**Otieno Maurice Baraza**

**<otienom@lafayette.edu>**

**Electrical and computer engineering**

**Date: Sun, 14th May**

# HEALTH MONITOR REPORT

**Table of contents**

## ABSTRACT

This project focuses on the design of a simple health monitor. The health monitor consists of two major parts i.e., a pulse sensor equipped with a pulse monitor and a reaction timer. The pulse sensor is fitted on the finger and detects pulses as blood in an artery flows through that part of the finger while the reaction timer records the time it takes for the user to react.

## INTRODUCTION

- Health monitors are important devices in our day-to-day activities. Whether one is an athlete, medical personnel, a life guard or just a health enthusiast these devices help us get information useful information of how our bodies function with time.
- An example of a health monitor is a pulse monitor, a device that measures the rate at which pulses flow through a given section in the body.
- Such health monitors come in various sizes and with advancements in technology they have been fit into a small device like an apple watch.
- In this project we designed a simple health monitor that detected a pulse and showed the value of that pulse on a seven-segment display. We also designed a reaction timer to show the reaction time of a user .

## SYSTEM DESIGN

### 1. System specification

The health monitor has the following specifications.

- System inputs

a. A mode select switch (SW0 on the PCB)
b. Pulse sensor (connected to the JXADC port on the PCB)
c. System reset button (push button BTNL)
d. A start button to start the reaction timer.
e. An enter button for the reaction timer.
f.
g.

- System outputs

a. The eight-digit seven segment display
b. The Red LED Lamp (RN16) for the reaction timer
c. The Green LED Lamp (GR11) for the reaction timer
d. The Blue LED Lamp (BG14) for the reaction timer
e. The Red pulse LED Lamp (RN15) for the pulse sensor
f. The green LEDS from H17 to V11 for the pulse sensor

- Modes of operation

The switch will be used to alternate between the pulse monitor and the reaction timer.

➢ **The pulse Monitor.**

- The pulse monitor receives a pulse signal from the analog pulse sensor attached to a daughterboard plugged into the PMOD connector of the PCB.
- It counts the number of pulses in five-minute intervals while storing the last three samples to calculate the user's pulse as a continuous average. It displays the user's pulse in beats per minute.

➢ **The Reaction timer**

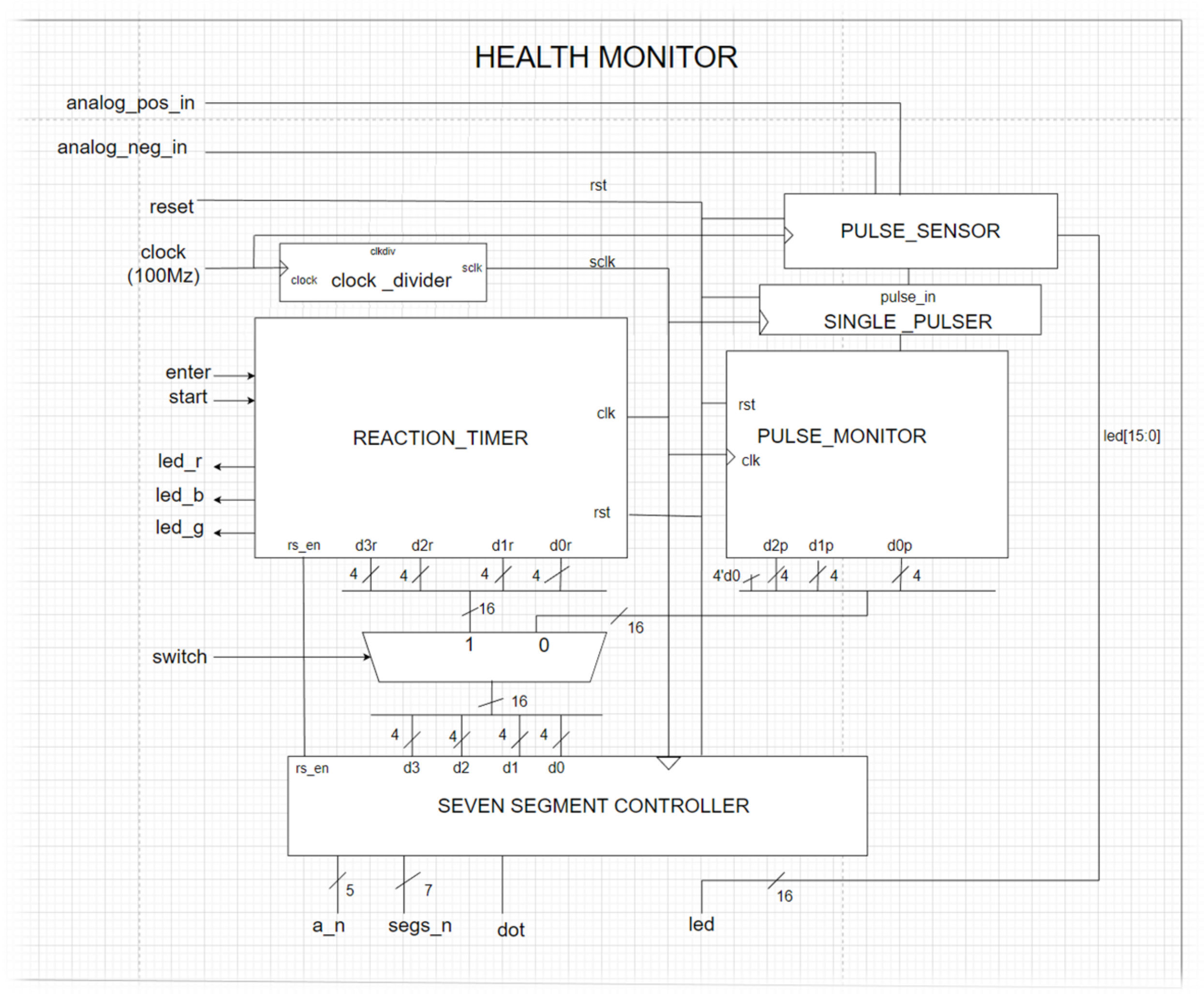
- When the reset button is pressed the, the system reaction timer lamp will turn GREEN to indicate GO and wait for the start button to be pressed.
- When the Start button is pressed the reaction timer lamp will go OFF to indicate WAIT. The system will wait for a random number of seconds between one second and nine seconds before the user presses the enter button. The reaction timer Lamp will then turn WHITE to indicate that the user is free to press the ENTER button. If the user presses the enter button before the lamp turns white, the lamp will turn RED to indicate an ERROR.
- Once the lamp turns red, it will continue displaying red for five seconds before automatically going back to display green and wait for the start button to be pressed.

- Once the lamp turns white, the system will count the amount of time the user takes to press the enter button. If the user presses the enter button while the lamp is white, the four lower digits of the seven-segment display will turn on and display the time taken to press the enter button. The reaction timer lamp will turn green and continue displaying green until the start button is pressed again. Likewise, the seven-segment display will continue to display the time until the start button is pressed then go off.
- If the enter button is not pressed within five seconds after the reaction timer lamp turns white, the lamp will turn YELLOW to indicate an ERROR meaning the user waited for too long to respond. It will continue displaying yellow for five seconds before turning green and waiting for the start button to be pressed.

## 2. High Level System Design

- The design of the health monitor consists of a reaction timer and a pulse monitor. The design of the entire project includes the pulse sensor, a clock divider and a multiplexer to choose what to display when the switch is or is not asserted.

*Figure 1 health monitor high level design diagram*



## 3. System Implementation

- The designs for all the modules wer done using system Verilog and simulated in vivado
- The module that implements the health monitor in system Verilog in its entirety is the lab10top
- Figure 1 shows the high-level design diagram for the health monitor project. It consists of three huge modules and two smaller modules and a multiplexer.

**Inputs:**

- analog_pos_in – brings in positive analog signal from the pulse sensor
- analog_neg_in – brings in negative analog signal from the pulse sensor
- enter – signal to stop time count and display the value on the seven-segment display
- start – signal to start the reaction timer
- reset – resets the entire system
- clock – clock signal at which the sensor operates. Must be divided for other components to 1kHz

**outputs:**

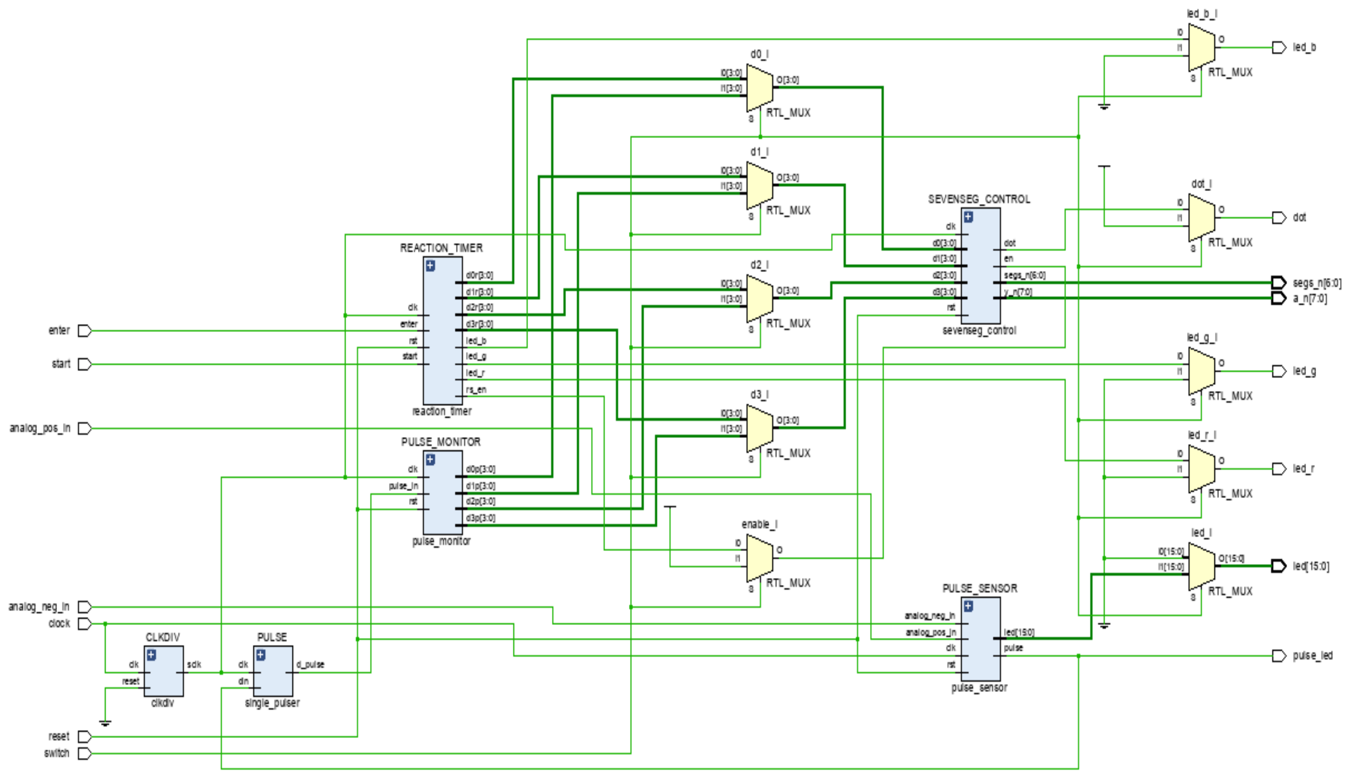
- led -display digitized pulse sensor value
- pulse – display pulse as red LED
- a_n- turn on the seven-segment display digits,
- segs_n – display value of data received
- dot – display decimal point,
- led_r,led_g,led_b, - display the colors on the reaction timer LED

a) lab10top.sv

```
//------------------------------------------------------------------------------

// Module Name   : lab10_top

// Project       : ECE 211 Digital Circuits 1

//------------------------------------------------------------------------------

// Author        : Otieno Maurice <otienom@lafayette.edu>

                       John Nestor  <nestorj@lafayette.edu>

// Created       : April 2023

//------------------------------------------------------------------------------

// Description   : Top-level module for Lab 10 - Health Monitor

//------------------------------------------------------------------------------



module lab10_top(

    input logic clock, switch,  // CLOCK 100 MHz
```

```
 input logic reset,enter, start,

 output logic led_r,led_g,led_b,

 input analog_pos_in,       // pulse sensor positive input

 input analog_neg_in,      // pulse sensor negative input

 output logic [15:0] led,  // display digitized pulse sensor value

 output logic dot, pulse_led,     // display pulse as red LED

 output logic [7:0] a_n,

 output logic [6:0] segs_n

 );

 logic enable, pulse_in;

 logic [3:0] d3p, d2p, d1p, d0p;

 logic [3:0] d3r,d2r, d1r, d0r;

 logic [3:0] d0, d1, d2, d3;

 logic [15:0] reaction_value, pulse_value;

 logic sclk, rs_en; // DIVIDED CLOCK 1KHz

 logic rst,led_r1,led_g1,led_b1,din;


 pulse_monitor PULSE_MONITOR(.clk(sclk), .pulse_in(d_pulse), .d0p, .d1p,.d2p,.d3p, .rst );


  single_pulser PULSE(.d_pulse,.clk(sclk),.din(pulse_in));


 reaction_timer REACTION_TIMER(.clk(sclk),.start, .enter,.rst,.d0r,.d1r,.d2r,

      .d3r,.led_r(led_r1),.led_b(led_b1),.led_g(led_g1),.rs_en);


clkdiv CLKDIV( .clk(clock),.sclk(sclk),.reset(1'b0));


 sevenseg_control SEVENSEG_CONTROL(.clk(sclk),.rst, .d0(d0), .d1(d1), .d2(d2),

   .d3(d3), .y_n(a_n),.segs_n,.dot, .en(enable));


 pulse_sensor PULSE_SENSOR(.clk(clock), .rst, .analog_pos_in, .analog_neg_in, .led, .pulse(pulse_in));


 // add clock divider and other modules here


 assign rst = reset;

 assign reaction_value = {d3r ,d2r ,d1r ,d0r};

 assign pulse_value = {d3p, d2p, d1p, d0p};
 assign pulse_led = pulse_in;


 //Code

 always_comb

 if (switch)

     begin

         led_r = led_r1; led_g = led_g1;l ed_b = led_b1;

         enable = rs_en;

         d0 = reaction_value [3:0] ;

         d1 = reaction_value [7:4] ;

         d2 = reaction_value [11:8] ;

         d3 = reaction_value [15:12] ;

     end


 else

     begin

         enable = 1; led_r =0; led_g = 0; led_b =0;

         d0 = pulse_value [3:0] ;

         d1 = pulse_value [7:4] ;

         d2 = pulse_value [11:8] ;

         d3 = pulse_value [15:12] ;

     end
```

