FPGA Health Monitor Report

Tafita Rakotozandry

December 2nd, 2019

1. Abstract

A health monitor is essential for individuals to keep track of their health for their well beings. This report develops a design used to make one health monitor system using FPGA. That health monitor is equipped with a pulse monitor which tracks the heartbeat speed of the user and a reaction timer which measures the reaction time of the user. The report shows that the final system meets with the majority of all requirements of the specification.

2. Introduction

Being healthy is key for every individual's happiness. Therefore, it is important to stay healthy. Nowadays, it is expensive to buy medications or to consult doctors . That is why people prefer to prevent diseases rather than to cure disease. People take their health seriously. A health monitor is a device that allows an individual to keep track of his/her health situation. Electrical engineers are working on several designs to make health monitors as efficient and affordable as possible . That is why technology like the Apple Watch has become in high demand. Health monitors are equipped with different sensors that allow them to read the heartbeat and blood pressure of the user in real-time. In this project, we are using an FPGA in order to design a health monitor module. That module provides the following specification to its user:

- Provide a simple pulse monitor that reads the heart beat of its user every 5 seconds
- Display the average of the measured heart beat in a 7 segment displays

- Measure the reaction speed of the user using a timer system
- Display the reaction speed on the 7 segment display for a possibility of recording the data
- Provide the user the option to choose between these two modes via a controller switch.

This technical report will examine all the modules that make the health monitor starting from the high level module to the submodule.

3. System Design

3.1. High-Level Design

Figure 1 depicts a diagram of the complete health monitor module. This module encapsulates pulse monitor and reaction timer. It uses a switch mode to select between the two sub modules. The reading output of each operation will be displayed in a 7 segment display as well as in an LED.

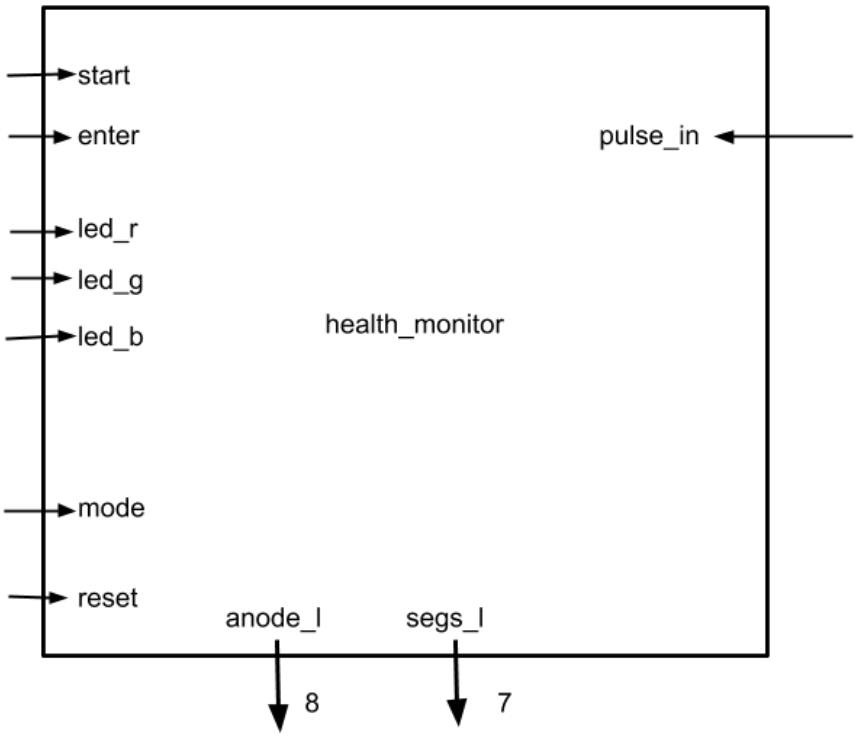

Figure 1: High-Level Module

3.2 Implementation

This section describes the modules that make up the health monitor high module. The overall modules were implemented using System Verilog. A Nexys 4 DDR board was used to test every implementation.

3.2.1 Top-level

The top-level module instantiates all the submodules within the design. Figure 2. Depict the organization of the top-level module.

a. Inputs

- clk: The 100Mhz clock provided by an external oscillator on the development board.
- reset. A push-button input that resets the health monitor back to an initial state
- pulse_in: introduce the signal read by an analog sensor that reads the user ‘s heartbeat
- start: A push-button input that starts reaction timer operation
- enter: A push-button input that performs the reaction of the user

b. Outputs

- led_r: A 3-bit active high signal that controls the red color of an LED
- Led_g: A 3-bit active high signal that controls the green color of an LED
- Led_b: A 3-bit active high signal that controls the blue color of an LED

c. Implementation and Design

Figure 2 depicts the organization of the top-level module. The top level module consists of 5 main modules: clock divider (clkdiv), pulse monitor, reaction timer, 16 bit 2 to 1 multiplexer and a seven-segment controller. The pulse monitor module and the reaction timer submodule are connected to each other by the multiplexer as presented in Figure 2.

