ECE 3300 – Lab 8 Report
RGB LED PWM Controller
Team Name: Group V
Date: August 14, 2025

Group Members

Nathan Marlow, Khristian Chan

1. Objective

We used a 4 slot loading system in this lab in our load_fsm finite state machine to step through four functional states each time the debounced load button (btn_r/Button Right) is pressed. The slots are arranged as Reset, Red Value Load, Green Value Load, and Blue Value Load. When in the Reset slot, all three PWM duty cycle registers are cleared, which turns on our RGB LED because it is active-low. Pressing the load button advances the FSM to the next slot, indicated on LED[3:0] as 0001 (Reset), 0010 (Red), 0100 (Green), or 1000 (Blue). In any color slot, the value on SW[7:0] (basically any value between 0-255, similar to hexadecimal 00-ff) is loaded into that color's duty register when the slot is first entered. This design follows a "reset + increment-by-1" pattern, where each button press moves to the next slot in sequence and wraps around after the Blue slot back to the Reset State. This allows us to easily program the brightness for each channel one at a time using just a single load control, while still also having full 8-bit resolution for each color.

2. Design Modules

2.1 clock_divider_fixed.v

```
`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 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
```

A clock divider was implemented using parameters and local parameters. Without having to do the math ourselves,we can use parameters and local parameters to set up our clock divider, specifying clock Hz, along with desired output frequency and using $clog2 to determine the bit-width needed for our exact clock divider circuit.

2.2 debounce_onepulse

```
`timescale 1ns/1ps
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
```

Given a stable button press, a single pulse will be outputted from the button.

2.3 load_fsm.v

```
`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,
    output reg wr_r,
    output reg wr_g,
    output reg 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
```

Although this module looks similar to our 7 segment drivers, it works slightly differently. Instead of loading a hex value, we load a pulse which is our pwm signal.

2.4 rgb_led_driver.v

```
`timescale 1ns/1ps
module rgb_led_driver #(
    parameter ACTIVE_LOW = 1
)(
    input wire pwm_r,
    input wire pwm_g,
    input wire pwm_b,
    output wire led_r,
    output wire led_g,
    output wire 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
```

This module assigns each led to a pwm signal. Additionally, it can swap between an active low and active high rgb led.

2.5 pwm_core

```
`timescale 1ns/1ps
module pwm_core(
    input wire clk,
    input wire rst_n,
    input wire [7:0] period,
    input wire [7:0] duty_r,
    input wire [7:0] duty_g,
    input wire [7:0] duty_b,
    output reg pwm_r,
    output reg pwm_g,
    output reg 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
```

This module is a pwm signal generator for our RGB led's. In order to generate a pwm signal, we take in our period, along with the duty cycle that we would like. While a counter increments to the max period, our pwm output is determined by seeing if cnt is less than our desired duty cycle. If it is, PWM goes high.

2.6 Top-Level Integration

```
`timescale 1ns/1ps
module top_lab8(
    input wire clk100mhz,
    input wire rst_n, //rst_n
    input wire btnr,
    input wire [7:0] sw,
    output wire [3:0] led,
    output wire rgb_r,
    output wire rgb_g,
    output wire rgb_b
);
    wire clk_1k, clk_pwm;
    clock_divider_fixed 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;
    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'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
```

3. Implementation Results

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs	117	0	0	63400	0.18
LUT as Logic	117	0	0	63400	0.18
LUT as Memory	0	0	0	19000	0.00
Slice Registers	152	0	0	126800	0.12
Register as Flip Flop	152	0	0	126800	0.12
Register as Latch	0	0	0	126800	0.00
F7 Muxes	0	0	0	31700	0.00
F8 Muxes	0	0	0	15850	0.00

* Warning! LUT value is adjusted to account for LUT combining.

Total On-Chip Power (W)	0.086
Design Power Budget (W)	Unspecified*
Power Budget Margin (W)	NA
Dynamic (W)	0.002
Device Static (W)	0.084
Effective TJA (C/W)	4.6
Max Ambient (C)	84.6
Junction Temperature (C)	25.4
Confidence Level	Low
Setting File	---
Simulation Activity File	---
Design Nets Matched	NA

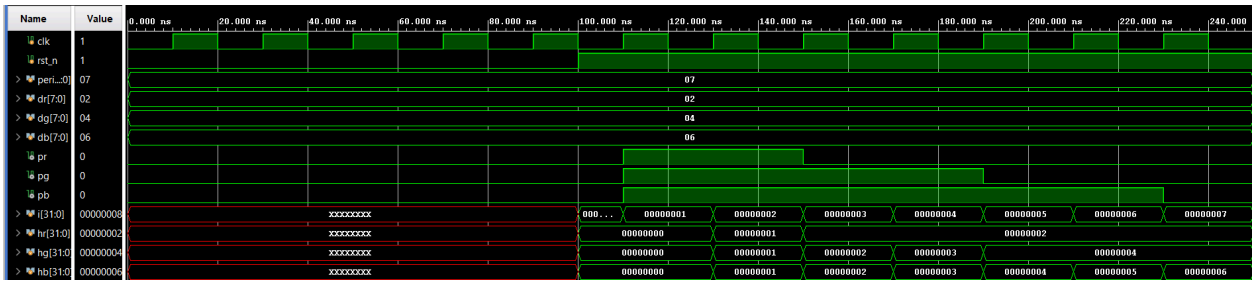
* Specify Design Power Budget using, set_operating_conditions -design_power_bu

