FPGA Health Monitor Technical Report

Vassilia Chrysanthopoulos

December 10, 2018

1 Abstract

The report presents a design of a health monitor which includes the functions of a reaction timer and pulse sensor. Both of these functions are implemented on one FPGA board. The reaction timer demonstrates the user's ability to react quickly to an LED. The pulse monitor displays a user's pulse on the FPGA board in beats per minute. This report includes a high-level description and design, how operations work, and information on the performance of the system as well as the success in functionality. The report explicitly shows that the final system meets all the requirements of the specification.

2 Introduction

Reaction timers are useful for many applications. They test a person's attention, as well their ability to react quickly. Different types of reaction timers are even used medically in order to help diagnose people with ADHD/ADD. However, this reaction timer differs in many of its functionalities. For example, medical reaction timers do not immediately show a user his or her results and therefore the user does not know how well they were able to pay attention and react. This simple reaction timer exceeds previously made reaction timers by completing the following objectives:

1. Provide a reaction timer, with simple yet informative commands for FPGA owners.
2. Provide simple controls using buttons such as `enter,reset,run.`
3. Once the `start` button is pressed, a green LED is displayed anywhere between 1-9 seconds.
4. The user is expected to respond to this green LED by pressing the `enter` button.
5. Include technical documentation, which allows the user to read the different results when buttons are pressed at different times. The technical documentation also explains how the health monitor works.

Pulse monitors have the ability to display the pulse of anyone just by the touch of a finger. This application is already used in many other devices, such as the Apple Watch. Similarly, this implementation of a pulse monitor is able to display the pulse of any user, when the FPGA board in the pulse monitor mode. The FPGA pulse monitor has implemented this function using the following objectives:

1. Provides a simple way to obtain the pulse of any user.
2. Displays the user's pulse in beats per minute (BPM) up to a maximum of 255.

3. Calculates this pulse as a moving average, so pulse will fluctuate depending on the users state.

This technical report describes in detail the completed health monitor. Section 3 shows the design of the health monitor, which includes a high-level description, block diagrams of the modules which include the organization of each, and description of each of the modules. Section 4 explains how the project passes all of its test, and demonstrates that the design meets the requirements specified. Supporting material, such as specifications, are located in the Appendix.

3 System Design

3.1 High-Level Design

Figure 1 depicts a high-level organization of the complete FPGA health monitor. The design of this health monitor involves the creation of an overall organization. It combines a set of combinational and sequential building blocks in order to implement the function properly. The health monitor displays its output on either the seven segment display or using LEDs depending on which mode the board is in. If switch 0 (SW0) on the FPGA board is switched on, the user's pulse can be measured, and if the switch (SW0) is in the off position, it tests the user's reaction time. Regardless of the mode of the FPGA board, the seven segment control should have the left four digits always off. In the reaction timer mode, the right four digits should display the reaction time. In the pulse monitor mode, three digits on the right should display the pulse of the user.

The inputs for this top level design are as follows:

1. `rst`- This button automatically resets the timer back to 0.000 for the reaction timer or 000 for the pulse monitor.
2. `start` - When in the reaction timer mode, this button begins the reaction timer, and the GO LED flashes after a random delay anywhere between 1-9 seconds.
3. `enter` - The user presses this button when they see the GO LED when using the reaction timer. This button will essentially stop the time from when the GO LED was displayed, to calculate how fast the user reacted.
4. `mode`- When switch 0 (SW0) is in the on position on the FPGA board, the pulse monitor can be used. When switch 0 (SW0) is in the off position, the reaction timer can be used.
5. `clk` - The synchronous design runs off a single clock signal, of 1000 Hz (1 kHz).

