CSEE 4280: Advanced Digital Design

Trent Jones

Abhi Boggavarapu

Assignment: Exercise 5

02 / 14 / 2025

Table of Contents

Roles and Responsibilities

Abhi	Software/Documentation
Trent	Software/Documentation

Github Code Link

The below is a link to the repository and folder within that includes the code used for this project.

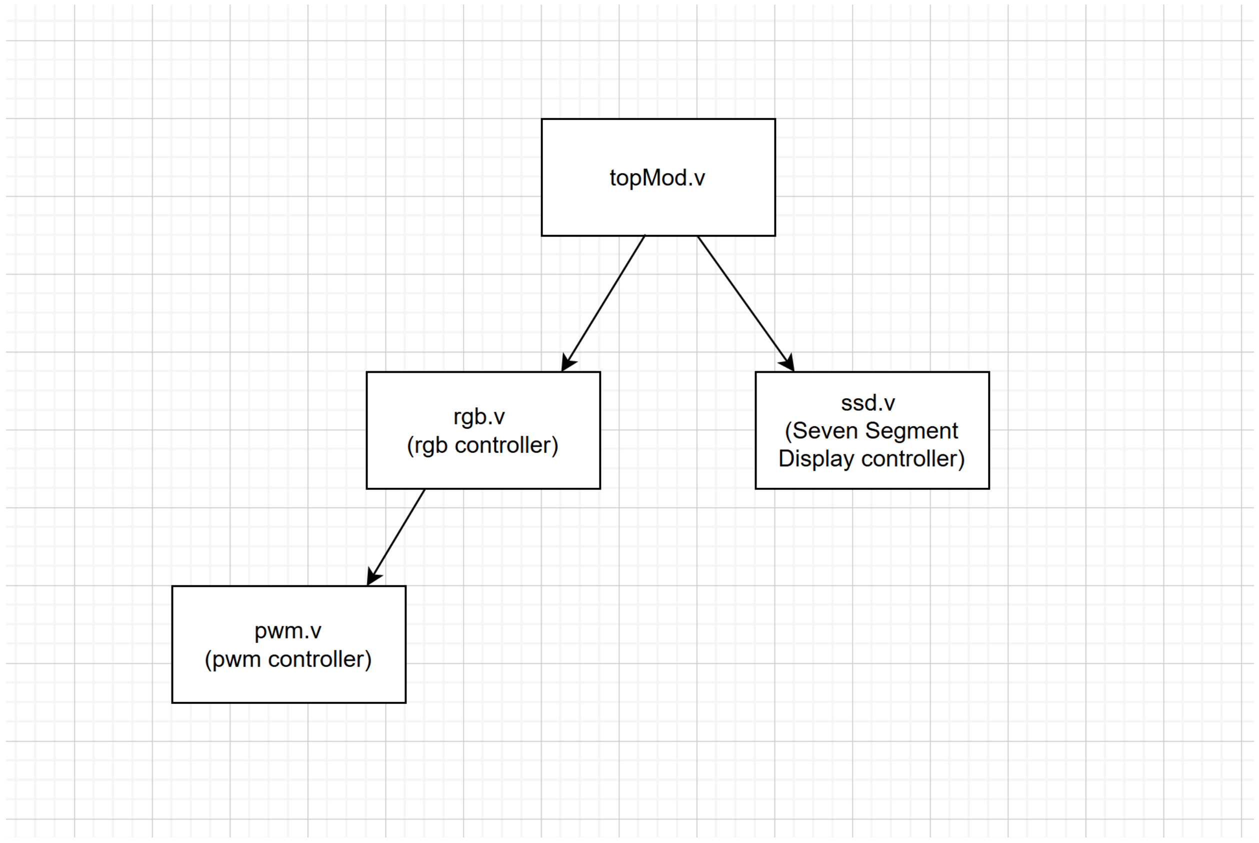
https://github.com/abhibogga/CSEE4280Workspace/tree/main/5_Exercise

Deliverables

1:

Hierarchy in your design. Build one controller main block. Inside the controller block, you will call three separate PWM machines.

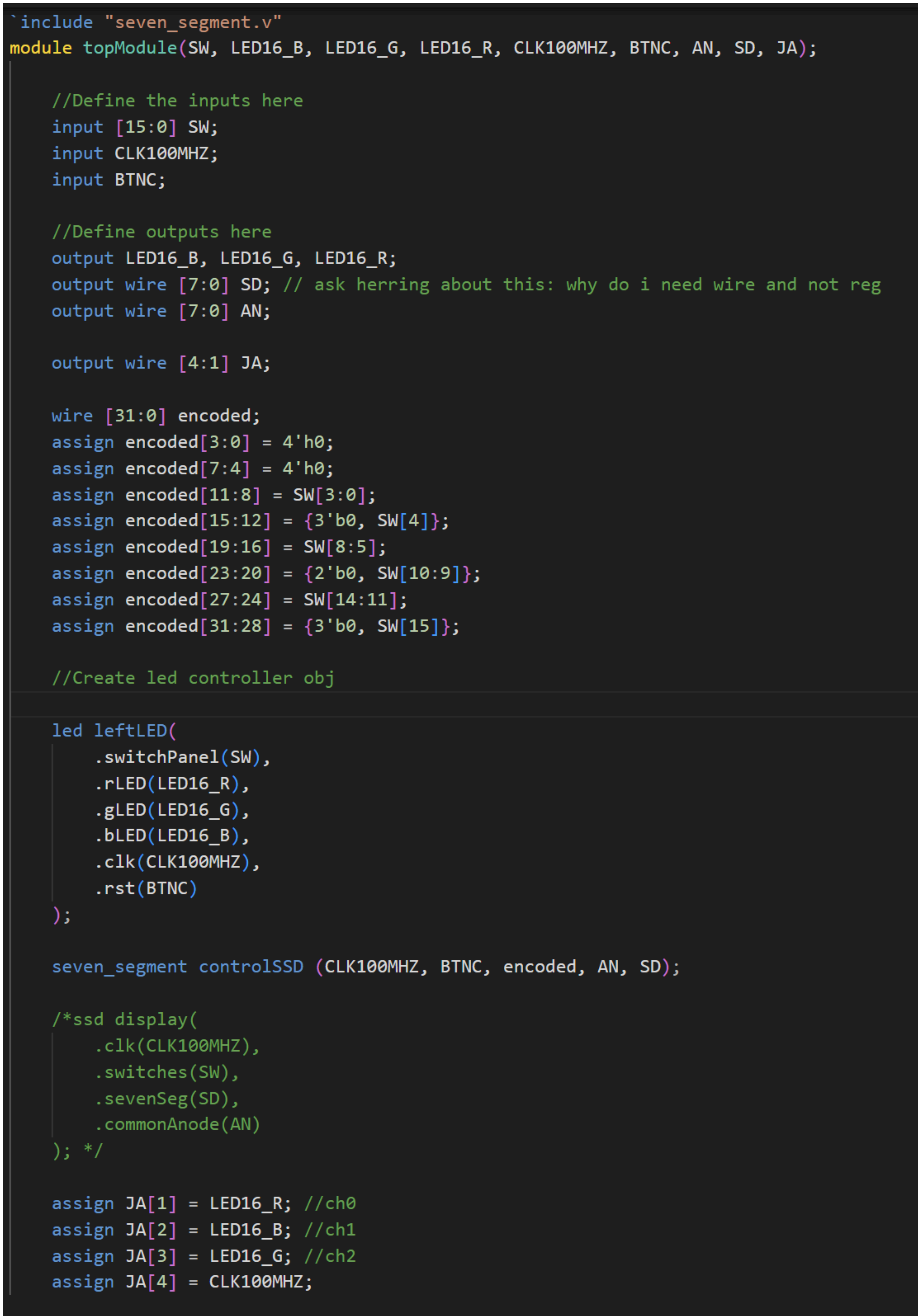
Within the design, there is a 3 stage hierarchy. At the very top there is a top level controller called topMod.v. This drives the rgb controller and the seven segment display modules. These 2 controllers are the second level of the hierarchy. The roles of these 2 modules are to control everything that happens on the input and output level of the project. The rgb controller changes the LED based on the input of the switches and the seven segment display module will display a hex number according to the configuration of the switches. Under the rgb controller module there exists the bottom of the hierarchy which is the pwm module. Within any given rgb controller there will be 3 pwm modules that modulate the duty cycle of the red, green, and blue lights that exist within the LED itself. Model shown below.



