Health Monitor Technical Report

Irwin Frimpong

Digital Circuits I

Prof. John Nestor

12/10/18

Irwin Frimpong
ECE 211 – Digital Circuits I
Prof. John Nestor
Health Monitor Technical Report

Abstract

Having access to health monitors is very useful for it allows one to continually monitor the status of their health. This report presents the design of a Health Monitor that implements a pulse sensor and a reaction timer on an FPGA while also including a high-level description of the design, details on the inner working of the health monitor, as well as information on the system performance and validation.

Introduction

Today, a lot of emphasis is placed on health, fitness, and lifestyles as individuals try to lead healthy lifestyle in order to extend their life expectancies. This has led many people in investing in health monitors to be aware of their own health status. Some of the more common health monitors are embedded into watches, like what Apple Inc. created and continually markets: The Apple Watch. Fitbit is another well-known company for manufacturing heart-rate monitors.

The FPGA health provides the following specifications to its user:

- Provides an easy to use heart rate monitor and reaction times for users who may not be as technologically inclined to purchase other rather expensive competitors
- Displays heart rate/reaction time on bright seven segment display
- Provides the user with the option of toggling between two modes via switch: reaction timer or pulse counter while presenting simple controls such as a start, enter, reset, as well as LED signals.

This technical report will examine the workings of the health monitor by first exploring the High-Level Organization of the Health Monitor and then delving into the individual modules that constitute the high-level description.

3. System Design

3.1 High Level Design

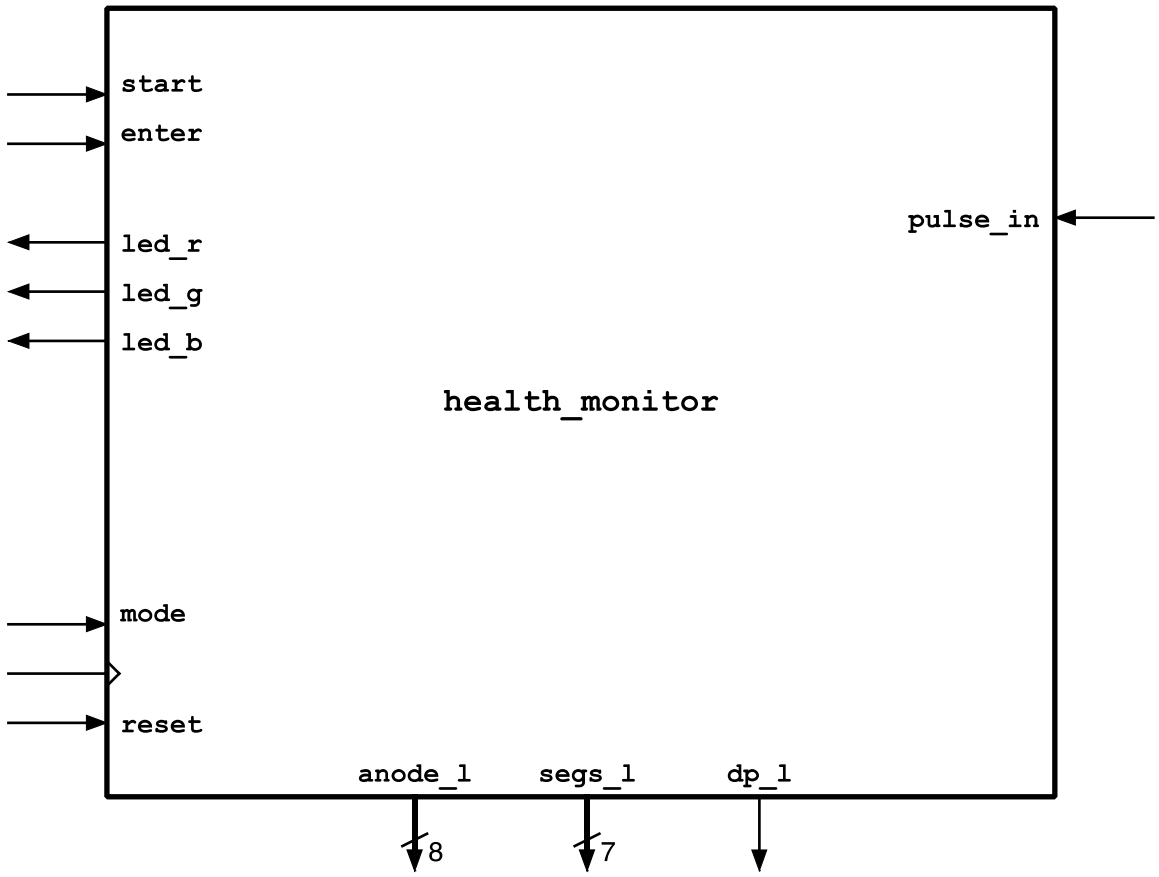

Diagram Drawn by Professor John Nestor

Figure 1-A: High-Level Organization Of The Health Monitor

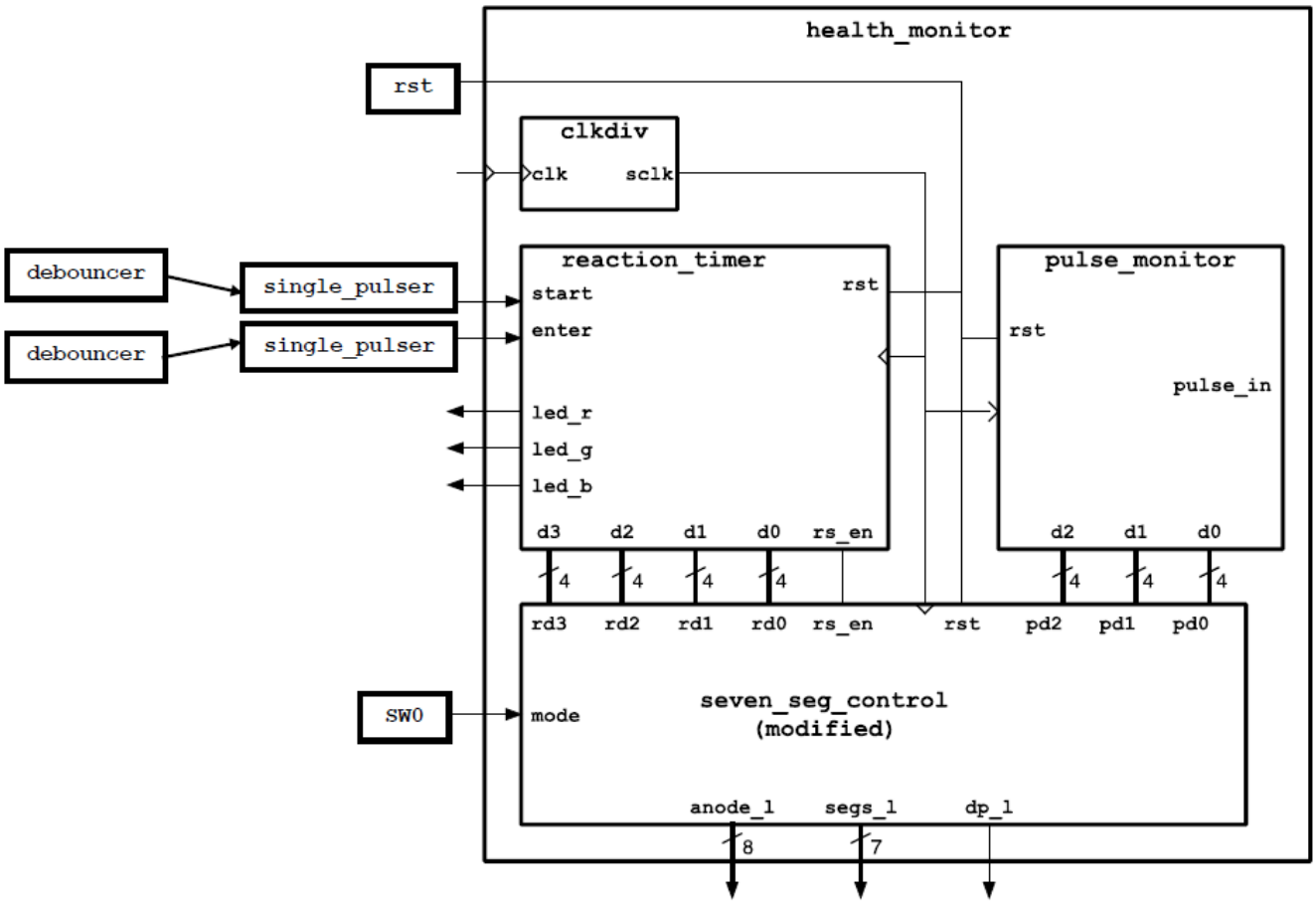

Diagram Modified by Max Huang Originally Drawn by Professor John Nestor

Figure 1-B: High-Level Organization Of The Health Monitor

This health monitor was programmed with two modes available to the user, a brief description of the two modes can be found below.

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 a randomly generated amount time between 1-9 seconds; this function accomplished in the random wait module of the reaction timer module. The user is prompted by a green LED which would be on for the duration of the time in which they are required to respond by pressing the 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. Cases are handled in the event the user presses the enter button without being promoted to as well as reacting too slowly. Considering that there are constraints surrounding the user enter button being pressed to record the reaction time, a Finite State Machine was employed to deal with these cases which will be detailed in a further section of this report.

The pulse monitor receives a pulse signal from an analog pulse sensor which is attached to the daughterboard via the PMOD connection. The monitor counts the number of heartbeats from the user over three, five-second intervals while maintaining the samples from each five-seconds trails for a conversion to beats-per-minute. The BPM value is then displayed on the seven-seg-display.

The High-Level Diagram of the complete health monitor is detailed in *Figure 1*. Given that there are two modes that make up this health monitor, the two modules `reaction_timer` and `pulse_monitor` correspond to the different functions that take place.

The `clkdiv` module generates a 1 kHz clock signal that is fed in as an input clock signal to all the modules in the High -Level Diagram.