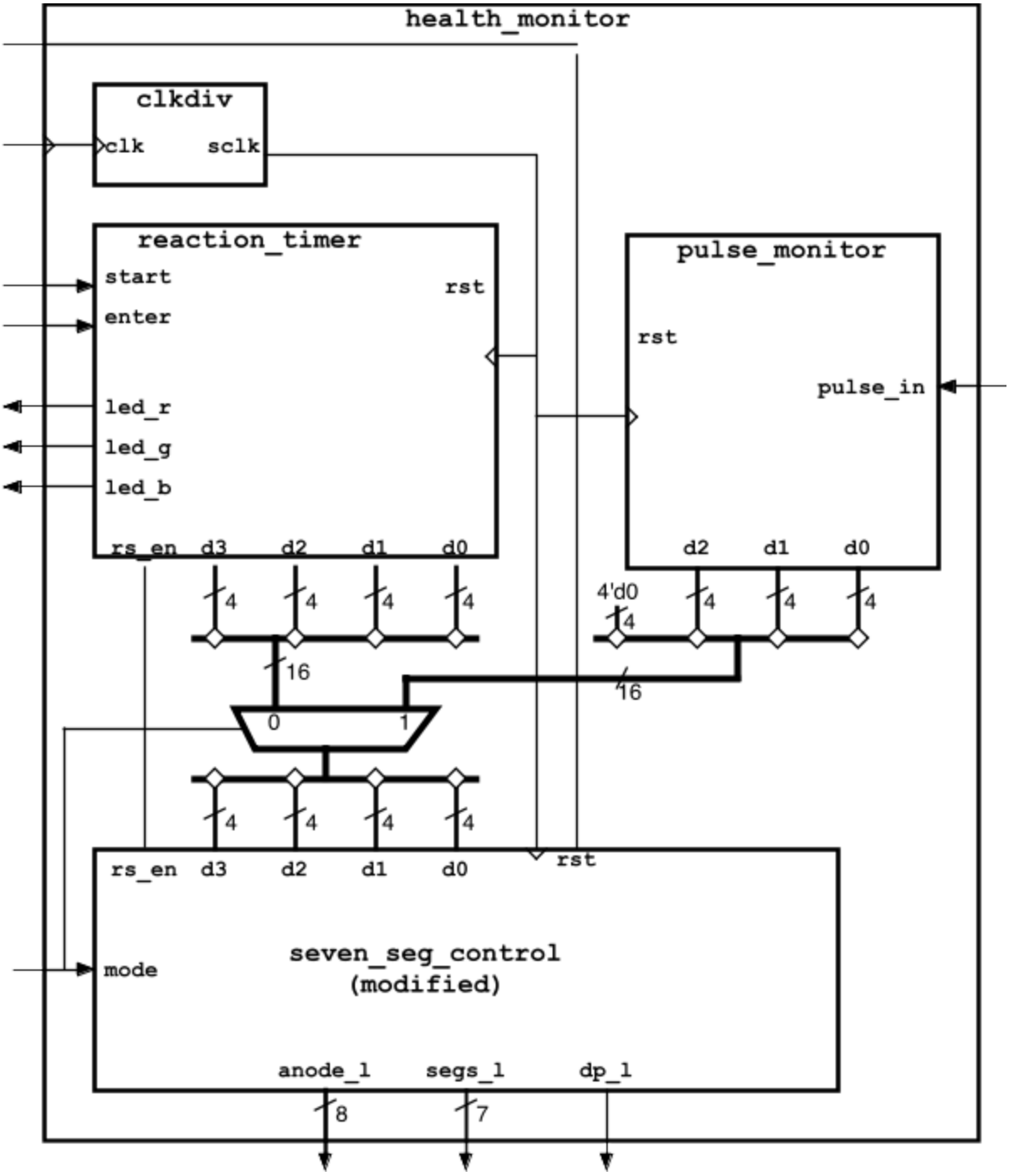

Figure 2: High-level module detailed

Code:

```
`timescale 1ns / 1ps

module project_top(input logic clk100MHz, rst, start, enter, mode,
pulse_in,
                   output logic led_r, led_g, led_b, dp_1,
                   output logic [7:0] anode_1,
                   output logic [6:0] segs_1 );

logic [3:0] d0,d1,d2,d3,pd,pd0,pd1,pd2;
logic rs_en;
logic clk_1000Hz;
logic start_in , start_out ;
 logic enter_in, enter_out ;
clk_div#(.DIVFREQ(1000))      CLK(.clk(clk100MHz),      .reset(1'b0),
.sclk(clk_1000Hz));
      debounce      START_DEBOUNCE(.clk(clk),      .pb(start)      ,
.pb_debounced(start_in));
single_pulser         START_PULSER(.clk(clk),         .din(start_in),
.d_pulse(start_out));
debounce        ENTER_DEBOUNCE(.clk(clk),        .pb(enter)        ,
.pb_debounced(enter_in));
single_pulser         ENTER_PULSER(.clk(clk),         .din(enter_in),
.d_pulse(enter_out));
pulse_monitor_top          PULSE(.clk(clk_1000Hz),          .rst(rst),
.pulse_in(pulse_in), .d0(pd), .d1(pd0), .d2(pd1), .d3(pd2) );
reaction_top  REACTION(.clk(clk_1000Hz),  .rst(rst),  .start(start),
.enter(enter),.d0(d0),.d1(d1),.d2(d2),.d3(d3),         .led_r(led_r),
.led_g(led_g), .led_b(led_b), .re_en(re_en));

logic [15:0] pm_out , rt_out;
assign pm_out = {pd2,pd1,pd0,pd} ;
assign rt_out = {d3,d2,d1,d0};
logic [15:0] q;
logic [6:0] segs;
logic [7:0] anode;
logic dp;
mux_16bit_2_1     MUX(.mode(mode)      ,     .reaction_timer(rt_out),
.pulse_mon(pm_out) , .q(q)) ;
sevenseg_top               SEVENSEG(               .clk(clk_1000Hz),
.rst(rst),.mode(mode),.rs_en(rs_en),     .d0(q[3:0]),     .d1(q[7:4]),
.d2(q[11:8]),    .d3(q[15:12]),    .d4(4'd0),    .d5(4'd0),.d6(4'd0),
.d7(4'd0), .segs_1(segs), .anode_1(anode), .dp_1(dp));
assign segs_1 = segs;
assign anode_1 = anode;
assign dp_1 = dp;
endmodule
```

3.2.2 Reaction timer

3.2.2.1 Top Level

The reaction timer is a module that measures the reaction time of its user. It counts how long the user reacts after a run signal is launched. Figure 4 represents the top module of the reaction timer