2:

Have a top level Verilog for the main program with inputs from your switches, clocks, etc.. and outputs to the RGB.

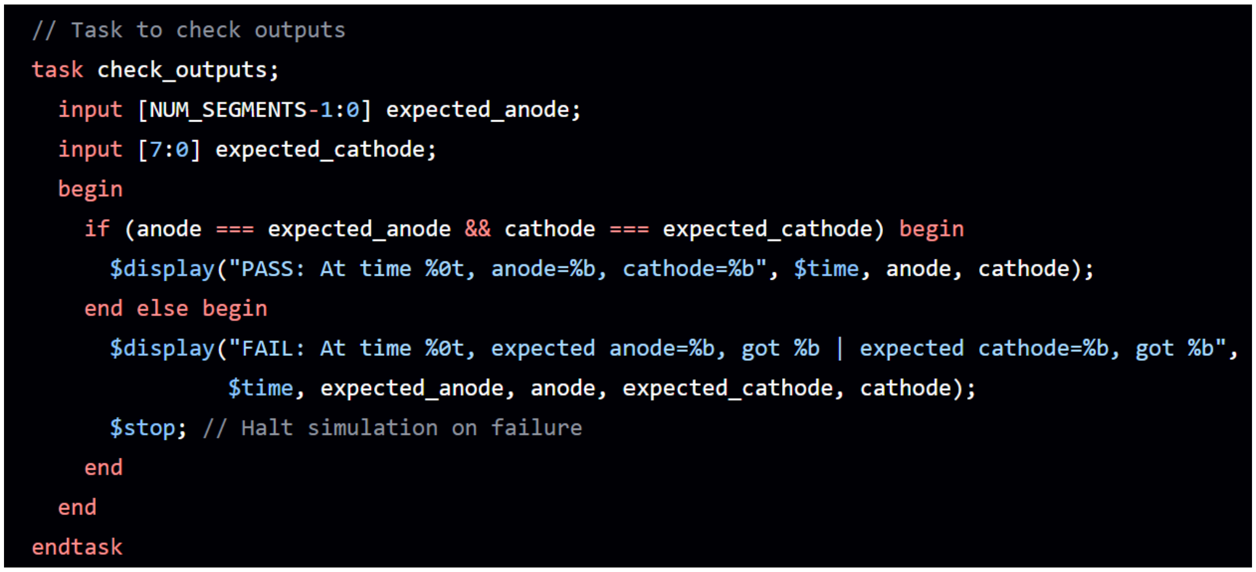
As shown above, the topMod.v file controls all of the code and is the main code that you run when you synthesize and implement in vivado. Code shown below



```
`include "seven_segment.v"
module topModule(SW, LED16_B, LED16_G, LED16_R, CLK100MHZ, BTNC, AN, SD, JA);

    //Define the inputs here
    input [15:0] SW;
    input CLK100MHZ;
    input BTNC;

    //Define outputs here
    output LED16_B, LED16_G, LED16_R;
    output wire [7:0] SD; // ask herring about this: why do i need wire and not reg
    output wire [7:0] AN;

    output wire [4:1] JA;

    wire [31:0] encoded;
    assign encoded[3:0] = 4'h0;
    assign encoded[7:4] = 4'h0;
    assign encoded[11:8] = SW[3:0];
    assign encoded[15:12] = {3'b0, SW[4]};
    assign encoded[19:16] = SW[8:5];
    assign encoded[23:20] = {2'b0, SW[10:9]};
    assign encoded[27:24] = SW[14:11];
    assign encoded[31:28] = {3'b0, SW[15]};

    //Create led controller obj


    led leftLED(
        .switchPanel(SW),
        .rLED(LED16_R),
        .gLED(LED16_G),
        .bLED(LED16_B),
        .clk(CLK100MHZ),
        .rst(BTNC)
    );

    seven_segment controlSSD (CLK100MHZ, BTNC, encoded, AN, SD);

    /*ssd display(
        .clk(CLK100MHZ),
        .switches(SW),
        .sevenSeg(SD),
        .commonAnode(AN)
    ); */

    assign JA[1] = LED16_R; //ch0
    assign JA[2] = LED16_B; //ch1
    assign JA[3] = LED16_G; //ch2
    assign JA[4] = CLK100MHZ;
```