*FIGURE 2 lab10top schematic diagram*



## THE PULSE SENSOR

- The pulse sensor collects pulses from the finger as blood flows through that part of the finger. This information is transmitted and received by the FPGA in analog form through the JAXDC ports on the PCB.
- The FPGA has an inbuilt analog to digital converter to convert the analog signal to a digital signal pulse that can be counted.
- This pulse is then transmitted to the single pulser to produce a pulse.
- The module that implements the conversion of the signal to a pulse is the pulse_sensor module. It runs at 100 MHz.
- It receives two inputs analog_pos_ in and analog_neg_in and produces a pulse.
  **Inputs:**
  clk – clock signal 100 MHz, rst – synchronous reset signal
  analog_pos_in – pulse sensor positive input
  analog_neg_in – pulse sensor negative input
  **outputs:**
  led[15:0] – a fifteen-bit output to display the digitized pulser sensor value
  pulse – delivered to the single pulser

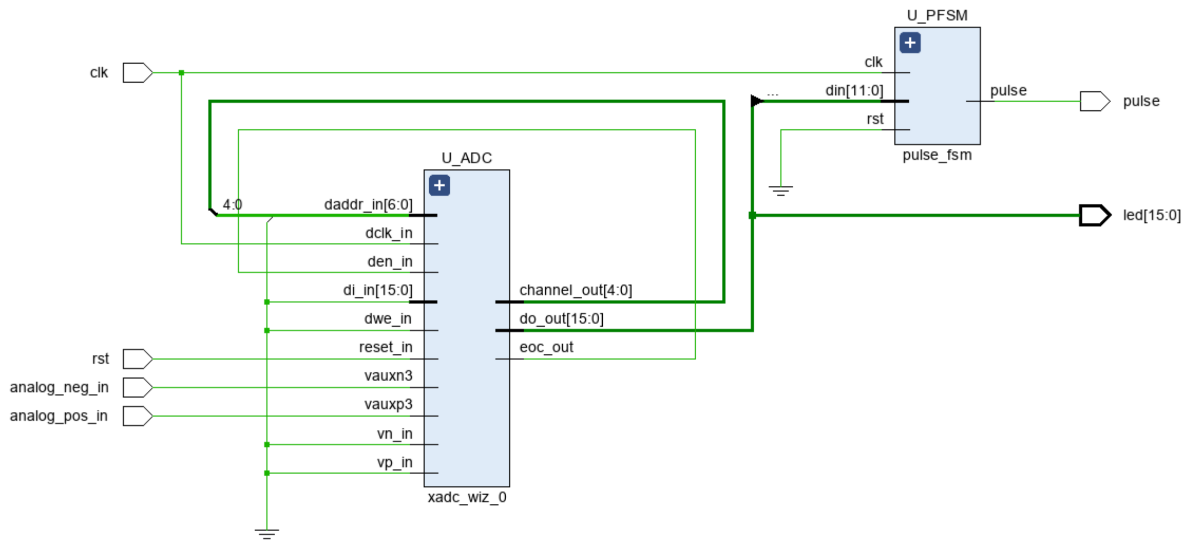
The module that implements the pulse sensor is shown.

It consist of two submodules a finite state machine that implements a simple comparator with hysteresis and analog to digital converter.

pulse_sensor.sv

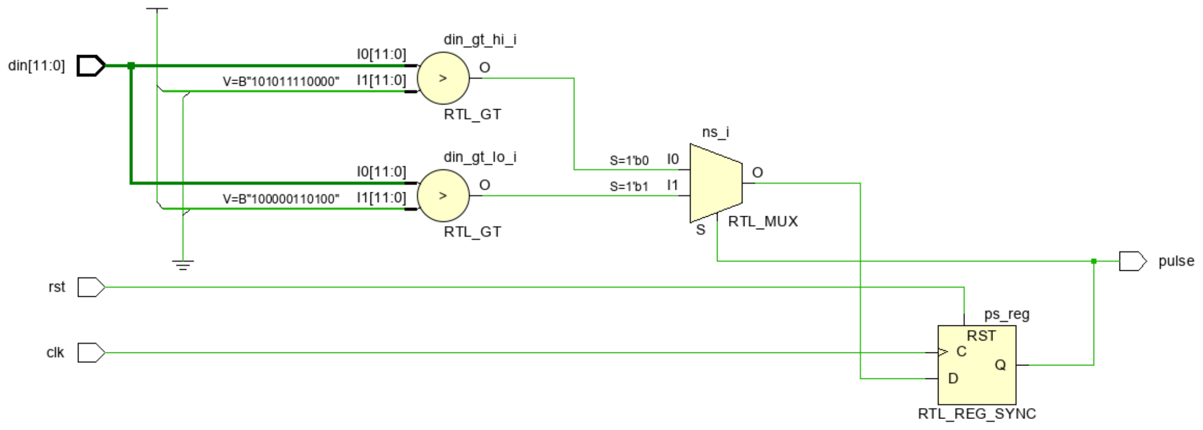
```
//-----------------------------------------------------------------------------
// Module Name   : pulse_sensor
// Project       : ECE 211 Digital Circuits 1
//-----------------------------------------------------------------------------
// Author        : John Nestor  <nestorj@lafayette.edu>
// Created       : April 2023
//-----------------------------------------------------------------------------
// Description   : Pulse sensor using the XADC A/D converter
//                 This circuit uses a simple digital comparator with hysteresis
//                 It also outputs the A/D output (digitized pulse signal) for
//                 display on LEDs
//-----------------------------------------------------------------------------

module pulse_sensor(
    input logic clk, // FAST CLOCK 100MHz
    input logic rst,
    input analog_pos_in,       // pulse sensor positive input
    input analog_neg_in,      // pulse sensor negative input
    output logic [15:0] led,  // display digitized pulse sensor value
    output logic pulse        // display pulse as red LED
    );

    logic [15:0] di_in, do_out;
    logic        eoc_out;
    logic [4:0] channel_out;
    // the following signals are not connected to anything
    logic busy_out, drdy_out, eos_out, ot_out, vccaux_alarm_out, vccint_alarm_out;
    logic user_temp_alarm_out, alarm_out;
    logic vp_in, vn_in;  // not connected analog signals
    assign vp_in = 0;
    assign vn_in = 0;

    assign led = do_out;
    assign di_in = '0;    // not actally used here

    logic [11:0] adc_data;
    assign adc_data = do_out[15:4];

    pulse_fsm U_PFSM( .clk, .rst(1'b0), .din(adc_data), .pulse );

    xadc_wiz_0 U_ADC
          (
          .daddr_in(channel_out),    // Address bus for the dynamic reconfiguration port
          .dclk_in(clk),             // Clock input for the dynamic reconfiguration port
          .den_in(eoc_out),          // Enable Signal for the dynamic reconfiguration port
          .di_in,                    // Input data bus for the dynamic reconfiguration port
          .dwe_in(1'b0),             // Write Enable for the dynamic reconfiguration port
          .reset_in(rst),            // Reset signal for the System Monitor control logic
          .vauxp3(analog_pos_in),    // Auxiliary channel 3 - connected to JXADC pins 6, 12
          .vauxn3(analog_neg_in),
          .busy_out,                 // ADC Busy signal
          .channel_out,              // Channel Selection Outputs
          .do_out,                   // Output data bus for dynamic reconfiguration port
          .drdy_out,                 // Data ready signal for the dynamic reconfiguration port
          .eoc_out,                  // End of Conversion Signal
          .eos_out,                  // End of Sequence Signal
          .ot_out,                   // Over-Temperature alarm output
          .vccaux_alarm_out,         // VCCAUX-sensor alarm output
          .vccint_alarm_out,         //  VCCINT-sensor alarm output
          .user_temp_alarm_out,      // Temperature-sensor alarm output
          .alarm_out,                // OR'ed output of all the Alarms
          .vp_in,                    // Dedicated Analog Input Pair
          .vn_in
          );

Endmodule
```

*Figure 3 pulse_sensor schematic diagram*


U_PFSM
clk
din[11:0]
rst
pulse
pulse_fsm
pulse
led[15:0]
clk
U_ADC
4:0
daddr_in[6:0]
dclk_in
den_in
di_in[15:0]
dwe_in
reset_in
vauxn3
vauxp3
vn_in
vp_in
channel_out[4:0]
do_out[15:0]
eoc_out
xadc_wiz_0
rst
analog_neg_in
analog_pos_in

```
//-----------------------------------------------------------------------------
// Module Name   : pulse_fsm
// Project       : ECE 211 Digital Circuits 1
//-----------------------------------------------------------------------------
// Author        : John Nestor  <nestorj@lafayette.edu>
// Created       : April 2023
//-----------------------------------------------------------------------------
// Description   : State machine that implements a simple comparator with hysteresis
//                 Threshold parameters THRESH_HI and THRESH_LO can be overriden
//                 on instantation to adjust behavior
//-----------------------------------------------------------------------------


module pulse_fsm(
    input logic clk,
    input logic rst,
    input logic [11:0] din,
    output logic pulse
    );

    parameter THRESH_HI = 12'd2800;
    parameter THRESH_LO = 12'd2100;

    logic din_gt_hi, din_gt_lo;

    // comparators
    assign din_gt_hi = (din > THRESH_HI);
    assign din_gt_lo = (din > THRESH_LO);

    typedef enum logic { LO, HI } states_t;

    states_t ps, ns;

    always_ff @(posedge clk)
        if (rst) ps <= LO;
        else ps <= ns;

    always_comb begin
       ns = LO;
       pulse = 0;
       case (ps)
            LO: begin
                pulse = 0;
                if (din_gt_hi) ns = HI;
                else ns = LO;
            end
            HI: begin
                pulse = 1;
                if (din_gt_lo) ns = HI;
                else ns = LO;
            end
        endcase
    end
endmodule
```