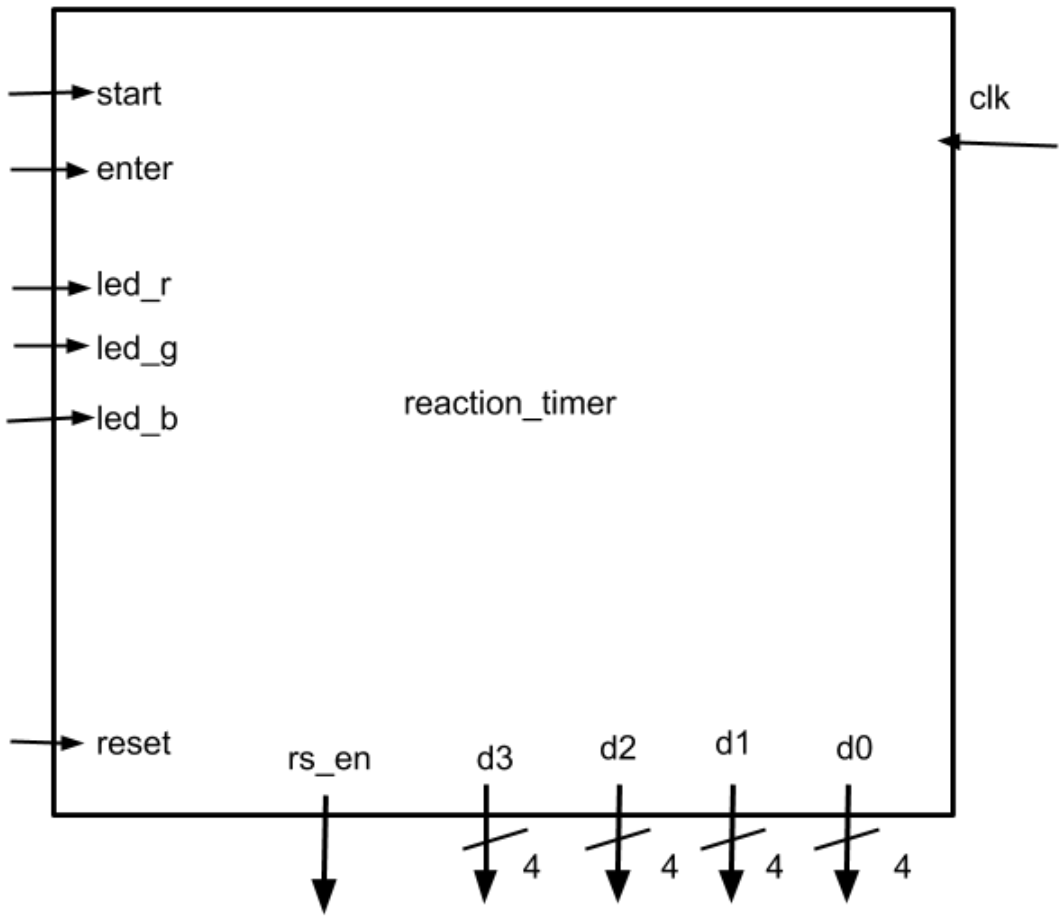

Figure 3: Reaction timer module

a. Inputs

- start: A push-button input that starts reaction timer operation
- enter: A push-button input that performs the reaction of the user
- rst: A push-button to restart the operation
- clk : clock divided

b. Outputs

The reaction timer posses 7 outputs :

- led_r: A 3-bit active high signal that controls the red color of an LED
- led_g: A 3-bit active high signal that controls the green color of an LED
- led_b: A 3-bit active high signal that controls the blue color of an LED
- d3: ones
- d2: tenth
- d1: hundredth
- d0: thousandth

c. Implementation and Design

Figure 4 represents the submodules of the reaction timer. The reaction timer top-level module instantiates different submodules of the reaction timer. It possesses 4 main submodules:

- reaction_fsm which describes the state diagram of the reaction timer into a hardware description language
- random_wait which generate a random wait time before the start signal will be given to the user
- delay_counter which counts during 5 seconds
- rgb_pwm which takes care of the color processing for the LED output
- time_count which counts the time until the user reacts to the starting signal.

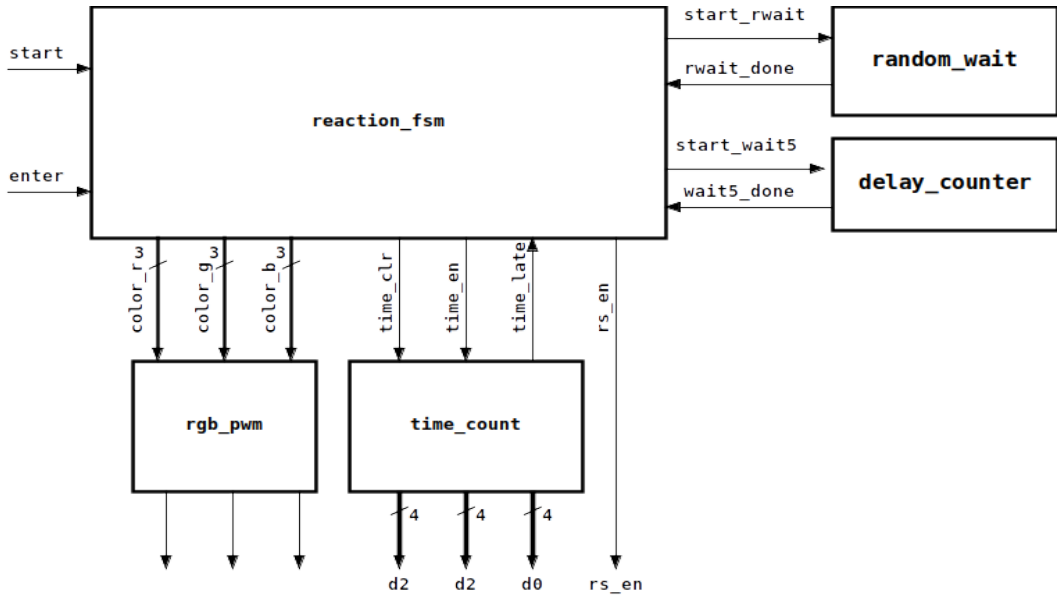

Figure 4: Reaction timer sub-modules configuration

3.2.2.2 reaction_fsm

a. Inputs

- Start: A push-button input that starts the overall module operation
- Enter: A push-button input that needs to be pressed only when the GO_LED signal is launched . An error state will happen if it is pressed on any other state
- rwait_done: high or low output sent by the random_wait module to signal whether the random wait time, that the user has to adhere by before being allowed to hit the enter, has passed.
- wait5_done: high or low output sent by the delay_counter to signal whether the five-second count has passed.
- time_late: a high or low signal sent by the time_count module, which essentially tells the user that the 10 second time period in which they had to press enter pushbutton the reaction time is up.

b. Outputs

• color_r [2:0]: turn on the red light of the RGB LED
• color_g [2:0]: turns on the green light of the RGB LED
• color_b [2:0] : turns on the blue light of the RGB LED
• time_clr: output signal which as the reset signal for the time_count module
• time_en: output signal to initiate the reaction timer in the time_count module
• rs_en: output signal sent to the seven_seg_control to display the current state of the reaction timer, once the enter button is pressed.
• start_rwait: output signal to the random_wait module to generate a random wait time for the
• start_wait5: output signal to the delay_counter module to start the five-second counter.

c. Functionality and design

The module's initial state is the IDLE state. When the start pushbutton is launched, the idle state will transition from that previous state into the next: r_wait. The system will stay for 5 seconds in this state before launching the GO LED. The reaction timer will wait until 10 seconds before changing to late state.