3:

Have a testbench (self-checking for switches to 7SD). PWM does not need to be self-checking.



```
// Task to check outputs
task check_outputs;
  input [NUM_SEGMENTS-1:0] expected_anode;
  input [7:0] expected_cathode;
  begin
    if (anode === expected_anode && cathode === expected_cathode) begin
      $display("PASS: At time %0t, anode=%b, cathode=%b", $time, anode, cathode);
    end else begin
      $display("FAIL: At time %0t, expected anode=%b, got %b | expected cathode=%b, got %b",
               $time, expected_anode, anode, expected_cathode, cathode);
      $stop; // Halt simulation on failure
    end
  end
endtask
```

Figure 3.1: 7SD Testbench (Self-checking algorithm)

The self-checking mechanism in this Verilog test bench is implemented through the **check_outputs** task, which compares the expected and actual values of the anode and cathode outputs. The improved version of this task ensures robust verification by checking if both signals match their expected values. If they do, a "PASS" message is displayed, showing the correct values at the given simulation time. If either value is incorrect, a "FAIL" message is printed with the expected and actual values, and the simulation halts using **$stop**, preventing further execution when an error occurs. This ensures that any mismatches are immediately flagged, making debugging more efficient. By enforcing strict output validation and stopping on failure, this approach transforms the test bench into a fully self-checking test, reducing the need for manual inspection of waveforms or simulation logs.

4:

Implement first in iVerilog as much as possible before moving to Vivado

This process was a great way to quickly test and compile code before moving into the heavy process of compiling on vivado.

5:

Once it passes in iVerilog move to Vivado

When moving to vivado, it allowed the development process to really take off when implementing code into the hardware as the debug cores that vivado provides are great resources to make sure the verilog code is doing what is doing what it is supposed to do.

6:

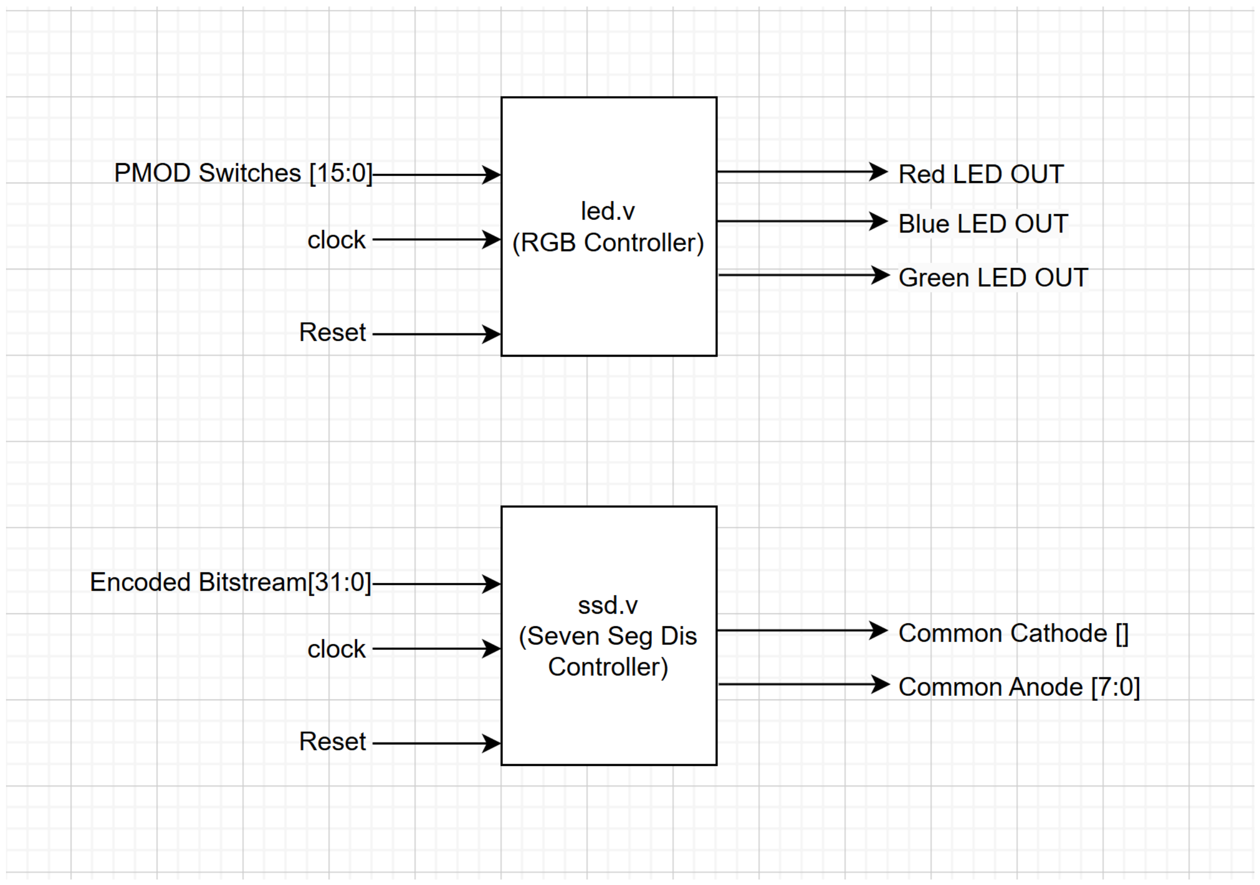
Final Verilog for iVerilog and for Vivado on Github

The GitHub repository requested is found above in its own section of the document.