The `reaction_timer` module, which will be explored in detail in the following section, takes in inputs `start` and `enter` which are debounced and single pulsed to ensure that each button press is responded to exactly once. The reaction timer has two sets of outputs, the first set of outputs involve the RGB Led assertions, corresponding to the states of the Finite State Machine. The second set of outputs are the four 4 bit outputs values that carry a user's reaction time to the 16 bit two to one multiplexer.

The `pulse_monitor` takes in a `pulse_in` input which is fed in from the pulse sensor and outputs the pulse of the user. This is, of course, a more general overview of what this module does; further discussion will

go into the steps taken from the point in which pulse is detected by the pulse sensor to how the calculations are done before the user's pulse is displayed.

Since the user has the flexibility of toggling between the reaction timer and pulse counter the `mode`, connected to SW0 on the FPGA board, controls what is displayed on the seven-segment display.

3.1.2 Health Monitor Top Level Code:

```
module health_monitor( input logic clk100Mhz, rst, start, enter, pulse_in, mode,
                       output logic [7:0] an_l ,
                       output logic [6:0] segs_l ,
                       output logic dp_l, led_r, led_b, led_g);
       // Logic Declarations
         logic clk ;
         logic rs_en ;
       //Outputs for the Pulse Moniter
         logic [3:0] d0,d1,d2,d3 ;
       //Outputs for the Reaction Timer
         logic [3:0] r0,r1,r2,r3 ;
        // Single Pulse and Debounce of Start Button
         logic start_debounce , start_go ;
        //Single Pulse and Debounce of Enter Button
         logic enter_debounce, enter_go ;
        // Holds the concatenation output of the Pulse Monitor & Reaction Timer
         logic [15:0] pulsemon_out , reaction_out ,q;

// Clock Divider for the clk signal of the entire circuit
clkdiv #(.DIVFREQ(1000)) U_CLKDIV(.clk(clk100Mhz), .reset(1'b0), .sclk(clk));

// Debouncer and Single Pulser  START
   debounce START_1(.clk(clk), .pb(start) , .pb_debounced(start_debounce));
   single_pulser START_2(.clk(clk), .din(start_debounce), .d_pulse(start_go));
// Debouncer and Single Pulser ENTER
   debounce ENTER_1(.clk(clk), .pb(enter) , .pb_debounced(enter_debounce));
    single_pulser ENTER_2(.clk(clk), .din(enter_debounce), .d_pulse(enter_go));
//Creating an instance of Pulse Monitor
pulse_monitor PULSE (.clk(clk),.rst(rst), .pulse_in(pulse_in), .pd0(d0), .pd1(d1), .pd2(d2), .pd3(d3));
//Creating an instance of Reaction Timer
```

```
reaction_timer REACT(.clk(clk), .rst(rst), .start(start_go), .enter(enter_go), .led_r(led_r), .led_g(led_g),
.led_b(led_b),.rs_en(rs_en),.d0(r0),.d1(r1),.d2(r2),.d3(r3)) ;

// Concatenating the Outputs of the Pulse Monitor and the Reaction Timer

        assign pulsemon_out = {d3,d2,d1,d0} ;

        assign reaction_out = {r3,r2,r1,r0};

// Creating an instance of 16 bit 2 to 1 Multiplexer

mux_16bit_2to1 SEL(.mode(mode) , .reaction_timer(reaction_out), .pulse_mon(pulsemon_out) , .q(q)) ;


sevenseg_control_hm U_C_5(.clk(clk), .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_l(segs_l), .an_l(an_l), .dp_l(dp_l));

endmodule
```

16-bit 2 to 1 Multiplexer

3.1.3.1 Inputs

- **[15:0] reaction_timer, pulse_mon:** concatenated values the corresponding module's output
- **Mode:** input value corresponding the state of SW0;

3.1.3.2 Outputs

- **[15:0] q :** outputs of the mux depending on the mode which acts as the select

3.1.3.3 Functionality and Design

The 16-bit 2-1 multiplexer takes in a mode input as a select. When mode is high, the select signal for the multiplexer is one which then outputs the concatenated output vales of the pulse monitor. Nonetheless, when the mode is low, the select signal for the select signal for the multiplexer is zero, which then outputs the concatenate output values of the reaction timer.

Mux_16-bit_2to1 Module Code:

```
module 16bit_2to1_mux(input logic mode,
                      input logic [15:0] reaction_timer ,pulse_mon,
                      output logic 15:0 q);
always_comb
        case(mode)
        // SW0 is off, so mode is reaction timer
                1'd0 : q = reaction_timer ;
        // SW1 is on, so mode is pulse moniter
                1'd1 : q = pulse_mon ;
                default : q = 16'd0;
        endcase
endmodule
```

Seven -Segment Controller

3.1.4.1 Inputs

- **clk:** clock signal
- **rst:** input signal used to reset the values on the seven seg display
- **mode:** used to control what is displayed on the seven-segment display
- **rs_en:** output signal from the reaction timer that is used as an input signal in the seven-segment controller to enable the seven segment displays in the DISPLAY state of the reaction FSM.
- **[3:0] d0,d1,d2,d4,d5,d6,d7:** hold the outputs of both the reaction timer and the pulse monitor depending the on the mode

3.1.4.2 Outputs

- **[6:0] segs_l : output** of the sevenseg_hex / sevenseg_decoder
- **[7:0] an_l:** outputs of the 3-8 decoder
- **dp_l:** used for configuring the decimal point

3.1.4.3 Functionality and Design

The seven-segment controller essentially takes in eight three-bit input digits which are the outputs of the two modes available on the health monitor; these inputs are fed into the four-bit 8-1 multiplexer that multiplexes and displays each input simultaneously.

The three-bit counter instantiated in this module is responsible for counting from zero to seven; these values being key in determining which of the eight seven-segment displayed were on at a moment. Essentially the counter value of zero is leftmost seven-segment display and each iterative value represented one of the eight displays on its rightward path; a value of 7 represented the rightmost seven_seg display. The output of the counter serves as input values for both the 3-8 decoder and the sevenseg_hex / sevenseg_decoder in means of activating the correct display on the FPGA board with its respective value.

Since the counting would be done in milliseconds of the reaction timer, the decimal point on the fourth segment from the left is activated.

Seven-Segment Controller Code:

```
module sevenseg_control_hm(input logic clk, rst, mode, rs_en,
                           input logic [3:0] d0,d1,d2,d3,d4,d5,d6,d7,
                           output logic [6:0]segs_l,
                           output logic [7:0] an_l,
                            output logic dp_l );


logic [3:0] data;
logic [2:0] count;
count_3bit U_C_1 (.clk(clk),.rst(rst),.q(count)) ;
dec_3_8_hm U_C_2 (.a(count),.mode(mode),.rs_en(rs_en),.an_l);
mux_4bit_8to1 U_C_3 (.sel(count),.d0,.d1,.d2,.d3,.d4,.d5,.d6,.d7,.y(data)) ;
```

```
sevenseg_hex U_C_4(.data(data), .segs_l);

//Setting the decimal point

assign dp_l = ~(an_l == 8'b11110111);

endmodule
```

3-8 Decoder

3.1.5.1 Inputs

- **[2:0] a :** count output from the 3-bit counter
- **mode:** based the state of SW0; value can be 0 (reaction timer) or 1 (pulse monitor) depending on whether SW0 is on or off
- **rs_en:** signal sent from the reaction fsm when in the reaction timer

3.1.5.2 Outputs

- **[7:0] an_l :** 8-bit output of the segment to be turned on

3.1.5.3 Functionality and Design

The `mode` and the `rs_en` play essential roles in the decoder considering that each mode had specification for displaying the output of the seven-segment display. In the module, if mode is logical high, meaning that SW0 is on, we are in the pulse monitor mode where the max number of segments needed is three, considering that a user can have a maximum pulse in BPM of 255. Nonetheless, when mode has a logic value of 0, the health monitor is in the reaction timer mode where a maximum of 4 seven-segment displays are needed to display the reaction time in the ms format of X.XXX .

As seen in the code provided below, the condition statement for assigning the output value `an_l` solely depends on the `rs_en` signal which is used in the reaction timer and the mode. Essentially, if either `rs_en` or `mode` is high, `an_l` would be the inverted value of `an` considering that the segments are active low. The inclusion of the "|| mode" in the condition statement was to ensure that segments did not remain off even when the health monitor was in the pulse monitor mode where it didn't have an output value of `rs_en`.

3-8 Decoder Code:

```
module dec_3_8_hm( input logic [2:0] a,
                   input logic mode, rs_en,
                   output logic [7:0] an_l);

logic [7:0] an;

always_comb

// If Mode is zero, we are in reaction timer

if(~mode)

    begin

        case (a)

           3'd0: an= 8'b00000001;

           3'd1: an= 8'b00000010;

           3'd2: an= 8'b00000100;
```

```
            3'd3: an= 8'b00001000;
            default: an = 8'b00000000;
         endcase
       end
else // Pulse Monitor
    begin
        case (a)
               3'd0: an= 8'b00000001;
               3'd1: an= 8'b00000010;
               3'd2: an= 8'b00000100;
                default: an = 8'b00000000;
         endcase
    end
    // Since we only want the segments on when rs_en is on
        assign an_l = (rs_en || mode) ? ~an : 8'b11111111 ;
  endmodule
```

Seven-Segment Decoder

3.1.6.1 Inputs

- **[3:0] data:** 4-bit binary coded decimal number

3.1.6.2 Outputs

- **[6:0] segments:** output value to control how segments on the seven segments are displayed

3.1.6.3 Functionality and Design

The seven-segment decoder converts the 4-bit input data to a 7-bit Boolean value to display that value on the segments of the Seven-Segment display.