*Figure 4 pulse_fsm schematic diagram*

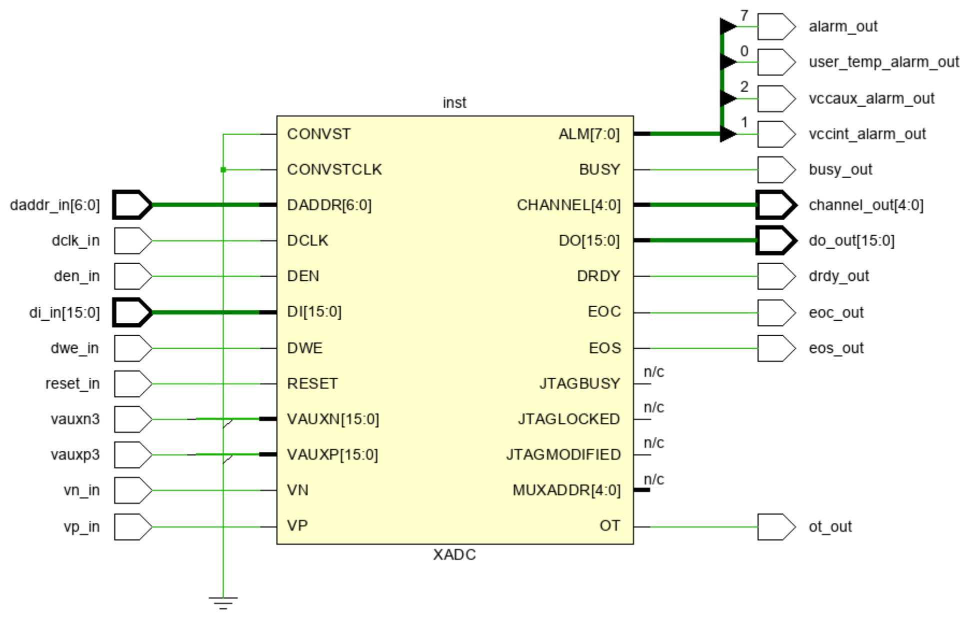


xadc_wiz_0.sv

```
// file: xadc_wiz_0.v
// (c) Copyright 2009 - 2013 Xilinx, Inc. All rights reserved.
//
// This file contains confidential and proprietary information
// of Xilinx, Inc. and is protected under U.S. and
// international copyright and other intellectual property
// laws.
//
// DISCLAIMER
// This disclaimer is not a license and does not grant any
// rights to the materials distributed herewith. Except as
// otherwise provided in a valid license issued to you by
// Xilinx, and to the maximum extent permitted by applicable
// law: (1) THESE MATERIALS ARE MADE AVAILABLE "AS IS" AND
// WITH ALL FAULTS, AND XILINX HEREBY DISCLAIMS ALL WARRANTIES
// AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING
// BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-
// INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and
// (2) Xilinx shall not be liable (whether in contract or tort,
// including negligence, or under any other theory of
// liability) for any loss or damage of any kind or nature
// related to, arising under or in connection with these
// materials, including for any direct, or any indirect,
// special, incidental, or consequential loss or damage
// (including loss of data, profits, goodwill, or any type of
// loss or damage suffered as a result of any action brought
// by a third party) even if such damage or loss was
// reasonably foreseeable or Xilinx had been advised of the
// possibility of the same.
//
// CRITICAL APPLICATIONS
// Xilinx products are not designed or intended to be fail-
// safe, or for use in any application requiring fail-safe
// performance, such as life-support or safety devices or
// systems, Class III medical devices, nuclear facilities,
// applications related to the deployment of airbags, or any
// other applications that could lead to death, personal
// injury, or severe property or environmental damage
// (individually and collectively, "Critical
// Applications"). Customer assumes the sole risk and
// liability of any use of Xilinx products in Critical
// Applications, subject only to applicable laws and
// regulations governing limitations on product liability.
//
// THIS COPYRIGHT NOTICE AND DISCLAIMER MUST BE RETAINED AS
// PART OF THIS FILE AT ALL TIMES.
`timescale 1ns / 1 ps


(* CORE_GENERATION_INFO =
"xadc_wiz_0,xadc_wiz_v3_3_8,{component_name=xadc_wiz_0,enable_axi=false,enable_axi4stream=false,dclk_frequ
ency=100,enable_busy=true,enable_convst=false,enable_convstclk=false,enable_dclk=true,enable_drp=true,enab
le_eoc=true,enable_eos=true,enable_vbram_alaram=false,enable_vccddro_alaram=false,enable_Vccint_Alaram=tru
e,enable_Vccaux_alaram=true,enable_vccpaux_alaram=false,enable_vccpint_alaram=false,ot_alaram=true,user_te
mp_alaram=true,timing_mode=continuous,channel_averaging=None,sequencer_mode=off,startup_channel_selection=
single_channel}" *)



module xadc_wiz_0

          (

          daddr_in,            // Address bus for the dynamic reconfiguration port

          dclk_in,             // Clock input for the dynamic reconfiguration port

          den_in,              // Enable Signal for the dynamic reconfiguration port

          di_in,               // Input data bus for the dynamic reconfiguration port

          dwe_in,              // Write Enable for the dynamic reconfiguration port

          reset_in,            // Reset signal for the System Monitor control logic

          vauxp3,              // Auxiliary channel 3

          vauxn3,

          busy_out,            // ADC Busy signal

          channel_out,         // Channel Selection Outputs

          do_out,              // Output data bus for dynamic reconfiguration port

          drdy_out,            // Data ready signal for the dynamic reconfiguration port

          eoc_out,             // End of Conversion Signal

          eos_out,             // End of Sequence Signal

          ot_out,              // Over-Temperature alarm output

          vccaux_alarm_out,    // VCCAUX-sensor alarm output

          vccint_alarm_out,    //  VCCINT-sensor alarm output

          user_temp_alarm_out, // Temperature-sensor alarm output

          alarm_out,           // OR'ed output of all the Alarms

          vp_in,               // Dedicated Analog Input Pair

          vn_in);


          input [6:0] daddr_in;

          input dclk_in;

          input den_in;

          input [15:0] di_in;

          input dwe_in;

          input reset_in;

          input vauxp3;

          input vauxn3;
          input vp_in;
          input vn_in;

          output busy_out;
          output [4:0] channel_out;
          output [15:0] do_out;
          output drdy_out;
          output eoc_out;
          output eos_out;
          output ot_out;
          output vccaux_alarm_out;
          output vccint_alarm_out;
          output user_temp_alarm_out;
          output alarm_out;

          wire GND_BIT;
          assign GND_BIT = 0;
          wire [15:0] aux_channel_p;
          wire [15:0] aux_channel_n;
          wire [7:0]  alm_int;
          assign alarm_out = alm_int[7];
          assign vccaux_alarm_out = alm_int[2];
          assign vccint_alarm_out = alm_int[1];
          assign user_temp_alarm_out = alm_int[0];
          assign aux_channel_p[0] = 1'b0;
          assign aux_channel_n[0] = 1'b0;

          assign aux_channel_p[1] = 1'b0;
          assign aux_channel_n[1] = 1'b0;

          assign aux_channel_p[2] = 1'b0;
          assign aux_channel_n[2] = 1'b0;

          assign aux_channel_p[3] = vauxp3;
          assign aux_channel_n[3] = vauxn3;
```

```
        assign aux_channel_p[4] = 1'b0;

        assign aux_channel_n[4] = 1'b0;


        assign aux_channel_p[5] = 1'b0;

        assign aux_channel_n[5] = 1'b0;


        assign aux_channel_p[6] = 1'b0;

        assign aux_channel_n[6] = 1'b0;


        assign aux_channel_p[7] = 1'b0;

        assign aux_channel_n[7] = 1'b0;


        assign aux_channel_p[8] = 1'b0;

        assign aux_channel_n[8] = 1'b0;


        assign aux_channel_p[9] = 1'b0;

        assign aux_channel_n[9] = 1'b0;


        assign aux_channel_p[10] = 1'b0;

        assign aux_channel_n[10] = 1'b0;


        assign aux_channel_p[11] = 1'b0;

        assign aux_channel_n[11] = 1'b0;


        assign aux_channel_p[12] = 1'b0;

        assign aux_channel_n[12] = 1'b0;


        assign aux_channel_p[13] = 1'b0;

        assign aux_channel_n[13] = 1'b0;


        assign aux_channel_p[14] = 1'b0;

        assign aux_channel_n[14] = 1'b0;


        assign aux_channel_p[15] = 1'b0;

        assign aux_channel_n[15] = 1'b0;

XADC #(
        .INIT_40(16'h0013), // config reg 0

        .INIT_41(16'h31A0), // config reg 1

        .INIT_42(16'h0400), // config reg 2

        .INIT_48(16'h0100), // Sequencer channel selection

        .INIT_49(16'h0000), // Sequencer channel selection

        .INIT_4A(16'h0000), // Sequencer Average selection

        .INIT_4B(16'h0000), // Sequencer Average selection

        .INIT_4C(16'h0000), // Sequencer Bipolar selection

        .INIT_4D(16'h0000), // Sequencer Bipolar selection

        .INIT_4E(16'h0000), // Sequencer Acq time selection

        .INIT_4F(16'h0000), // Sequencer Acq time selection

        .INIT_50(16'hB5ED), // Temp alarm trigger

        .INIT_51(16'h57E4), // Vccint upper alarm limit

        .INIT_52(16'hA147), // Vccaux upper alarm limit

        .INIT_53(16'hCA33),  // Temp alarm OT upper

        .INIT_54(16'hA93A), // Temp alarm reset

        .INIT_55(16'h52C6), // Vccint lower alarm limit

        .INIT_56(16'h9555), // Vccaux lower alarm limit

        .INIT_57(16'hAE4E),  // Temp alarm OT reset

        .INIT_58(16'h5999), // VCCBRAM upper alarm limit

        .INIT_5C(16'h5111),  //  VCCBRAM lower alarm limit

        .SIM_DEVICE("7SERIES"),

        .SIM_MONITOR_FILE("design.txt")

)


inst (

        .CONVST(GND_BIT),

        .CONVSTCLK(GND_BIT),

        .DADDR(daddr_in[6:0]),

        .DCLK(dclk_in),

        .DEN(den_in),

        .DI(di_in[15:0]),

        .DWE(dwe_in),

        .RESET(reset_in),

        .VAUXN(aux_channel_n[15:0]),

        .VAUXP(aux_channel_p[15:0]),
        .ALM(alm_int),

        .BUSY(busy_out),

        .CHANNEL(channel_out[4:0]),

        .DO(do_out[15:0]),

        .DRDY(drdy_out),

        .EOC(eoc_out),

        .EOS(eos_out),

        .JTAGBUSY(),

        .JTAGLOCKED(),

        .JTAGMODIFIED(),

        .OT(ot_out),

        .MUXADDR(),

        .VP(vp_in),

        .VN(vn_in)

          );

Endmodule
```

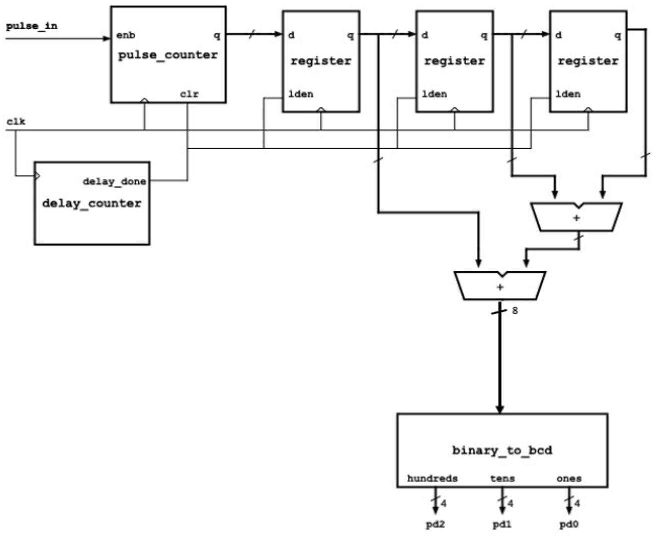
Figure 5 Xadc_wiz_0 schematic diagram



## THE PULSE MONITOR

The Figure below shows the high-level diagram for the pulse monitor.

*Figure 6 high level diagram for the health monitor*



- The pulse monitor receives a single pulse from the single pulser and counts the number of pulses in five second intervals, stores three successive counts, averages the pulses, changes the amount to beats per minute and converts the number to binary coded decimal.
- It operates at 1kHz
- It consists of a pulse counter to count the pulses, a series of shift registers to store the counted pulse, a delay counter to clear the pulse counter and provide signal to shift the value of the registers, three multibit adders and a binary to bcd converter.

Functionality

- The module that implements the pulse monitor is pulse_monitor.
- It consists of two submodules named pulse_registers and binary_to_bcd.

- pulse_registers - the module that computes the binary value of the pulse
- binary_to_bcd converts the binary value from pulse_registers to BCD that is viewable on the seven-segment display

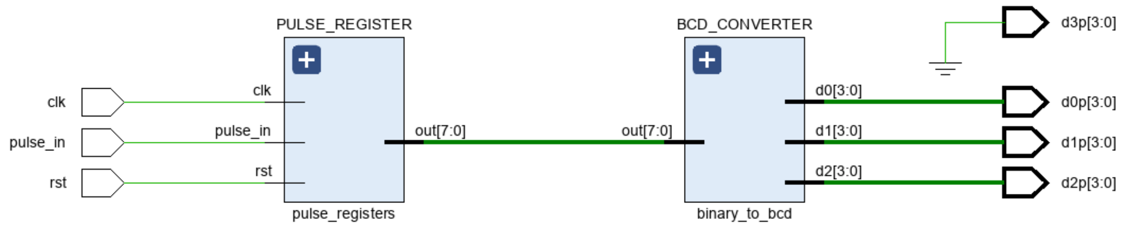
  **inputs:**
  clk – delivers the clock signal from clock divider
  pulse_in – delivers the pulse from the single pulser
  rst – synchronous reset signal
  **outputs:**
  dop, d1p, d2p, d3p – 4-bit data values from the bcd converter

it is important to know that the fourth value d3p is always active low since the maximum number of pulses is 255.
The code for the design of the pulse monitor is shown below

pulse_monitor.sv

```
`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

// Company:

// Engineer:

//

// Create Date: 05/04/2023 11:26:40 AM

// Design Name: PULSE_MONITOR

// Module Name: pulse_monitor

// Project Name:

// Target Devices: NEXYSA7 100T

module pulse_monitor(

input logic clk, pulse_in, rst,

output logic [3:0] d0p, d1p, d2p,d3p    // d3p is a 0 bit output

);

logic [7:0] out;

   binary_to_bcd BCD_CONVERTER(.d0(d0p), .d1(d1p), .d2(d2p), .out);

   pulse_registers PULSE_REGISTER(.clk, .pulse_in, .out, .rst);

assign d3p = 4'b0;

endmodule
```

*FIGURE 7 pulse_monitor schematic diagram*



1. Pulse Register

This is the first component of the pulse monitor

Functionality

- consists of five submodules an adder and a 2-bit shifter
- The submodules include:
  - pulse_counter: counts pulses
  - pulse_reg: stores the most immediate value from pulse_counter
  - pulse_reg_2: stores the value from pulse_reg_1
  - pulse_reg_3: stores the value from pulse_reg_2
  - pulse_delay_counter – clears resets pulse_counter, signals the shift of values from one register to the other
- The values from the three registers are added together by adders and then shifted by 2 which is the equivalent of multiplying by 4. This converts the beats from BPS to BPM

  **inputs:** clk – clock signal

  pulse_in – pulse from single pulser

  rst – synchronous reset signal

  **outputs:**

  out – 8-bit binary value

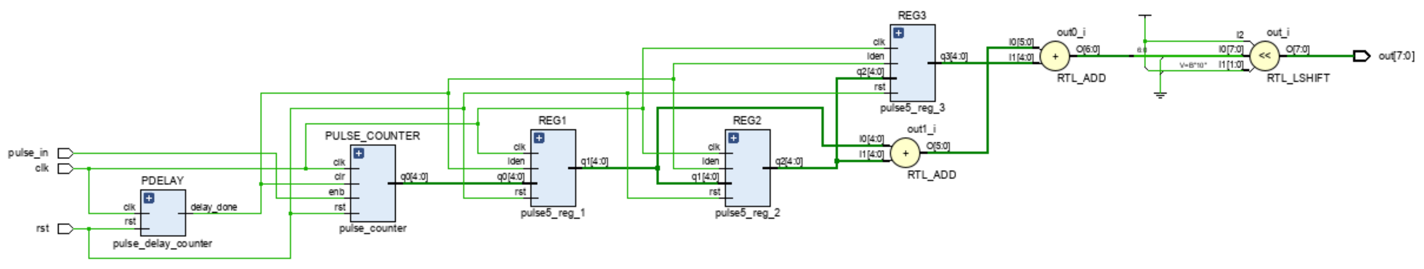
- The code that implements the pulse register is given below

pulse_registers.sv

```
`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////
```

```
// Company: Lafayette college

// Engineer: OTIENO mAURICE

//

// Create Date: 05/04/2023 10:52:36 AM

// Design Name:

// Module Name: pulse_registers

// Project Name:

// Target Devices:


module pulse_registers(


input logic clk, pulse_in, rst,

output logic [7:0] out


    );

         logic lden;

         logic [4:0] q0, q1, q2, q3;

         logic [4:0]d0, d1, d2;


          assign clr = lden;


  pulse_counter PULSE_COUNTER(.clk, .enb(pulse_in), .clr, .q0, .rst); /*  counts the number of pulses
every 5s */

  pulse5_reg_1 REG1(.clk, .lden, .q0, .q1, .rst);

  pulse5_reg_2 REG2(.clk, .lden, .q1, .q2, .rst);

  pulse5_reg_3 REG3(.clk, .lden, .q2, .q3, .rst);

  pulse_delay_counter PDELAY(.clk, .delay_done(lden), .rst);


        assign d0 = q1;

        assign d1 = q2;

        assign d2 = q3;


        assign out = (d0 + d1 + d2) << 2;

/* the pulses received is in beats per second so we shift the value of the pulse by 4 to give the average
in beats per minute */

endmodule
```

*Figure 8 pulse_registers schematic diagram*



a. The Pulse Counter

- Receives pulse from the single pulser
- Counts the number of pulses received in five seconds
- Resets every five seconds
- Feeds its output to the first register

  **Inputs:**

  clk- clock signal

  rst – synchronous reset signal

  enb: enable signal; when asserted, it begins counting. The pulse from the single pulser is the enable signal. Every time there is a pulse, the value of q0 increments by 1