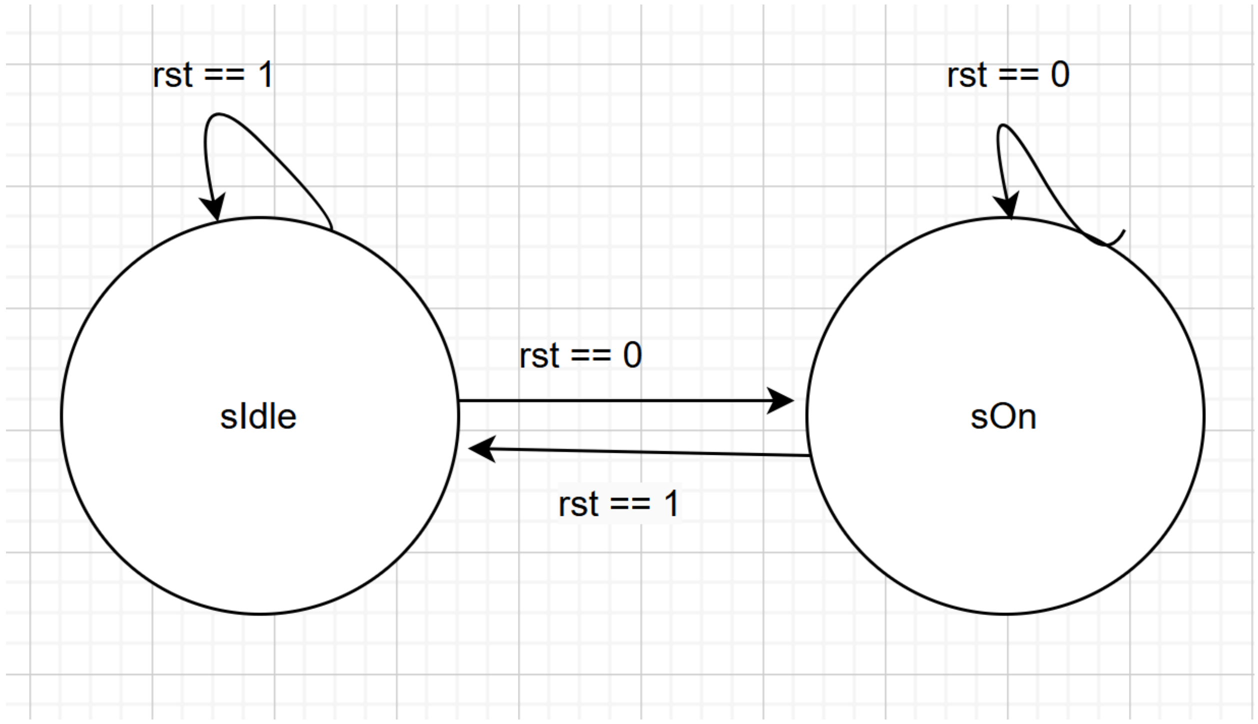
7:

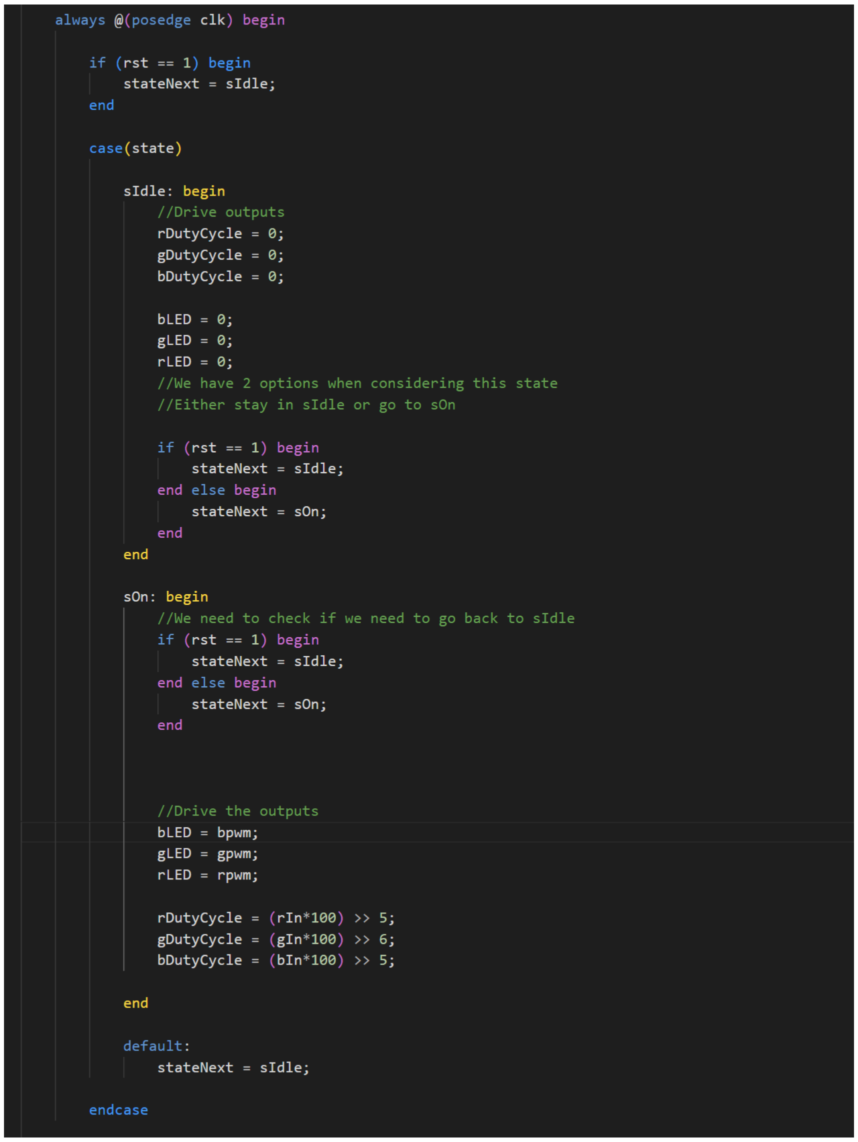
A document for your design (block diagram) and state machine drawings.

Below is the block diagram for the led.v and the ssd.v modules. The workings of the rgb controller contains 3 inputs and 3 outputs. The 3 inputs that it takes is a 15 bit array which directly corresponds to the PMOD switches as shown below in the diagram. The other 2 inputs that it takes are the clock module and the reset button. The clock module is connected to the FPGAs RTC. The reset will drive the outputs to a low. This reset button is set to the center button on the FPGA. The next module is the seven segment display module. This takes the same clock and reset inputs as the rgb controller but the only difference is the 31 bit array input. This is also set to the PMOD switches but is configured to read hex values and certain 1 bit values for the overflow 5th bit.



