Lab 8: RGB LED PWM Controller

3300L

Dia Agrawal and Robert Lainez Torres

Objective:

The objective of this lab is to design and implement a hardware-based RGB LED controller using Pulse Width Modulation (PWM) on the Nexys A7 FPGA board. By developing modular Verilog components and integrating them into a top-level system, students learn to manipulate LED brightness and blend colors based on user inputs. The lab involves capturing and processing button presses, storing user-selected values for PWM period and duty cycles, and generating smooth, flicker-free color output. This hands-on experience builds practical skills in digital logic design, finite state machines, signal debouncing, clock division, and hardware interfacing.

Design description

Clock divider Module:

The clock_divider_fixed module generates two separate lower-frequency clocks from the 100 MHz system clock: a 1 kHz clock used for debouncing buttons and controlling state transitions, and a 20 kHz clock for driving the PWM signals without visible flicker. This is essential because the default FPGA clock is too fast for direct human-interactive operations and for LED PWM. The divider counts clock ticks and toggles output clocks after specific intervals, enabling timing control across the design.

```
timescale 1ns / 1ps


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 DIV1K = (INPUT_HZ/TICK1_HZ)/2;
    localparam integer DIVPWM = (INPUT_HZ/PWM_HZ)/2;
    reg [$clog2(DIV1K):0] c1;
    reg [$clog2(DIVPWM):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 == DIV1K-1) begin
            c1 <= 0;
            clk_1k <= ~clk_1k;
         end else
         c1 <= c1+1;

            if (c2 == DIVPWM-1) begin c2 <= 0; clk_pwm <= ~clk_pwm; end else c2
 <= c2+1;
        end
    end
endmodule
```

Debouncing and one pulse:

The debounce_onepulse module processes mechanical button signals to eliminate noise and bouncing effects. Mechanical switches naturally produce spurious transitions when pressed or released, which this module filters out by checking for stable input over multiple clock cycles. It also ensures that each valid press generates a single, clean pulse, regardless of how long the button is held. This pulse is used to trigger finite state machine

transitions and data loading.

```
3
4   module debounce_onepulse #(
5       parameter integer STABLE_TICKS = 20
6   )(
7       input wire clk,
8       input wire rst_n,
9       input wire din,
10      output reg pulse
11  );
12
13      reg d0, d1;
14      reg stable, stable_q;
15      reg [$clog2(STABLE_TICKS+1)-1:0] cnt;
16      always @(posedge clk or negedge rst_n) begin
17          if (!rst_n) begin d0<=0; d1<=0; end else begin d0<=din; d1<=d0; end
18      end
19      always @(posedge clk or negedge rst_n) begin
20          if (!rst_n) begin cnt<=0; stable<=0; end
21          else if (d1 != stable) begin
22              if (cnt==STABLE_TICKS) begin stable<=d1; cnt<=0; end
23
24              else cnt<=cnt+1;
25          end else cnt<=0;
26      end
27
28      always @(posedge clk or negedge rst_n) begin
29          if (!rst_n) begin stable_q<=0; pulse<=0; end
30          else begin pulse <= (~stable_q) & stable; stable_q <= stable; end
31      end
32  endmodule
```

Load FSM (Finite State Machine):

The load_fsm module manages the process of cycling through four memory slots that hold PWM configuration values. Since the board only has one load button, this FSM cycles through different load states resolution, red duty cycle, green duty cycle, and blue duty cycle every time the button is pressed. It also generates write-enable signals for the appropriate registers, ensuring the correct value is captured from the switches at the right time.

```
`timescale 1ns / 1ps


module load_fsm(
    input wire clk,
    input wire rst_n,
    input wire load_pulse,
    output reg [1:0] slot,
    output wire [3:0] slot_onehot,
    output reg wr_res, wr_r, wr_g, wr_b
);
    assign slot_onehot = 4'b0001 << slot;
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) slot <= 2'd0;
        else if (load_pulse) slot <= slot + 2'd1;
    end

    always @* begin
        wr_res = 0; wr_r = 0; wr_g = 0; wr_b = 0;
        case (slot)
            2'd0: wr_res = load_pulse;
            2'd1: wr_r = load_pulse;
            2'd2: wr_g = load_pulse;
            2'd3: wr_b = load_pulse;
        endcase
    end
endmodule

