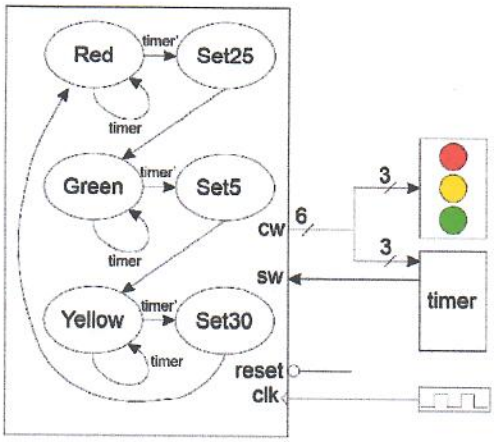
Lecture 31

The Traffic Light Controller

A control unit is a finite state machine that sequence the operation of some datapath. Hence, it's a term used for many digital systems, not just computers. All the FSM's that you design in this class will be Moore machines, meaning that their output depends only on the current state. To see how to build a Moore-type FSM in Verilog we'll examine the construction of a FSM to control a traffic light shown in **Error! Reference source not found.**. This traffic light operates as you would expect, cycling between red, green and yellow.



Each light has its own input and when that input is 1, the light illuminates. The timer takes the 3-bit input, that follows the following truth table. While the timer is counting down, the output equals 1 and when the timer reaches 0 seconds, its output equals 0 and stays there until the timer is set.

The body of the Verilog code for the traffic light FSM contains the following sections listed in bold typeface.

State definitions

We will use a dense coding of states in our Verilog code. A dense code uses $\lceil \log_2(N) \rceil$ bits to give each of N objects a unique binary code. The $\lceil\ \rceil$ brackets mean round up. Thus, for the 6 states of the traffic light controller, we need a $\lceil \log_2(6) \rceil = 3$-bit code for each state. Assign each state a unique binary code (the choice is immaterial) using the localparam statement. Append all your state names with "_STATE" to help you tell what sort of thing you are looking at.

```
localparam    LGT_RD_STATE = 3'b000;
```

Define the reset state

In all our circuits, we will briefly hold the reset line low at startup. This reset signal is used to specify the initial state of our FSM as LGT_RD_STATE. The else condition assigns the nextstate to the current state on the positive edge of the clk.

```
always @(posedge clk or negedge reset)
if (!reset)
    state <= LGT_RD_STATE;
else
    state <= nextstate;
```

Next state logic

The memory input equations are captured in the next state logic section. Unlike the MIE's which capture the arc going into each state, the next state logic captures the logic describing the outgoing arcs from each state. Each state will have a case statement where you describe the next state in terms of the input associated with each arc. When you use the if/then structure you **must always** include a final else statement.

```
LGT_RD_STATE:
    begin
        if (timer == 1'b1)     nextstate = SET_25_STATE;
        else                   nextstate = LGT_RD_STATE;
    end
```

Output logic

The output equations are captured in the output logic section. This always/case statement has a case for each state where the output associated with that state is listed. Assign the control word associated with each state using the localparam statement. This will make your code a lot easier to read your code and maintain it.

```
localparam   LR_CW  = 6'b100000;
LGT_RD_STATE:    cw = LR_CW;
```

Create symbolic alias' for state binary codes in the .do file

When you run a testbench for the FSM, the state and nextstate variables will be displayed using the 3-bit code you assigned them using the `localparam LGT_RD_STATE = 3'b000;` statement you wrote earlier. As a human, I would rather know that my FSM is in state LGT_RD rather than in state 3'b000. I searched for a solution until I came across the section on radix define statement in the ModelSim® Command Reference Manual. In their words:

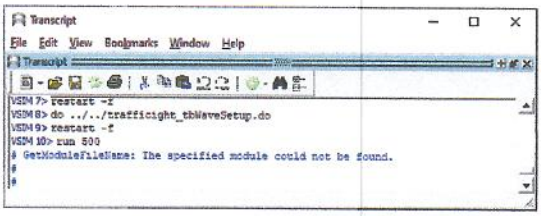
> *This command is used to create or modify a user-defined radix. A user definable radix is used to map bit patterns to a set of enumeration labels or setup a fixed or floating point radix. Userdefined radices are available for use in the most windows and with the examine command.*