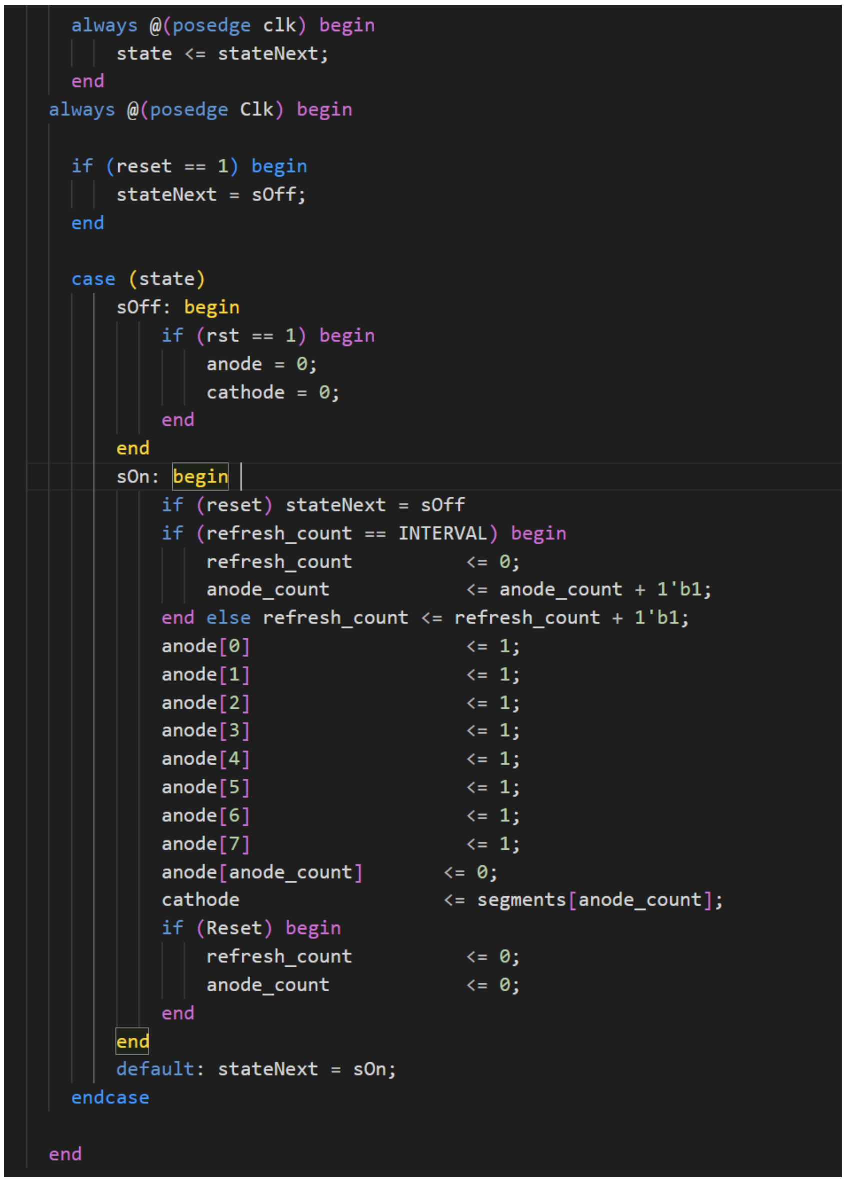
The next diagram is the state machine for BOTH the rgb controller and the seven segment display controller. Both have a very simple state machine design where only when the reset is turned on the outputs will get driven to a low. For the RGB controller, when the reset is turned on, the led will turn off and when the reset turns high for the seven segment display all the seven segment displays will turn off. As soon as the reset or the center button is released, the state will move to the on state. Code for both modules are shown below.


rst == 1
rst == 0
sIdle
rst == 0
sOn
rst == 1



```
    always @(posedge clk) begin

        if (rst == 1) begin
            stateNext = sIdle;
        end

        case(state)

            sIdle: begin
                //Drive outputs
                rDutyCycle = 0;
                gDutyCycle = 0;
                bDutyCycle = 0;

                bLED = 0;
                gLED = 0;
                rLED = 0;
                //We have 2 options when considering this state
                //Either stay in sIdle or go to sOn

                if (rst == 1) begin
                    stateNext = sIdle;
                end else begin
                    stateNext = sOn;
                end
            end

            sOn: begin
                //We need to check if we need to go back to sIdle
                if (rst == 1) begin
                    stateNext = sIdle;
                end else begin
                    stateNext = sOn;
                end



                //Drive the outputs
                bLED = bpwm;
                gLED = gpwm;
                rLED = rpwm;

                rDutyCycle = (rIn*100) >> 5;
                gDutyCycle = (gIn*100) >> 6;
                bDutyCycle = (bIn*100) >> 5;

            end

            default:
                stateNext = sIdle;

        endcase
```

Figure: LED State Machine Code



```
    always @(posedge clk) begin
        state <= stateNext;
    end
  always @(posedge Clk) begin

    if (reset == 1) begin
        stateNext = sOff;
    end

    case (state)
        sOff: begin
            if (rst == 1) begin
                anode = 0;
                cathode = 0;
            end
        end
        sOn: begin
            if (reset) stateNext = sOff
            if (refresh_count == INTERVAL) begin
                refresh_count           <= 0;
                anode_count             <= anode_count + 1'b1;
            end else refresh_count <= refresh_count + 1'b1;
            anode[0]                    <= 1;
            anode[1]                    <= 1;
            anode[2]                    <= 1;
            anode[3]                    <= 1;
            anode[4]                    <= 1;
            anode[5]                    <= 1;
            anode[6]                    <= 1;
            anode[7]                    <= 1;
            anode[anode_count]        <= 0;
            cathode                   <= segments[anode_count];
            if (Reset) begin
                refresh_count           <= 0;
                anode_count             <= 0;
            end
        end
        default: stateNext = sOn;
    endcase

  end
```

Figure: State Machine for Seven Segment Display

8:

Include GTKWave of simulated outputs using 1:1:1 and 1F:3F:1F as inputs for switches.