1.1 On-Chip Components

On-Chip	Power (W)	Used	Available	Utilization (%)
Clocks	0.001	3	---	---
Slice Logic	<0.001	325	---	---
LUT as Logic	<0.001	117	63400	0.18
Register	<0.001	152	126800	0.12
CARRY4	<0.001	20	15850	0.13
Others	0.000	7	---	---
BUFG	0.000	2	32	6.25
Signals	<0.001	262	---	---
I/O	<0.001	18	210	8.57
Static Power	0.084			
Total	0.086			

4. Simulation Results:

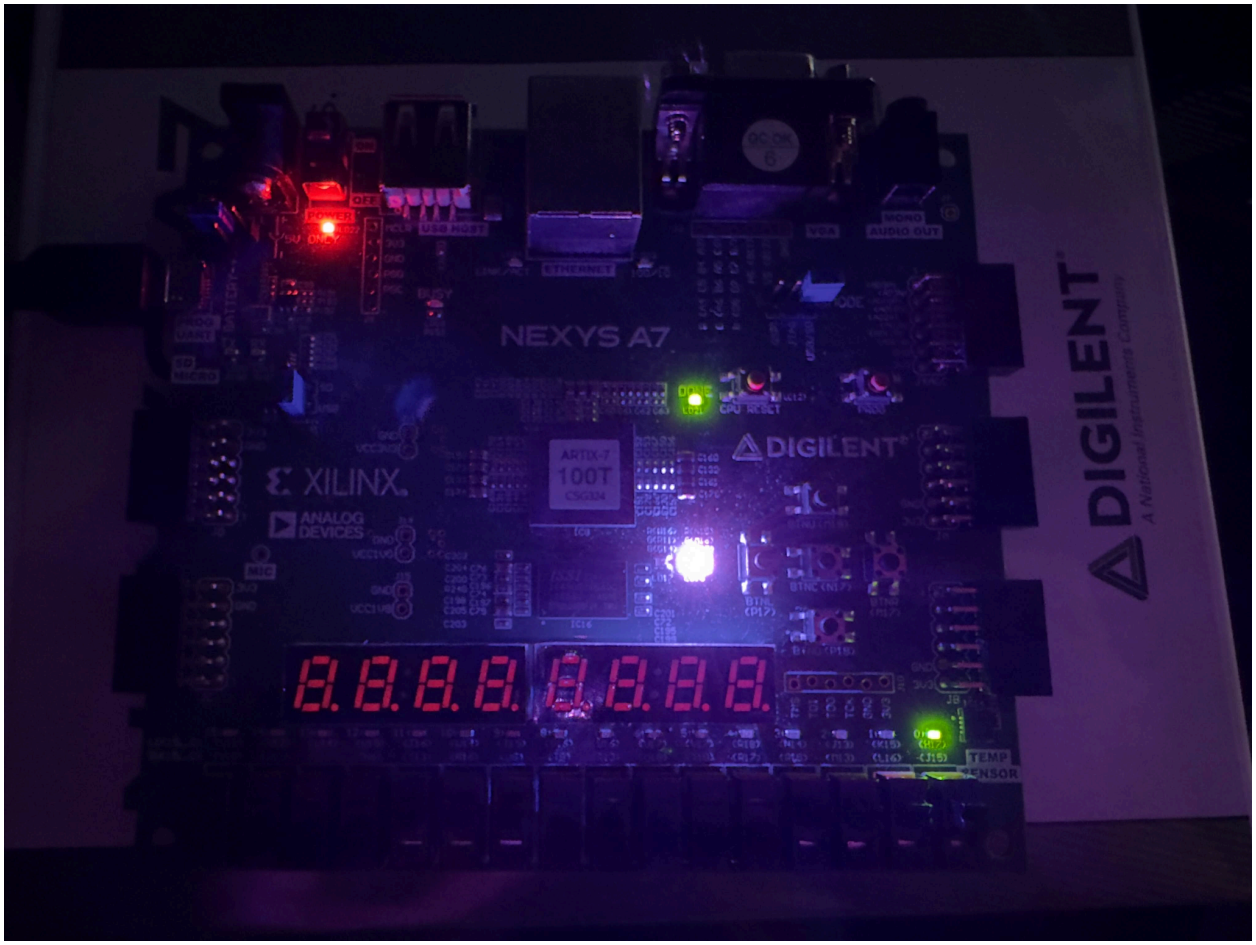
4.1 - pwm_core_tb



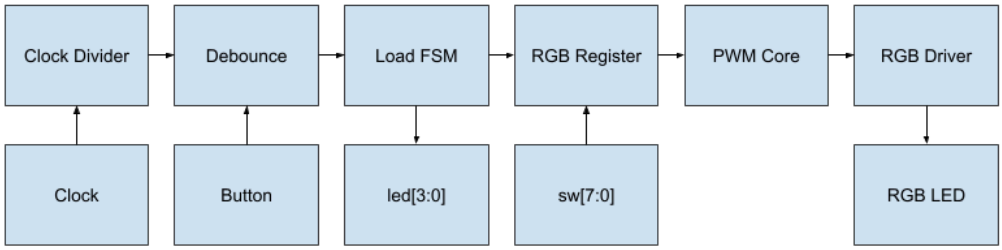

```
Time resolution is 1 ps
R=2/8 G=4/8 B=6/8 (expected: ~2,4,6)
$finish called at time : 250 ns : File "C:/Users/khris/lab_8/lab_8.srcs/sim_1/new/pwm_core_tb.v" Line 33
```

We see that our simulation is correct. Our expected output is 2, 4, and 6 and we see those values in both our waveform and log.

4. Board Photo



6. Block Diagram



7. Demonstration Video

https://youtu.be/23jqtF8r5a0?si=1WLqw7LKQn4hRh1-

Contributions:

Khristian Chan- 50%: Testbench, Simulation, Debugging, Report, Demo
Nathan Marlow- 50%: Implementation, Code, XDC, Report