This command allows you to define symbolic names for the 3-bit binary codes associated with each state. I put this declaration at the top of my .do file used to setup the timing simulation for my FSMs. The symbolic names can be anything you want and do not have to be the same as the `localparam LGT_RD_STATE = 3'b000;` statement you wrote earlier. The contents of the .do file used for our traffic light controller are shown below.

```
################################################################################
################################################################################
radix define States {

    3'b101 "SET_30"  -color red,
    3'b000 "LGT_RD"  -color red,

    3'b001 "SET_25"  -color green,
    3'b100 "LGT_GR"  -color green,

    3'b010 "LGT_YW"  -color yellow,
    3'b011 "SET_05"  -color yellow,

    -default hex
    -defaultcolor orange
}
restart -f
delete wave *
add wave -position end  sim:/trafficLightController_tb/uut/clk
add wave -position end  sim:/trafficLightController_tb/uut/reset
add wave -position end  sim:/trafficLightController_tb/uut/sw
add wave -position end  -radix States sim:/trafficLightController_tb/uut/state
add wave -position end  -radix hex sim:/trafficLightController_tb/uut/cw
```

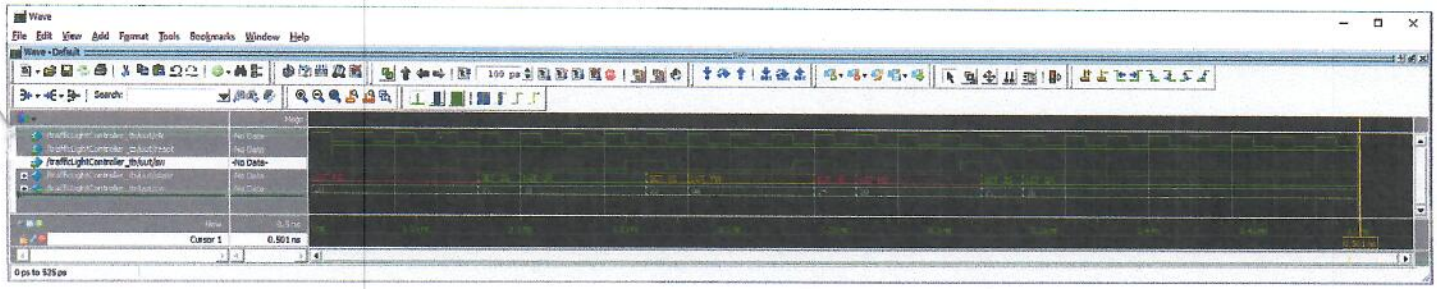
Working in the Transcript window

All the buttons that you have been pressing to run simulations in ModelSim are just surrogates for commands that you can type in the Transcript window at the bottom of the screen. I have found that a few of these commands, shown below, to be very helpful to debugging our increasingly sophisticated designs.



The timing diagram for the traffic light controller

Color coding the state names using the `radix define` command really helps show the progression of the FSM for the traffic light controller.



```
//*****************************************************************
// Name:    Dr. Chris Coulston
//*****************************************************************
module trafficLightController(clk, reset, cw, sw);

        input wire          clk, reset;
        output reg [5:0]    cw;
        input wire          sw;

        wire timer;
        assign timer = sw;

        //-------------------------------------------------------------
        // The state and nextstate signals must be the same size
        //-------------------------------------------------------------
        reg     [2:0]   state, nextstate;

         localparam     LGT_RD_STATE  = 3'b000;
         localparam     SET_25_STATE  = 3'b001;
         localparam     LGT_YW_STATE  = 3'b010;
         localparam     SET_05_STATE  = 3'b011;
         localparam     LGT_GR_STATE  = 3'b100;
         localparam     SET_30_STATE  = 3'b101;

        //-------------------------------------------------------------
        //-------------------------------------------------------------
         localparam   LR_CW  = 6'b100000;
         localparam   s25_CW = 6'b100001;
         localparam   LG_CW  = 6'b010000;
         localparam   s05_CW = 6'b100010;
         localparam   LY_CW  = 6'b001000;
         localparam   s30_CW = 6'b100100;

        //-------------------------------------------------------------
        // Assign the initial state (reset) or next state (clk edge)
        //-------------------------------------------------------------
        always @(posedge clk or negedge reset)
        if (!reset)
                state <= LGT_RD_STATE;
        else
                state <= nextstate;

        //-------------------------------------------------------------
        // Logic to generate the output based on the current state
        //-------------------------------------------------------------
        always @(state)
        begin
                case(state)
                        LGT_RD_STATE:   cw = LR_CW;
                        SET_25_STATE:   cw = s25_CW;
                        default:        cw = LR_CW;
                endcase
        end

        //-------------------------------------------------------------
        // Logic to generate the next state based on the current state and inputs
        //-------------------------------------------------------------
        always @(*)
        begin
                case(state)
                        LGT_RD_STATE:
                                begin
                                if (timer == 1'b1) nextstate = SET_25_STATE;
                                else               nextstate = LGT_RD_STATE;
                                end

                        SET_30_STATE:
                                nextstate = LGT_RD_STATE;

                        default:                   nextstate = LGT_RD_STATE;
           endcase
         end
endmodule
```