The state diagram of the overall is shown below

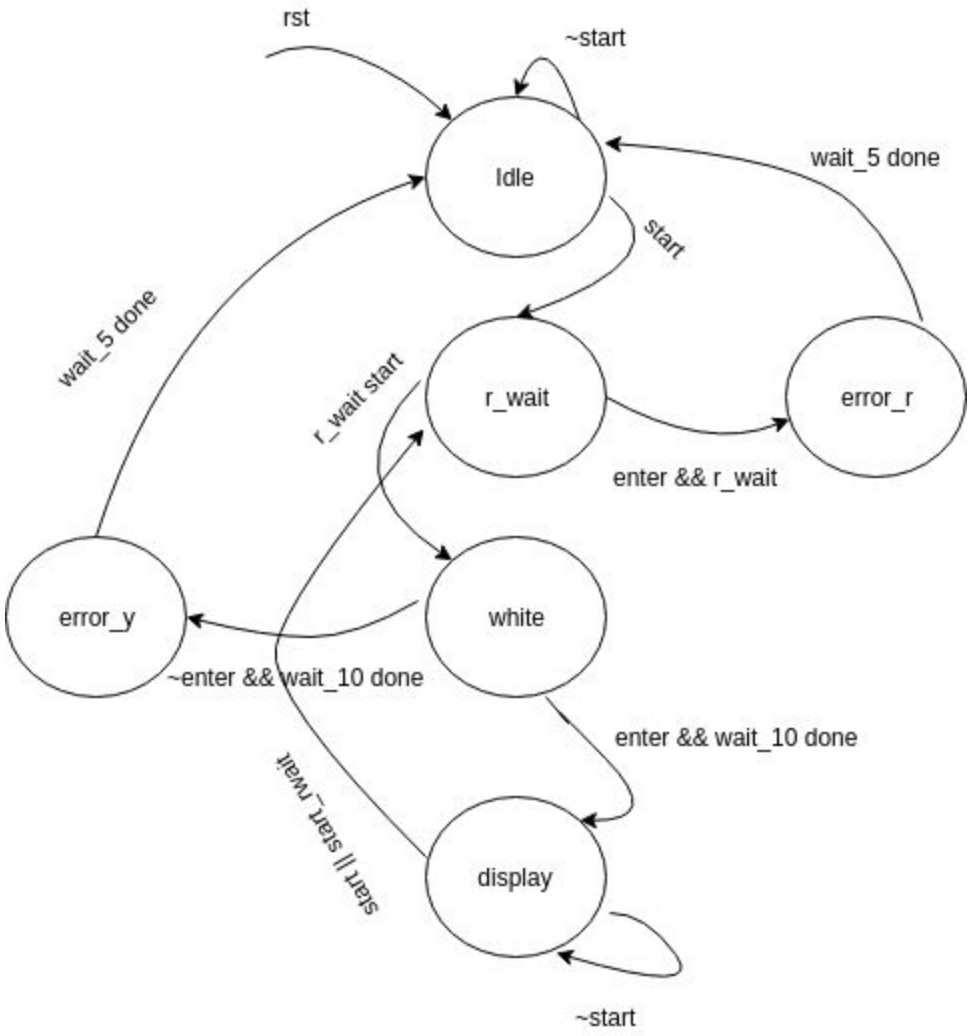

Figure 5: Finite State machine of the reaction FSM

Code:

```
`timescale 1ns / 1ps

module reaction_fsm(input logic clk, start, enter, rst, rwait_done,
wait5_done, time_late,
                    output logic start_rwait, start_wait5, time_clr,
time_en, rs_en, output logic [2:0] color_r, color_g, color_b);

typedef enum logic [2:0] {
idle= 3'b000, rwait= 3'b001, white= 3'b010, display= 3'b011, error_y=
3'b100, error_r= 3'b101
} state;

state p_s, n_s;
```

```
always_ff @(posedge clk)
begin
if (rst)  p_s <= idle;
else      p_s <= n_s;
end

localparam logic on = 1'b1 , off = 1'b0;


always_comb
 begin

color_r <= 3'd0;
color_g <= 3'd0;
color_b <= 3'd0;
start_rwait <= off;
start_wait5 <= off;
rs_en <= off;
time_clr <= off;
time_en<= off;
n_s =idle;

case(p_s)
idle:
begin
color_r <= 3'd0;
color_g <= 3'd0;
color_b <= 3'd0;
start_wait5 <= off;
rs_en <= off;
time_clr <= off;
time_en<= off;


if(start) begin
n_s <= rwait;
start_rwait <= on;
end
else begin
n_s <= idle;
end
```

```
end


rwait:
begin
color_r <= 3'd1;
color_g <= 3'd1;
color_b <= 3'd1;
start_rwait <= off;
start_wait5 <= off;
time_clr <= on;
time_en<= off;

if(enter && !rwait_done) begin
n_s <= white;
end
else if(!enter && !rwait_done) n_s <=rwait;
else if (rwait_done)begin
n_s <= error_r;
time_en=on;
end
else n_s=rwait;
end


white:
begin
color_r <= 3'd1;
color_g <= 3'd0;
color_b <= 3'd0;
start_rwait <= off;
start_wait5 <= off;
rs_en <= off;
time_clr <= off;
time_en = on;

if(enter && ~time_late) begin
time_en=off;
n_s <= display;
end
else if (~enter && ~time_late)
n_s = error_y;
else if (time_late)time_clr
```

```
n_s = error_r;
else n_s=white;
end


error_y:
begin
color_r = 3'd1;
color_g = 3'd0;
color_b = 3'd0;
start_rwait <= off;
start_wait5 <= on;
rs_en <= off;
time_clr <= off;
if (~wait5_done)
n_s = error_y;
 else
n_s = idle;
end


error_r:
begin
color_r <= 3'd3;
color_g <= 3'd1;
color_b <= 3'd0;
start_rwait <= off;
start_wait5 <= on;
rs_en <= off;
time_clr <= off;
time_en<= off;

if(wait5_done) begin
n_s <= idle;
end
else begin
n_s <= error_r;
end
end


display:
begin
```

```
color_r <= 3'd0;
color_g <= 3'd0;
color_b <= 3'd0;
start_rwait <= off;
rs_en <= on;
time_clr <= off;
time_en<= off;

if(start) begin
n_s <= rwait;
start_rwait <= on;
end
else begin
n_s <= display;
end
end

endcase

end
endmodule
```

3.2.2.3 random_wait

The random wait is part of the module that is supposed to generate a random wait time However, in this project , that number was fixed to a specific number for the sake of simplicity.

a. Inputs

- start_rwait: input signal sent by the reaction timer FSM to initiate the random counter
- clk: clock signal
- rst: reset signal
-

b. Output

rwait_done: output signal sent to the reaction timer FSM to signal the change of state to allow the user to user to enter push button to record reaction time

c. Design Implementation

It is made up of a counter that increments and a comparator that compares the incremented value with the aimed number

Code :

```
`timescale 1ns / 1ps

module random_wait(input logic clk, rst, start_rwait, output logic
rwait_done);

always_ff @(posedge clk) begin

if(start_rwait)
delay_counter WAIT (.clk(clk), .rst(rst), .out(rwait_done));

end

endmodule
```

3.2.2.4 delay_counter

a. Input

start_wait5: output signal from the reaction FSM to initiate the five-second counter

b. Output

wait5_done: a signal sent to the reaction FSM to signal that the five-second counter is complete

c. Implementation

this module will count until 5 seconds before it asserts high output.

