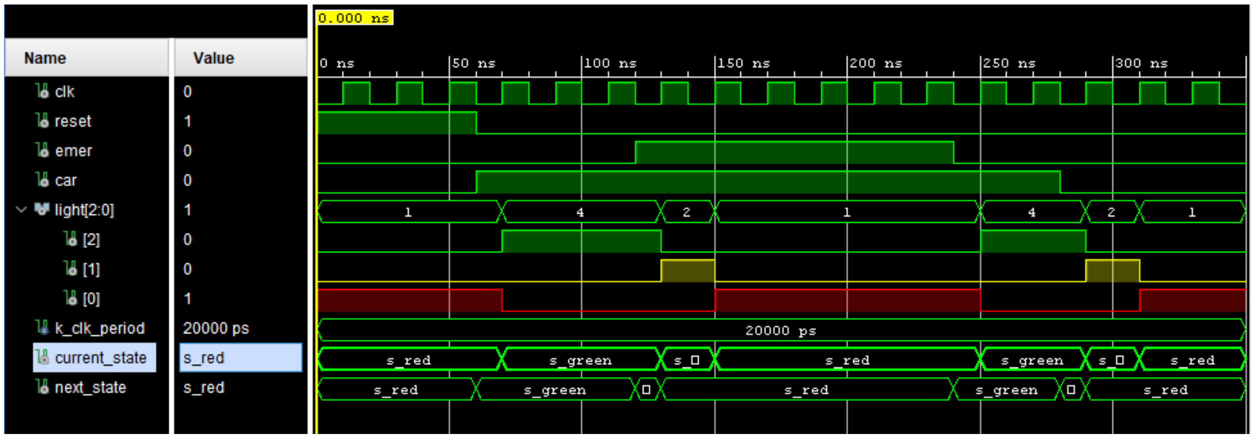
Now that we have our next state logic defined, it is time to create the process that will drive the transitions. This is extremely simple in this case, where the process will only be monitoring two signals (i.e. the sensitivity list) clk, and reset. I won't give you the code for this, but let's discuss a bit further:

Associating enumerated types with output logic:

Now that we have our state transitions defined, and a process to control them, we must associate this enumerated type back to some form of state encoding. In the example below, I chose to use one-hot encoding for each of the three states.

```
light <=   "001" when current_state =        else
           "010" when current_state =          else
           "100" when current_state =        ;
```

That's all there is to it. We can now create a test bench that allows each of the transitions to be tested.



Name	Value
clk	0
reset	1
emer	0
car	0
light[2:0]	1
[2]	0
[1]	0
[0]	1
k_clk_period	20000 ps
current_state	s_red
next_state	s_red