```

PWM Core Module:

The pwm_core module is responsible for generating the actual PWM signals for each color channel (Red, Green, Blue) based on the stored duty cycles and period (resolution). It compares a counter against the duty cycle to determine the high or low state of each PWM signal. By adjusting these values, different brightness levels and color combinations are achieved. This module ensures accurate color representation through timed digital outputs**.**

```
module pwm_core(
    input wire clk,
    input wire rst_n,
    input wire [7:0] period,
    input wire [7:0] duty_r, duty_g, duty_b,
    output reg pwm_r, pwm_g, pwm_b
);

    wire [8:0] eff_period = {1'b0, period} + 9'd1;

    function [8:0] clamp9(input [7:0] d);
        clamp9 = ( {1'b0,d} >= eff_period ) ? (eff_period - 9'd1) : {1'b0,d};
    endfunction

    reg [8:0] cnt;
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) cnt <= 0;
        else if (cnt == eff_period - 1) cnt <= 0;
        else cnt <= cnt + 1;
    end

    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) {pwm_r, pwm_g, pwm_b} <= 0;
        else begin
            pwm_r <= (cnt < clamp9(duty_r));
            pwm_g <= (cnt < clamp9(duty_g));
            pwm_b <= (cnt < clamp9(duty_b));
        end
    end
endmodule
```

RGB LED Driver:

The rgb_led_driver module adapts the active logic level of the PWM signals to match the hardware requirements of the RGB LEDs. On the Nexys A7 board, LEDs are active-low, meaning they turn on when the signal is low. This module inverts the PWM output signals if needed, based on the ACTIVE_LOW parameter. It provides flexibility for compatibility with various LED configurations.

```
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
```

Top-Level Integration Module:

The top_lab8 module ties together all the other modules into a single, functional system. It coordinates clock domains, handles user inputs (button and switches), stores values in registers, and drives the PWM outputs to the RGB LEDs. It also ensures proper crossing between the 1 kHz (user input) and 20 kHz (PWM) domains using flip-flop synchronization. This top-level design is the complete RGB LED controller system that implements user-configurable, real-time LED brightness and color blending.

```
reg [7:0] reg_res, reg_r, reg_g, reg_b;
always @(posedge clk_1k or negedge rst_n) begin
    if (!rst_n) begin
        reg_res<=8'd0;
        reg_r<=0;
        reg_g<=0;
        reg_b<=0;
    end
    else begin
        if (wr_res) reg_res <=sw;  //fixed RES
        if (wr_r) reg_r <=  sw;   // fixed RED Duty cycle
        if (wr_g) reg_g <=  sw;   // fixed GREEN Duty Cycle
        if (wr_b) reg_b <=  sw; // fixed BLUE Duty Cycle
    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
```

```
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_1k, clk_pwm;

    clock_divider_fixed #(.INPUT_HZ(100_00_000)) u_div(
     .clk_in(clk100mhz),
     .rst_n(rst_n),
     .clk_1k(clk_1k),
     .clk_pwm(clk_pwm));

    wire load_pulse;

  debounce_onepulse #(.STABLE_TICKS(20)) u_db(
    .clk(clk_1k),
    .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_1k),
        .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;
```