  **outputs:**

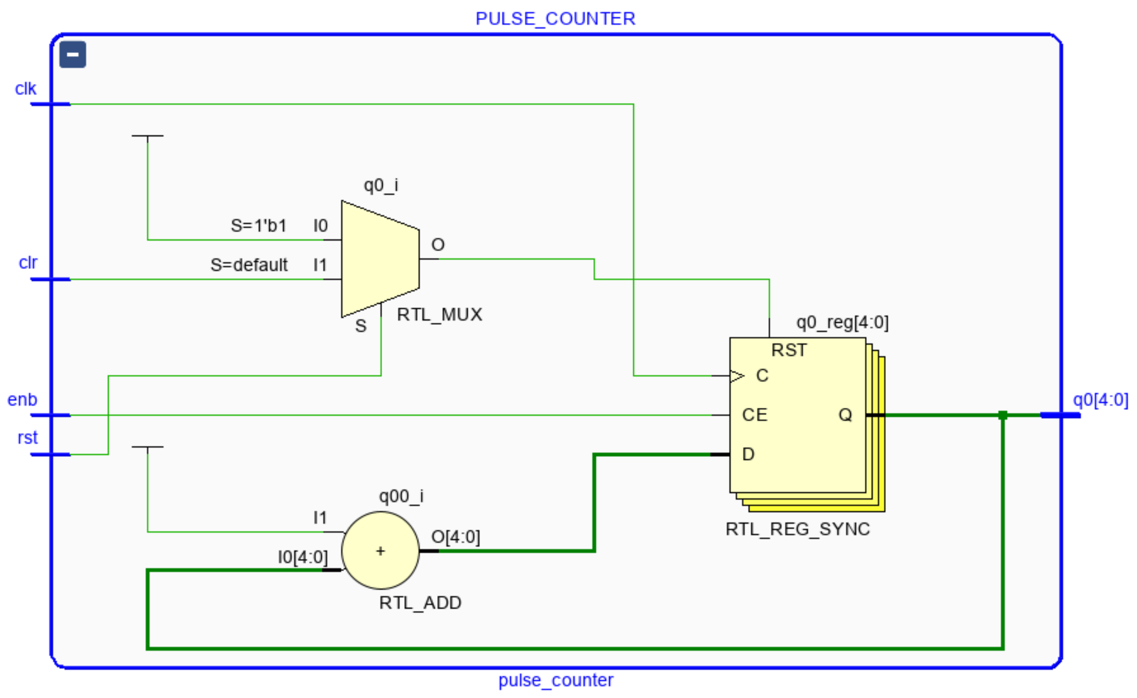
  q0 – total number of pulses in five seconds
- The code for the design of the pulse_counter is shown

pulse_counter.sv

```
`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

// Company: Lafayette College

// Engineer: Otieno Maurice

//

// Create Date: 05/04/2023 09:34:28 AM

// Design Name: PULSE_COUNTER

// Module Name: pulse5_reg

// Project Name: Health Monitor

// Target Devices:  NEXYSA7 100T

// Tool Versions:

// Description:

//  THIS COUNTER RECEIVES INPUT PULSE INPUT FROM THE SINGLE PULSER AND COUNTS THE NUMBER OF PULSES IN A 5
SECOND INTERVAL
```

```
// Revision 0.01 - File Created

// Additional Comments:


module pulse_counter(


  input logic clk, enb, clr, rst,

  output logic [4:0] q0


    );

      always_ff @(posedge clk)


         if (rst) q0 <= '0;


         else if (clr) q0 <= '0;


         else if(enb) q0 <= q0 + 1;


endmodule
```

*Figure 9 pulse_counter schematic diagram*



## a. REGISTER 1

The registers just shift values to store them as a function of time

**Inputs:**

clk – clock signal, rst – synchronous reset

q0 – 4-bit input value from pulse_counter

**Outputs:**

lden – update signal; connected to delay_done of pulse_delay_counter. q1 updates every time this signal is asserted

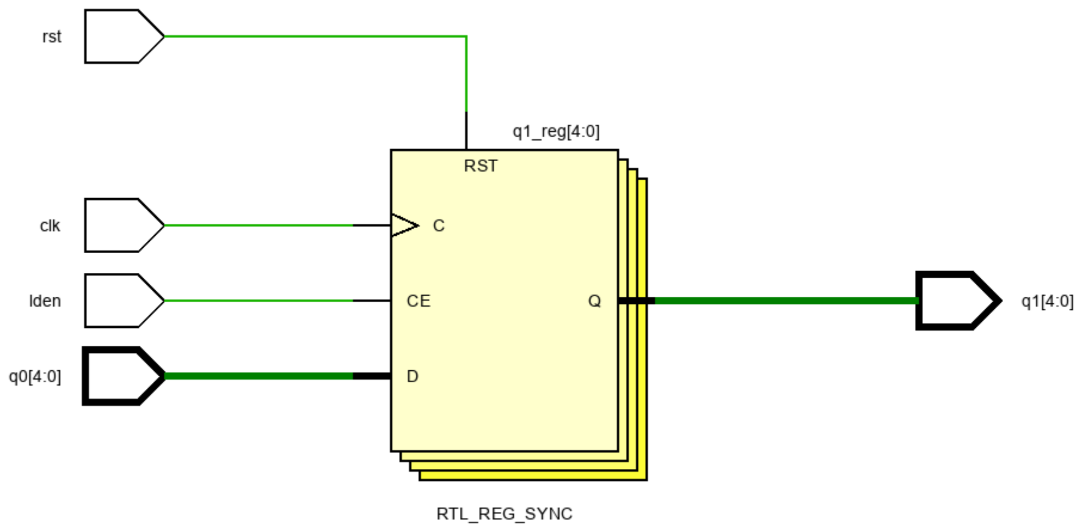
q1 – 4-bit updated output

- The design of the other two registers follows pretty much the same model

pulse5_reg_1.sv

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: Lafayette college
// Engineer: Otieno Maurice
//
// Create Date: 05/04/2023 09:34:28 AM
// Design Name: pulse register
// Module Name: pulse5_reg_2
//
//////////////////////////////////////////////////////////////////////////////////

`timescale 1ns / 1ps

module pulse5_reg_1(
   input logic clk, lden, rst,
   input logic [4:0] q0,
    output logic [4:0] q1
  );

      always_ff @(posedge clk)
         if (rst) q1 <= '0;
         else if (lden) q1 <= q0;
endmodule
```

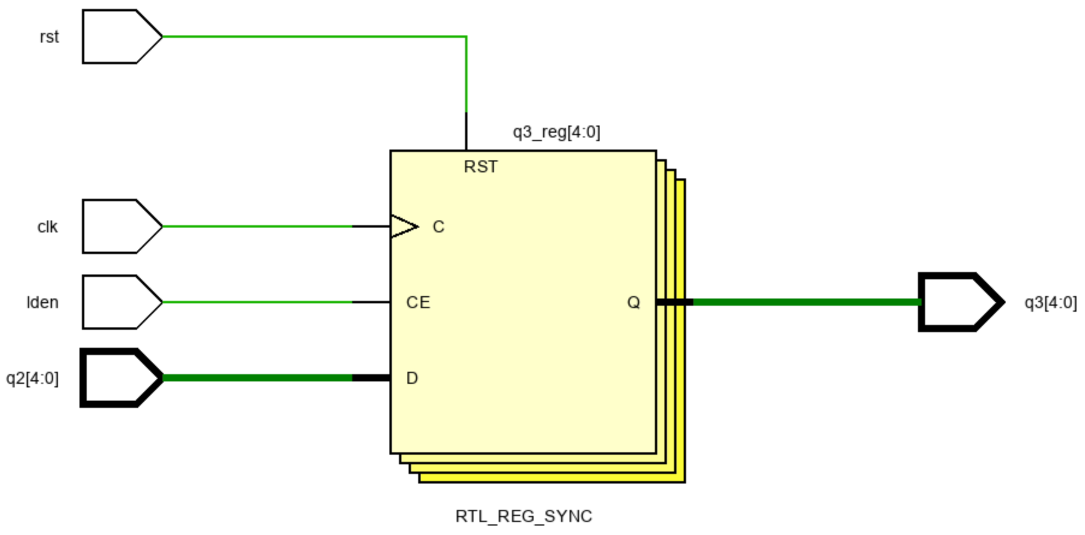
*Figure 10 register 1 schematic*



**b. REGISTER 2**

```
module pulse5_reg_2(


input logic clk, lden, rst,

input logic [4:0] q1,

output logic [4:0] q2


    );


    always_ff @(posedge clk)

    if (rst) q2 <= '0;


    else if (lden) q2 <= q1;

endmodule
```

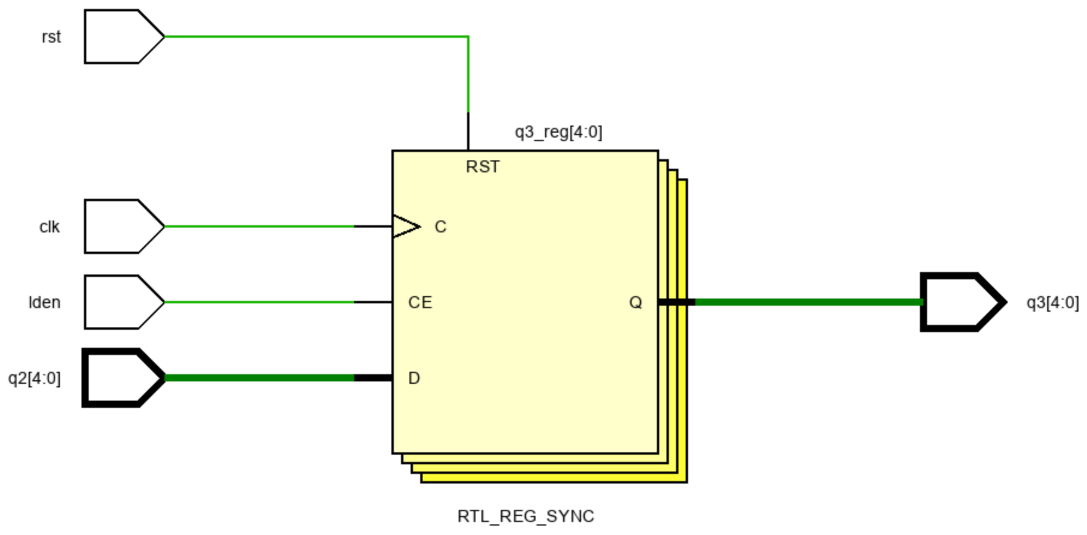
*Figure 11 register 2 schematic diagram*



## c. REGISTER 3

```
 `timescale 1ns / 1ps
module pulse5_reg_3(


  input logic clk, lden, rst,
  input logic [4:0] q2,
  output logic [4:0] q3
);
      always_ff @(posedge clk)
           if (rst) q3 <= '0;
           else if (lden) q3 <= q2;
endmodule
```

FIGURE 12 register 3 schematic diagram



d. The Pulse Delay Counter

Functionality

**Inputs:**

clk – clock signal

rst – synchronous reset

**Outputs:**

delay_done – the clear signal – connects to all the registers and the pulse counter

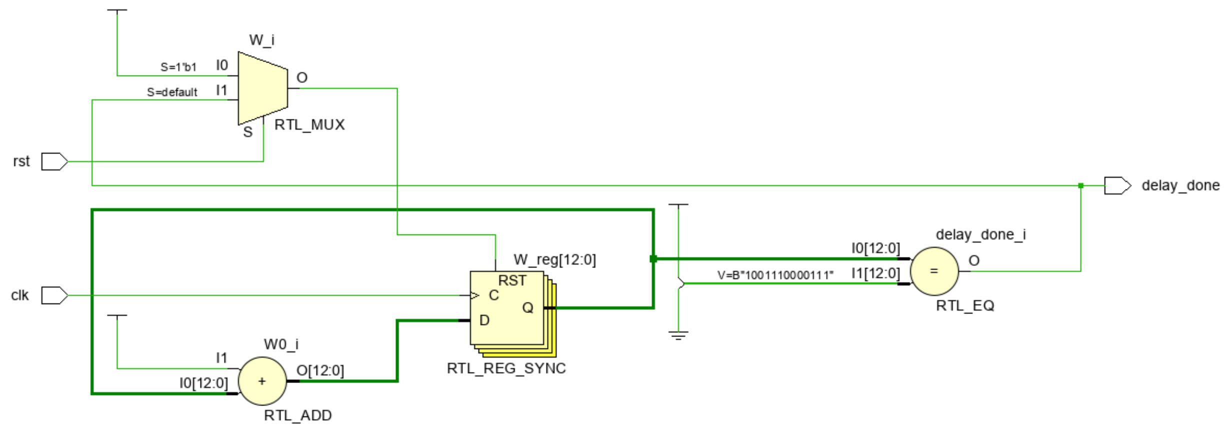
- The delay counter is responsible for the signal that resets the pulse counter and the signal that determines the shift in the registers.
- it consists of a 12- bit internal variable w that increments every clock cycle until the value of w is equal to 5000 (corresponds to 5 seconds). When this is done, the counter asserts the delay_done signal and resets.

pulse_delay_counter.sv

```
`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

// Company:

// Engineer:
```

```
//
// Create Date: 04/27/2023 10:13:19 AM
// Design Name: PDELAY
// Module Name: delay_counter5
// Project Name: HEALTH MONITOR
// Target Devices: NEXYSA7 100T
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module pulse_delay_counter(
input logic clk, rst,
output  delay_done
);
 // Internal Wire that contains the hz counting
logic [12:0] W;

always_ff @(posedge clk)
    if (rst) W <= '0;
```

*Figure 13 pulse_delay_counter schematic diagram*



2. The Binary-to-bcd Converter

- This is responsible for converting the binary value of the pulses to binary coded decimal Functionality
- The module that implements the conversion is known as binary_to_bcd
- It has several instantiations of the add3 module that checks if a binary number is greater than 5 and adds three before doing a 1-bit left shift.

**Inputs:**

out – the data value to be converted to bcd

**Outputs:**

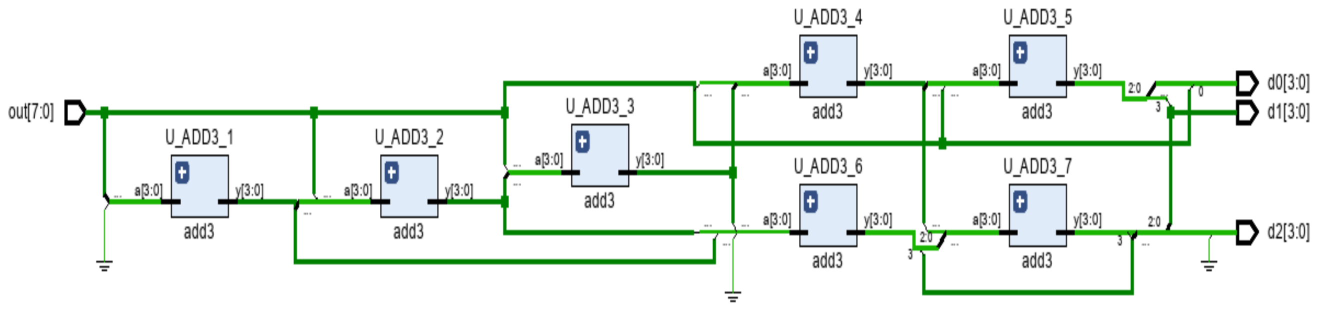
D2, d1, d0 – 4-bit binary coded data ready to be displayed on the seven-segment display hundreds, tens, ones

binary_to_bcd.sv code