Seven-Segment Decoder Code:

```
module sevenseg_hex(  input logic [3:0] data,
                      output logic [6:0] segs_l);
 always_comb
 begin
    case(data)
        4'd0: segs_l = 7'b0000001;
        4'd1: segs_l = 7'b1001111;
        4'd2: segs_l = 7'b0010010;
        4'd3: segs_l = 7'b0000110;
        4'd4: segs_l = 7'b1001100;
        4'd5: segs_l = 7'b0100100;
        4'd6: segs_l = 7'b0100000;
        4'd7: segs_l = 7'b0001111;
        4'd8: segs_l = 7'b0000000;
        4'd9: segs_l = 7'b0001100;
        default: segs_l = 7'b1111111;
    endcase
 end
endmodule
```

3.2 Reaction Timer

The reaction timer consists of the reaction_fsm, random_wait, delay_counter, rgb_pwm, and time_count modules.

3.2.1 Reaction Timer Top Level

The Reaction Timer top level instantiates all the submodules within the design. Depicted in *Figure 2* below is the configuration of the submodules in the top-level module.

Reaction Timer Top Level Code:

```
module reaction_timer( input logic clk ,start , enter, rst,
                       output logic led_r ,led_g , led_b, rs_en ,
                       output logic [3:0] d0,d1,d2,d3);
// Logic Instantiations
logic start_rwait, rwait_done;  //Random Wait
logic start_wait5 , wait5_done ; //Delay Counter
logic [2:0] color_r, color_g, color_b ; //RGB PWM
logic time_clr, time_en, time_late ; //Time Count

//Creating Instance of Reaction FSM
reaction_fsm FSM(.clk(clk),.rst(rst),.start(start),.enter(enter), .rwait_done(rwait_done),
.wait5_done(wait5_done),
 .time_late(time_late) , .start_rwait(start_rwait),.start_wait5(start_wait5), .rs_en(rs_en),
.time_clr(time_clr),.time_en(time_en),.color_r(color_r) , .color_g(color_g), .color_b(color_b));

 // Creating Instance of Random Wait
random_wait RW(.clk(clk), .rst(rst), .start_wait(start_rwait), .rwait_done(rwait_done));

// Creating Instance of Delay Counter
delay_counter_react DELAY (.clk(clk),.rst(rst),.start_wait5(start_wait5), .wait5_done(wait5_done));

//Creating Instance of Time Count
time_count TIME_COUNT(.clk(clk), .time_clr(time_clr),.rst(rst), .time_en(time_en),.time_late(time_late),
.d0(d0), .d1(d1), .d2(d2), .d3(d3));

// Creating an instance of rgb_pwm
rgb_pwm RGB(.clk(clk),.rst(rst), .color_r(color_r), .color_g(color_g) , .color_b(color_b), .rgb_r(led_r) ,
.rgb_g(led_g), .rgb_b(led_b));

endmodule
```

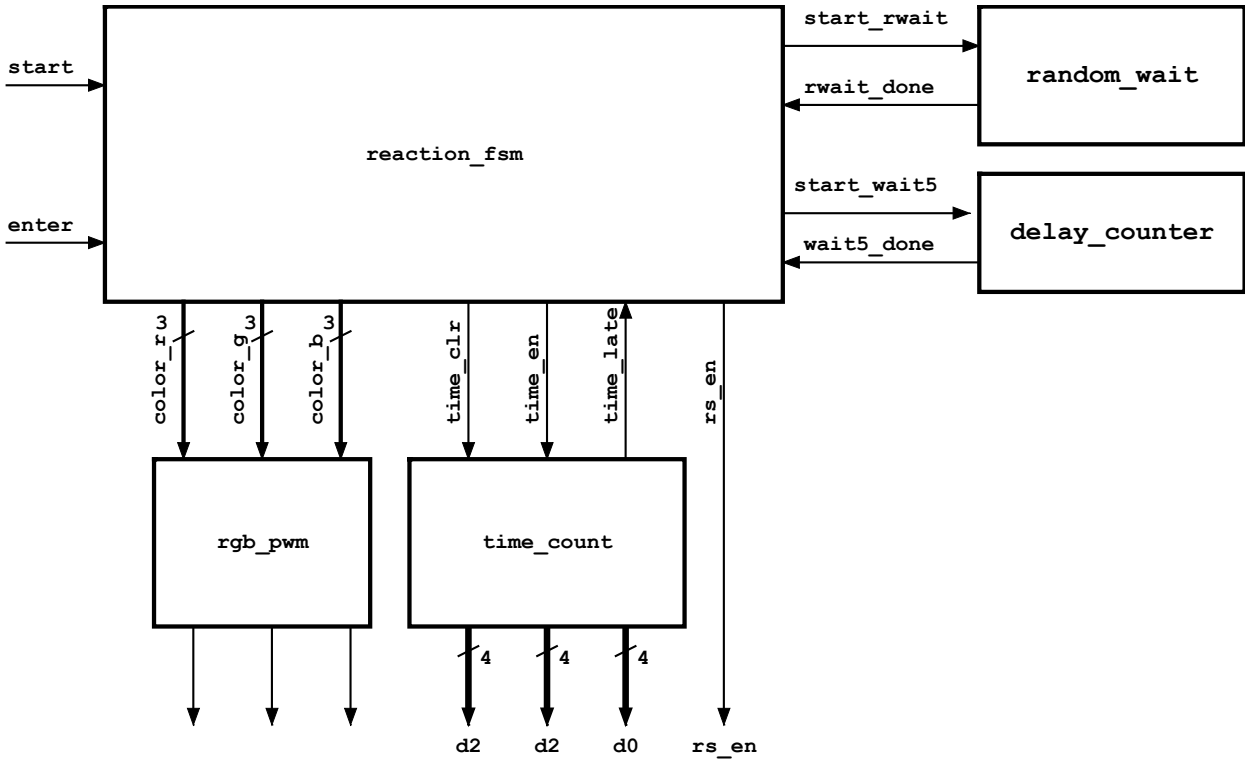

Diagram Drawn by Professor John Nestor

Figure 2: Top- Level Organization of the Reaction Timer

Reaction Finite State Machine

3.2.1.1 Inputs

- **Start:** A push button input that is single pulsed and debounced shown in Figure 1, initiates the reaction timer when pressed by sending a high signal to the reaction finite state machine
- **Enter:** A push button that is single pulsed as well as debounced, allows the user to stop the timer that records the reaction, once the signals are displayed on the FPGA via the RGB Led lights.
- **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. This signal is imperative for the EARLY and LATE states of the finite state machine.
- **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.

3.2.1.2 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 user.
- **start_wait5:** output signal to the `delay_counter` module to start the five second counter.

3.2.1.3 Functionality and Design

The circuit starts off in the IDLE state, where a high signal from the start push button will transition the circuit to the R_WAIT state where the random timer is initiated. Since RGB Led's are used as indicators of specific states in the system, for the IDLE and R_WAIT all the led's remain off. The random time is a time roughly between one and nine seconds. Note that during these three initial states the seven-segment display is off. Once the random timer is terminated, the green led is turned on, signally "GO". The user is then required to press the enter button between a 10 seconds interval from the time the green light is turned on.

If the user presses the enter button before the random wait signal is asserted, the circuit the reroutes to the EARLY state, where the red led is turned on for five seconds to signal the EARLY state. It then goes back to the idle state for the user to initiate the reaction timer once more.

If the user, successfully presses the enter button before the 10 second time constraint, the DISPLAY state is encountered where the seven-segment display is turned on and displays the reaction time. This reaction time will remain displayed until the user presses the user presses the start button again to initiate a new cycle of the reaction timer.

If the user, fails to do the aforementioned within the time constraint, the system goes into the LATE state, where a yellow led is displayed for five seconds before returning back to the IDLE state.