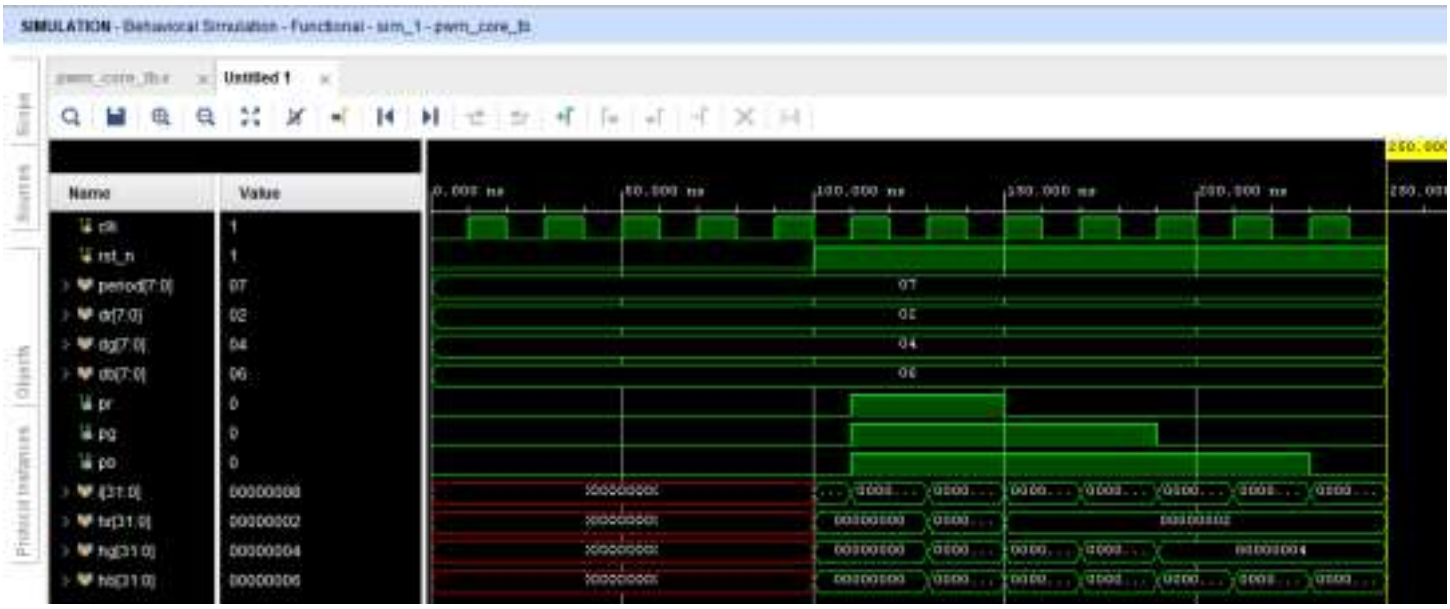
```
wire pwm_r, pwm_g, pwm_b;

    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
```

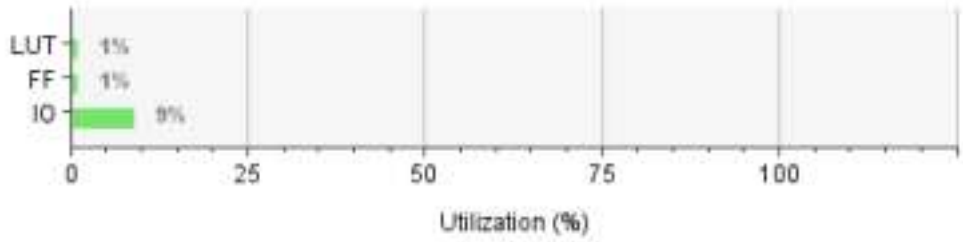
Test Bench



The pwm_core_tb.v testbench is designed to verify the functionality of the pwm_core module by simulating its behavior in a controlled environment. It applies a known PWM period and specific duty cycle values for the red, green, and blue channels, then counts how many clock cycles each output is high over a full PWM period. By simulating 8 clock cycles with a period of 7 (effective period of 8), it expects the red channel to be high for 2 cycles, green for 4, and blue for 6, corresponding to their respective duty cycles. The testbench prints out the high-time counts for each channel, allowing the user to confirm that the pwm_core module is producing correct PWM outputs. This automated test helps ensure that the module behaves as intended before integrating it into the full system.

Utilization report:

Resource	Utilization	Available	Utilization %
LUT	103	63400	0.16
FF	145	126800	0.11
IO	18	210	8.57



Team Contributions:

As usual, we both contributed throughout the design, testing, and documentation process, often working together to integrate and verify our modules.

Robert's Contributions:

I created my own modules and worked on errors with my partner. I got the testbench simulation and utilization report.

Dia's Contributions:

I also created my own modules and worked on the errors. I uploaded the code to the board and created the demo.

We both participated in debugging, testbench design, and reviewing each other's code to ensure correctness. Final documentation and polishing were done collaboratively.