```
// Company: Lafayette College

// Engineer: Otieno Maurice

//

// Create Date: 03/02/2023 10:06:29 AM

// Design Name:

// Module Name: binary_to_bcd

// Project Name: HEALTH MONITOR

// Target Devices: NEXYSA7 100T

// Tool Versions:

// Description:

//

// Dependencies:

//
```

```
// Revision:

// Revision 0.01 - File Created

// Additional Comments:

-----------------------------------------------------------------------------------------------------------
------// THIS IS JUST CONVERTING BINARY TO Binary Coded Decimal

//

//////////////////////////////////////////////////////////////////////////////////

module binary_to_bcd (

  input logic [7:0] out,

  output logic [3:0] d2,

  output logic [3:0] d1,

  output logic [3:0] d0

);


   logic [7:0]b;

   assign b = out;

   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 d2 = {1'b0,1'b0, y6[3],y7[3]};

        assign d1 = {y7[2:0], y5[3]};
        assign d0 = {y5[2:0], b[0]};
endmodule
```

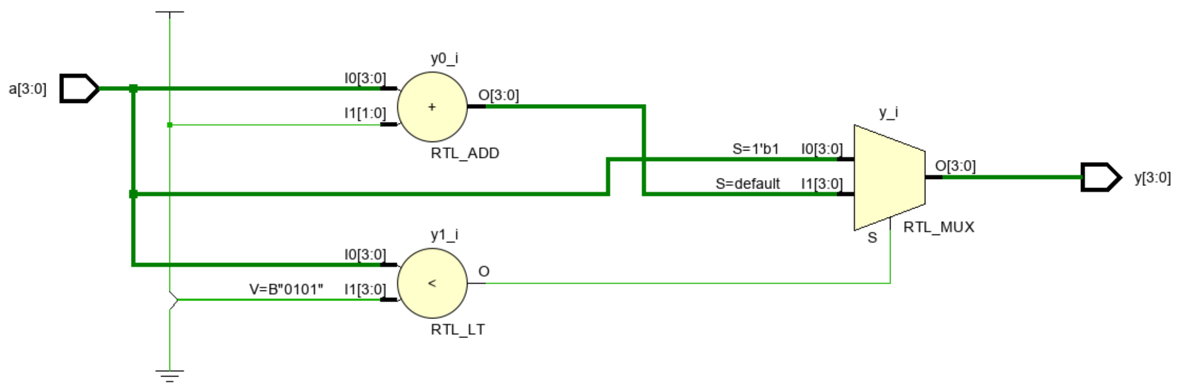
*Figure 14 binary_to_bcd schematic diagram*



add3.sv

```
module add3(
   input logic [3:0] a,
   output logic [3:0] y);
        always_comb
          begin
        if (a<5) y = a;
          else y = a+3;
    end
endmodule
```

*FIGURE 15 add3 schematic diagram*



## THE SINGLE PULSER

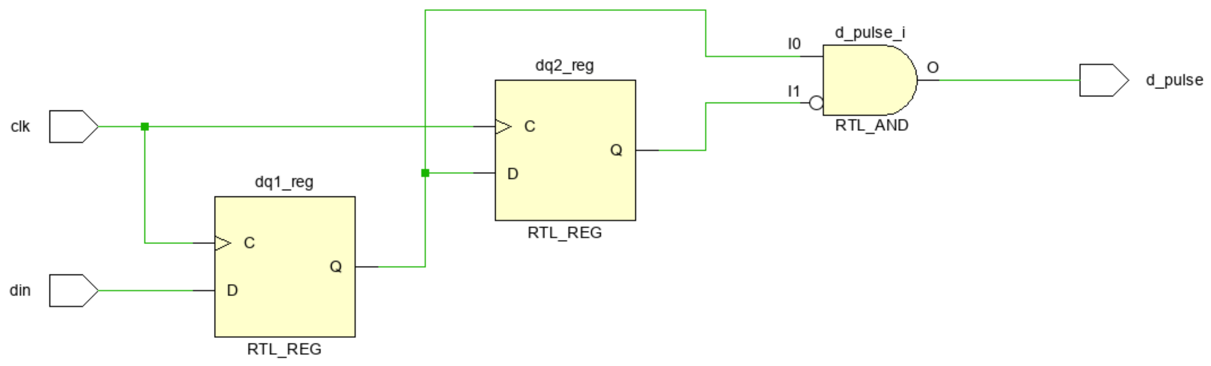
2) The pulse sensor runs on a very fast 100 MHz clock and therefore detects a lot of signals in a short period of time. The single pulser provides only a single pulse at 1 kHz that corresponds to the heartbeat we desire to monitor.
3) The single pulser accepts a series of pulse signals from the pulser monitor through its input **din** and outputs one pulse through its output **d_pulse**.
4) The single pulser is essentially a shift register with an and gate. Every clock signal the input of the first register is transferred to the second register and the current inputs of both registers travel though an and gate with an inverted input to deliver a single pulse.
5) This design is done by the module single_pulser in System Verilog as shown.

Single_pulser.sv

```
//-----------------------------------------------------------------------------

// Title         : single_pulser - detects a rising edge and outputs a single pulse

// Project       : ECE 491 - Senior Design I

//-----------------------------------------------------------------------------

// File          : single_pulser.v

// Author        : John Nestor

// Created       : 02.09.2009

// Last modified : 02.09.2009

//-----------------------------------------------------------------------------

// Description :

// This circuit detects a rising edge on the input din.  WHen the rising edge occurs,

// it outputs a single pulse one clock period in length.  It is based on the

// single pulser circuit described in Prosser & Winkel's book "The Art of Digital Design

//-----------------------------------------------------------------------------
```

```
// Modification history :

// 02.09.2009 : created

//-----------------------------------------------------------------------------


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

   logic dq1, dq2;


   always_ff @(posedge clk)

     begin

           dq1 <= din;

           dq2 <= dq1;

     end


   assign d_pulse = dq1 & ~dq2;

endmodule // single_pulser
```

*FIGURE 16 Single_pulser schematic diagram*



## THE CLOCK DIVIDER

**6)** The clock generator on the NEXYSA7 100T printed circuit board delivers a signal at 100 MHz. This signal is high enough to use for the pulse sensor but too high to be used by the pulse monitor.

**7)** The module clkdiv divides the 100MHz clock to 1 kHz for use by all components of the pulse monitor, the seven-segment display and the reaction timer.

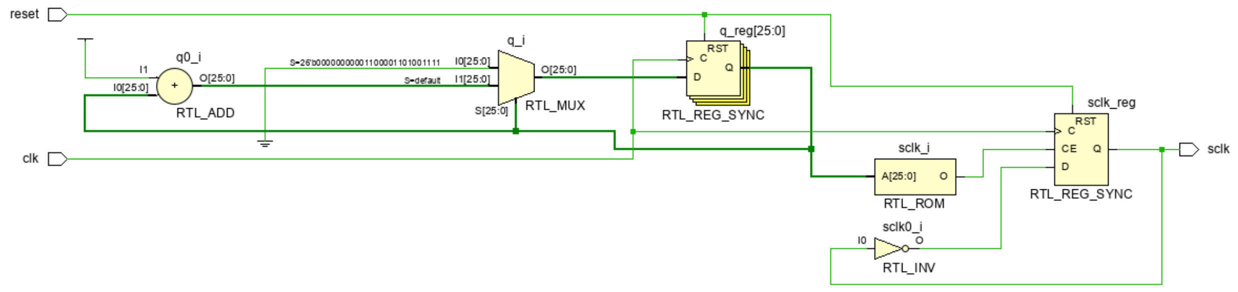
**8)** It receives an input **clock** from the PCB clock generator and outputs a slower clock signal **sclk.** The clock divider is essentially a counter which outputs a clock signal once every counting sequence is done.

9) It has a **reset** input here that is always active low

clkdiv.sv

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
//-----------------------------------------------------------------------------
// Title         : clkdiv - clock divider
// Project       : ECE 211 - Digital Circuits 1
//-----------------------------------------------------------------------------
// File          : clkdiv.sv
// Author        : John Nestor
// Created       : 09.08.2011
// Last modified : 10.16.2015
//-----------------------------------------------------------------------------
// Description :
// This module divides the 100MHz clock on the Nexys4 board down to a lower
// frequency.  Set the DIVFREQ parameter to the desired frequency in Hz.
// If a frequency lower than 1 Hz is desired, the DIVBITS bitwidth parameter
// must be increased.  If a higher frequency is used, we assume that synthesis
// will trim the unused most signficant counter bits and logic.
// To use, instantiate this module whiel setting the DIVFREQ parameter to the
//  desired frequency in Hz.
// For example, to generate a 1 Hz clock, instantiate a module as follows:
//
//  clkdiv #(.DIVFREQ(1)) U_DIV (clk, reset, sclk);
//
// Where:
//    clk    is the 100MHz system clock that arrives on an input pin of the FPGA
//           (see the documentation)
//    reset  is a signal that resets the clock divider counter
//           (connect it to zero if unused)
//    sclk   is the output clock - connect this to your logic
//
//-----------------------------------------------------------------------------

module clkdiv(input logic clk, input logic reset, output logic sclk);
   parameter DIVFREQ = 1000;  // 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
```

*FIGURE 17 clkdiv schematic diagram*



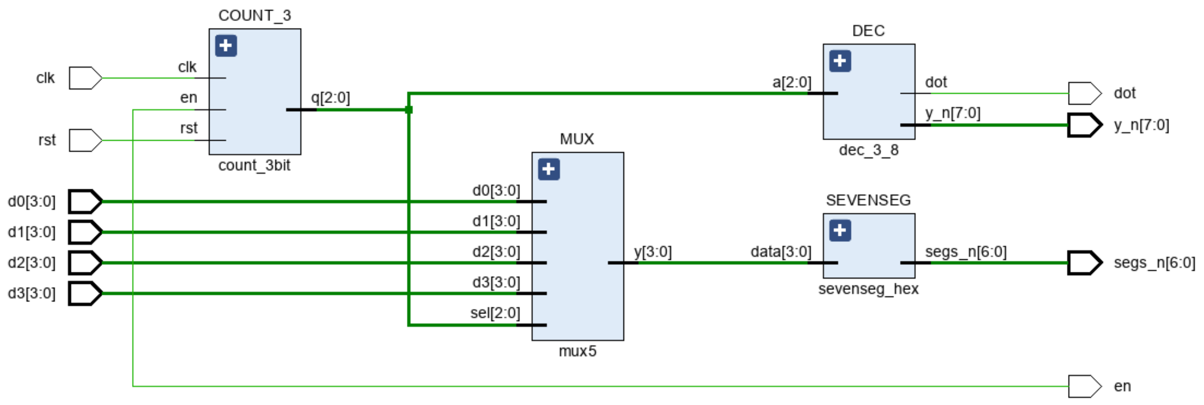
# The seven segment controller

sevenseg_control.sv

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 03/23/2023 09:02:46 AM
// Design Name:
// Module Name: sevenseg_control
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module sevenseg_control(
    input logic clk, rst,
    input logic [3:0] d0, d1, d2, d3,
    output logic dot,en,  // dot for the decimal point
                               // en for the enable input
         output logic [7:0] y_n,
         output logic [6:0] segs_n);
         logic [2:0] sel;
         logic [3:0] y;
            count_3bit COUNT_3(.clk,.rst,.q(sel),.en);
            dec_3_5 DEC(.a(sel), .y_n, .dot);
            mux8 MUX(.d0,.d1,.d2,.d3,.sel, .y);
            sevenseg_hex SEVENSEG(.data(y),.segs_n);
```

```
 ////////////////
Endmodule
```

*FIGURE 18 sevenseg_control schematic diagram*

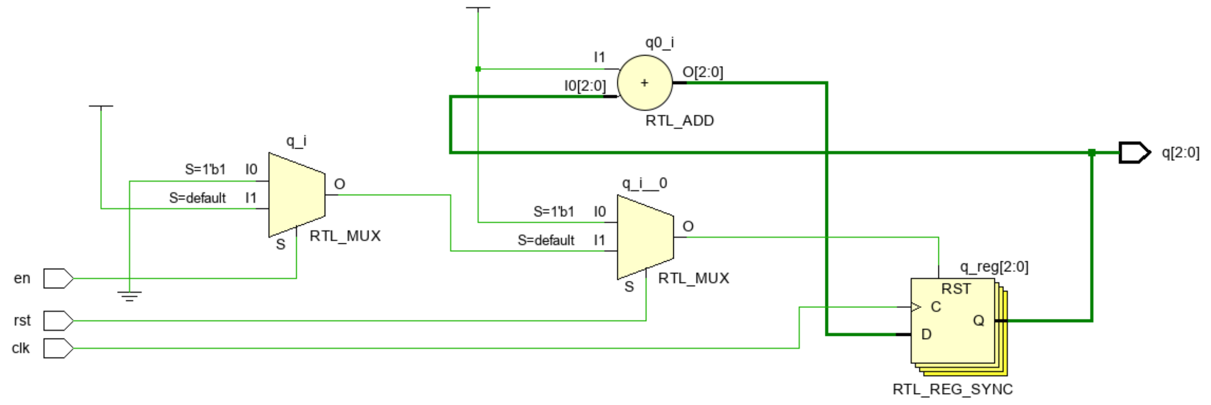


## The 3-bit counter

Count_3bit.sv