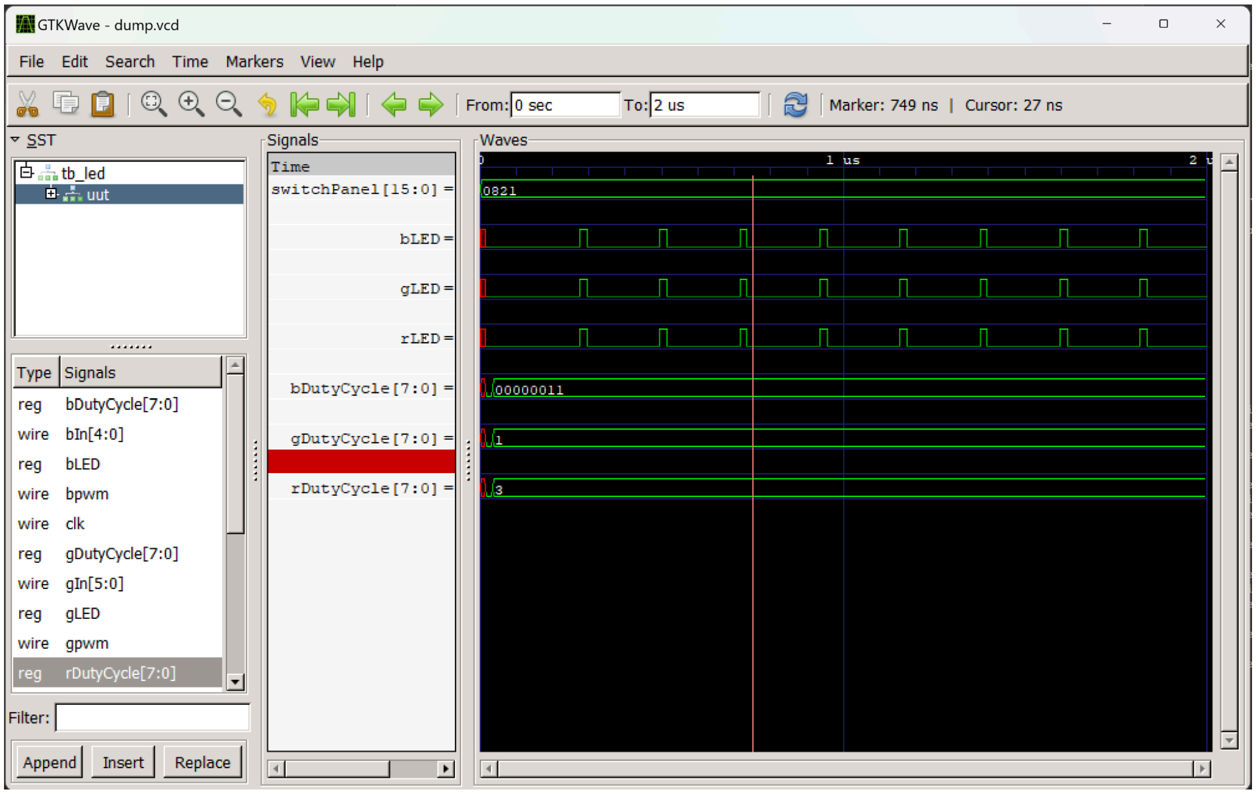


Figure 8.1: GTKWave simulation of RGB at 1:1:1

The simulation above represents the red, green, and blue portions of the RGB LED when all set to 01. All the waveforms produced show one single bit activated per cycle. Looking at the duty cycles, because the green LED has one more bit to process, the cycle is going to be smaller than the red and blue. This is depicted at the bottom of the simulation where blue and red share the same duty cycle, but green is smaller than the others.

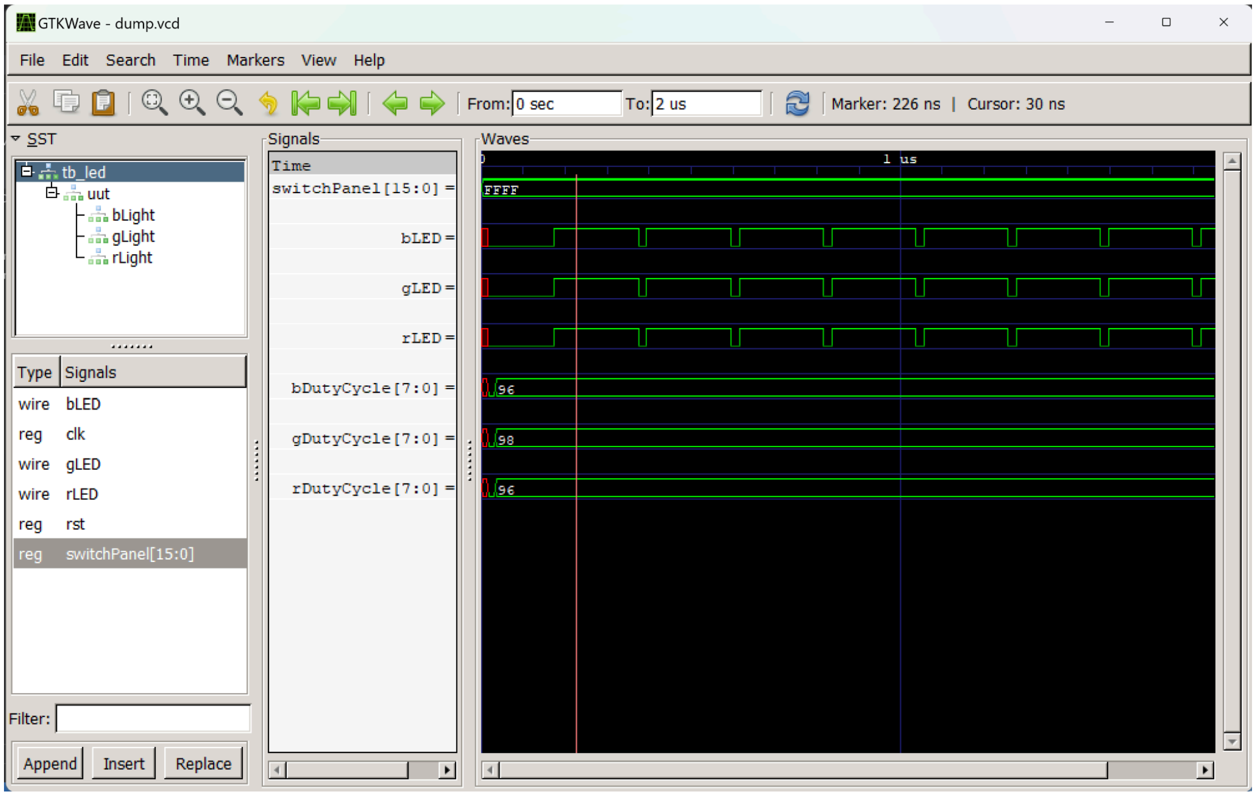


Figure 8.2: GTKWave simulation of RGB at 1F:3F:1F

The simulation above represents the same RGB LED portions, but instead of at the lowest detectable pulse they are all set to the largest value. The green LED has one more bit than the others, so the value is 3F instead of 1F for red and blue. Looking at the graph, all three show a waveform that is majority on as expected. Looking at the duty cycles once again both the red and blue LED share the same cycle, whereas the green now has a larger duty cycle.