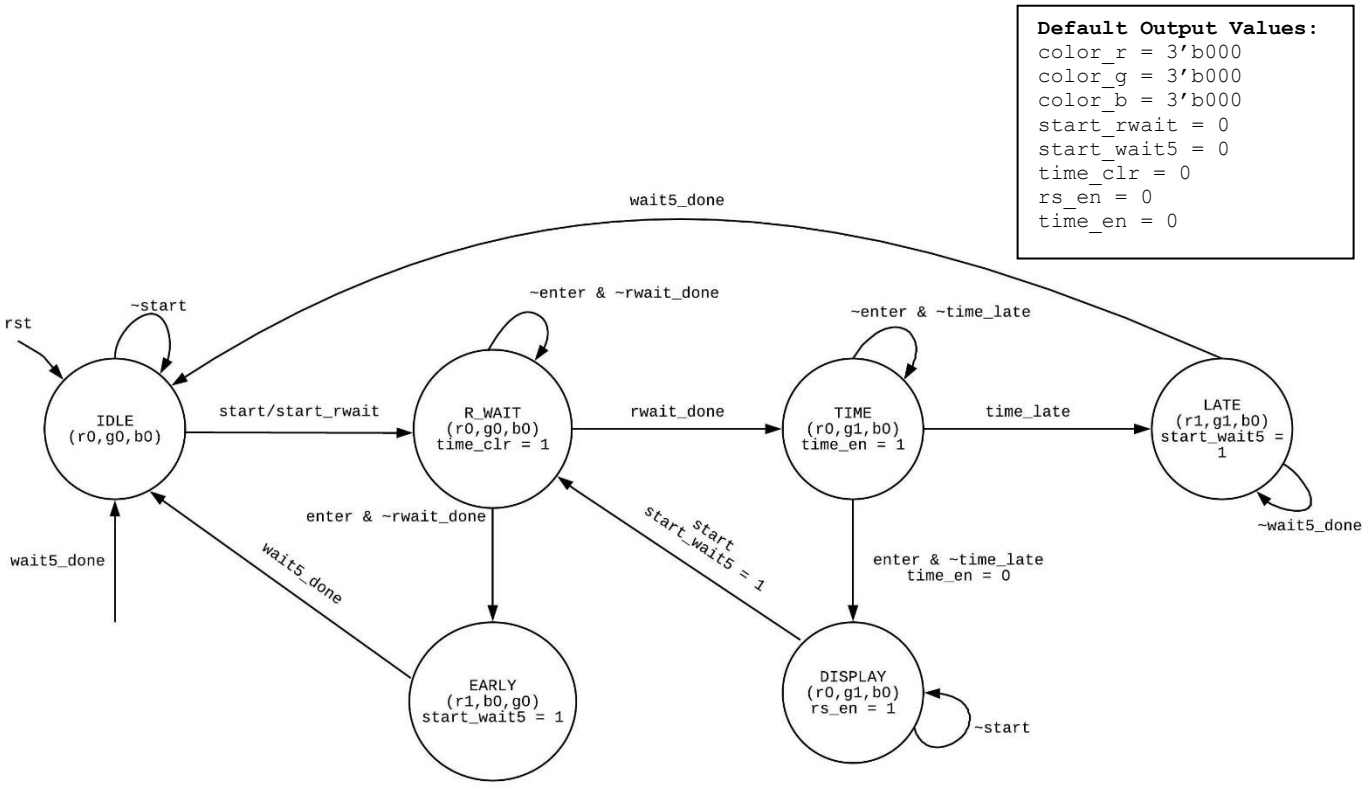

Diagram Drawn by Max Huang

Figure 3 : State Transition Diagram of FSM

Reaction Timer Finite State Machine Code:

```
module reaction_fsm(input logic clk,rst,start,enter,rwait_done,wait5_done,time_late,
                    output logic start_rwait, start_wait5, rs_en, time_clr, time_en,
                     output logic [2:0] color_r , color_g , color_b);
          //States for the FSM
 typedef enum logic [2:0]{
    IDLE= 3'b000, R_WAIT = 3'b001, EARLY = 3'b010, TIME = 3'b011, DISPLAY = 3'b100, LATE = 3'b101
    }state_t ;
 state_t state, next ;

 always_ff @(posedge clk)
    if(rst) state <= IDLE ;
    else state <= next ;
    always_comb
    begin
                // Want all the led's to be turned off in the beginging
                color_r = 3'b000;
                color_g = 3'b000;
                color_b = 3'b000;
                start_rwait = 0 ;
                start_wait5 = 0 ;
                time_clr = 0;
                rs_en = 0 ;
                time_en = 0;

                next = IDLE;
```

```
            // Starting the case statements for the states
            case(state)
                        IDLE:
                                    begin
                                                color_r = 3'b000;
                                                color_g = 3'b000;
                                                color_b = 3'b000;
                                                rs_en = 0;

            // If the start button is pressed in the idle, we want to initiate the random wait
                                    if(start)
                                    begin
                                                start_rwait = 1;
                                                next = R_WAIT;
                                    end
                       //if the start button is not pressed, we must remain in the idle state
                                    else
                                                next = IDLE;
                                    end

                        R_WAIT:
                                    begin
                                                // Want to clear the time when the start button
                                                time_clr = 1;

                                                // LEDS are off in Random Wait
                                                color_r = 3'b000;
                                                color_g = 3'b000;
                                                color_b = 3'b000;
                                                rs_en = 0;

                         //If to check if Enter is pressed while the random wait isn't finished
                                    if (enter && ~rwait_done)

                          // Go into the early state
                                                 next = EARLY;
                      // If enter is not pressed and random wait is not finished
                                                 else if (~enter && ~rwait_done)
                      // Remain in the R_wait state
                                                       next = R_WAIT;
                      // If the random counter is finished, we want to enter the time state
                                                 else if (rwait_done)
                                                 begin
                                                       time_en = 1;
                                                          next = TIME;
                                                 end
                                    end
                        EARLY:
                                    begin
                      // Turn on the RED light
                                                      color_r = 3'b001;
                                                      color_g = 3'b000;
                                                      color_b = 3'b000;

                     // Start the delay counter
                                                start_wait5 = 1;

                   // After the delay counter is done, since we want the user to have to use the start button to
                  //initiate a new reaction sequence, we go back to the idle state
                                                if (~wait5_done)
                                                next = EARLY;
                     // If the 5 second timer is not done we want to remain in idle
```

```
                                        else
                                      next = IDLE;
                                end
                        TIME:
                                begin
                // Want to initiate the timer keeping track of the reaction time
                                        time_en = 1;

                // Turn on the Green LED light
                                        color_r = 3'b000;
                                        color_g = 3'b001;
                                        color_b = 3'b000;
                                     if (enter && ~time_late)
                                        begin
                                                time_en= 0; // Stop the timer
                                                next = DISPLAY;
                                        end
                                     else if (~enter && ~time_late)
                                        next = TIME;
                                     else if (time_late)
                                        next = LATE;
                                end
                        DISPLAY:
                                begin
                //Green LED remains on
                                        color_r = 3'b000;
                                        color_g = 3'b001;
                                        color_b = 3'b000;
                // Tells the seven seg to display the digits
                                     rs_en = 1;
                // If the start button is pressed, we want to go to the R_Wait State
                                if (start)
                                        begin
                                                start_wait5 = 1;
                                                next = R_WAIT;
                                        end
                                else
                // If start button is not pressed
                                        next = DISPLAY;
                                     end
                        LATE:
                                begin
                // Initiate our five second counter
                                     start_wait5 = 1;
                //Yellow light must be on
                                        color_r = 3'b001;
                                        color_g = 3'b001;
                                        color_b = 3'b000;
                // Once the five second time period is up we want to return to the idle state
                                        if(~wait5_done)
                                                next = LATE;
                                        else if (wait5_done)
                                                next = IDLE;
                                end
                        endcase
                end
endmodule
```

Random Wait Module

3.2.2.1 Inputs

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

3.2.2.2 Outputs

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

3.2.2.3 Functionality and Design

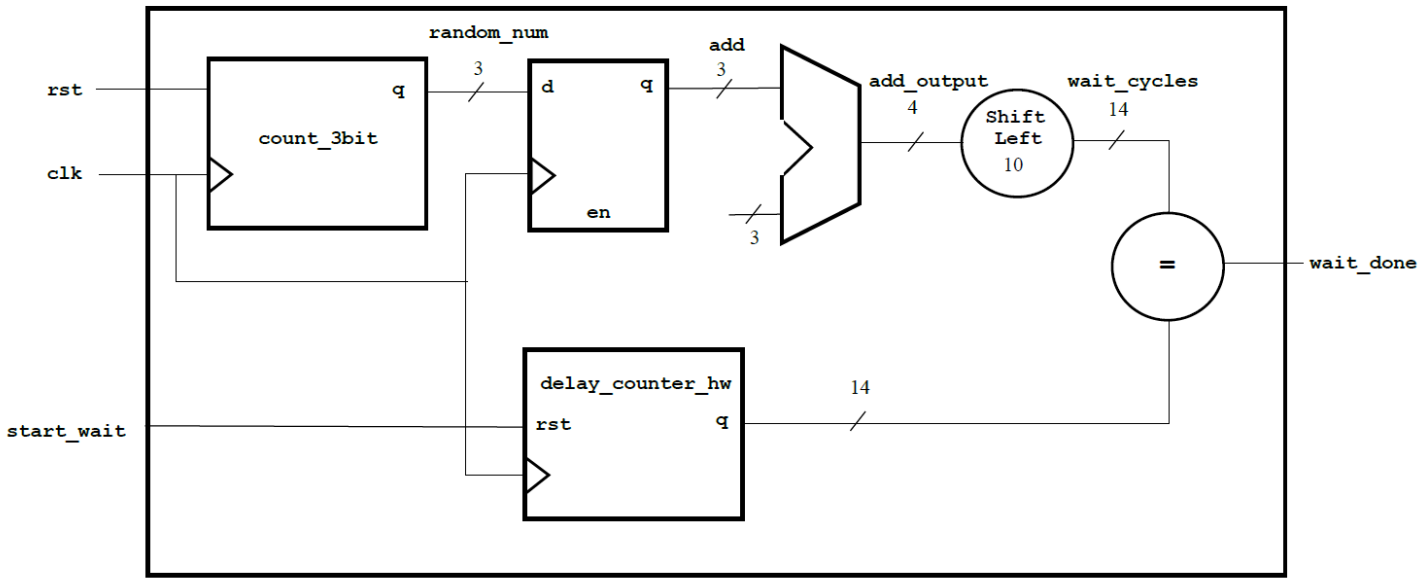

Diagram Drawn By Max Huang

Figure 4: Random Wait Module Block Diagram

The random_wait module runs a 3bit counter at the generated clk div clock signal of 1kHz; once the start push button is pressed the current count that the timer had at that clock edge is sampled and passed into the flipflop. This 3-bit value is then passed into an adder that adds one to that random number. The result of the add is multiplied by 1024 which is essentially the same as computing a bitwise shift of 10 to the left.

On the other, with the delay counter, once the start push button is pressed by the user, the delay counter is reset and then begins to count; at every clock edge the number that the delay counter is at, is sampled to see if it matches the results of the left bitwise shift on the adder's output. If there is a match, wait done is set to logic high which indicates that the random delay time has been completed. This output is sent to the reaction FSM for it to transition to the TIME state.