```
module count_3bit (

   input logic clk, rst, en,

   output logic [2:0] q

 );


        always_ff @(posedge clk)

           if (rst) q <= 3'd0;

           else if (en) q <= q + 3'd1;

           else q <= 0;

endmodule // count
```

*FIGURE 19 count_3bit schematic diagram*

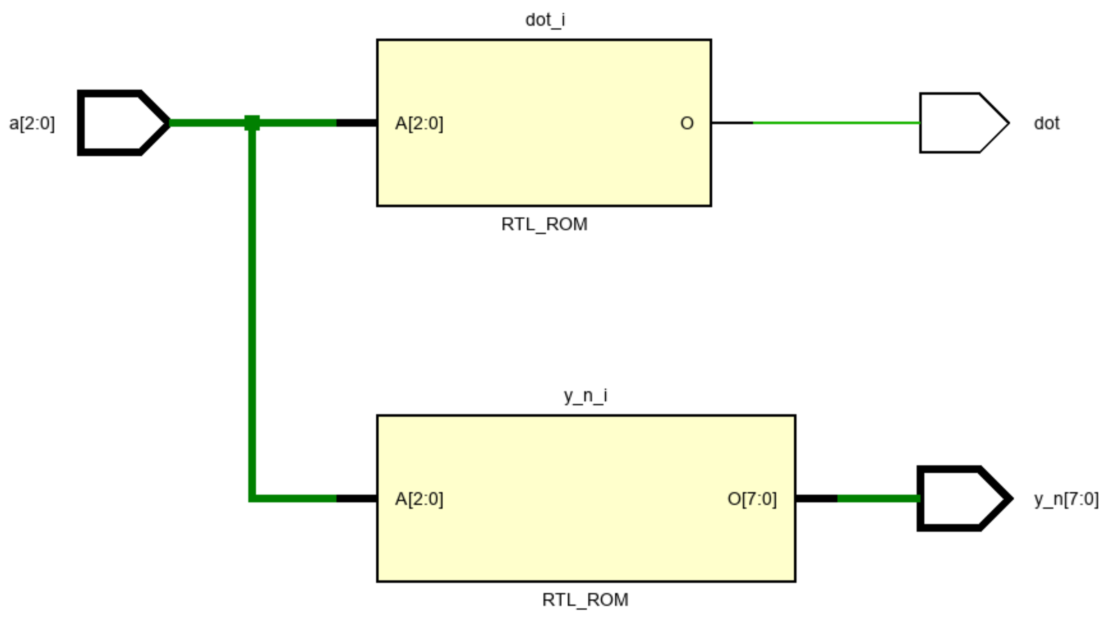


## The Three Input Five Output Decoder

dec_3_ 5.sv

```
`timescale 1ns / 1ps

module dec_3_5(

   input logic [2:0] a,

   output logic dot,

   output logic [7:0] y_n);

        always_comb

          unique case (a)

                3'd0: begin y_n = 8'b11111111; dot = 1'b1; end

                3'd1: begin y_n = 8'b11111110; dot = 1'b1; end

                3'd2: begin y_n = 8'b11111101; dot = 1'b1; end

                3'd3: begin y_n = 8'b11111011; dot = 1'b1; end

                3'd4: begin y_n = 8'b11110111; dot = 1'b0; end

                default: begin y_n = 8'b11111111; dot = 1'b1; end

            endcase

endmodule
```

*FIGURE 20 dec_3_8 schematic diagram*



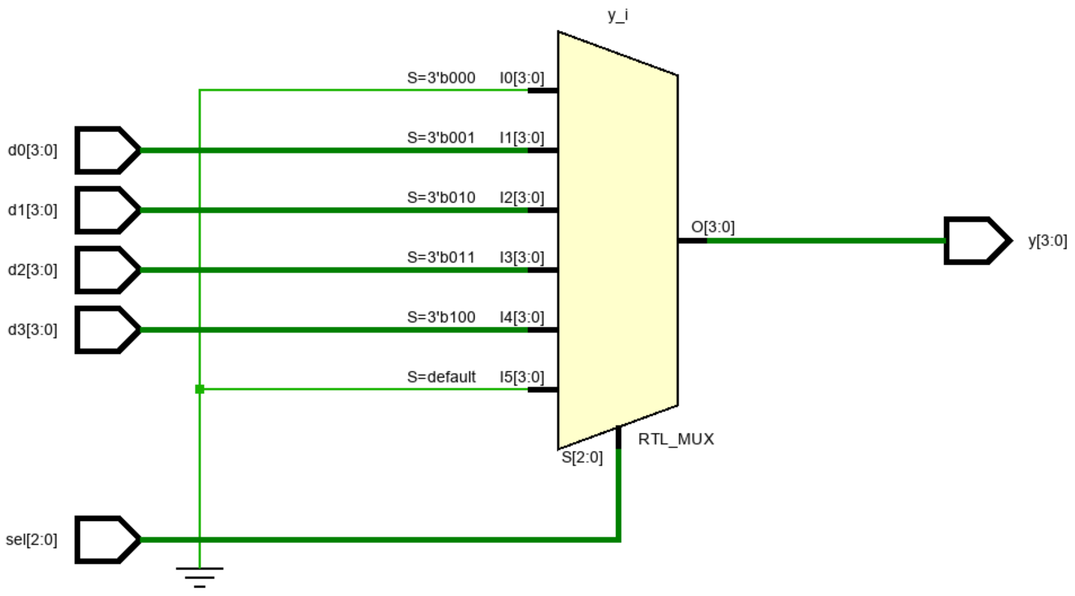
## The 5-1 Multiplexer

mux5.sv

```
`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

//-----------------------------------------------------------------------------

// Module Name : mux8

// Project : RTL Hardware Design and Verification using SystemVerilog

//-----------------------------------------------------------------------------

// Author :Otieno Maurice <otienom@lafayette.edu>

// Created : Feb 2020

//-----------------------------------------------------------------------------

// Description : 8-1 mux parameterized by bitwidthw W

//-----------------------------------------------------------------------------

module mux5(

  input logic [3:0] d0, d1, d2, d3,

  input logic [2:0] sel,

  output logic [3:0] y);
```

```
        always_comb
          unique case (sel)
                 3'd0 : y = 1'b0;
                 3'd1 : y = d0;
                 3'd2 : y = d1;
                3'd3 : y = d2;
                 3'd4 : y = d3;
                default : y = '0;
           endcase
endmodule: mux5
```

*FIGURE 21 mux5 schematic diagram*

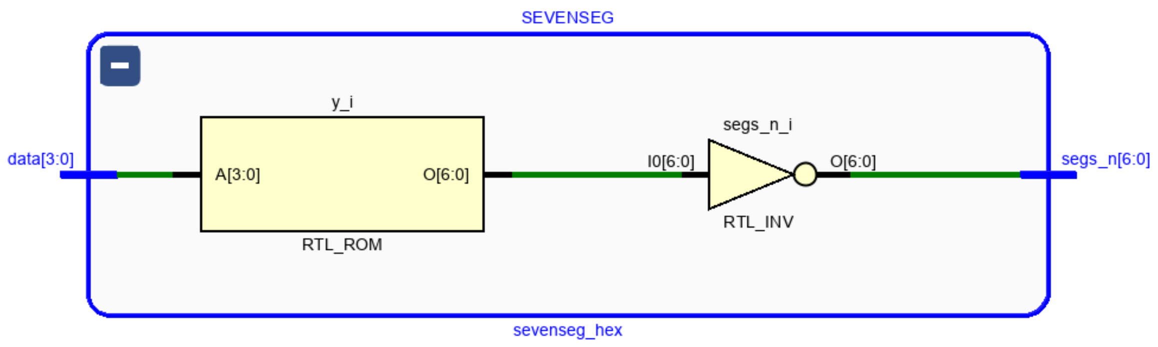


## The Seven-Segment Decorder

sevenseg_hex.sv

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
module sevenseg_hex(
input logic [3:0] data,
output logic [6:0] segs_n
 );
logic [7:0] y;
always_comb
 case(data)
 4'd0: y = 7'b111_1110;
 4'd1: y = 7'b011_0000;
 4'd2: y = 7'b110_1101;
 4'd3: y = 7'b111_1001;
 4'd4: y = 7'b011_0011;
 4'd5: y = 7'b101_1011;
 4'd6: y = 7'b101_1111;
 4'd7: y = 7'b111_0000;
 4'd8: y = 7'b111_1111;
 4'd9: y = 7'b111_0011;
 4'ha: y = 7'b111_0111;
 4'hb: y = 7'b110_0000;
 4'hc: y = 7'b000_1101;
 4'hd: y = 7'b011_1101;
 4'he: y = 7'b100_1111;
 4'hf: y = 7'b100_0111;
 default: y = 7'b000_0000;
 endcase
assign segs_n = ~y ;
 endmodule
```

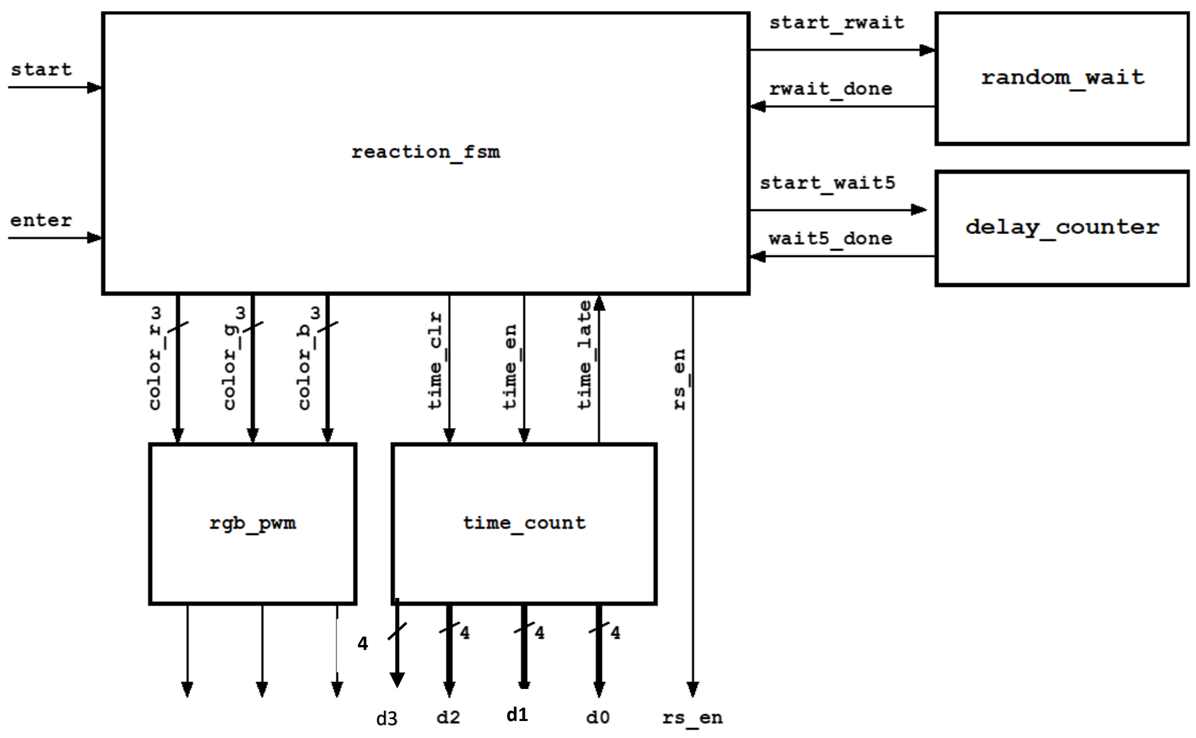
*FIGURE 22 sevenseg_hex schematic diagram*



## THE REACTION TIMER

- The diagram below shows the top-level design of the reaction timer
- These specifications of the reaction timer dictate that it should have three inputs reset, start, enter and 7 outputs i.e., the time data buses and the red, green and blue light signal controllers
- It consists of a stopwatch, a finite state machine, a random wait function, a delay counter and the pulse width modulation controller

*Figure 23 reaction timer top level diagram*



Functionality:

**Inputs:**

start – start the system
enter – display the time taken to before enter button is pressed
rst – synchronous reset
clk – clock signal

**Outputs:**

- led_r – red LED lamp duty ratio controller
- led_g - green LED lamp duty ratio controller
- led_b - blue LED lamp duty ratio controller

- The module that implements the reaction timer is known as reaction_timer.
- It consists of five modules:
- • allcounters: counts the time when the system finishes the random wait procedure
- • rgb_pwm: controls the duty ratio of the rgb LED lamps
- • random_wait: sends the system into an automatic wait period
- • delay_counter5: delays the system for five second
- • reaction_FSM: determines the state the system will be in
- The module has the following important internal signals
    - start_wait5: sends the system into waiting for five seconds. Connects to delay_counter5
    - wait5_done: tells the system that its done waiting for five seconds
    - time_en: it is the enable input for the stop watch
    - time_clr: the reset input for the stop watch
    - time_late: tells the system that 5 seconds have elapsed
    - start_rwait: sends the system into a random wait
    - rwait_done: tells the system that the random wait is done
    - rs_en: turns on the seven-segment display

The finite state machine, the most important module controls all the activities of the reaction timer