Code :

```
`timescale 1ns / 1ps

module delay_counter(input logic clk, rst,
                     output logic out);

logic [12:0] q;
always_ff @ (posedge clk)
begin

q <= q +1;
if (q == 13'd5000)
begin
```

```
q <= 0;
out <= 1 ;
end
else
out <= 0;
end


endmodule
```

3.2.2.5 rgb_pwm

rgb_pwm uses the input about the color rgb into actual color configuration color in the LED.

a. Input

[2:0] color_r, color_g, color_b: inputs represent the intensities of the red, green, and blue LEDs

b. Output

led_r, led_g, led_b: output signals to turn on/ off corresponding LEDs on the FPGA

c. Functionality and Design

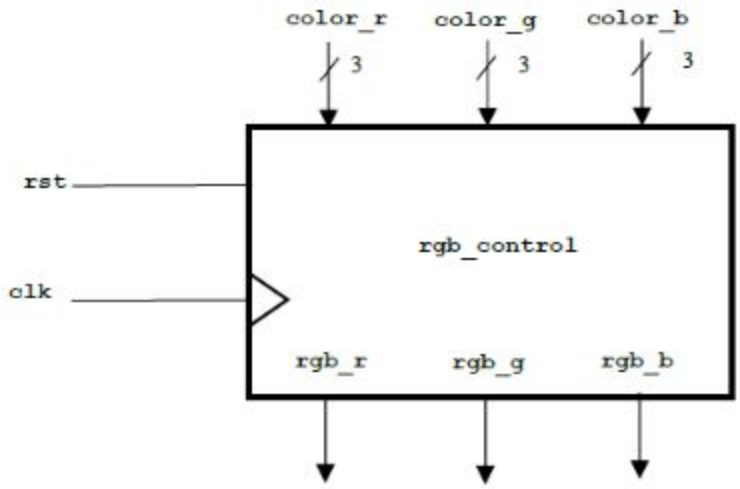

Figure 6: rgb-pwm module

Code:

```
`timescale 1ns / 1ps


module rgb_pwm(input logic clk, rst,
            input [2:0]  color_r, color_g, color_b,
            output logic led_r, led_g, led_b);

   logic [3:0] rbgcount;

   always_ff @(posedge clk)
     if (rst) rbgcount <= 0;
     else rbgcount <= rbgcount + 1;

   assign led_r = (rbgcount < color_r);
   assign led_g = (rbgcount < color_g);
   assign led_b = (rbgcount < color_b);

endmodule // rgb_pwm
```

3.2.2.6 time_count

a. Input

clk: clock signal
time_clr : clear time
time_en: enable time
time_late: late time

b. Output

[3:0] d0: thousands
[3:0] d1: hundredths
[3:0] d0: tenths
[3:0] d0: ones

c. Functionality and Design

Code:

```
`timescale 1ns / 1ps

module time_count( input logic clk, time_clr, time_en,
                   output logic time_late,
                   output logic [3:0] d0, d1, d2, d3);

logic cout_1, cout_2, cout_3, cout_4;
bcd_counter   COUNTER_1(   .clk(clk),   .rst(time_clr),   .enb(time_en),
.out(d0), .cout(cout_1));
bcd_counter   COUNTER_2(   .clk(clk),   .rst(time_clr),   .enb(cout_1),
.out(d1), .cout(cout_2));
bcd_counter   COUNTER_3(   .clk(clk),   .rst(time_clr),   .enb(cout_2),
.out(d2), .cout(cout_3));
bcd_counter   COUNTER_4(   .clk(clk),   .rst(time_clr),   .enb(cout_3),
.out(d3), .cout(cout_4));

ten_count LATE(.clk(clk) , .enb(time_en), .time_late(time_late));

endmodule
```

3.2.3 Pulse Monitor

3.2.3.1 Top Level

The pulse monitor module is a module that measures the heartbeat of the user. It uses a provided analog sensor to read the heart bit. Only the signal every 5 cycles will be processes in that input.

a . Inputs

- pulse_in: take the pulse from the sensor
- rst: A push-button to restart the operation
- clk : clock divided

b. Outputs

- pd3: ones
- pd2: tenth
- pd1: hundredth
- pd0: thousandth

c. Implementation and Design

The figure below represents the organization of the submodules of the pulse monitor module. The reaction timer is concerned with recording the reaction time of the user and it is initialized by the user pressing the start button. The user must then wait for 5 seconds and after a green LED will signal prompt the user to click the enter button. Time will be counted until the user count enter button. The time in which the user takes to press the enter button while the GO-LED remains on is displayed on the seven-segment display.