Random Wait Module Code:

```
module random_wait( input logic clk, rst, start_wait,
                    output logic rwait_done );
    //Logic Instantiations
       logic [2:0] random_num;
       logic [2:0] add;
       logic [13:0] wait_cycles;
       logic [13:0] d_count;
       logic [3:0] add_output;


    // Instantiating Count 3 Module
      count_3bit COUNT3 (.clk(clk), .rst(rst), .q(random_num)) ;
   // Instantiating FlipFLop for Random Num capture
      r_num_rw RANDOM_NUM (.clk(clk) , .en(start_wait),.d(random_num), .q(add));
   // Creatting instance of Delay Counter
      delay_counter_rw DC (.clk(clk), .rst(start_wait), .q_delay(d_count));
  // Computing the addition
      assign add_output = add + 1 ;
  // Computing the multiplication by performing a shift
  //Shifting to the left 10 which is representative of multiplying by 1024
      assign wait_cycles =  add_output << 10 ;
// Checking if the values of wait cycle equals the output of the delay counter so we can assert wait_done true
      assign rwait_done = (wait_cycles == d_count) ;
 endmodule
```

Code for submodules in random wait :

Count 3bit Code:

```
module count_3bit ( input logic clk, rst,
                      output logic [2:0] q );
       always_ff @(posedge clk)
       if (rst) q<= 3'd0;
       else q<= q + 3'd1;
       endmodule // count_3bit
```

R_num_rw Code:

```
module r_num_rw( input logic clk ,en,
                 input logic [2:0] d,
                 output logic [2:0] q);
```

```
always_ff @(posedge clk)
// If en is asserted q gets d
if (en) q <= d ;
endmodule // r_num_rw
```

delay_conter_rw Code:

```
module delay_counter_rw( input logic clk, rst,
                         output logic [13:0] q_delay);
always_ff @(posedge clk)
        if (rst) q_delay<= 0;
        else q_delay<= q_delay + 1;
endmodule // delay counter_rw
```

Delay Counter Module

3.2.3.1 Inputs

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

3.2.3.2 Outputs

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

3.2.3.3 Functionality and Design

The delay counter module is solely responsible for counting up to five seconds, by adding one to the internal output value q at every clock edge as long as the value of q has not exceeded the 13-bit q value of 5,000. This delay counter is used by the EARLY and LATE states of the reaction FSM.

Delay Counter Module Code:

```
module delay_counter_react( input logic clk, rst, start_wait5,
                            output logic wait5_done);
// Logic Instantiation
logic [12:0] q ;
always_ff @ (posedge clk)
begin
        if(rst)
         q <= 0 ;
        else if(start_wait5 && q == 13'd5000)
        begin
        // Want to set wait done equal to one
               wait5_done <= 1 ;
```

```
                q <= 0 ;
        end
        else if (start_wait5 && q != 13'd5000)
           begin
                q <= q+1 ;
                wait5_done <= 0 ;
           end
        end
endmodule
```

Time Count Module

3.2.4.1 Inputs

- **time_clr:** input signal which acts as a reset for the dec counters
- **time_en:** input signal to start the counters
- **rst:** reset signal

3.2.4.2 Outputs

- **[3:0] d0, d1,d2,d3 :** outputs which hold the values of the dec counters

3.2.4.3 Functionality and Design

The time count module as mentioned previously instantiates four `dec_counter` modules connected to create a shift register counter. Each of these modules corresponding to a place values in the decimal: thousandths, hundredths, tenths, and ones.

The `dec_counter` module creates a register will a clock, reset and enable input as well as a 3-bit q output and a carry output. This shift register increments the values of q at every clock edge if the `enb` signal is high and the value of q is not 9. If the latter is true, the value of carry is asserted to be true, which is then fed in as an `enb` input to the next `dec_counter` for it to begin its counting. Note the once carry is high, the counter register resets to zero and repeats the cycle.

Moreover, to return the `time_late` signal, another register is instantiated called `ten_sec_count` which runs on the same clock signal and is fed the same `time_en` input signal that initiates the 10-sec timer. Once this time is complete, the `time_late` signal is asserted high and further returned.

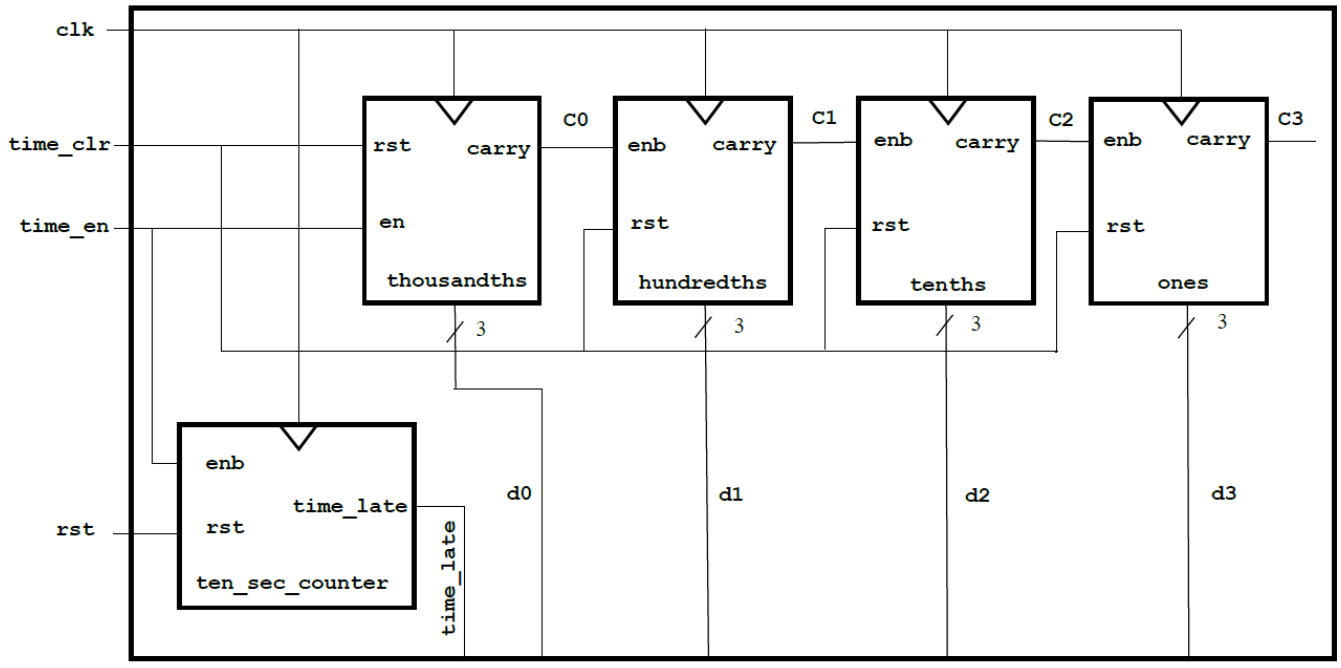

Diagram Drawn by Max Huang

Figure 5: Block Diagram of Time Count Module

Time Count Module Code:

```
module time_count(     input logic clk, time_clr, time_en,rst,
                       output logic time_late,
                       output logic [3:0] d0,d1,d2,d3);
// Logic Instantiations
logic c0,c1,c2,c3; // Used for the carry of our counters
logic q ;
dec_counter milisecond(.clk(clk),.rst(time_clr),.enb(time_en),.q(d0), .carry(c0)) ;
dec_counter hundredths(.clk(clk), .rst(time_clr),.enb(c0) ,.q(d1), .carry(c1)) ;
dec_counter tenths(.clk(clk), .rst(time_clr), .enb(c1),.q(d2), .carry(c2)) ;
dec_counter ones(.clk(clk), .rst(time_clr), .enb(c2),.q(d3), .carry(c3)) ;

// Ten second counter used for sending in the time_late signal
ten_sec_count tensec_count(.clk(clk) , .rst(rst), .enb(time_en), .time_late(time_late));
endmodule
```

** Ten second counter could be replaced by an assign statement [assign time_late = (c3 == 1)], however a Timing Loop Warning was encountered.*

Irwin Frimpong
ECE 211 – Digital Circuits I
Prof. John Nestor
Health Monitor Technical Report

Dec Counter Module Code:

```
module dec_counter(    input logic clk, rst, enb,
                       output logic [3:0] q,
                       output logic carry);

assign carry = (q == 4'd9) && enb;

always_ff @(posedge clk )

       begin

              if (rst || carry) q <= 0;
              else if (enb) q <= q + 1;
       end
endmodule // dec_counter
```

Ten Second Count Module Code:

```
module ten_sec_count( input logic clk, enb,
                      output logic time_late);
// Logic Instantiation
logic [13:0] q ;

always_ff @ (posedge clk)
        begin
              if (enb && q != 14'd10000)
               begin
                    q<= q + 1 ;

                  time_late <= 0;
               end
          else if (enb && q == 14'd10000)
               begin
                    time_late <= 1 ;
                    q <= 0 ;
               end
       end
endmodule
```

RGB Pulse Width Modulation Module

3.2.5.1 Inputs

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

3.2.5.2 Outputs

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

3.2.5.3 Functionality and Design

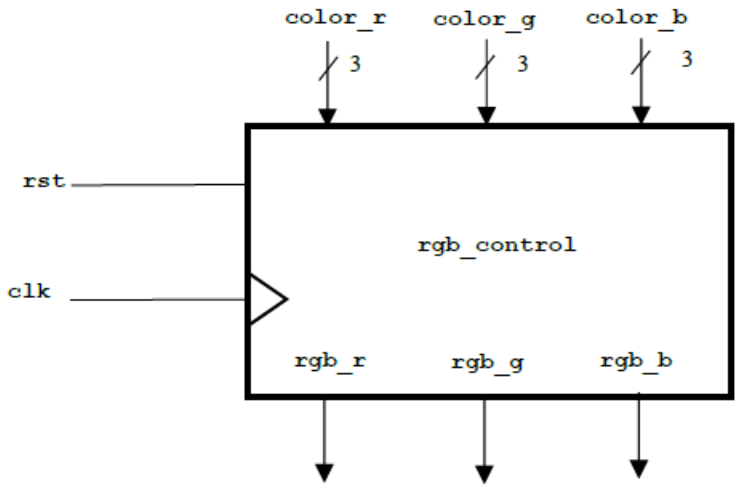

Diagram drawn by Max Huang

Figure 6: Block Diagram of rgb_pwm

RGB PWM Module Code:

```
module rgb_pwm (input logic clk, rst,
                input logic [2:0] color_r , color_g, color_b,
                output logic rgb_r, rgb_g, rgb_b )
// Logic Instantiations
logic [3:0]q

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

always_comb

begin
        if (q < color_r) rgb_r <= 1;
        else rgb_r <= 0 ;

        if (q < color_g) rgb_g <= 1;
        else rgb_g <= 0 ;
        if (q < color_b) rgb_b <= 1;
        else rgb_b <= 0 ;

        end
endmodule;
```

3.3 Pulse Monitor

The pulse monitor consists of the pulse_counter, delay_counter,pcount_registers, pulse_adder, convert_to_bpm, and binary_to_bcd modules.

