ECE 3300L

Lab Report #8

Group A

Edwin Estrada (#015897050)

Michelle Lau (#016027427)

August 14, 2025

Design

Fixed Clock Divider

```
module clock_divider_fixed #(
    parameter integer INPUT_HZ = 100_000_000,
    parameter integer TICK1_HZ = 1_000,
    parameter integer PWM_HZ = 20_000
)(
    input wire clk_in,
    input wire rst_n,
    output reg clk_1k,
    output reg clk_pwm
);
    localparam integer DIV1H = (INPUT_HZ/TICK1_HZ)/2;
    localparam integer DIVPMH = (INPUT_HZ/PWM_HZ)/2;
    reg [$clog2(DIV1H):0] c1;
    reg [$clog2(DIVPMH):0] c2;

    always @(posedge clk_in or negedge rst_n) begin
        if (!rst_n) begin c1 <= 0; clk_1k <= 0; c2 <= 0; clk_pwm <= 0; end
        else begin
            if (c1 == DIV1H-1) begin c1 <= 0; clk_1k <= ~clk_1k; end else c1 <= c1+1;
            if (c2 == DIVPMH-1) begin c2 <= 0; clk_pwm <= ~clk_pwm; end else c2 <= c2+1;
        end
    end
endmodule
```

This module creates two clocks from the 100Mhz base FPGA clock: The 1kHz clock for the debounce button module and for handling the FSM.

Debounce Pulse

```
module debounce_onepulse #(
parameter integer STABLE_TICKS = 20
)(
    input wire clk,
    input wire rst_n,
    input wire din,
    output reg pulse
);

reg d0, d1;
reg stable, stable_q;
reg [$clog2(STABLE_TICKS+1)-1:0] cnt;

always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin d0<=0; d1<=0; end else begin d0<=din; d1<=d0; end
end

always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin cnt<=0; stable<=0; end
    else if (d1 != stable) begin
        if (cnt==STABLE_TICKS) begin stable<=d1; cnt<=0; end
        else cnt<=cnt+1;
    end else cnt<=0;
end

always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin stable_q<=0; pulse<=0; end
    else begin pulse <= (~stable_q) & stable; stable_q <= stable; end
end
endmodule
```

This module handles debounce in order to prevent multiple peaks and troughs generated throughout the button press. Additionally, the pulse effect is created so the cycle button has to be released and pressed again to execute the cycle process again.

Load FSM Implementation

```
1  module load_fsm(
2      input wire clk,
3      input wire rst_n,
4      input wire load_pulse,
5      output reg [1:0] slot,
6      output wire [3:0] slot_onehot,
7      output reg wr_res, wr_r, wr_g, wr_b
8  );
9
10     assign slot_onehot = 4'b0001 << slot;
11
12     always @(posedge clk or negedge rst_n) begin
13         if (!rst_n) slot <= 2'd0;
14         else if (load_pulse) slot <= slot + 2'd1;
15     end
16
17     always @* begin
18         wr_res = 0; wr_r = 0; wr_g = 0; wr_b = 0;
19         case (slot)
20             2'd0: wr_res = load_pulse;
21             2'd1: wr_r = load_pulse;
22             2'd2: wr_g = load_pulse;
23             2'd3: wr_b = load_pulse;
24         endcase
25     end
26 endmodule
```

This module handles the current state the circuit is in. There are four states: PWM resolution, Red, Green, and Blue. Depending on the current state of the board, the resolution, red, green, and blue are changed in a single state. When the right button is pressed, the state is cycled.

PWM Core

```
23  module pwm_core(
24      input wire clk,
25      input wire rst_n,
26      input wire [7:0] period,
27      input wire [7:0] duty_r, duty_g, duty_b,
28      output reg pwm_r, pwm_g, pwm_b
29  );
30
31      wire [8:0] eff_period = {1'b0, period} + 9'd1;
32
33      function [8:0] clamp9(input [7:0] d);
34          clamp9 = ( {1'b0,d} >= eff_period ) ? (eff_period - 9'd1) : {1'b0,d};
35      endfunction
36
37      reg [8:0] cnt;
38
39      always @(posedge clk or negedge rst_n) begin
40          if (!rst_n) cnt <= 0;
41          else if (cnt == eff_period - 1) cnt <= 0;
42          else cnt <= cnt + 1;
43      end
44
45      always @(posedge clk or negedge rst_n) begin
46          if (!rst_n) {pwm_r, pwm_g, pwm_b} <= 0;
47          else begin
48              pwm_r <= (cnt < clamp9(duty_r));
49              pwm_g <= (cnt < clamp9(duty_g));
50              pwm_b <= (cnt < clamp9(duty_b));
51          end
52      end
53  endmodule
54
```

PWM period is decided by PWM resolution. The higher the period input, the higher the period is. To control the impact the R, G, and B have on the LED, a duty is applied to each color, where the counter counts through the whole period and duty splits the period to produce the impact each color has on the LED.