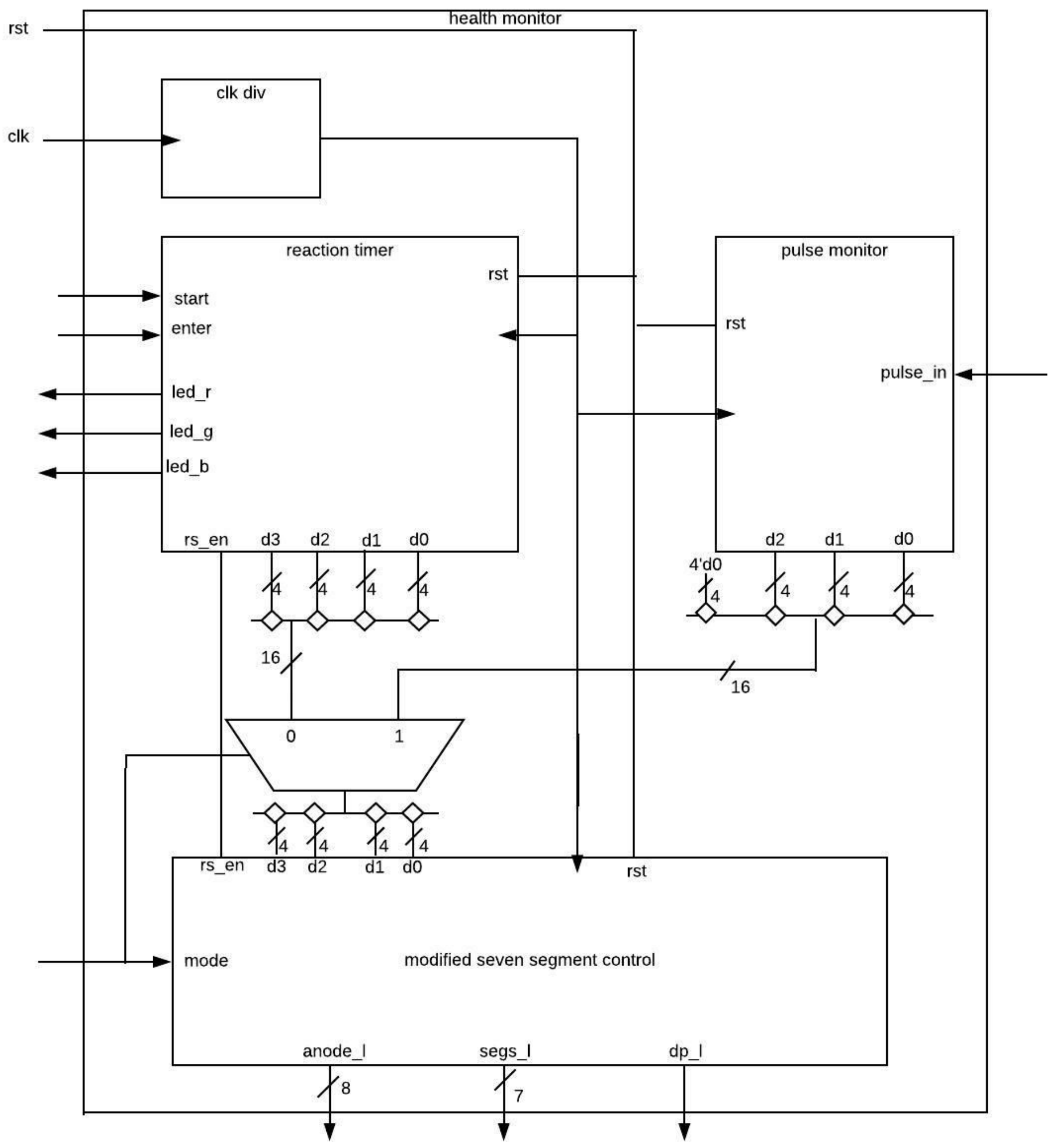

Figure 1. High-level block diagram of Health Monitor
Created by Lia Chrysanthopoulos

Details of the health monitor system in Figure 1 are described in Section 3.2

3.2 Implementation

This section describes the modules shown in Figure 1 that make up the health monitor. In Figure 1, the modules were implemented using Vivado, and were written in SystemVerilog.

3.2.1 Reaction Timer Top Level

A top level module, displays all of the submodules within a design. Figure 2 depicts the internal organization of the reaction timer.

3.2.1.1 Inputs

- `start`- This sends a signal to the reaction timer in order to generate a random number of seconds to wait and display the GO LED.
- `enter`- The user presses this button when they see the GO LED, and stops the timer from counting, so the time that has elapsed from when the GO LED turned on and the `enter` button was pressed is displayed on the seven segment control.

3.2.1.2 Outputs

- `led_r, led_g, led_b` - These three outputs together set the specific color of the LED that is displayed on the Nexys4 board, which is determined in the PWM module.
- `d3` - The amount of seconds it took for the user to press the `enter` button after the GO LED was displayed. This is displayed on the 4th place from the left on the seven segment display.
- `d2` - The amount of tenths of seconds it took for the user to press the `enter` button after the GO LED was displayed. This is displayed on the 3rd place from the left on the seven segment display.
- `d1` - The amount of hundredths of seconds it took for the user to press the `enter` button after the GO LED was displayed. This is displayed on the 2nd place from the left on the seven segment display.
- `d0` - The amount of thousandths of seconds it took for the user to press the `enter` button after the GO LED was displayed. This is displayed on the left most spot on the seven segment display.
- `rs_en` - When the `rs_en` is asserted false, the seven segment display is blank. This gets passed into the seven segment control module.

3.2.1.3 Functionality and Design

Figure 2 shows the organization of the top-level module. The system consists of a reaction timer finite state machine which is connected to a random wait time generator, a delay counter, a red green blue pulse width module, and a time counter. The inputs are first put through a debouncer and single pulser in order to filter out any noise in the signal.

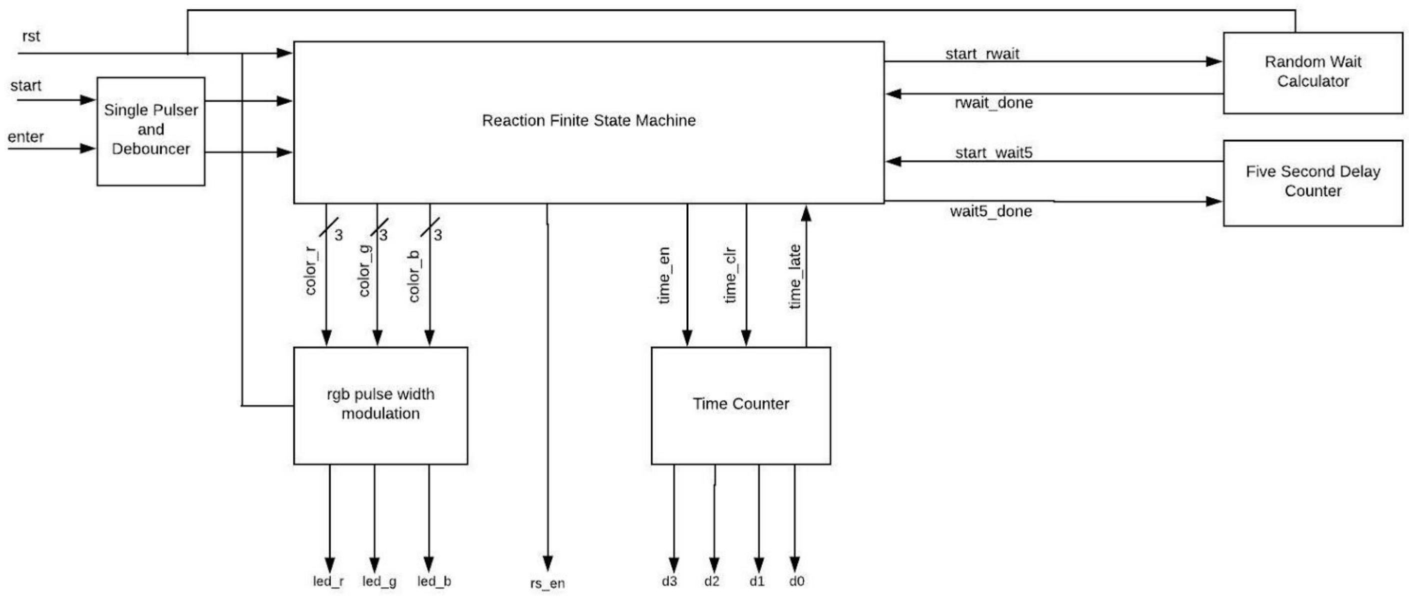

Figure 2 Reaction Timer top-level block diagram
Created by Zachary Martin

The functionality of these block diagrams is described below.