3.3.1 Pulse Monitor Top Level

The Pulse Monitor Top Level module instantiates all the submodules within the design. Depicted in *Figure 7* below is the configuration of the submodules in the top-level module. The `pulse_in` signal coming from the pulse senor as is single pulsed and is supplied to the instance of pulse count, keeping track of the number of heartbeats at every clock edge. The count value that the pulse counter has at every five-second mark is passed into the instance of `pcount_registers` that houses the shift registers to store the heartbeats recorded. To calculate the moving average of the three heartbeats, the outputs of the three shift registers are passed into the `pulse_adder` module where these values are added; their sum is passed into an instance of `convert_to_bpm` module where left-bit shifting is performed on the value to convert the average beats for five seconds to BPM(Beats per minute). Lastly, the BPM is passed into the `binary_to_bcd` module where the BPM's place values (hundreds, tens, ones) are assigned for display on the seven-segment display.

3.3.1.2 Pulse Monitor Top Level Code

```
module pulse_moniter( input logic clk,rst, pulse_in ,
                      output logic [3:0] pd0, pd1, pd2, pd3);

 //Logic Instantiations for Module
  logic pulse_go; // Single Pulsed pulse in put

  logic d_done; // Wire holding the output of the delay counter

  //Wire for the register that will hold the beats
  logic [3:0] q0, q1, q2 , p_counter ;
  // Wires for Results for Adder
  logic [5:0] adder_sum;

  //Wire for BPM Convertion
  logic [7:0] bpm_out;

   //Wire for 15 sec counter
  logic add_time ;

  // pd3 is connected to 0
  assign pd3 = 4'd0;

 // Single Pulser for Pulse_In input
   single_pulser S_PULSE(.clk(clk), .din(pulse_in), .d_pulse(pulse_go)) ;

 //Creating an instance of delay counter
    delay_counter DELAY (.clk(clk), .delay_done(d_done));

 // Creating an instance of pulse counter
    pulse_counter PULSE (.clk(clk), .clr(d_done), .enb(pulse_go) , .q(p_counter));

   // Instance of Registers
```

```
  pcount_registers PCOUNT (.clk(clk),.iden(d_done), .rst(rst), .q_in(p_counter), .c1(q0) , .c2(q1), .c3(q2)) ;

  // Instantiating the adder module to sum up the pulse
    pulse_adder ADD (.q1(q0), .q2(q1),.q3(q2), .sum(adder_sum));

  // Converting to BPM
  convert_to_bpm BPM (.sum(adder_sum), .bpm(bpm_out));

  //Instantiate Binary_to_bcd module
   binary_to_bcd BTBCD (.b(bpm_out), .hundreds(pd2), .tens(pd1) , .ones(pd0));

endmodule
```

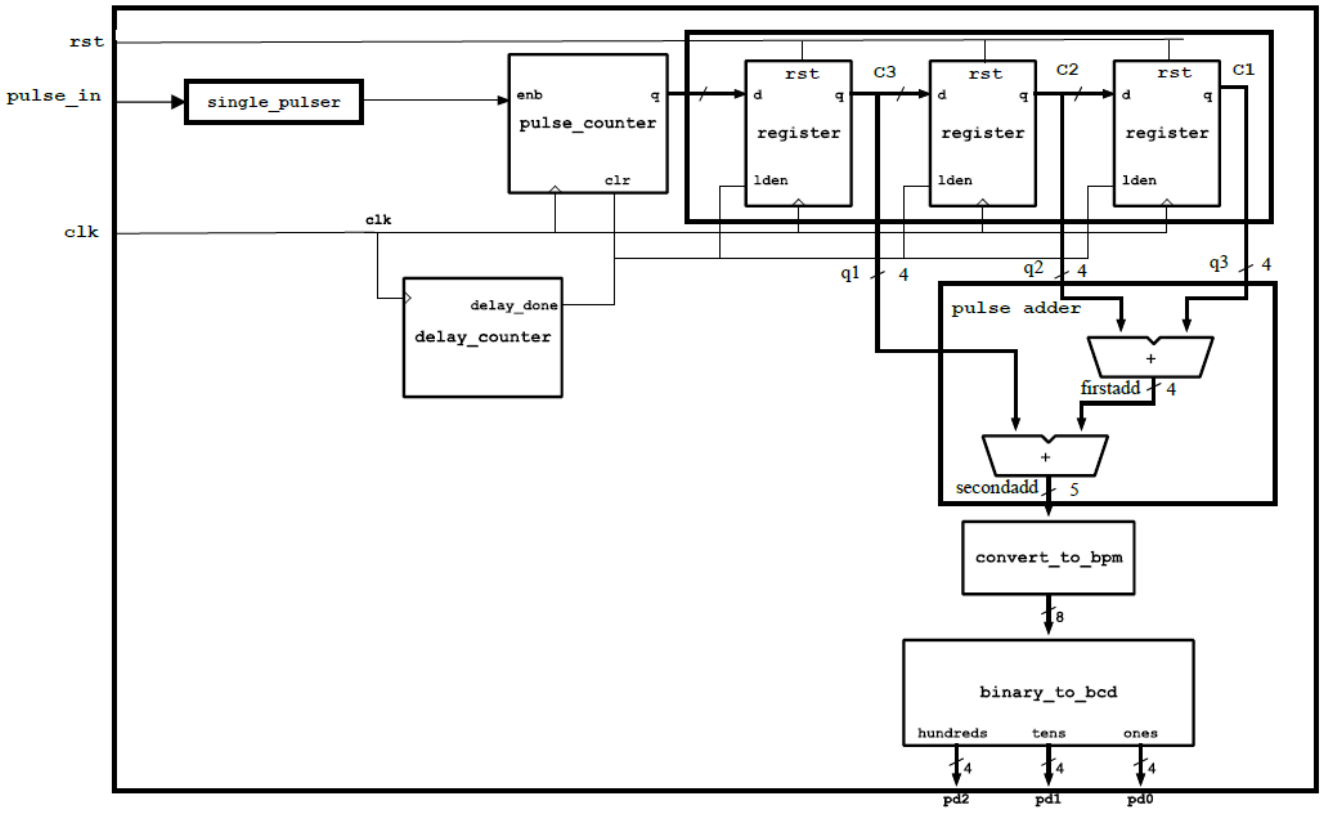

Diagram Modified by Max Huang Originally Drawn by Professor John Nestor

Figure 7: Block Diagram of Pulse Monitor

Pulse Counter

3.3.2 .1 Inputs

- **pulse_in:** input signal from the pulse sensor
- **clk :** clock signal from the clock divider
- **clr:** input signal from the delay counter

3.3.2.2 Outputs

- **q:** outputs from the pulse for a five second period

3.3.2.3 Functionality and Design

The pulse counter takes in an enable input which in this case is the pulse_in input coming from the pulse sensor; Essentially whenever the pulse sensor detects a pulse, it will assert the pulse_in input to be high which then translate to the pulse counter incrementing the value of q. If a clr signal is received from the delay_counter, the value of q is passed into one of the shift register depicted in Figure 7 and then further cleared for the next cycle of pulse counting. After three trails of pulse counting, each register of the shift register would have a value corresponding the pulse counted in one of the three trails.

Pulse Counter Module Code:

```
module pulse_counter( input logic clk, clr, enb ,
                      output logic [3:0] q ) ;
  always_ff @(posedge clk)

      if ( enb)  q <= q + 1 ;
    else if ( clr) q <= 0 ;

endmodule
```

Delay Counter

3.3.3.1 Inputs

- **clk:** clock signal from the clock divider

3.3.3.2 Outputs

- **delay_done:** output signal signaling the counter has reached the 5-second mark

3.3.3.3 Functionality and Design

The delay is a simple register counter that counts up to the 13-bit value of 5,000, representing five seconds, upon which a delay_done signal is asserted high for the pulse counter to reset its counter.

Delay Counter Module Code :

```
module delay_counter( input logic clk,
                      output logic delay_done);
         // Logic Instantiation
         logic [12:0] q;
 always_ff @(posedge clk)

       begin
           // Increment the counter by 1
            q <= q +1;
        if (q == 13'd5000)
             begin
                 q <= 0;
                delay_done <= 1 ;
             end

        else
            delay_done <= 0;
        end

     endmodule
```

Pulse Adder Module

3.3.4.1 Inputs

- **[3:0] q1,q2, q3:** 3-bit inputs which are essentially the outputs of the registers holding the pulse count of the five-second interval

3.3.4.2 Outputs

- **[5:0] sum:** holds the results of the addition of the input values

3.3.4.3 Functionality and Design

The pulse adder takes in the three 4-bit output values from the shift-register, and first computes the sum of the first two input values. Since they are both 4-bit values, the value of the output of their sum must be stored as a 5-bit wire called `firstadd` to account for an overflow if any. The next set of add is computed using the firstadd value and the third 3-bit input value; their 6-bit result is returned by the module

Pulse Adder Module Code:

```
module pulse_adder( input logic [3:0] q1, q2, q3,

                    output logic [5:0] sum);

    // Wires for adder

  logic [4:0] firstadd;

  logic [5:0] secondadd;

always_comb

    begin

//Computing first add

         firstadd = q1 + q2 ;

 //Computing second add
```

```
        secondadd= firstadd + q3 ;
    end
  assign sum = secondadd ;
  endmodule
```

Pulse Count Registers

3.3.5.1 Inputs

- **clk:** clock signal
- **iden:** is essentially the `delay_done` assertion from the delay counter and serves as an enable
- **rst:** rst signal
- **[3:0] q_in:** the output of the pulse counter to be fed into a register

3.3.5.2 Outputs

- **[3:0] c1, c2, c3:** outputs of each register

3.3.5.3 Functionality and Design

This module instantiates three registers that would act as shift registers, so essentially as the user pulse count from the pulse counter is fed in as an input every five seconds, the previously recorded pulse value shifts to the next register as the new pulse is then recorded. The goal is to record three consecutive heart rates over three five-second intervals; these values will be then used in the calculation of beats per minute.