9:

Show these same results on your LA. You may take a screenshot or snip of the results. Make sure they are clearly identified.

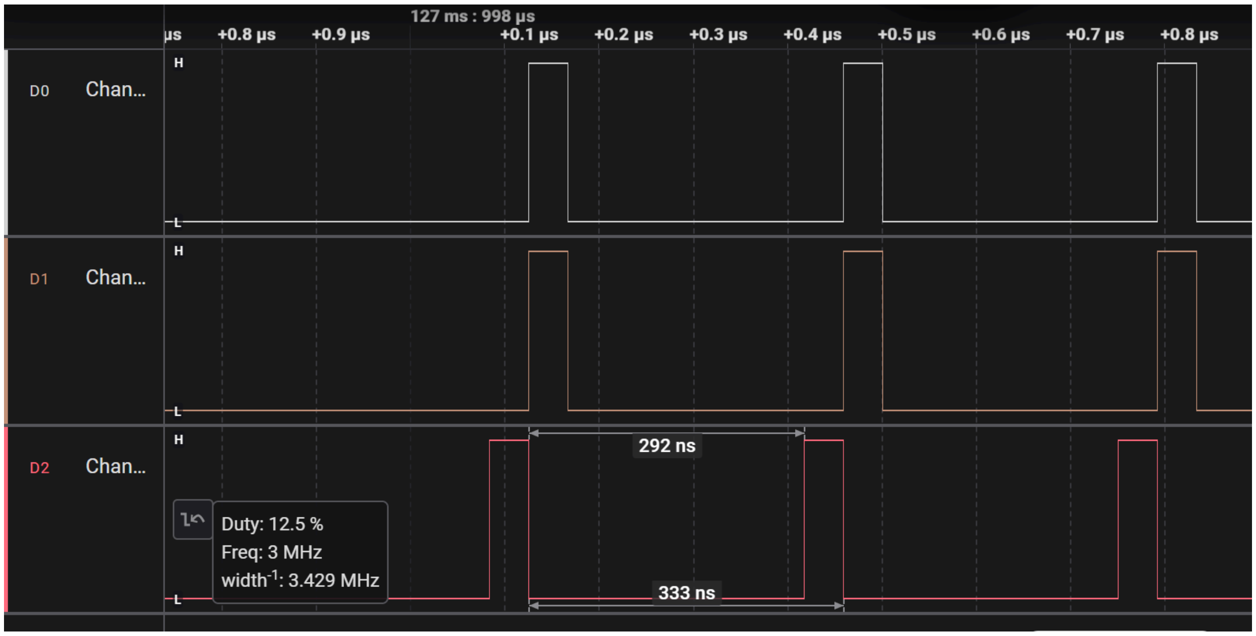


Figure 9.1: Logic Analyzer waveform from PMOD at 1:1:1

The waveform graph above shows each component of the RGB LED in the order red, blue, green. Similar to the GTKWave simulation, it is apparent both the red and blue LED's share the same duty cycle and the green LED has a shorter cycle. Some of the discrepancy in values could be attributed to trouble connecting cleanly to the PMOD inputs.

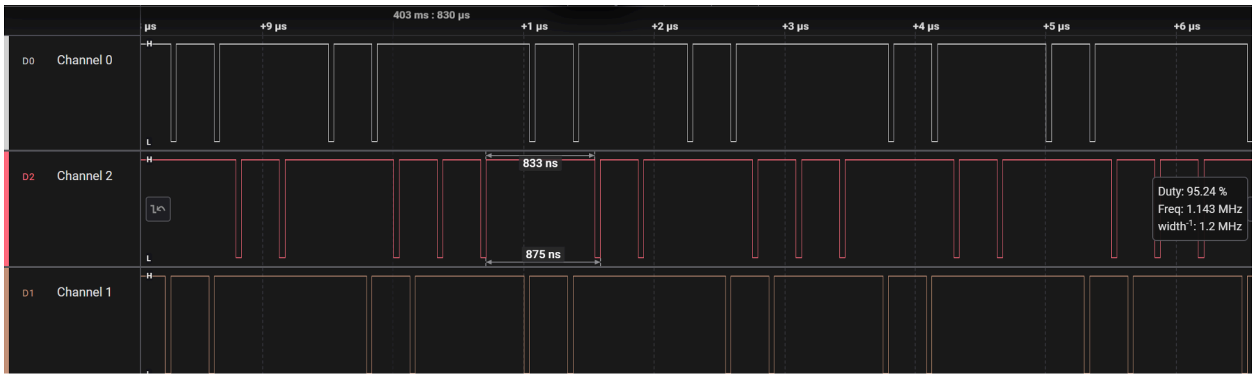


Figure 9.2: Logic Analyzer Waveform from PMOD at 1F:3F:1F

The waveform graph above shows the RGB LED components in the order red, green, blue. Again, similar to the GTKWave simulation there is a similarity in duty cycle between red and blue LED components. The green LED looks as though the duty cycle is different than the other, and it looks as if there is more to process with the extra bit that goes into its "3F" value.

10:

Describe how you tested and verified the design.

This process was very intensive on looking at any waveform diagram. Most of the initial testing was done on gtkWave and processing and dialing the pwm outputs. When moving towards vivado and implementing on the FPGA. The debug cores and internal logic analyzer were very important to see exactly how the seven segment display and PMOD switches were acting. There was a large bug within the beginning of the hardware implementation process where sometimes the PMOD switches would have a floating high value. This was a lifesaver in the implementation process because it allowed the software development process to proceed at a much faster pace. In terms of working on gtkWave, it was also very helpful to see the duty cycles and the pwm outputs being copied to the outputs.