3.2.2 Reaction Finite State Machine

The reaction finite state machine, controls the reaction timer in terms what which signals are set as logic high or low. Figure 2.1 shows the state transition diagram of how the finite state machine operates. Once, the board is turned on and in the reaction timer mode, the FSM is in the IDLE state, until the `start` button is pressed. Once, the `start` button is pressed, the next state in the FSM is the RWAIT state, and `start_rwait` is asserted to true; essentially the random wait timer begins.

If `enter` is pressed before `rwait_done` is asserted true (the random number of seconds between 1-9 is not complete), the next state is the EARLY state. A red LED is displayed and `start_wait5` is enabled to true. The timer waits 5 seconds, `wait5_done` is asserted to false, and the next state is the IDLE state.

If `enter` is not pressed at all and `rwait_done` is asserted true. The next state is the TIME state. In the TIME state, `time_en` and `color_g` are asserted to true, meaning that the timer starts counting and a the GO LED is displayed. If the `enter` button is not pressed after 9.999 seconds, the next state is the LATE state. Here, `start_wait5, color_g,` and `color_r` are enabled to true, this starts the count of 5 seconds, and displays a yellow LED.

When 5 seconds have passed, `wait5_done` is set to true, and the next state in the FSM is the IDLE state. If the `enter` button is pressed within 9.999 seconds in the TIME state, the next state is the DISPLAY state. Here, `rs_en` is asserted to true, and the seven- segment control displays the time . If the `start` button is pressed or 5 seconds have passed then the next state is RWAIT. Otherwise, the FSM will remain in the DISPLAY state.

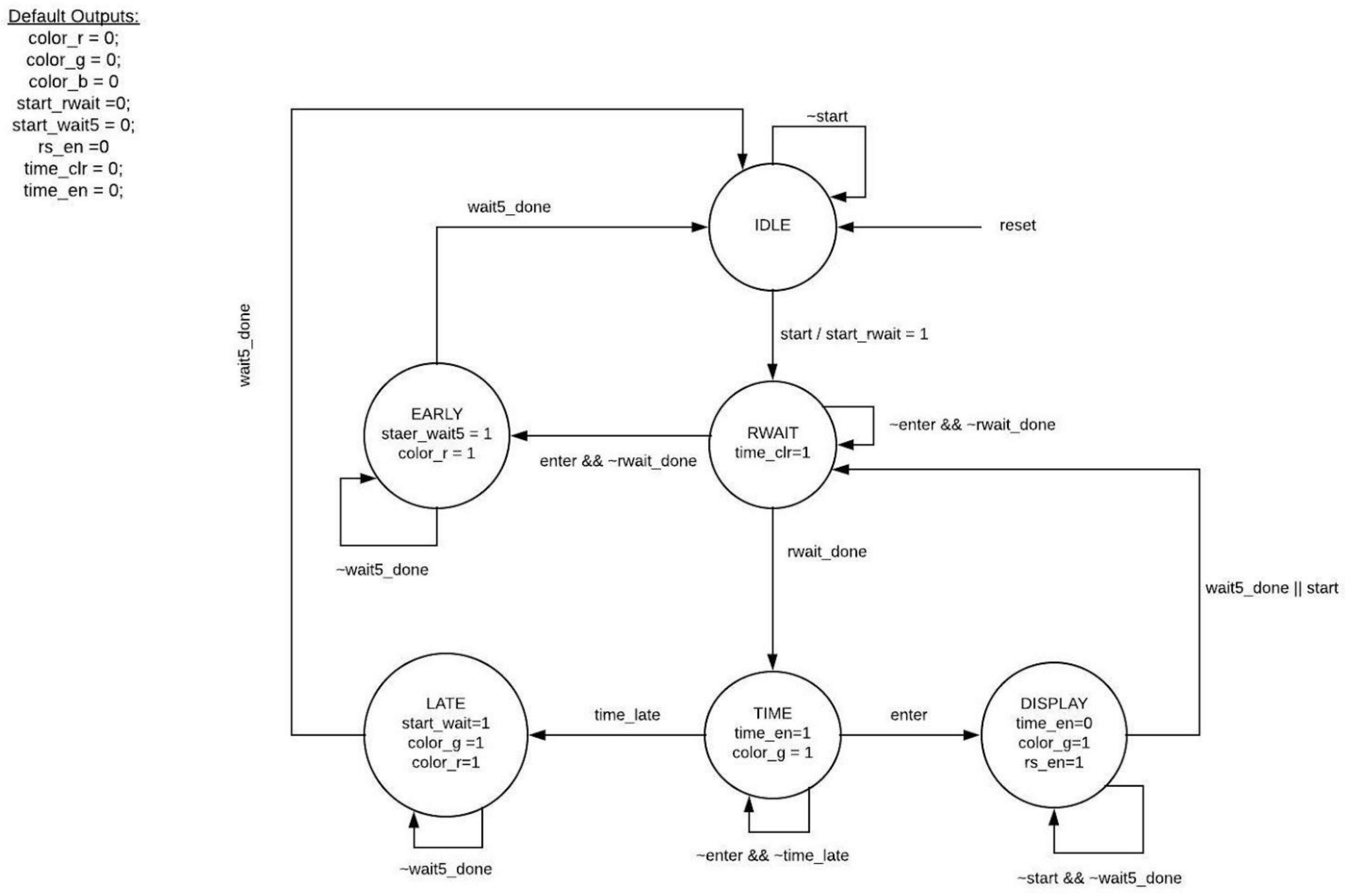

Figure 2.1 Finite State Machine: Reaction Timer
Created By Lia Chrysanthopoulos

3.2.3 Random Wait Calculator

The random wait calculator, counts from 1-9 continuously until the `start` button is pressed, the finite state machine asserts the `start_rwait` (random wait time) to be true. Once `rwait_done` is asserted to true, a random amount of seconds between 1-9 has passed, and the finite state machine goes into the TIME state and the GO LED is displayed on the FPGA board.

3.2.4 Five Second Delay Counter
The five second delay counter keeps track of when 5 seconds have passed after `start_wait5` has been enabled to true. Once, 5 seconds have passed, the signal `wait5_done` is asserted true and the module stops counting.