RGB LED Driver

```
module rgb_led_driver #(parameter ACTIVE_LOW=1)(
    input wire pwm_r, pwm_g, pwm_b,
    output wire led_r, led_g, led_b
);
    generate
        if (ACTIVE_LOW) begin
            assign led_r = ~pwm_r;
            assign led_g = ~pwm_g;
            assign led_b = ~pwm_b;
        end else begin
            assign led_r = pwm_r;
            assign led_g = pwm_g;
            assign led_b = pwm_b;
        end
    endgenerate
endmodule
```

Because the LED's are active low, the pwm signals need to be reversed to properly activate the RGB LED's.

High Level Implementation

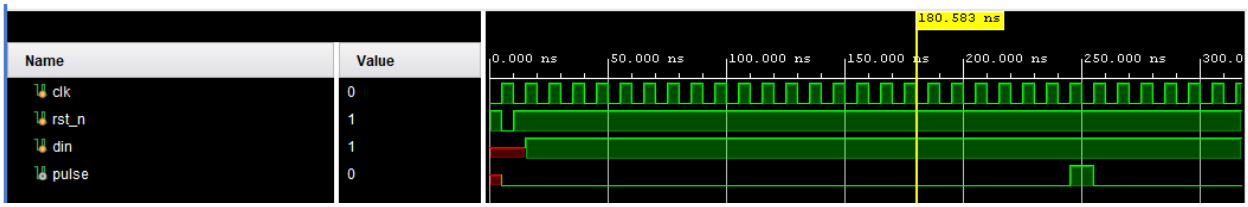
```
module top_lab8(
    input wire clk100mhz,
    input wire btnc_n,
    input wire btnr,
    input wire [7:0] sw,
    output wire [3:0] led,
    output wire rgb_r, rgb_g, rgb_b
);

    wire rst_n = ~btnc_n;
    wire clk_lk, clk_pwm;

    clock_divider_fixed
u_div(.clk_in(clk100mhz), .rst_n(rst_n), .clk_lk(clk_lk), .clk_pwm(clk_pwm));
    wire load_pulse;
    debounce_onepulse #(.STABLE_TICKS(20))
u_db(.clk(clk_lk), .rst_n(rst_n), .din(btnr), .pulse(load_pulse));
    wire [1:0] slot;
    wire [3:0] slot_oh;
    wire wr_res, wr_r, wr_g, wr_b;
    load_fsm u_fsm(
        .clk(clk_lk), .rst_n(rst_n), .load_pulse(load_pulse),
        .slot(slot), .slot_onehot(slot_oh),
        .wr_res(wr_res), .wr_r(wr_r), .wr_g(wr_g), .wr_b(wr_b)
    );
    assign led = slot_oh;

    reg [7:0] reg_res, reg_r, reg_g, reg_b;
    always @(posedge clk_lk or negedge rst_n) begin
        if (!rst_n) begin reg_res<=8'd63; reg_r<=0; reg_g<=0; reg_b<=0; end
```

```
        else begin
            if (wr_res) reg_res <= sw;
            if (wr_r) reg_r <= sw;
            if (wr_g) reg_g <= sw;
            if (wr_b) reg_b <= sw;
        end
    end

    reg [7:0] res_q1,res_q2,r_q1,r_q2,g_q1,g_q2,b_q1,b_q2;
    always @(posedge clk_pwm or negedge rst_n) begin
        if (!rst_n) begin res_q1<=0; res_q2<=0; r_q1<=0; r_q2<=0; g_q1<=0;
g_q2<=0; b_q1<=0; b_q2<=0; end
        else begin
            res_q1<=reg_res; res_q2<=res_q1;
            r_q1<=reg_r; r_q2<=r_q1; g_q1<=reg_g; g_q2<=g_q1; b_q1<=reg_b;
b_q2<=b_q1;
        end
    end

    wire pwm_r, pwm_g, pwm_b;
    pwm_core u_pwm(.clk(clk_pwm), .rst_n(rst_n),
        .period(res_q2), .duty_r(r_q2), .duty_g(g_q2), .duty_b(b_q2),
        .pwm_r(pwm_r), .pwm_g(pwm_g), .pwm_b(pwm_b));

    rgb_led_driver #(.ACTIVE_LOW(1)) u_led(
        .pwm_r(pwm_r), .pwm_g(pwm_g), .pwm_b(pwm_b),
        .led_r(rgb_r), .led_g(rgb_g), .led_b(rgb_b));
endmodule
```

High level implementation connects all modules together to control the RGB LED. The center button resets the LED and controls. The right button toggles board states to select, R, G, B, or PWM resolution. When the right button is pressed, the element of whatever mode the board is on is set and the RGB LED reacts accordingly.