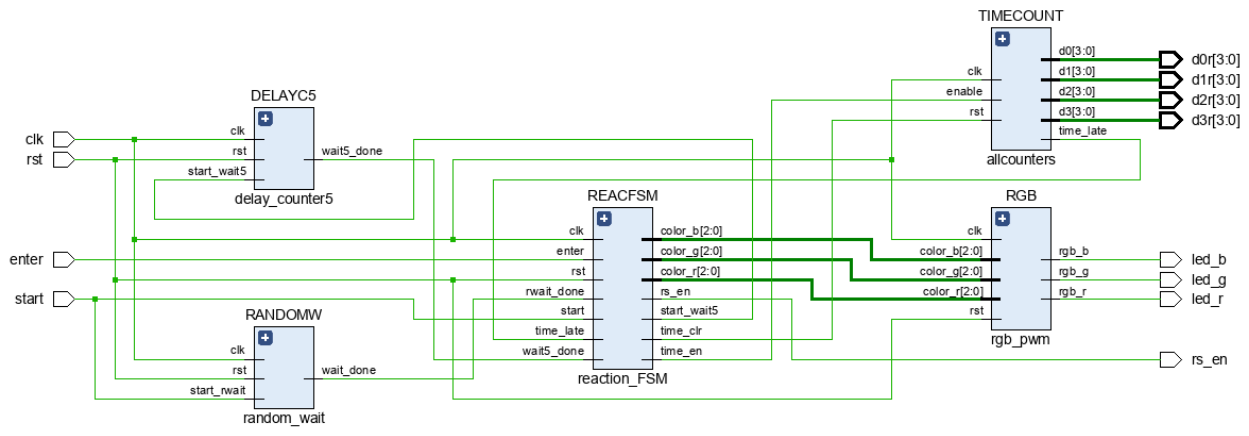
The code for the design of the reaction timer is given below

reaction_timer.sv

```
`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

// Company:

// Engineer:

//

// Create Date: 04/20/2023 08:54:50 AM

// Design Name: REACTION_TIMER

// Module Name: reaction_timer

// Project Name: HEALTH MONITOR

// Target Devices: NEXYSA7 100T

// Tool Versions:

// Description:

//

// Dependencies:

//

// Revision:
```

```
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module reaction_timer(
   input logic start, enter, clk, rst,
   output logic [3:0] d0r, d1r, d2r,d3r,
   output logic rs_en,
   output logic led_r, led_g, led_b
);
// Wires
  logic time_late,time_en,time_clr;
  logic [3:0] d0, d1, d2, d3;
  logic [2:0] color_r, color_g, color_b;

  logic start_rwait, rwait_done;
  logic start_wait5, wait5_done;

//We will have 5 modules

allcounters TIMECOUNT(.d0(d0r), .d1(d1r), .d2(d2r), .d3(d3r), .rst(time_clr),
    .clk(clk),.enable(time_en),.time_late);
rgb_pwm RGB(.color_r, .color_g, .color_b, .rst,.clk(clk), .rgb_r(led_r), .rgb_g(led_g), .rgb_b(led_b));

random_wait RANDOMW(.rst, .clk(clk), .wait_done(rwait_done), .start_rwait(start)); //done

delay_counter5 DELAYC5(.rst, .clk(clk), .wait5_done, .start_wait5);

reaction_FSM REACFSM(.start, .enter, .color_r, .color_g, .color_b, .time_clr, .time_en, .start_wait5,
    .start_rwait, .rs_en, .rwait_done, .wait5_done, .time_late, .rst, .clk(clk));

endmodule
```

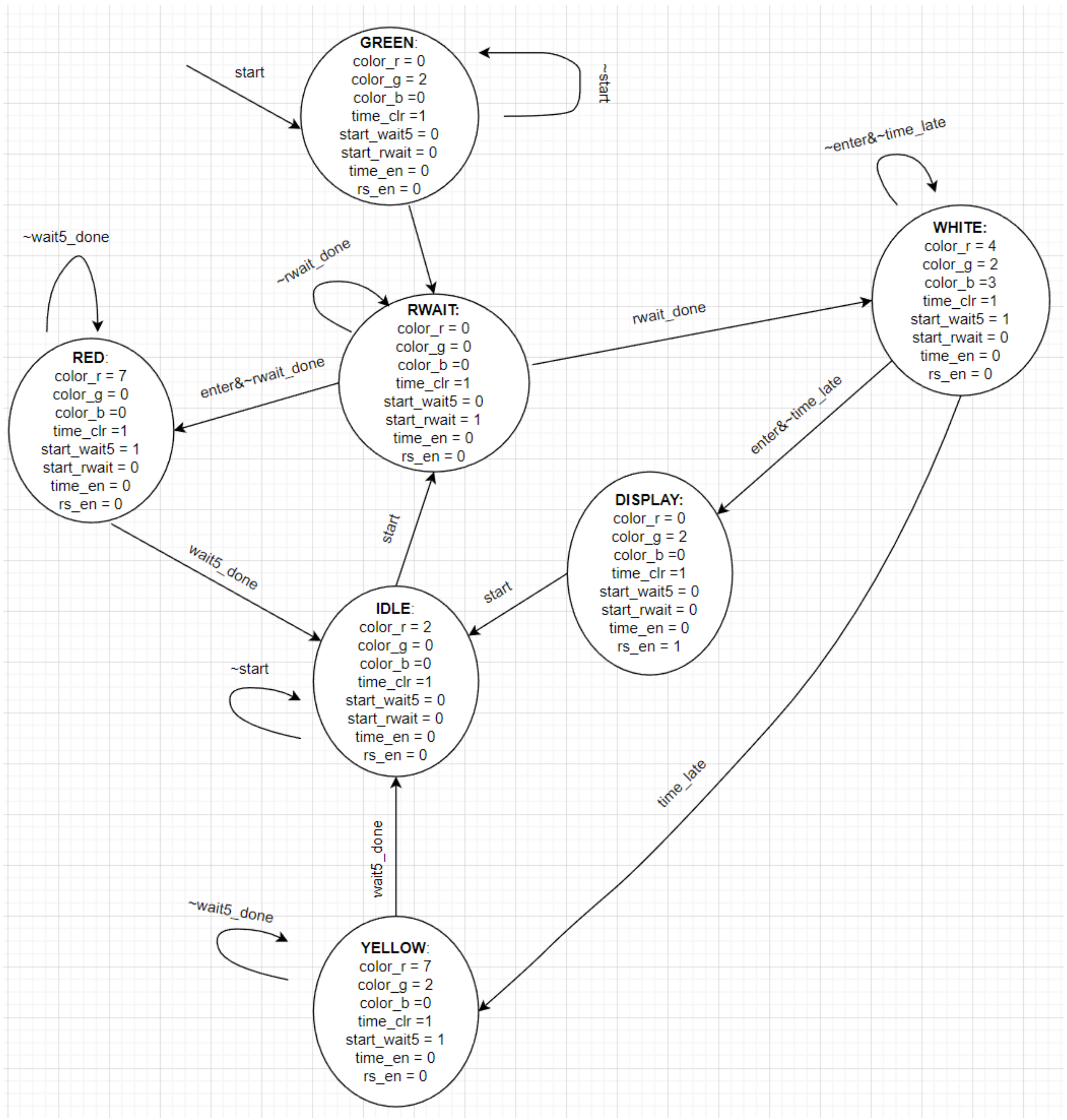
*FIGURE 24 reaction_timer schematic diagram*



a. The Reaction timer finite state machine

- The diagram below is a MOORE finite state machine transition diagram with the outputs in the state bubble.
- The state determines what happens to the LED lamps of the reaction timer and the display on the
- The colors on the LED lamps have to have the same intensity. The finite state machine here determines the intensity of the color. The Color green is more intense for the same duty cycle hence we lowered the duty cycle to 2. Red with the lowest intensity has a duty ratio of 7.
- To produce Yellow, both the red and green LED lamps have to be on with green having a duty cycle of 2 and red having a duty cycle of 7
- To produce white all lamps, have to be on with green having a duty cycle of 2, red having a duty cycle of 4 and blue having a duty cycle of 3.
- The finite state machine was simulated to check for errors and proper functioning. the code for simulation and the scope are given below

*FIGURE 25 finite state machine transition diagram*

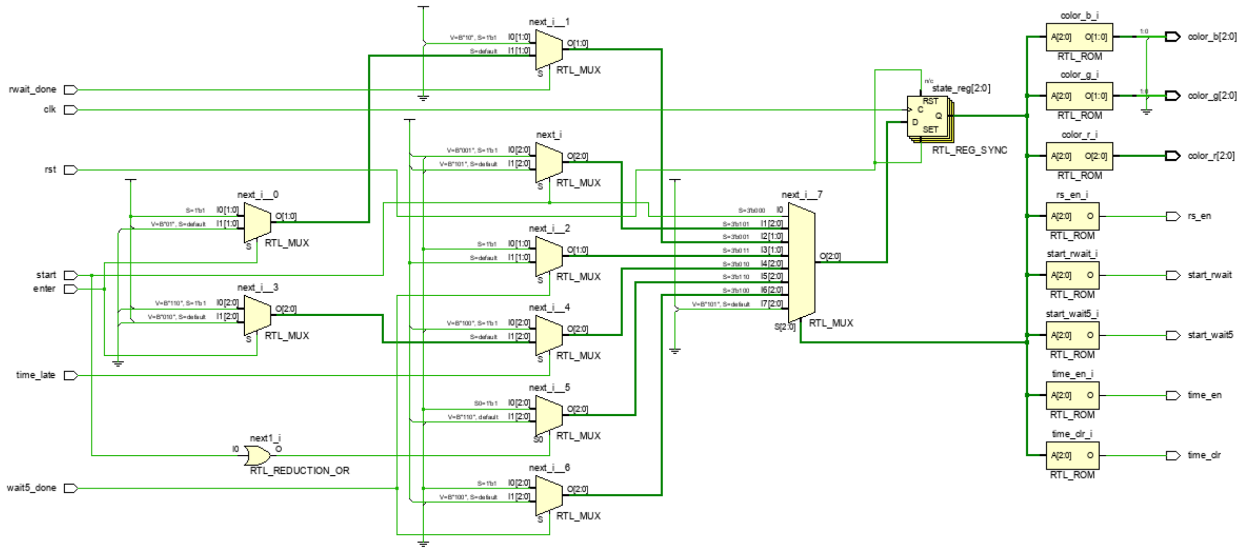


```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: LAFAYETTE COLLEGE
// Engineer: OTIENO MAURICE AND ALEX VILLALBA
//
// Create Date: 04/20/2023 09:43:05 AM
// Design Name: REACFSM
// Module Name: reaction_FSM

// Project Name: HEALTH MONITOR

// Target Devices: NEXYSA7 1007

// Tool Versions:

// Description:

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

//  WE ASSIGN A VALUE 0-7 TO THE COLORS TO CONTROL THE INTENSITY


module reaction_FSM(

   input logic start, enter, rwait_done, wait5_done, time_late, rst, clk,

   output logic [2:0] color_r, color_g, color_b,

   output logic time_clr, time_en, start_wait5, start_rwait, rs_en

);


     typedef enum logic [2:0] {

         IDLE = 3'b000, RWAIT = 3'b001, WHITE = 3'b010, RED = 3'b011,

          YELLOW = 3'b100, GREEN = 3'b101, DISPLAY = 3'b110

      } states_t ;


   states_t state, next;




     always_ff @(posedge clk)

               begin

                 if (rst) state <= GREEN;

                 else state <= next;

                end
```

```
always_comb
   begin
     next = GREEN;
     unique  case(state)
        IDLE: begin
                 color_r = 0; color_g =2; color_b =0; time_clr = 1;
                 start_rwait =0; start_wait5 = 0;rs_en = 0; time_en =0;
                 if (start) next = RWAIT;
                 else next = IDLE;
               end
       GREEN: begin
                 color_r = 0; color_g =2; color_b =0;time_clr = 1;
                 start_rwait =0; time_en = 0; start_wait5 = 0; rs_en = 0;
                 if (start)  next = RWAIT;
                  else next = GREEN;
               end
      RWAIT: begin
                 color_r = 0; color_g =0; color_b =0; time_clr = 1;
                 start_rwait =1;time_en = 0;start_wait5 = 0; rs_en = 0;
                 if (rwait_done) next = WHITE;
                 else  if ( enter & ~rwait_done) next = RED;
                 else next = RWAIT;
               end
       RED: begin
                color_r = 7; color_g =0; color_b =0; start_wait5 = 1;
                time_clr = 1; start_rwait =0; time_en = 0;rs_en =0;
                if (wait5_done) next = IDLE;
                else next = RED;
              end
       WHITE: begin
                color_r = 4; color_g =2; color_b =3; start_rwait = 0;
                rs_en = 0; time_en = 1; start_wait5 = 0;time_clr = 0;
                if (time_late) begin  next = YELLOW; end
                else if (enter & ~time_late) next = DISPLAY;
                else  next = WHITE;
              end
```

```
            DISPLAY: begin
                      color_r = 0; color_g =2; color_b =0; start_rwait =0;
                      rs_en = 1; time_clr = 0; time_en = 0; start_wait5 = 0;
                      if (start) next = IDLE;
                      else next = DISPLAY;
                    end
            YELLOW: begin
                      color_r = 7; color_g =2; color_b =0; time_clr = 1;
                      start_rwait =0; time_en = 0; start_wait5 = 1;rs_en = 0;
                      if (wait5_done) next = IDLE; else next = YELLOW;
                    end
            default: begin
                      color_r = 0; color_g =0; color_b =0; time_clr = 1;
                      start_rwait =0; start_wait5 = 0;rs_en = 0;time_en = 0;
                     end
             endcase
          end
endmodule
```