3.2.5 Time Counter
In the Finite state machine, the initial state for the input `time_clr` is low. Once the `start` button is pressed, the next state, RWAIT, sets `time_clr` to true, which displays nothing on the seven segment display. Once the random time is completed (`rwait_done`), the wire `time_en` is set to true and the clock begins counting, indicating that the GO LED has been turned on the FPGA board. The FSM waits until the `enter` button is pressed, in this case the `time_en` is equal to zero and the FSM is now in the DISPLAY state, which displays a green LED and the time elapsed on the seven segment display. Which are the outputs `d[0]:d[3]`.

3.2.6 RGB Pulse Width Modulation
To control the intensity of an LED, the concept called a *pulse- width modulation* (PWM) is used. This concept allows each LED output to connect to a periodic signal that has a varying duty cycle, this duty cycle is the the percentage of the period in which the signal is on. If the duty cycle is at a fast enough rate then the human eye filters out the flickering of the LED. The RGB PWM module, outputs a specific color depending on how the user reacts.

Looking at the FSM, when in the TIME state, if the `enter` button was pressed within 9.999 seconds, the signal is passed through the time counter and the time is displayed on the seven segment display.

If the `enter` button is not pressed within 9.999 seconds then the next state is the LATE state and the rgb_pwm displays a yellow LED. If the `enter` button is pressed before the green GO LED is displayed, the next state is the EARLY state and red LED is displayed. In both scenarios, the system waits five seconds by asserting `start_wait5` to true. After 5 seconds have passed, the signal `wait5_done` will be sent back to the reaction finite state machine turing off the red or yellow LED.

3.2.7 Debouncer
A debouncer is a counter that is incremented every sample pulse while the button is pressed. When the button is not pressed the counter is reset to zero. This ensures, that the button is actually pressed.

3.2.8 Single Pulser

The single pulser detects a rising edge of the input data, so that it counts each pulse exactly once. When a button is pressed, it outputs one signal that is a clock cycle long. In a circuit, the digital machine will only test the single-pulser output instead of directly using the pushbutton signal. This is very important because it will only detect each pulse exactly once.

3.3.1 Pulse Monitor Top Level

A top level module, displays all of the submodules within a design. Figure 3, depicts the internal organization of the pulse monitor.

3.3.1.1 Inputs

- `pulse_in` - The pulse received from an analog pulse sensor on an attached daughterboard plugged into the PMOD JB connector input pin 1.
- `clk`- The 100 MHz clock provided by an external oscillator circuit that produces a signal at the desired frequency.

3.3.1.2 Outputs

- `pd2` - The hundreds place of the users pulse. This is displayed on the 3rd spot form the right on the seven segment display.
- `pd1` - The tens place of the users pulse. This is displayed on the 2nd spot form the right on the seven segment display.
- `pd0` - The ones place of the users pulse. This is displayed on the rightmost spot on the seven segment display.

3.3.1.3 Functionality and Design

Figure 3 shows the organization of the top-level module for the pulse monitor. The system consists of a pulse counter, 3 registers, a delay counter, a convert to BMP module, and a convert from binary to bcd module.