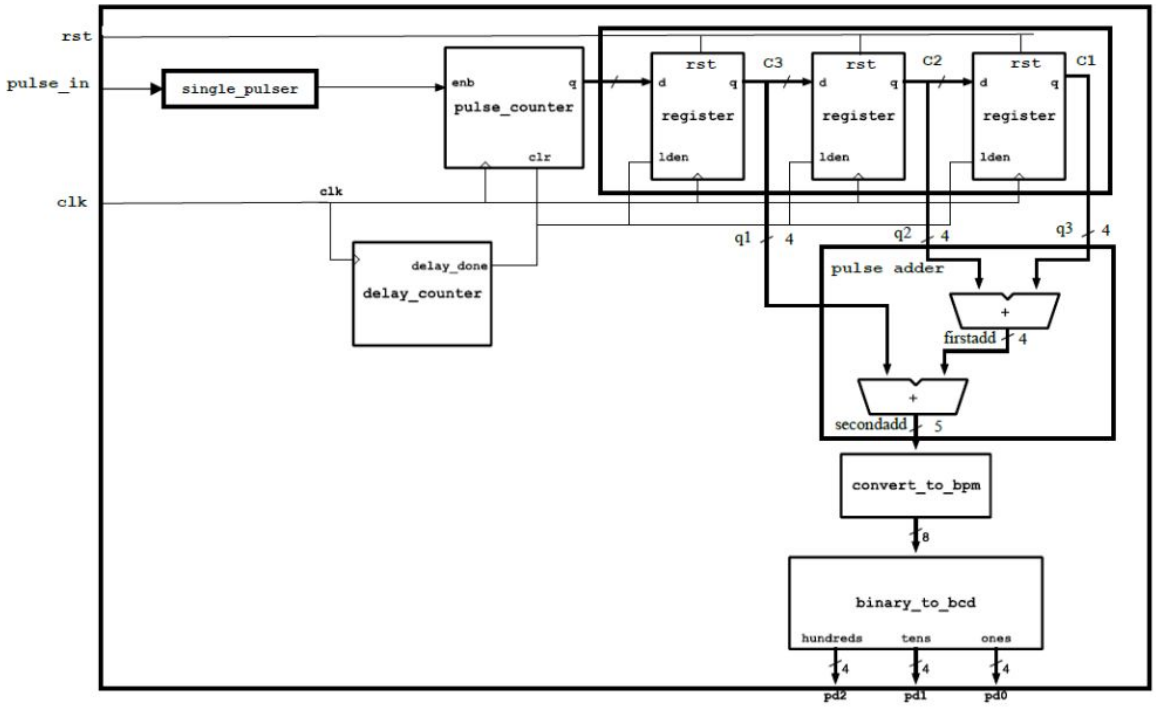

Figure 7: Top module pulse monitor

Code:

```
`timescale 1ns / 1ps

module pulse_monitor_top( input logic clk, rst, pulse_in,
                output logic [3:0]  d0, d1, d2, d3 );


logic pulse_in_deb;
logic single_pulse;
logic reset;
logic [3:0] number;
logic [3:0] reg_1;
```

```
logic [3:0] reg_2;
logic [3:0] reg_3;
logic [7:0] total;
 assign d3 = 4'd0;

debounce DEBOUNCE ( .clk(clk), .pb(pulse_in), .pb_debounced(pulse_in_deb));
single_pulser SINGLE_PULSE(.clk(clk), .din(pulse_in_deb), .d_pulse(single_pulse) );
delay_counter DELAY_COUNTER(.clk(clk), .rst(rst), .out(reset));
counter COUNTER(.clk(clk), .rst(reset), .pulse_in(single_pulse), .q(number));
Register REGISTER1(.clk(clk), .rst(rst), .enb(reset), .d(number), .q(reg_3));
Register REGISTER2(.clk(clk), .rst(rst), .enb(reset), .d(reg_3), .q(reg_2));
Register REGISTER3(.clk(clk), .rst(rst), .enb(reset), .d(reg_2), .q(reg_1));
convert_to_bpm BPM(.a(reg_1), .b(reg_2), .c(reg_3), .y(total));
binary_to_bcd DATA(.b(total), .hunds(d2), .tens(d1), .ones(d0));
```

3.2.3.2 single_pulser

a. Inputs

clk : clock signal
din : input signal

b. Outputs

d_pulse : single pulse output

c. Implementation and Design

The heartbeat is counted in single beats at a time. Therefore we will need a single pulse circuit that will take the digital input and gives a single pulse output to be counted.
The delay counter simulation waveform looks like the following

Code:

```
`timescale 1ns / 1ps

module single_pulser(input logic clk, din, output logic d_pulse);
   logic dq1, dq2;

   always_ff @(posedge clk)
     begin
        dq1 <= din;
```

```
        dq2 <= dq1;
    end

  assign d_pulse = dq1 & ~dq2;
endmodule // single_pulser
```

3.2. 3.3 delay_counter

a. Inputs

clk : clock signal
rst: reset button

b. Outputs

out : output state

c. Implementation and design

Delay counter will be used to reset the counter every 5 seconds as we want three samples with 5-second intervals. The signal for the delay counter will rise every 5 seconds and will be used as the reset for the counter and the clock edge for the registers so they can take the data every 5 seconds from the counter.

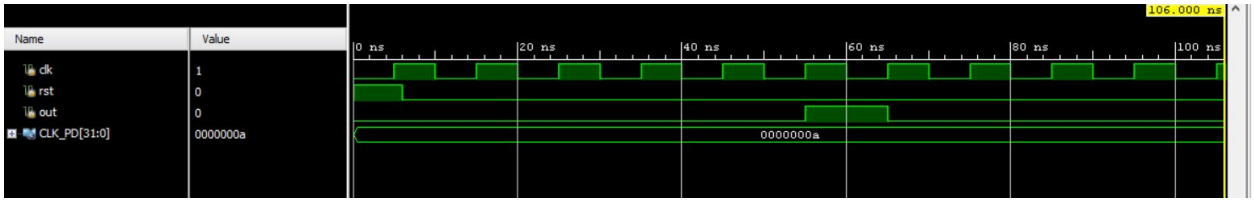

Figure 8: Simulation of delay counter

Code:

```
`timescale 1ns / 1ps

module delay_counter(input logic clk, rst,
                     output logic out);

logic [12:0] q;
always_ff @ (posedge clk)
begin

q <= q +1;
```

```
if (q == 13'd5000)
begin
q <= 0;
out <= 1 ;
end
else
out <= 0;
end


endmodule
```

3.2. 3.4 time_count

The time_count module is used to count the time while waiting for the user's input

a. Inputs

clk: clock signal
time_clr: clearing time
time_en: enabling time
rst: reset

b. Outputs

time_late: last time
[3:0] d0: thousandths
[3:0] d1: hundredths
[3:0] d2: tenth
[3:0] d3: ones

c. Implementation

The time_count module is made up of 4 registers that shift the carry out everytime the register reaches more than 9 incrementation. That techniques is used in order to have the appropriate data input for each of the 7 segment display.
For example in the display we have 0001, the number will increment until 0009. When it reaches that number, the carry out will be sent to the neighboring register. It will make the next number to 0010.