Pulse Count Registers Code:

```
module pcount_registers( input logic clk, iden,rst,
                         input logic [3:0] q_in,
                         output logic[3:0] c1 ,c2,c3 );


//Instantiating Registers
p_register R3 (.clk,.iden,.rst(rst), .d(q_in),.q(c3)) ;
p_register R2 (.clk,.iden,.rst(rst), .d(c3),.q(c2)) ;
p_register R1 (.clk,.iden,.rst(rst), .d(c2),.q(c1)) ;
endmodule
```

P Register

3.3.6.1 Inputs

- **clk:** clock signal
- **iden:** output of delay counter and serves as enable for the register
- **rst:** reset signal
- **[3:0] d:** 4-bit input for the pulse counter value

3.3.6.2 Outputs

- **[3:0] q:** 4-bit output to return the stored pulse counter value

3.3.6.3 Functionality and Design

This module houses the registers that are instantiated in the `pcount_registers` module to store the pulse counts. The `iden` input value serves as an enable for the registers, in which when asserted high, allows the register to accept a 4-bit input `d` and output the 4-bit value q to the next register; this shown in Figure 7 .

Code for P Register :

```
module p_register( input logic clk , iden, rst,
                   input logic [3:0] d ,
                   output logic [3:0] q  );
 // Iden will tell the register to store its inputted value from the previous register
   always_ff @(posedge clk)
     begin
       if (iden)
              q <= d ;
       else if (rst)
               q <= 0;
        end
endmodule
```

Convert To BPM

3.3.7.1 Inputs

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

3.3.7.2 Outputs

- **[7:0] bpm:** output with converted beat per minute

3.3.7.3 Functionality and Design

Since the pulse counter only samples three 5 seconds set of pulse readings, to calculate beat per minute it would necessary for one to find the average of the three counts and then multiply the value by 12 since 5sec is 1/12 of

a minute. However, since multiplication is expensive and difficult to realize in hardware, a faster alternative was to perform a left bitwise shift by 2 on the sum input, which is what the bpm module accomplishes.

Convert to BPM Module Code :

```
module convert_to_bpm( input logic [5:0] sum ,

                       output logic [7:0] bpm );

// Shifting the sum (the avg of beats for 5sec) to convert to bpm

 assign bpm = sum << 2 ;

endmodule
```

Binary To BCD

3.3.8.1 Inputs

- **[7:0] bpm :** input from the convert_to_bpm module

3.3.8.2 Outputs

- **[3:0] pd0:** ones
- **[3:0] pd1:** tens
- **[3:0] pd2:** hundreds

3.3.8.3 Functionality and Design

Binary to BCD Code:

```
module binary_to_bcd ( input logic [7:0] b,
                       output logic [3:0] hundreds,
                       output logic [3:0] tens,
                       output logic [3:0] ones );
// Logic Instantiations
 logic [3:0] a1, a2, a3, a4, a5, a6, a7;
 logic [3:0] y1, y2, y3, y4, y5, y6, y7;
//Creating instances of add3
 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 ones = {y5[2:0],b[0]};
 assign tens = {y7[2:0],y5[3]};
 assign hundreds = {2'b0,y6[3],y7[3]};
endmodule
```

4 System Validation and Performance

Details on the test procedures will be documented in Appendix B. To ensure the proper functioning of the health monitor, a series of tests were carried out, the test and their results are documented below.

Reaction Timer:

Test	*Action*	*Results*	*Pass/Fail*	*Initial*
1	SW0 is off	All seven-seg displays are off	PASS	I.F.
2	Pressed start push button	Arbitrary wait time before the green LED turned on	PASS	I.F.
3	Pressed start and then pressed the enter button before the "Go" LED signal	Red LED was displayed for five seconds	PASS	I.F.
4	Pressed start button but waited for the green LED to remain on for 10 secs	Yellow LED is displayed for five seconds **verified the wait time w/ smartphone stopwatch*	PASS	I.F.
5	Pressed enter push button after green LED turned on	Trail 1: 1.293 s Trail 2: 2.463 s Trail 3: 3.796 s **Display is initially off after the enter button is pressed, display remains lit with reaction time until either the start/ reset push button is pressed.*	PASS	I.F.
6	Pressed Reset after DISPLAY state of reaction timer	Device enters the IDLE mode; the seven-seg displays turn off	PASS	I.F.
7	Pressed reset push button while in the "GO" state of reaction timer	Returns to the IDLE state	PASS	I.F.
8	Pressed Enter button before start button to initiate the reaction fsm	Seven-seg display remains off; all LED's are off	PASS	I.F.
9	Pressed Enter button after Reaction time is displayed	Reaction time remained on the seven-segment displays. Nothing is supposed to happen	PASS	I.F.
10	Pressed the start button three times while in the "GO" state	Green Led Remains on. Promptly after pressing the enter button, reaction time was displayed.	PASS	I.F.
	Verifying the random_wait times	Trail 1: 7.168 s Trail 2: 3.072 s Trail 3: 5.120 s	PASS	I.F

Irwin Frimpong
ECE 211 – Digital Circuits I
Prof. John Nestor
Health Monitor Technical Report

Pulse Monitor:

Test	*Action*	*Results*	*Pass/Fail*	*Initial*
1	SWO is flipped on	Display is displaying XXXXX000 **The X's represents the segments that are off*	PASS	I.F.
2	Placed finger on the sensor to measure pulse	**Pulse Monitor 1:** 104 **Apple Watch:** 95 **Pulse Monitor 2:** 112 **Apple Watch:** 91 **Pulse Monitor 3:** 95 **Apple Watch:** 84	PASS	I.F.
3	Reset Button pressed 5 times on different occasions	Display is reset to displaying XXXXX000	PASS	I.F.
4	No finger is placed on the sensor	Display reads XXXX000	PASS	I.F.

Simulations of Some Computational Modules of Pulse Monitor

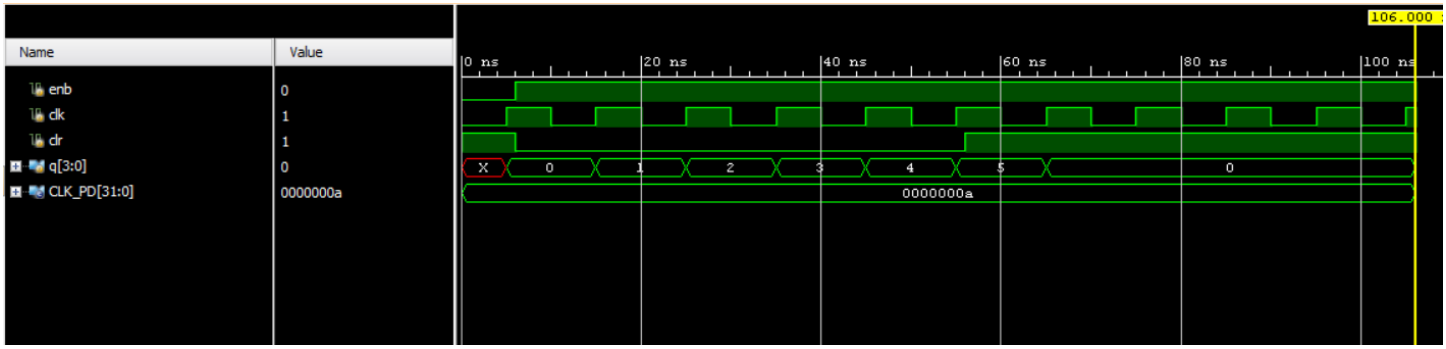

Figure 8: Simulation of Pulse Counter

Simple simulation was run on the `pulse_counter` module to see if it counted the right number of pulses for the five simulated clock cycles.

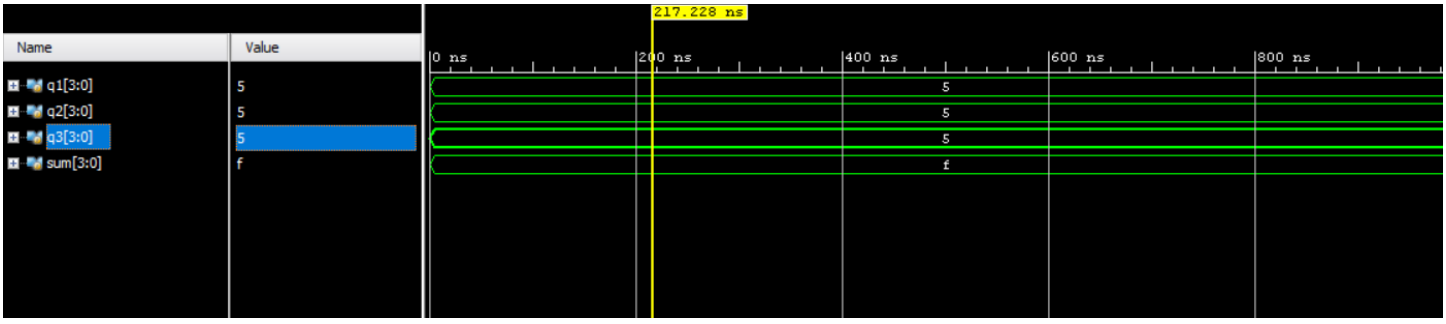

Figure 9: Simulation of Addition Module

Simulation run on the adder module to ensure that the values passed in as inputs were probably being passed and that the output sum value received the right value. In this case 5+5+5 yields f which is 15 in hexadecimal.

5 Summary

This report describes the implementation and testing of the health monitor implemented on an FPGA board. Through testing, it is verified that the final design of the reaction timer and pulse monitor, the two modes that make up the health monitor, successfully met all the design requirements.