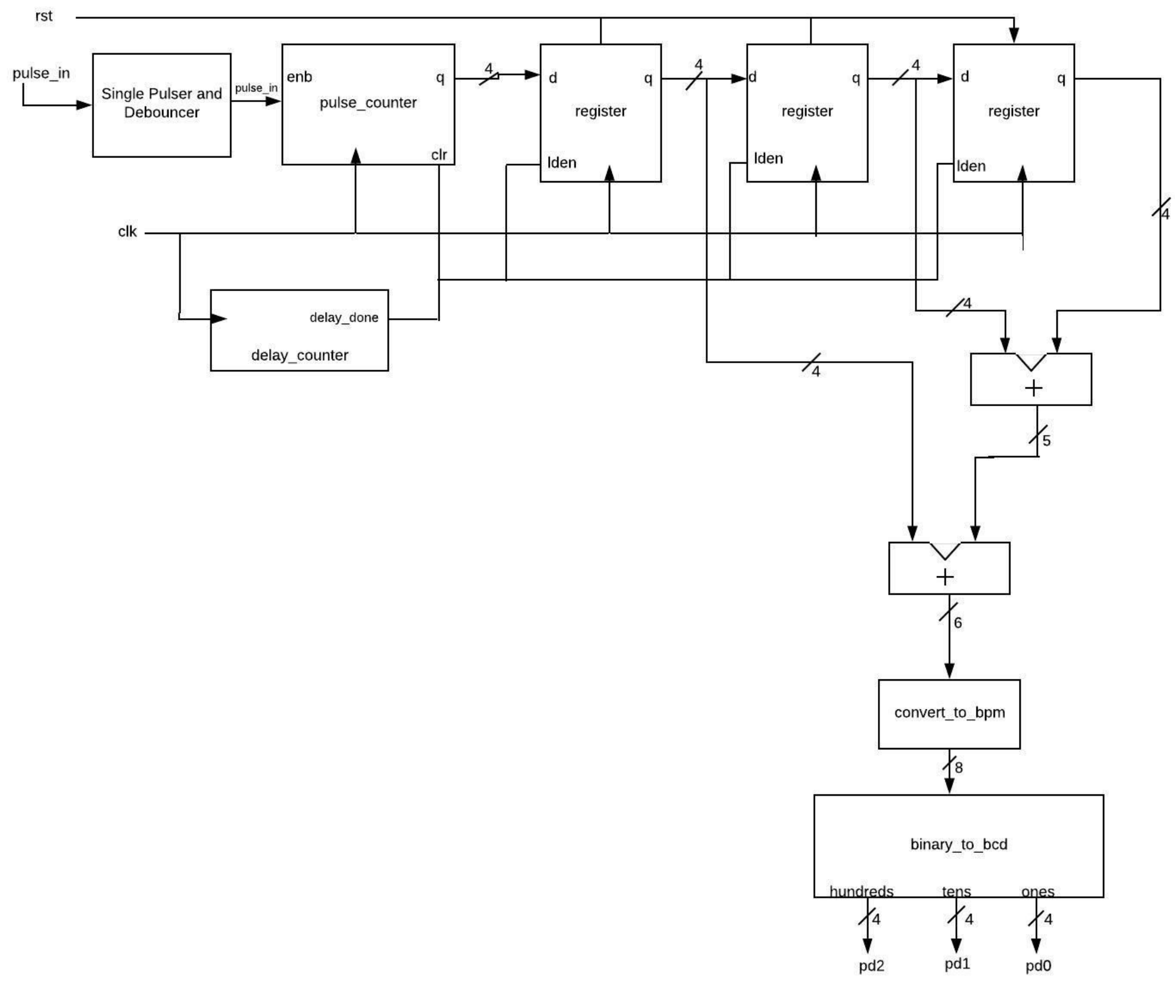

Figure 3: Pulse Monitor top- level block diagram
Created By Lia Chrysanthopoulos

3.3.2 Single Pulser and Debouncer

The Single pulser and debouncers first filter the signal of the pulse. This module helps the system distinguish real input from outside noise.
The single pulser detects a rising edge of the input data. When a button is pressed, it outputs a single signal that is a clock cycle long. In a circuit, the digital machine will only test the single-pulser output instead of directly using the pushbutton signal. This is very important because it will only detect each pulse exactly once.
A debouncer is a counter that is incremented every sample pulse while the button is pressed. When the button is not pressed the counter is reset to zero. This ensures, that the button is actually pressed.

3.3.3 Pulse counter

The pulse counter module counts up by 1 every time it receives a signal (`pulse_en` equals 1). If the reset button is pressed, then the `pulse counter` and the three registers are all cleared.

3.3.4 Delay Counter

The delay counter counts for 5 seconds. After 5 seconds have passed, the wire `delay_done` is asserted to 1 and the pulse counter is cleared. When `delay_done` is asserted to 1, `lden` "loads" the value into the registers by shifting the registers current value to the left. The `pulse counter` would send its current value signal to the register next to it.

3.3.5 Convert to BPM

The convert to BPM module would shift the bps number to the left 2 bits, and fill the lower bits with 0s. This is the same as multiplying the number by 4. This averages the pulse over a 60 second time period; since we had 3 registers taking data at 5 seconds each; 5(3) x 4 = 60, in order to obtain beats per minute.

3.3.6 Binary to BCD

The BPM value is the passed to the binary to bcd module, the double dabble algorithm is implemented here. The binary number is converted into 3 digits representing the BCD digits; hundreds, tens, and ones. These numbers are then outputted to the seven segment display.

3.4.1 Seven Segment Controller

The block diagram in Figure 4, shows a 7 seven segment controller with a clock, reset, rs_en, mode along with eight 4-bit inputs. Each of these 4-bit inputs represents a value that will be displayed on the FPGA board. The 8 digits will display one at a time at a remarkably fast rate; so fast, that the human eye cannot detect this refresh rate and all the digits will appear at the same time.

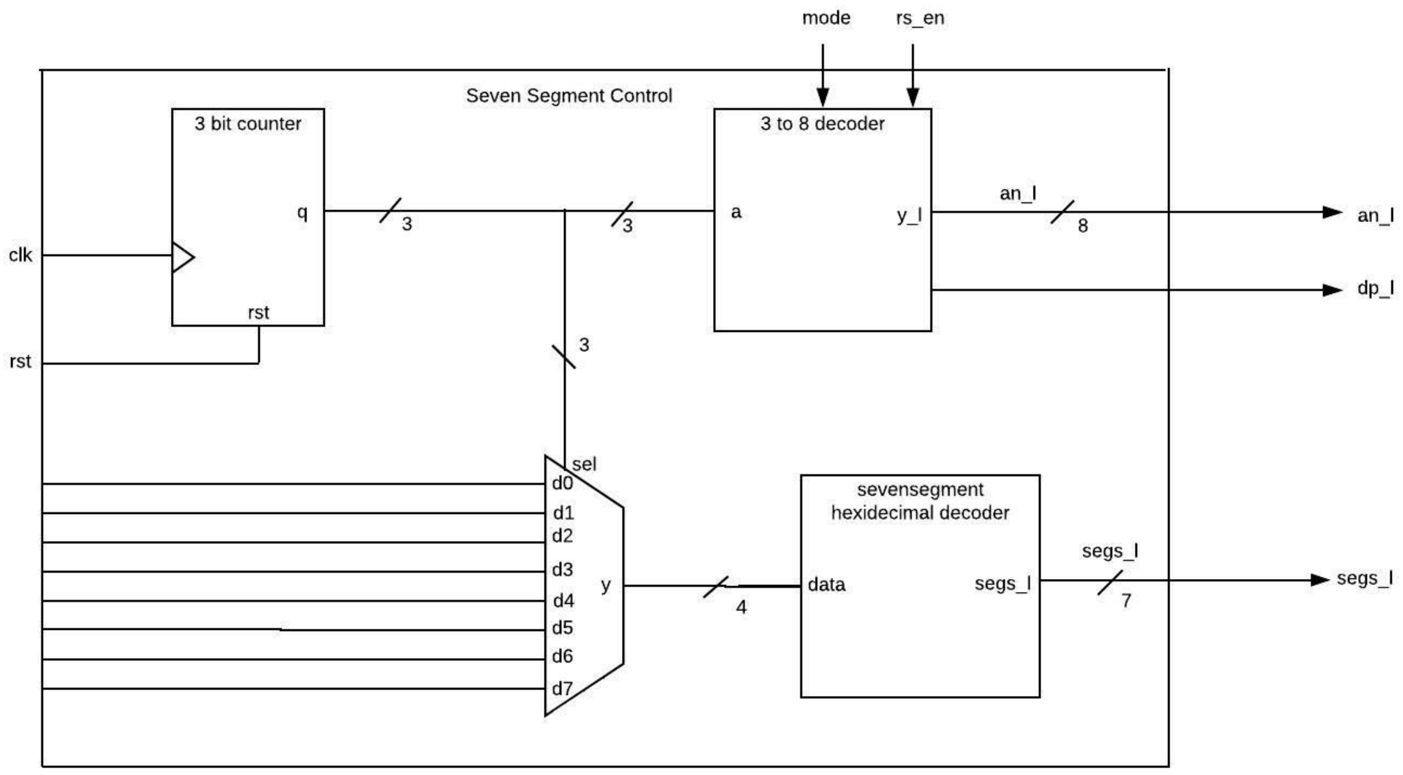

Figure 4: Seven segment controller organization
Created By Lia Chrysanthopoulos

3.4.1.1 Inputs

- `clk` - The synchronous design runs off a single clock signal, of 1000 Hz (1 kHz).
- `rst` - This button clears the `3 bit counter`, and sets the output values to all zeros. Therefore, sets the seven segment display to display only zeros.
- `rs_en` - This wire when asserted false, displays nothing to the 7-segment control.
- `mode` - This determines what mode the health monitor is in. If `mode` is equal to 0 the health monitor works as a reaction timer. If `mode` is equal to 1 then the health monitor works as a pulse monitor.
- `[d0:d7]` - These 4-bit inputs each represent the value that will be displayed on the eight digit display on the Nexys4 board. `d0` represents the rightmost digit on the seven segment display and `d7` represents the leftmost digit.