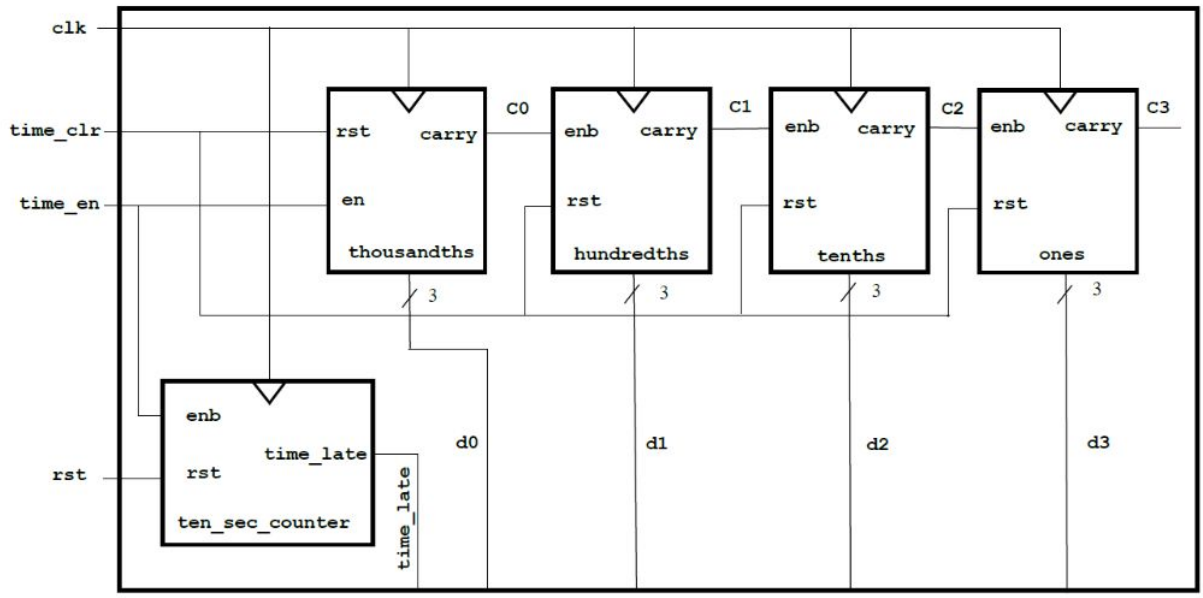

Figure 9: 4 cascades of register

Code :

```
`timescale 1ns / 1ps


module time_count( input logic clk, time_clr, time_en,rst,
output logic time_late,
output logic [3:0] d0,d1,d2,d3);

logic cout0,cout1,cout2,cout3;
logic q ;
bcd_counter
milisecond(.clk(clk),.rst(time_clr),.enb(time_en),.out(d0),
.cout(cout0)) ;
bcd_counter     hundredths(.clk(clk),     .rst(time_clr),.enb(cout0)
,.out(d1), .cout(cout1)) ;
bcd_counter  tenths(.clk(clk),  .rst(time_clr),  .enb(cout1),.out(d2),
.cout(cout2)) ;
bcd_counter  ones(.clk(clk),  .rst(time_clr),  .enb(cout2),.out(d3),
.cout(cout3)) ;
ten_sec_count      tensec_count(.clk(clk)      ,      .enb(time_en),
.time_late(time_late));
endmodule
```

3.2.3.5 Convert beats per minute

a. Inputs

[5:0] Sum: input which is essentially the output from the adders

b. Outputs

[5:0] Sum: input which is essentially the output from the adders

c. Implementation

The module will take the average of the 5-second pulse values of the three registers and get an average value for the beats per 5 seconds. Then the module will multiply this by 12 to make it beats per 60 seconds, essentially making the output beats per minute.

Code:

```
module convert_to_bpm( input logic [5:0] sum ,
output logic [7:0] bpm );
assign bpm = sum << 2 ;
```

3.2.3.6 Convert binary to BCD

a. Inputs

[7:0] b : pulse number

b. Outputs

[3:0] hunds : hundreds
[3:0] tens : tens
[3:0] ones : ones

c. Implementation and design

The module will convert the binary pulse number to BCD for it to be displayed on the seven segment display later on.

Code:

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

```
module binary_to_bcd ( input logic [7:0] b,
output logic [3:0] hunds,
output logic [3:0] tens,
output logic [3:0] ones );

logic [3:0] a1, a2, a3, a4, a5, a6, a7;
logic [3:0] y1, y2, y3, y4, y5, y6, y7;

add3 U_ADD3_1 (.a(a1), .y(y1));
add3 U_ADD3_2 (.a(a2), .y(y2));
add3 U_ADD3_3 (.a(a3), .y(y3));
add3 U_ADD3_4 (.a(a4), .y(y4));
add3 U_ADD3_5 (.a(a5), .y(y5));
add3 U_ADD3_6 (.a(a6), .y(y6));
add3 U_ADD3_7 (.a(a7), .y(y7));

assign a1 = {1'b0, b[7:5]};
assign a2 = {y1[2:0], b[4]};
assign a3 = {y2[2:0], b[3]};
assign a4 = {y3[2:0], b[2]};
assign a5 = {y4[2:0], b[1]};
assign a6 = {1'b0, y1[3],y2[3],y3[3]};
assign a7 = {y6[2:0], y4[3]};

assign hunds = {2'd0, y6[3], y7[3]};
assign tens = {y7[2:0], y5[3]};
assign ones = {y5[2:0], b[0]};

endmodule
```

3.2.3 Clock divider

The clock divider divides the clock which is provided by the Nexys 44DDR to 100Mhz to 1000Hz.

a. Inputs

clk : clock signal
rst : reset signal

b. Output

sclk: clock signal divided

c. Implementation

The clock divider uses a specific formula to divide raw frequency into another frequency. This code was already provided.

Code:

```
`timescale 1ns / 1ps

module clkdiv(input logic clk, input logic reset, output logic sclk);
    parameter DIVFREQ = 100;  // desired frequency in Hz (change as
needed)
   parameter DIVBITS = 26;   // enough bits to divide 100MHz down to
1 Hz
   parameter CLKFREQ = 100_000_000;
   parameter DIVAMT = (CLKFREQ / DIVFREQ) / 2;

   logic [DIVBITS-1:0] q;

   always_ff @(posedge clk)
     if (reset) begin
         q <= 0;
         sclk <= 0;
     end
     else if (q == DIVAMT-1) begin
         q <= 0;
         sclk <= ~sclk;
     end
     else q <= q + 1;
endmodule // clkdiv
```

4. System Verification Performance

A-Reaction Timer

Test	Action	Result	PASS/ FAIL	COMMENT
1	SW0 off	All seven segment off	PASS	N/A
2	Press Start Button	LED goes white after 5 seconds of pressing the button	FAIL	In the actual project description, the initial waiting should be random
3	Press Start Button and after press the Enter	LED goes red and wait for 5 seconds and then goes off	PASS	N/A
4	Press Start Button and wait for 10 seconds	LED goes yellow	PASS	N/A
5	Press the enter button when the light is white	7 segments display 1322 (1s trial) , 4234 (2nd trial)	FAIL	In the project description, it should appear a decimal number
6	Press the reset while LED is on	All seven segment goes off	PASS	N/A

B-Pulse Monitor

Test	Action	Result	PASS/ FAIL	COMMENT
1	SW0 on	Seven segment displays 000	PASS	N/A
2	Hands put in the pulse monitor	Seven segment display numbers that increases until it reaches 100	FAIL	The measurement is not precise although the overall circuit is working
3	No finger is placed on the sensor	Seven segment displays 000	PASS	N/A
4	Press reset button	Seven segment displays 000	PASS	N/A

5.Summary

This report describes every component used to design a health monitor module implemented in a Nexys 4 DDR FPGA. That module has two main functions: reading the pulse of the user and displaying on the average display on a 7 segment displays and counting the reaction time of its user and displaying the time on the 7 segment display board. These two sub modules were designed separately. They are connected to each other using a multiplexer which possess a switch to select between them. One of the limitations of this project is the accuracy of the health monitor which is independent from the designed module. It may be the result from the reading sensor itself. Another problem that this module has is the random wait generator. It is a fixed value in contrast to the project criteria.