6 Appendix A – Specifications

This section details the requirements for the Health Monitor as presented in the lab manual provided by Professor John Nestor.

The specification of the Health Monitor's inputs, outputs, and functions is as follows:

*** Extracted from ECE 211- Digital Circuits I Lab 10 Health Monitor Lab Manual produced by Prof. John Nestor*

Inputs

• Mode select switch (slide switch SW0)

• Reaction time START button start (pushbutton BUTNC)

• Reaction time ENTER button (pushbutton BUTNL)

• System RESET button (pushbutton BTNL)

• 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

o Receives a pulse signal from an analog pulse sensor on an attached daughterboard plugged into the PMOD connector.

o 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.

o Displays the user's pulse in beats per minute (BPM) up to a maximum of 255 BPM.

o Unused digits on the 7-segment display should be blank.

Irwin Frimpong
ECE 211 – Digital Circuits I
Prof. John Nestor
Health Monitor Technical Report

Reaction Timer

o 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.

o 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.

o 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.

- If 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 clock generated by a clock divider.

• All sequential logic (except the clock divider and single pulser circuits) should include a synchronous 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.

7 Appendix B

This section details the test plan employed for the Health Monitor, results are detailed in section 4, System Validation and performance.

Reaction Timer

- To verify that SW0, properly toggled between the two modes, the first test performed on the reaction timer was setting the SW0 to a logic low (turning the switch off). This caused the seven segments to turn off and remain off until the reaction time was to be displayed.
- To test the "EARLY" state of the reaction timer, the enter button was pressed before the "Go" LED which then caused the red light of the RGB LED to turn on for five seconds.
- To test the "LATE" state of the reaction timer, the enter button was pressed in the ten-second interval after the "Go" LED is displayed. A yellow LED is thus displayed for five seconds (Display *times were verified with smartphone stopwatch*).
- Tested reaction time, by conducting three different trials where the enter button was pressed within the ten-second interval following the "Go" LED turning on. Reaction times are detailed in the table in Section 4 of this report.
- Pressed Reset button after displaying reaction timer and the seven segments turn off
- Pressed Reset button while in the "Go" state, the green LED turns off and the device returns to the idle state
- Pressed enter button while the health monitor was in the IDLE state, seven segment displays which were originally off, remained off which is valid because the only button that should trigger the device out of the IDLE state is the start button.
- Pressed Enter button after reaction time is displayed; reaction timer remains displayed on the segments. The reset or start button triggers a change of state when the health monitor is in the DISPLAY state.
- Pressed the start button three times while in the "Go" state, the green LED remains on and promptly after pressing the enter button is the reaction time displayed.
- Verified random wait times after the start button is pressed. Results are shown in section 4.

Pulse Monitor

- Verified that the SW0 when turned on, switched from the reaction timer mode to the pulse monitor. Upon the initial switch toggle, XXXXX000 was displayed on the seven-segment display
- Tested the functionality and accuracy of the pulse counter by conducting three trails of pulse readings and comparing these readings against those obtained from an Apple Watch. The table in section 4 details the pulse readings obtained by both the pulse monitor and the Apple Watch. The values obtained from both, although not matching identically, are not too far off considering the hardware differences between the two. The Apple Watch produces a more accurate reading of pulse considering the technology used for such a task, as opposed to the pulse sensor which may have picked false readings while reading one's pulse.
- Pressed Reset Button; Display is reset to zero XXXXX000.
- No finger is placed on the sensor the value displayed on the sensor is XXXXX000.

Irwin Frimpong
ECE 211 – Digital Circuits I
Prof. John Nestor
Health Monitor Technical Report

Health Monitor Constraints Code:

```
# Clock signal
    #Bank = 35, Pin name = IO_L12P_T1_MRCC_35,                          Sch name = CLK100MHZ
    set_property PACKAGE_PIN E3 [get_ports clk100Mhz]
          set_property IOSTANDARD LVCMOS33 [get_ports clk100Mhz]
          create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports clk100Mhz]

##Pmod Header JA
##Bank = 15, Pin name = IO_L1N_T0_AD0N_15,                            Sch name = JA1
       set_property PACKAGE_PIN B13 [get_ports {pulse_in}]
              set_property IOSTANDARD LVCMOS33 [get_ports {pulse_in}]

## Seven Segment Displays

 ##7 segment display
 ##Bank = 34, Pin name = IO_L2N_T0_34, Sch name = CA
       set_property PACKAGE_PIN L3 [get_ports {segs_l[6]}]
              set_property IOSTANDARD LVCMOS33 [get_ports {segs_l[6]}]

 ##Bank = 34, Pin name = IO_L3N_T0_DQS_34,                             Sch name = CB
       set_property PACKAGE_PIN N1 [get_ports {segs_l[5]}]
              set_property IOSTANDARD LVCMOS33 [get_ports {segs_l[5]}]

 ##Bank = 34, Pin name = IO_L6N_T0_VREF_34,                            Sch name = CC
       set_property PACKAGE_PIN L5 [get_ports {segs_l[4]}]
              set_property IOSTANDARD LVCMOS33 [get_ports {segs_l[4]}]

 ##Bank = 34, Pin name = IO_L5N_T0_34,                                 Sch name = CD
       set_property PACKAGE_PIN L4 [get_ports {segs_l[3]}]
              set_property IOSTANDARD LVCMOS33 [get_ports {segs_l[3]}]

 ##Bank = 34, Pin name = IO_L2P_T0_34,                                 Sch name = CE
       set_property PACKAGE_PIN K3 [get_ports {segs_l[2]}]
              set_property IOSTANDARD LVCMOS33 [get_ports {segs_l[2]}]

 ##Bank = 34, Pin name = IO_L4N_T0_34,                                 Sch name = CF
        set_property PACKAGE_PIN M2 [get_ports {segs_l[1]}]
              set_property IOSTANDARD LVCMOS33 [get_ports {segs_l[1]}]

 ##Bank = 34, Pin name = IO_L6P_T0_34,                                 Sch name = CG
       set_property PACKAGE_PIN L6 [get_ports {segs_l[0]}]
              set_property IOSTANDARD LVCMOS33 [get_ports {segs_l[0]}]

 ##Bank = 34, Pin name = IO_L18N_T2_34,                                Sch name = AN0
       set_property PACKAGE_PIN N6 [get_ports {an_l[0]}]
              set_property IOSTANDARD LVCMOS33 [get_ports {an_l[0]}]

 ##Bank = 34, Pin name = IO_L18P_T2_34,                                Sch name = AN1
       set_property PACKAGE_PIN M6 [get_ports {an_l[1]}]
              set_property IOSTANDARD LVCMOS33 [get_ports {an_l[1]}]

 ##Bank = 34, Pin name = IO_L4P_T0_34,                                 Sch name = AN2
       set_property PACKAGE_PIN M3 [get_ports {an_l[2]}]
              set_property IOSTANDARD LVCMOS33 [get_ports {an_l[2]}]

 ##Bank = 34, Pin name = IO_L13_T2_MRCC_34,                            Sch name = AN3
        set_property PACKAGE_PIN N5 [get_ports {an_l[3]}]
              set_property IOSTANDARD LVCMOS33 [get_ports {an_l[3]}]

 ##Bank = 34, Pin name = IO_L3P_T0_DQS_34,                             Sch name = AN4
        set_property PACKAGE_PIN N2 [get_ports {an_l[4]}]
              set_property IOSTANDARD LVCMOS33 [get_ports {an_l[4]}]

 ##Bank = 34, Pin name = IO_L16N_T2_34,                                Sch name = AN5
        set_property PACKAGE_PIN N4 [get_ports {an_l[5]}]
              set_property IOSTANDARD LVCMOS33 [get_ports {an_l[5]}]

 ##Bank = 34, Pin name = IO_L1P_T0_34,                                    Sch name = AN6
       set_property PACKAGE_PIN L1 [get_ports {an_l[6]}]
```

```
               set_property IOSTANDARD LVCMOS33 [get_ports {an_l[6]}]

##Bank = 34, Pin name = IO_L1N_T034,                                          Sch name = AN7
       set_property PACKAGE_PIN M1 [get_ports {an_l[7]}]
              set_property IOSTANDARD LVCMOS33 [get_ports {an_l[7]}]

 #Bank = 34, Pin name = IO_L16P_T2_34,                                       Sch name = DP
       set_property PACKAGE_PIN M4 [get_ports dp_l]
              set_property IOSTANDARD LVCMOS33 [get_ports dp_l]


## RESET BUTTON

##Bank = CONFIG, Pin name = IO_L15N_T2_DQS_DOUT_CSO_B_14,                    Sch name = BTNL
      set_property PACKAGE_PIN T16 [get_ports rst]
              set_property IOSTANDARD LVCMOS33 [get_ports rst]

## START BUTTON

##Bank = 15, Pin name = IO_L11N_T1_SRCC_15,                                  Sch name = BTNC
      set_property PACKAGE_PIN E16 [get_ports start]
              set_property IOSTANDARD LVCMOS33 [get_ports start]

## ENTER BUTTON

##Bank = 14, Pin name = IO_25_14,                                            Sch name = BTNR
      set_property PACKAGE_PIN R10 [get_ports enter]
              set_property IOSTANDARD LVCMOS33 [get_ports enter]

## LEDs

##Bank = 34, Pin name = IO_0_34,                                             Sch name = LED17_R
      set_property PACKAGE_PIN K6 [get_ports led_r]
              set_property IOSTANDARD LVCMOS33 [get_ports led_r]


##Bank = 35, Pin name = IO_24P_T3_35,                                        Sch name = LED17_G
      set_property PACKAGE_PIN H6 [get_ports led_g]
              set_property IOSTANDARD LVCMOS33 [get_ports led_g]

##Bank = CONFIG, Pin name = IO_L3N_T0_DQS_EMCCLK_14,                         Sch name = LED17_B
       set_property PACKAGE_PIN L16 [get_ports led_b]
              set_property IOSTANDARD LVCMOS33 [get_ports led_b]
```