3.4.1.2 Outputs

- `an_l` - This active low output represents the position of the digit on the 8 digit display on the FPGA board. `an_l[7]` represents the leftmost digit and `an_l[0]` represents the rightmost digit.
- `segs_l` - This active low output represents the segment of a digit on the FPGA board. Each of these determines which segment of the digit should be on or off, in order to display the correct digit.
- `dp_l` - This active low variable is the decimal point. The decimal point should be displayed only when the health monitor is in the reaction timer mode. The decimal point position will appear after 3 digits from the right on the seven segment display controller.

3.4.1.3 Functionality and Design

The 3 bit counter indicates which digit is currently logic high, the counter iterates through the 3 bit binary sequence. The 3:8 decoder enables a single anode control line for each counter value. and the digits are displayed on the seven segment display with `an_l[0]` being on the right most side of the display and `an_l[7]` being on the leftmost side. These are then connected to an 8:1 multiplexer, and each digit value is connected to the seven segment decoder. The correct number is then displayed on the FPGA board. The decimal point is either high or low depending on what mode the Health monitor is in. The decimal point will appear when asserterted to logic low, which will be when the health monitor is in the reaction timer mode. The decimal point will be displayed, after the third digit from the right, on the FPGA board.

4 System Validation and Performance

The test plan for the health monitor is provided in Appendix B. Proper operation of the health monitor proves to work properly according to the following plan. The results of each test for the reaction timer are labeled and determined in Table 1. The results of each test for the pulse monitor are labeled and determined in Table 2.

Table 1 - Reaction Timer

Test	Action	Result	Pass/Fail	Initial
1	Checked initial 7 segment control display.	7 segment display controller is blank.	PASS	VC
2	Pressed reset.	7 segment display controller is blank.	PASS	VC

3	Pressed `start` button.	Random time between 1-9 seconds passed before green GO LED turned on. Tested random time by running multiple trials using a seperate stopwatch. The stopwatch was started when the `start` button was pressed and stopped when the green GO LED displayed.	PASS	VC
4	Pressed `start` button and pressed `enter` button too early.	Red LED displayed for 5 seconds and shut off. Tested duration of 5 seconds by using a seperate stopwatch. Started stopwatch when red LED displayed and stopped it when red LED shut off.	PASS	VC
5	Pressed `start` button and didn’t press `enter` within 9.999 seconds.	LED turned yellow for 5 seconds then shut off. Tested duration of 5 seconds by using a seperate stopwatch. Started stopwatch when yellow LED displayed and stopped it when yellow LED shut off.	PASS	VC
6	Pressed `start` button and pressed `enter` button within 9.999 seconds.	LED remained green, time elapsed appeared on seven segment display controller.	PASS	VC
7	Pressed reset.	Blank on 7 segment display	PASS	VC

Table 2 - Pulse Monitor

Tes t	Action	Result	Pass/Fail	Initial
1	Check initial 7 segment control display.	7 - segment display shows 000.	PASS	VC
2	Pressed reset.	After button is pressed, 7-segment display shows 000.	PASS	VC

3	Put finger on pulse sensor PCB.	Average pulse over 5 second period appeared on 7 segment display. Tested accurate pulse reading by calculating my own pulse reading using a seperate watch and my fingers as a pulse sensor.	PASS	VC
4	Released finger from PCB.	7 segment display continues to average pulse. There is no pulse, and 7 segment display eventually displays 000	PASS	VC

5 Conclusion

This report describes the testing and implementation of a health monitor on a Nexys4 FPGA board. The design met all of the requirements and successfully worked.

6 Appendix A - Specifications

This section lists the requirements for the Health Monitor, provided by Professor Nestor in the document "Lab 10- Health Monitor". These are tested and verified by the elements outlined in Appendix B.

- When SW0 is 0 (off) the health monitor is in the reaction timer mode should initially display nothing on the seven segment display.
- When the user hits the `start` button (BUTNC) the LED should light up green after a random number of seconds between 1-9 have passed.
- After the random wait, the green GO LED will turn on. The user must press the `enter` button (BUTNL) and the 7 segment display will show the reaction time of the user.
    - The first 4 left most places of the seven segment display are off and the rest of the display reads the reaction time A.BCD - where A is measured in seconds, B is measured in tens of seconds, C is measured in hundredths of seconds, and D is measured in thousandths of seconds.
- After the green GO LED has been turned on, if the user does not press the `enter` button before 9.999 seconds have passed a yellow LED must turn on for 5 seconds then shut off.
    - The program should wait for the `start` button to be pressed again.
- If the user presses the `enter` button before the green GO LED turns on, the LED will turn red for 5 seconds then turn off.
    - The program should wait for the `start` button to be pressed again.
- If the reset button is pressed, the LED is turned off until the `start` button is pressed.

- When SW0 is 1 (on) the health monitor is in the pulse monitor mode and should initially display 000 on the 7-segment display.
- When a user places their finger on the PMOD JB connector the 7 segment display will display the users pulse in BPM.
- If the reset button is pressed at any point, the 7 segment display appears as 000 where only the right most 3 bits are on.

7 Appendix B - Test Plan

This section details the test plan for each of the requirements of Health Monitor provided to us by Professor Nestor, in the document "Lab 10- Health Monitor". This test plan, tests all functions listed in Appendix A.