11:

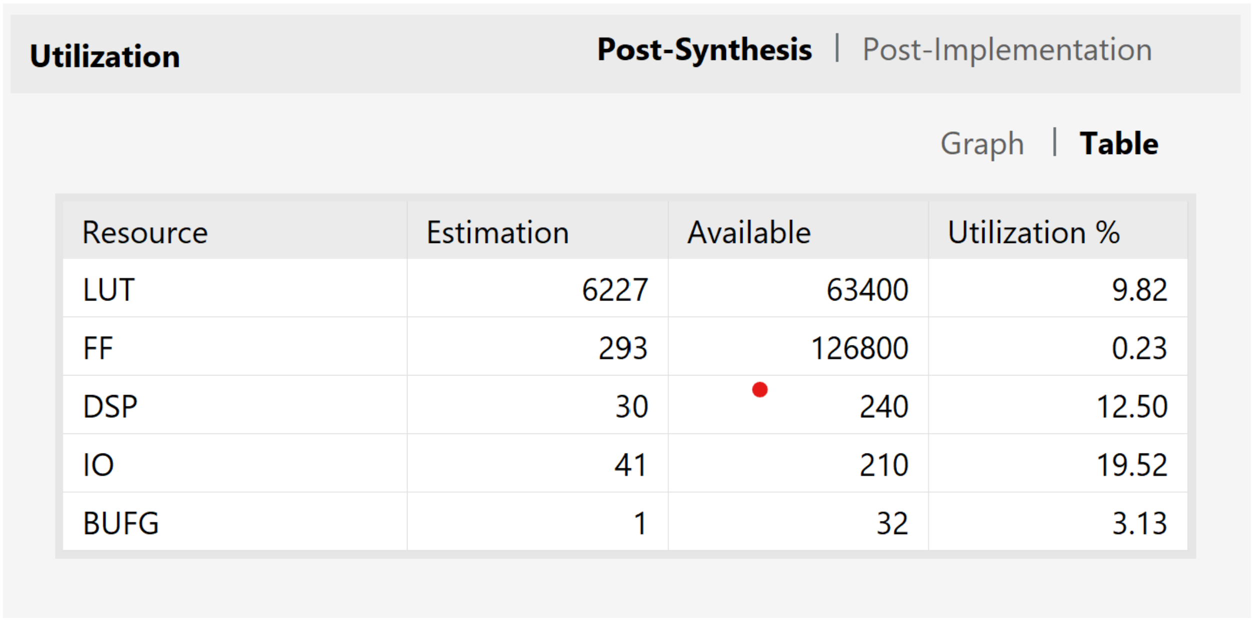
Include a video of your PWM LEDs working. It should show Only R, Only G, Only B and then mix it up. Keep it around 45 seconds or less. Narrate what you are doing. The video needs to show the RGB LED, the switches and the 7SD. Run in real time while you are changing switch values.

https://youtube.com/shorts/gC0QO26o6oQ?feature=share

12:

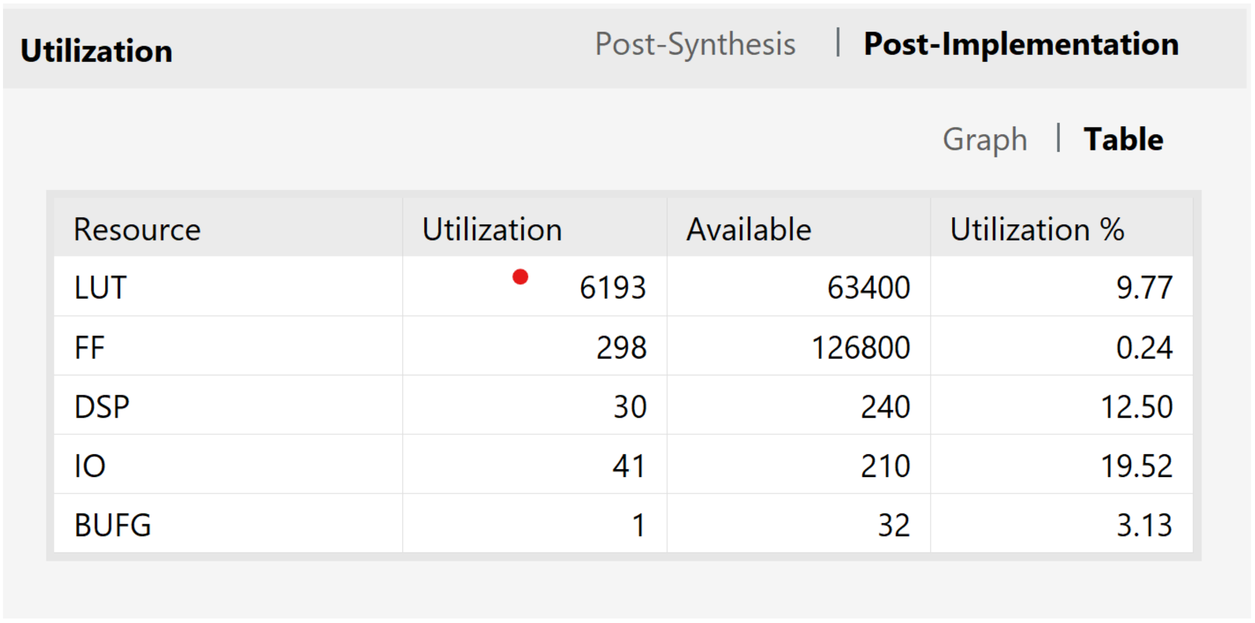
S**how utilization information from Vivado.**

The utilization below in **FIgure 12.1** represents the Verilog post-synthesis, and **FIgure 12.2** represents the utilization post-implementation.



Resource	Estimation	Available	Utilization %
LUT	6227	63400	9.82
FF	293	126800	0.23
DSP	30	240	12.50
IO	41	210	19.52
BUFG	1	32	3.13

Figure 12.1: Post-Synthesis Utilization Chart



Resource	Utilization	Available	Utilization %
LUT	6193	63400	9.77
FF	298	126800	0.24
DSP	30	240	12.50
IO	41	210	19.52
BUFG	1	32	3.13

Figure 12.2: Post-Implementation Utilization Chart

References

FPGA Handbook:
https://github.com/PacktPublishing/The-FPGAProgramming-Handbook-Second-Edition

Digilent NEXYS A7 Reference Document:

https://digilent.com/reference/programmable-logic/nexys-a7/reference-manual?srsltid=AfmBOoocSWEAjUvJc9W0T3uws_BGRb8yzAFHGeGOBHs4DKJbFoKus505

NEXYS A7 XADC Demo:

https://digilent.com/reference/programmable-logic/nexys-a7/demos/xadc?srsltid=AfmBOorAOvnyIDr39yPIEl0EhIcUJUQBQpfaY6WuBYAaIQKHeoct-Gk6