6. Appendix A- Specification

This section details the specification for the Health Monitor as presented in the lab manual provided by Prof Muppaneni

Inputs

- Mode select switch (slide switch SW0)
- Reaction time START button start (pushbutton BUTNC)
- Reaction time ENTER button (pushbutton BUTNL)
- System RESET
- Pulse Sensor (PMOD JB connector input pin 1)

Outputs

- 8-digit seven-segment display (anode_l, segs_l)
- Reaction Timer “Go” Lamp (RGB LED LD17)

Operation

- The health monitor provides two different functions: (a) when the mode select switch SW0 is on, it measures the user’s pulse, and (b) When the mode select switch SW0 is off, it tests the user’s reaction time.
- Pulse monitor

- Receives a pulse signal from an analog pulse sensor on an attached daughterboard plugged into the PMOD connector.
- Counts the number of heartbeats over five-second intervals while maintaining the last three samples to calculate the user's pulse as a moving average.
- Displays the user's pulse in beats per minute (BPM) up to a maximum of 255 BPM.
- Unused digits on the 7-segment display should be blank.

- Reaction Timer
  - When the START button is pressed, the seven-segment display should be turned off (if it isn't already). The circuit should then wait for a random amount of time between roughly 1 and 9 seconds. The wait time should be randomly selected from at least eight different delay values in this range.
  - After the random wait, turn on the GO LED and record the amount of time which passes before the user presses the ENTER button. The LED should be off except when waiting for the user to press ENTER.
  - Depending on when (and if) the user presses the ENTER button, the seven-segment display and LED will display the result of the reaction time test, as follows:
    - If the ENTER button is pressed up to 9.999 seconds after the GO LED turns on, the seven-segment display should be turned on and display the reaction time in the format x.xxx (in seconds). The circuit will continue to display this time until the START button is pressed again.
    - If the ENTER button is pressed before the GO LED turns on, the seven-segment display should remain off and the LED color should change to red for five seconds to indicate an error. It should remain lit for five seconds after which it should be turned off and the system should return to waiting for the START button to be pressed.
    - It the ENTER button has not been pressed 10 seconds after the GO LED turns on, the seven-segment display should remain off and the LED color should change to yellow for five seconds to indicate an error. It should remain lit for five seconds after which it should be turned off and the system should return to waiting for the START button to be pressed.

Additional requirements and constraints

- The circuit must be implemented as a fully synchronous circuit using a 1 kHz

The clock generated by a clock divider.

- All sequential logic (except the clock divider and single pulse circuits) should

include asynchronous reset and be connected to a single master RESET input.

- All storage in the circuit must be implemented using flip-flops - the circuit must

contain no latches. To check whether your circuit contains latches, use the Vivado Synthesis Report (or watch for warnings about latch inferences in the "messages" pane).

- The RGB LED should display outputs at a comfortable intensity and all colors should be displayed at approximately equal intensity.
- Unused digits in the 7-segment display should be blank in both modes of operation.

Appendix B- Constraints

```
set_property -dict { PACKAGE_PIN E3 IOSTANDARD LVCMOS33 }
[get_ports { clk100MHz }]; #IO_L12P_T1_MRCC_35 Sch=clk100mhz
set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets mode_IBUF]
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5}
[get_ports {clk100MHz}];


set_property PACKAGE_PIN N17 [get_ports {rst}]
set_property IOSTANDARD LVCMOS33 [get_ports {rst}]
set_property PACKAGE_PIN M18 [get_ports {start}]
set_property IOSTANDARD LVCMOS33 [get_ports {start}]
set_property PACKAGE_PIN P18 [get_ports {enter}]
set_property IOSTANDARD LVCMOS33 [get_ports {enter}]
set_property PACKAGE_PIN C17 [get_ports {pulse_in}]
set_property IOSTANDARD LVCMOS33 [get_ports {pulse_in}]


set_property PACKAGE_PIN N16 [get_ports {led_r}]
set_property IOSTANDARD LVCMOS33 [get_ports {led_r}]
set_property PACKAGE_PIN R11 [get_ports {led_g}]
set_property IOSTANDARD LVCMOS33 [get_ports {led_g}]
set_property PACKAGE_PIN G14 [get_ports {led_b}]
set_property IOSTANDARD LVCMOS33 [get_ports {led_b}]


set_property PACKAGE_PIN J15 [get_ports {mode}]
set_property IOSTANDARD LVCMOS33 [get_ports {mode}]


set_property PACKAGE_PIN J17 [get_ports {anode_1[0]}]
```

```
set_property IOSTANDARD LVCMOS33 [get_ports {anode_1[0]}]

set_property PACKAGE_PIN J18 [get_ports {anode_1[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {anode_1[1]}]

set_property PACKAGE_PIN T9 [get_ports {anode_1[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {anode_1[2]}]

set_property PACKAGE_PIN J14 [get_ports {anode_1[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {anode_1[3]}]

set_property PACKAGE_PIN P14 [get_ports {anode_1[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {anode_1[4]}]

set_property PACKAGE_PIN T14 [get_ports {anode_1[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {anode_1[5]}]

set_property PACKAGE_PIN K2 [get_ports {anode_1[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {anode_1[6]}]

set_property PACKAGE_PIN U13 [get_ports {anode_1[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {anode_1[7]}]



set_property PACKAGE_PIN L18 [get_ports {segs_1[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {segs_1[0]}]

set_property PACKAGE_PIN T11 [get_ports {segs_1[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {segs_1[1]}]

set_property PACKAGE_PIN P15 [get_ports {segs_1[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {segs_1[2]}]

set_property PACKAGE_PIN K13 [get_ports {segs_1[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {segs_1[3]}]

set_property PACKAGE_PIN K16 [get_ports {segs_1[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {segs_1[4]}]

set_property PACKAGE_PIN R10 [get_ports {segs_1[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {segs_1[5]}]
```

```
set_property PACKAGE_PIN T10 [get_ports {segs_1[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {segs_1[6]}]
```