1. Verify that when the SW0 is off, the FPGA program is in the reaction timer mode and the seven segment display is blank. Confirm displays correctly on the startup.
2. Verify that when the user hits the `start` button, the 7 segment display is blank and a green LED is displayed after a random number of seconds. Confirm that a random number of seconds between 1-9 has passed by creating multiple tests using an outside stopwatch, to record the time from when the start button was pressed to when then green LED turns on. Confirm that the 7 segment display is blank and the green LED lights up after a random amount of time between 1-9 seconds.
3. Verify that when the user hits the `enter` button within 9.999 seconds, the 7 segment display shows the reaction time of the user. Confirm the seven segmet display shows the format X.XXX.
4. Verify that when the user doesn't hits the `enter` button after 9.999 seconds, a yellow LED turns on and turns off after 5 seconds. Tested the correct elapsed time of 5 seconds by using a seperate stopwatch. The outside stopwatch started when yellow LED displayed and was stopped when yellow LED shut off. Confirm the yellow LED turns off after 5 seconds.
    a. Verify that when the `start` button was pressed again, the green GO LED displayed after a random amount of time between 1-9 seconds. Confirm that the green GO LED displayed again after the yellow LED shut off and the `start` button was pressed again.
5. Verify that when the user hits the `enter` button before the green GO LED turns on, a red LED turns on. Tested the correct elapsed time of 5 seconds by using a seperate stopwatch. The outside stopwatch started when red LED displayed and was stopped when red LED shut off. Confirm the red LED turns off after 5 seconds.
    a. Verify that when the `start` button was pressed again, the green GO LED displayed after a random amount of time between 1-9 seconds. Confirm that the green GO LED displayed again after the red LED shut off and the `start` button was pressed again.

6. Verify that when the user presses the `rst` button, the program resets. Confirm that the 7 segment display turns off and no LEDs are on.
7. Verify that when SW0 is on, the FPGA board in the pulse monitor mode. Confirm that the 7 segment display is blank for the first 5 digit places from the left and displays 000 on the rightmost side.
8. Verify that when the user places his or her finger on the PMOD connector the pulse monitor begins. Confirm that the pulse monitor displays the users pulse in BPM in the form XXX.
9. Verify that when the reset button is pressed, at any point, the 7 segment control is reset to 0. Confirm that the seven segment display shows 000 on the rightmost side

8 Code Listings

Health Monitor Top

```
'timescale 1ns / 1ps
module Health_Monitor(input logic clk_100MHz, rst, pulse_in, start, enter, mode,
                      output logic[7:0] an_l, output logic[6:0] segs_l, output logic dp, led_r,
                      led_g, led_b);

logic sclk, start_debounced, start_pulsed, enter_debounced, enter_pulsed, rs_en;
logic[3:0] d0, d1, d2, d3, pd0, pd1, pd2, pd3;
logic[15:0] timerOutput, monitorOutput, d;

clkdiv #(.DIVFREQ(1000)) U_CLKDIV(.clk(clk_100MHz), .reset(1'b0), .sclk(sclk));
//start
debounce debouncerSTART(.clk(sclk) , .pb(start), .pb_debounced(start_debounced));
single_pulser pulserSTART ( .clk(sclk), .din(start_debounced), .d_pulse(start_pulsed));
//enter
debounce debouncerENTER(.clk(sclk) , .pb(enter), .pb_debounced(enter_debounced));
single_pulser pulserENTER ( .clk(sclk), .din(enter_debounced), .d_pulse(enter_pulsed));

pulse_monitor monitor(.pulse_in(pulse_in), .clk(sclk) , .rst(rst), .pd0, .pd2, .pd1, .pd3);

reaction_timer reactTime(.start(start_pulsed), .enter(enter_pulsed), .clk(sclk), .rst, .led_r,
.led_g, .led_b, .rs_en, .d0, .d1, .d2, .d3);

assign timerOutput = {d3, d2, d1, d0};
assign monitorOutput = {pd3, pd2, pd1, pd0};

mux_16bit mux(.react(timerOutput), .pulse(monitorOutput), .mode, .d);

sevenseg_control segControl(.clk(sclk), .rst, .mode, .rs_en, .d0(d[3:0]), .d1(d[7:4]),
.d2(d[11:8]), .d3(d[15:12]),.d4(4'b0), .d5(4'b0), .d6(4'b0), .d7(4'b0), .an_l(an_l),
.segs_l(segs_l), .dp(dp));
endmodule
```

Reaction Timer Top

```
`timescale 1ns / 1ps
module reaction_timer( input logic start, enter, clk, rst, output logic led_r, led_g, led_b,
                      rs_en, output logic[3:0] d0, d1, d2, d3);
    //rgb_r, rgb_g, rgb_b
    logic rwait_done, wait5_done, start_rwait, start_wait5, time_late, time_clr, time_en;
    logic [2:0] color_r, color_g, color_b;
    reaction_fsm reactionControl(.start, .enter, .rwait_done, .wait5_done, .time_late, .rst, 
.clk,
                                 .start_rwait, .start_wait5, .time_clr, .time_en, .rs_en,
                                 .color_r, .color_g, .color_b );
    random_wait rwait(.clk,.rst, .start_wait(start_rwait), .wait_done(rwait_done));
    delay_counter_react delayCount(.clk, .start_wait5, .delay_done(wait5_done));
    time_count timer(.time_clr, .time_en, .clk, .d0, .d1, .d2, .d3, .time_late);
    rgb_pwm ledControl(.clk, .rst, .color_r, .color_g, .color_b, .rgb_r(led_r), .rgb_g(led_g),
                       .rgb_b(led_b))  ;

endmodule
```

Pulse Monitor Top

```
`timescale 1ns / 1ps

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

logic pulse_enb;
logic clear;
logic [3:0] q0, q1, q2, q; //registers
logic [4:0] adder1;
logic [5:0] adder2;
logic [7:0] bpm_final;
```

```
 assign pd3 = 4'b0;
    single_pulser pulser(.clk(clk), .din(pulse_in), .d_pulse(pulse_enb));
    delay_counter d_counter(.clk(clk), .delay_done(clear));
    pulse_counter pcounter(.enb(pulse_enb), .clk(clk), .clr(clear), .q(q));

 //registers
    always_ff @ (posedge clk)
    if(rst)
        begin
            q2 <=0;
            q1 <=0;
            q0 <=0;
        end
     else if(clear)
        begin
            q0 <=q1;
            q1 <=q2;
            q2 <=q;
        end
//adders
    always_comb
        begin
         adder1 = q0 + q1;
         adder2 = adder1 + q2;
        end

    convert_to_bpm convert(.bps(adder2), .bpm(bpm_final));
    binary_to_bcd binary2bcd (.b(bpm_final), .hundreds(pd2), .tens(pd1), .ones(pd0));

endmodule
```