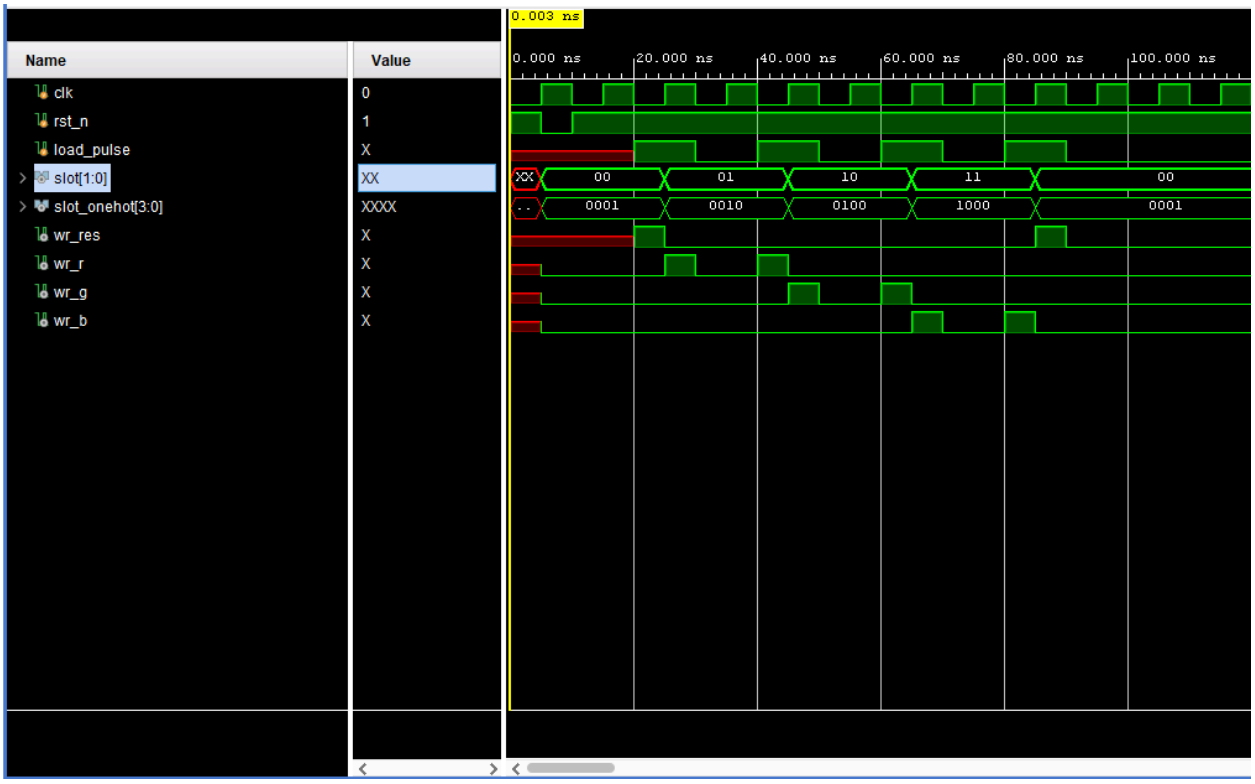
Simulation Waveform

Debounce Pulse



The screenshot shows the reset button being pressed, giving all output fields valid values. When the button is held for long enough, it is shown that a pulse is generated around 250ns into the simulation.

Load FSM



The screenshot again shows the reset button being pressed, giving all output fields valid values. Every time load_pulse is positive edge triggered (right button is pressed), the board cycles through the states of wr_res, wr_r, wr_g, and wr_b.

RGB LED Driver

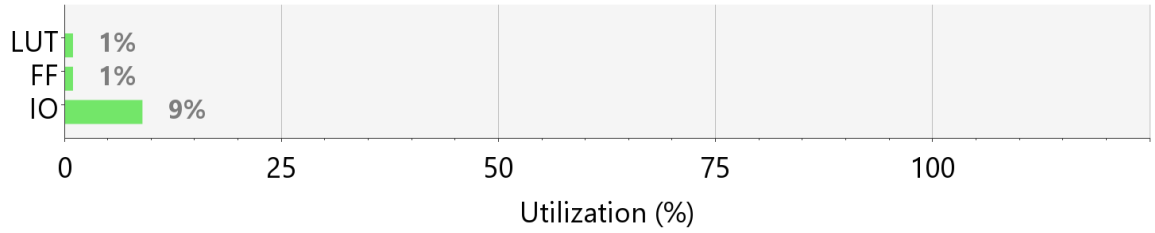
Because the LED's are active low, we need the input signals to be low to turn the LED on. For example, when RGB = 000, the LED is white (all segments of LED are on). When RGB = 111, however, the LED is off (all segments of LED are off).

Implementation

Utilization Table:

Summary

Resource	Utilization	Available	Utilization %
LUT	116	63400	0.18
FF	152	126800	0.12
IO	18	210	8.57



Timing Summary:

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	6.563 ns	Worst Hold Slack (WHS):	0.140 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	32	Total Number of Endpoints:	32	Total Number of Endpoints:	33

All user specified timing constraints are met.

Contribution

Michelle Lau (50%) - Debugging, implementation and board demo
Edwin Estrada (50%) - Programming, debugging, and test benching