Seven Segment Control Top

```
`timescale 1ns / 1ps

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

logic [2:0]sel;
logic [3:0] y_2_data;

  always_comb
    if(sel ==3'd3 && ~mode) dp =0;
    else dp =1;

count_3bit counter (.clk(clk), .rst(rst), .q(sel));
dec_3_8 decoder (.mode(mode), .rs_en(rs_en), .a(sel), .an_l(an_l));
mux_4bit mux (.d0(d0), .d1(d1), .d2(d2), .d3(d3), .d4(d4), .d5(d5), .d6(d6), .d7(d7), .s(sel),
              .y(y_2_data));
sevenseg_hex sevenseg (.data(y_2_data), .segs_l(segs_l));


endmodule
```

Clk DIV

```
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
```

Pulse Counter

```
`timescale 1ns / 1ps

module pulse_counter(input logic enb, clk, clr, output logic [3:0] q);
    always_ff @ (posedge clk)
        if (clr) q <=0;
        else if (enb) q<=q+1;
endmodule
```

Delay Counter (Pulse Monitor)

```
timescale 1ns / 1ps

module delay_counter(input logic clk, output logic delay_done);

logic [12:0] count;
    always_ff @ (posedge clk)
    begin
        count <= count+1;
        if(count == 13'd5000)
          begin
             delay_done <= 1;
             count <=0;
          end
        else delay_done <= 0;
    end


endmodule
```

convert to bpm

```
`timescale 1ns / 1ps

module convert_to_bpm(input logic [5:0] bps, output logic [7:0] bpm);

    assign bpm = bps<<2; //shift left, and fill lower bits with 0

endmodule
```

binaryto bcd

```
`timescale 1ns / 1ps

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 [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 ones = {y5[2:0],b[0]};
assign tens = {y7[2:0],y5[3]};
assign hundreds = {2'b0,y6[3],y7[3]};

endmodule
```

reaction fsm

```
`timescale 1ns / 1ps

module reaction_fsm(input logic start, enter, rwait_done, wait5_done, time_late, rst, clk,
                    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, 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;
     localparam logic [2:0] OFF = 3'b000, ON=3'b011;

     always_comb
        begin
            color_r = OFF;
            color_g = OFF;
            color_b = OFF;
            start_rwait = 0;
            start_wait5 = 0;
            rs_en = 0;
            time_clr = 0;
            time_en = 0;
            next = IDLE;
            case(state)
                IDLE:
                    begin
                        rs_en = 0;
                        color_r = OFF;
                        color_g = OFF;
                        color_b = OFF;
                        if(start)
                          begin
                            start_rwait = 1;
                            next <=RWAIT;
                          end
                        else  next <=IDLE;
                    end
               RWAIT:
                   begin
                        color_r = OFF;
                        color_g = OFF;
                        color_b = OFF;
                        rs_en = 0;
                        time_clr = 1;
                        if (~enter && ~rwait_done) next <= RWAIT;
                        else if(enter && ~rwait_done) next <= EARLY;
                        else if(rwait_done)
                         begin
                           time_en = 1;
                           next <= TIME;
                         end
                   end
              EARLY:
                    begin
                        start_wait5 = 1;
                        color_r = ON;
                        color_g = OFF;
                        color_b = OFF;
                        if(wait5_done) next <= IDLE;
                        else next <= EARLY;
```

```
        end
TIME:
        begin
            color_r = OFF;
            color_g = ON;
            color_b = OFF;
            time_en = 1;
            if(time_late)  next <= LATE;
            else if(enter)
                begin
                    time_en =0;

                    next <= DISPLAY;
                end
            else
             next <= TIME;

        end
LATE:
        begin
            start_wait5 =1;
            color_g = ON;
            color_b = OFF;
            color_r = ON;
            if(wait5_done) next<= IDLE;
            else next <= LATE;
        end
DISPLAY:
        begin
            color_r = OFF;
            color_g = ON;
            color_b = OFF;
            start_wait5 = 1;
            rs_en = 1;
```

```
                        if(wait5_done) next <= RWAIT;
                        if(start) next <= RWAIT;
                        else next <= DISPLAY;
                    end
     endcase
   end
endmodule
```

rgb pwm

```
`timescale 1ns / 1ps

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[3:0] count;

always_ff @ (posedge clk)
begin
   if(rst) count <= 0;
   else count <= count + 1;
   rgb_r <= {1'b0, color_r} > count;
   rgb_g <= {1'b0, color_g} > count;
   rgb_b <= {1'b0, color_b} > count;
   end
endmodule
```

time count

```
`timescale 1ns / 1ps

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

    bcd_counter_top bcd_timer( .en(time_en), .rst(time_clr), .clk(clk), .time_late(time_late),
.d0(d0), .d1(d1), .d2(d2), .d3(d3));

endmodule
```

delay counter (reaction timer)

```
`timescale 1ns / 1ps
module delay_counter_react(input logic clk, start_wait5, output logic delay_done);

logic [12:0] count;
    always_ff @ (posedge clk)
    if(start_wait5)
    begin
        count <= count+1;
        if(count == 13'd5000)
          begin
             delay_done <= 1;
             count <=0;
          end
        else delay_done <= 0;
    end

  endmodule
```

random wait

```
`timescale 1ns / 1ps

module random_wait(input logic clk, rst, start_wait, output logic wait_done);

logic[2:0] random_num, random_saved;
logic[3:0]  adder_rand;
logic[13:0] wait_cycles, delay_cycles;

always_ff @ (posedge clk)
   if(rst) random_num <= 3'd0;
   else if(start_wait)
         begin
            random_saved <= random_num;
            delay_cycles <= 14'd0;
         end
   else
         begin
            random_num <= random_num + 1;
            delay_cycles <= delay_cycles + 1;
          end

always_comb
      begin
         adder_rand = random_saved + 3'd1;
         wait_cycles = adder_rand << 10;
      end

assign wait_done = (delay_cycles == wait_cycles);

endmodule
```