*FIGURE 26 reaction_FSM schematic diagram*



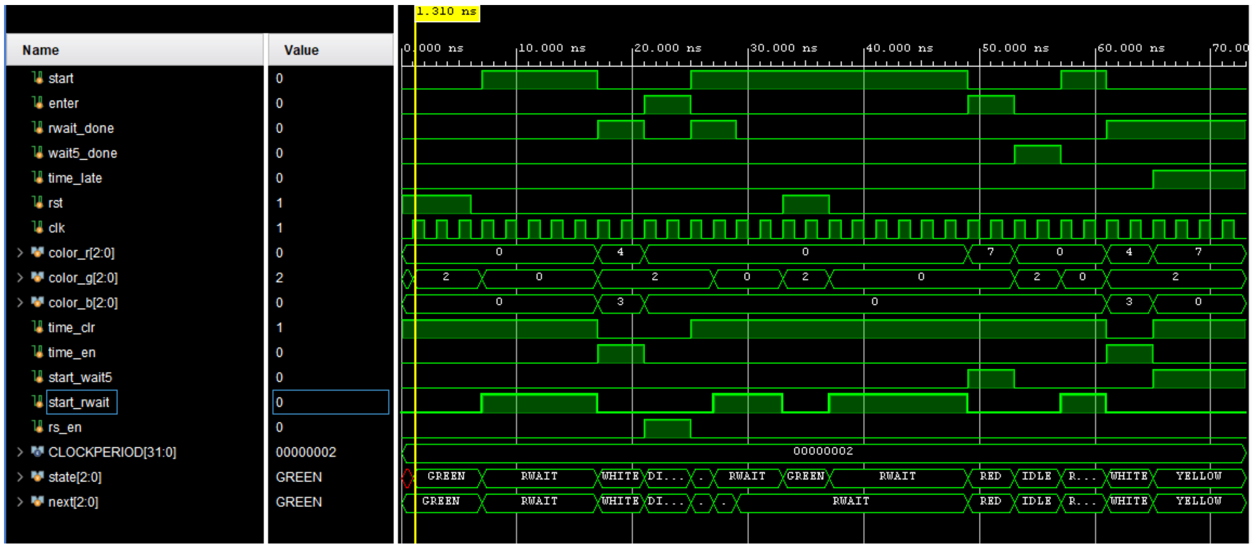
reaction_FSM_tb.sv

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

```
// Company: Lafayette college
// Engineer: Otieno Maurice
//
// Create Date: 04/24/2023 12:05:24 AM
// Design Name: reaction_FSM_TB
// Module Name: reaction_FSM_TB
// Project Name: HEALTH MONITOR
// Target Devices: NEXYSA7 100T
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module reaction_FSM_TB;
     logic start, enter, rwait_done, wait5_done, time_late, rst, clk;
     logic [2:0] color_r, color_g, color_b;
     logic  time_clr, time_en, start_wait5, start_rwait, rs_en;
        reaction_FSM DUV( .start, .enter, .rwait_done, .wait5_done, .time_late, .rst,
             .clk, .color_r, .color_g, .color_b, .time_clr, .time_en, .start_wait5, .start_rwait,.rs_en);

       parameter CLOCKPERIOD = 2;

       always begin
                clk = 0; #(CLOCKPERIOD/2);
                clk = 1; #(CLOCKPERIOD/2);
               end
      initial begin
            rst = 1;
            start = 0; enter = 0; rwait_done = 0; wait5_done = 0;
```

```
             time_late = 0;start_wait5 = 0; #(CLOCKPERIOD*2);
            # (CLOCKPERIOD);
            rst = 0;
            # (CLOCKPERIOD/2);
            start = 1; enter = 0; rwait_done = 0; wait5_done = 0; time_late = 0;#(CLOCKPERIOD*5);
            start = 0; enter = 0; rwait_done = 1; wait5_done = 0; time_late = 0;#(CLOCKPERIOD*2);
            start = 0; enter = 1; rwait_done = 0; wait5_done = 0; time_late = 0;#(CLOCKPERIOD*2);
            start = 1; enter = 0; rwait_done = 1; wait5_done = 0; time_late = 0;#(CLOCKPERIOD*2);
            start = 1; enter = 0; rwait_done = 0; wait5_done = 0; time_late = 0;#(CLOCKPERIOD*2);
            rst = 1; #4;
            rst = 0; #2;
            start = 1; enter = 0; rwait_done = 0; wait5_done = 0; time_late = 0;#(CLOCKPERIOD*5);
            start = 0; enter = 1; rwait_done = 0; wait5_done = 0; time_late = 0;#(CLOCKPERIOD*2);
            start = 0; enter = 0; rwait_done = 0; wait5_done = 1; time_late = 0;#(CLOCKPERIOD*2);
            start = 1; enter = 0; rwait_done = 0; wait5_done = 0; time_late = 0;#(CLOCKPERIOD*2);
            start = 0; enter = 0; rwait_done = 1; wait5_done = 0; time_late = 0;#(CLOCKPERIOD*2);
            start = 0; enter = 0; rwait_done = 1; wait5_done = 0; time_late = 1;#(CLOCKPERIOD*2);
            start = 0; enter = 0; rwait_done = 1; wait5_done = 0; time_late = 1;#(CLOCKPERIOD*2);
        $stop;
      end
endmodule
```

*FIGURE 27 reaction_FSM_TB simulation scope*



b. The time counter (essentially a watch)

The time counter is a collection of instances of the count_bcd module. The count_bcd module is basically a counter with enable

Functionality

**Inputs:**

- clk: clock signal
- rst: synchronous reset
- enable: when asserted, the counter begins counting the time. This enable input is connected to time_en of the finite state machine and is only asserted when the white state.

**Output:**

- d0, d1, d2, d3: 4-bit bcd data that will be displayed on the seven-segment display
- time_late: this out is asserted when d3 equals 5 meaning that five seconds have elapsed.

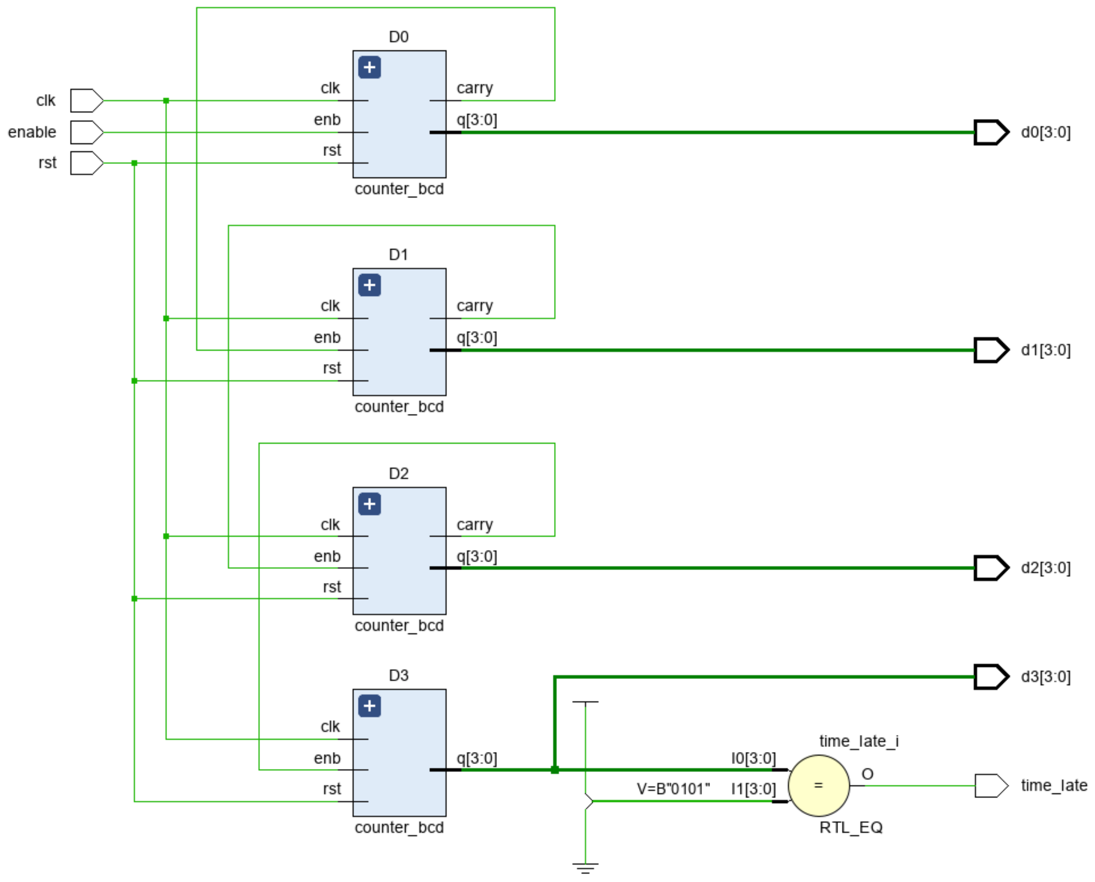
The code for all counters module is given below

allcounter.sv

```
`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

// Company: Lafayette college

// Engineer: Otieno Maurice

//

// Create Date: 04/06/2023 07:45:01 PM

// Design Name: TIMECOUNT

// Module Name: allcounters

// Project Name: HEALTH MONITOR

// Target Devices: NEXYS100 AT

// Tool Versions:

// Description:

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////
```

```
module allcounters(
  input logic clk, rst,enable,
  output logic [3:0] d0,d1,d2,d3,
  output logic time_late
 );
  logic carry1, carry2, carry3, carry4,clkcommon, rstcommon;
  logic [3:0] q0,q1,q2,q3;
   counter_bcd D0(.enb(enable), .rst(rstcommon),.clk(clkcommon), .carry(carry1),.q(q0));
   counter_bcd D1(.enb(carry1),.rst(rstcommon),.clk(clkcommon),.q(q1),.carry(carry2));
   counter_bcd D2(.enb(carry2),.rst(rstcommon),.clk(clkcommon),.q(q2),.carry(carry3));
   counter_bcd D3(.enb(carry3),.rst(rstcommon),.clk(clkcommon),.q(q3)/*.carry(carry4)*/);
     assign time_late = (q3 == 4'd5);
     assign d0 = q0;
     assign d1 = q1;
     assign d2 = q2;
     assign d3 = q3;
     assign clkcommon = clk;
     assign rstcommon = rst;
endmodule
```

FIGURE 28 allcounters schematic diagram



Counter_bcd

Functionality

This component of the allcounters module is the basis of the counter

**inputs:**

- clk: clock signal
- rst: reset signal
- enb: enable signal to start the counting

**outputs:**

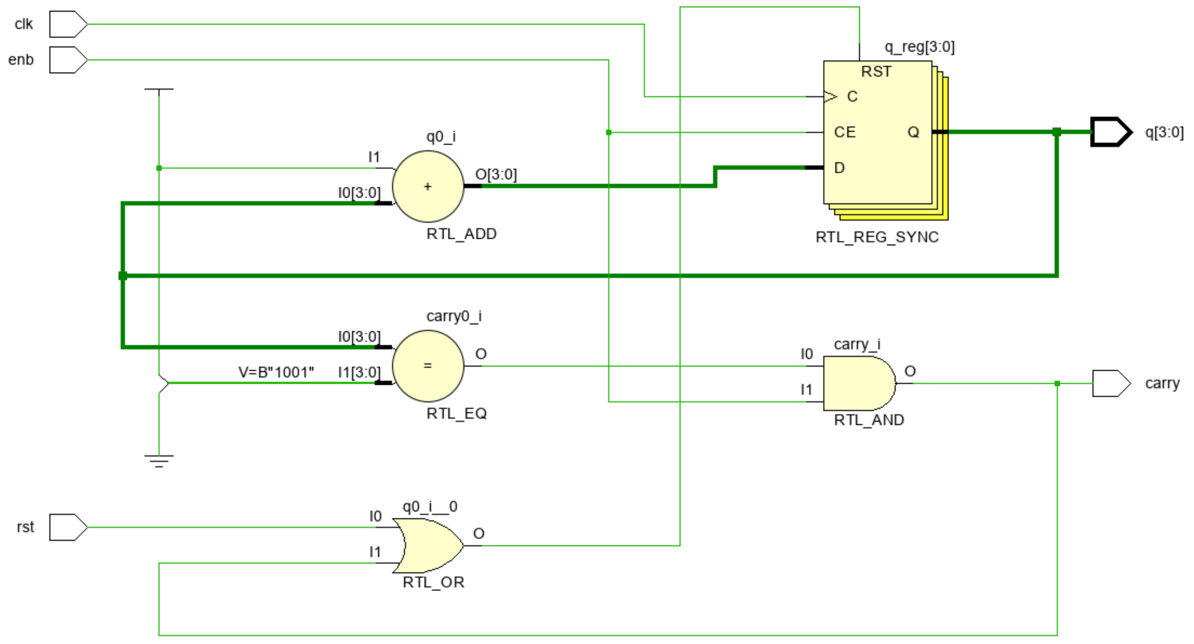
- q: 4-bit output
- carry: becomes the enable signal of the succeeding counters

Counter_bcd.sv

```
//----------------------------------------------------------------------------
```

```
// Title         : counter - simple 4-bit binary counter
// Project       : ECE 211 - Digital Circuits 1
//-----------------------------------------------------------------------------
// File          : counter_bcd.sv
// Author        : John Nestor
// Created       : 10.06.2014
// Last modified : 10.07.2018
//-----------------------------------------------------------------------------
// Description :
// This module is a simple 4-bit counter that you will extend in Lab
//
//-----------------------------------------------------------------------------
module counter_bcd(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 // counter
```

*FIGURE 29 counter_bcd schematic diagram*



c. 5 second delay counter

This delay counter is instrumental for the red and yellow states:

Functionality

- Basically, an incrementor with a comparator. Increments the value of an internal signal W every clock cycle and then compares it to 4999 corresponding to 5 seconds. Asserted when this comparison is true

**inputs:**

- clk: clock signal
- rst: synchronous reset
- start_wait5: is the enable signal of this counter. It receives its signal from the reaction finite state machine

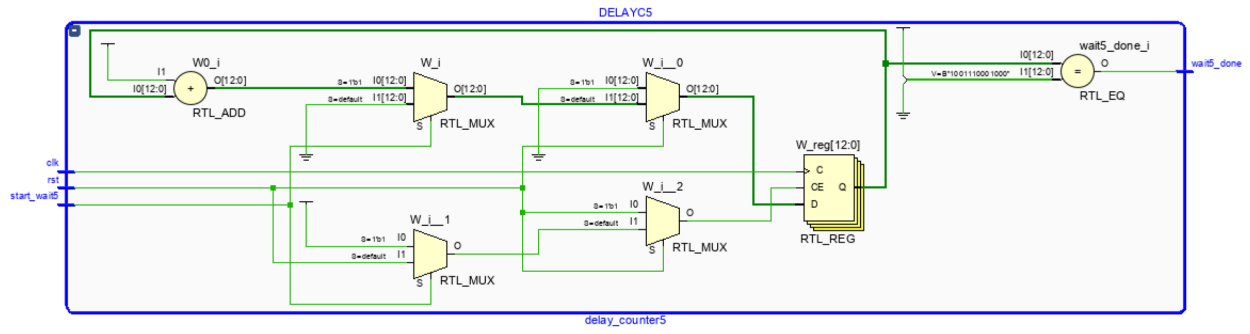
**outputs:**

- wait5_done: forces the system to wait for 5 seconds while in these states before going back to the idle state
-

delay_counter5.sv

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: LAFAYETTE COLLEGE
// Engineer: OTIENO MAURICE AND ALEX VILLABA
//
// Create Date: 04/27/2023 10:13:19 AM
// Design Name: DELAYC5
// Module Name: delay_counter5
// Project Name: HEALTH MONITOR
// Target Devices: NEXYSA7 100T
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module delay_counter5(
     input logic rst, clk, start_wait5,
     output  wait5_done
);
    logic [12:0] W;
        always_ff @(posedge clk)
         begin
           if (rst) W = '0;
           else if (start_wait5) W <= W + 1;
         end
    assign wait5_done = (W == 13'd5000);

endmodule
```

*Figure 30 Delay_Counter5 Schematic Diagram*



d. Red Green Blue Light Pulse Width Modulation controller

- This is another sub module of the reaction timer. This module controls the intensity of the rgb LED lamps by pulse width modulation.
- Each LED output will shine bright when logic high and thus light of very bright intensity. will be produced. We need to control this intensity.

Functionality

**inputs:**

- clk: clock signal
- rst: synchronous reset
- color_r: 3-bit data from fsm that determines the duty cycle of red LED
- color_g: 3-bit data from fsm that determines the duty cycle of red LED
- color_b: 3-bit data from fsm that determines the duty cycle of red LED

**outputs:**

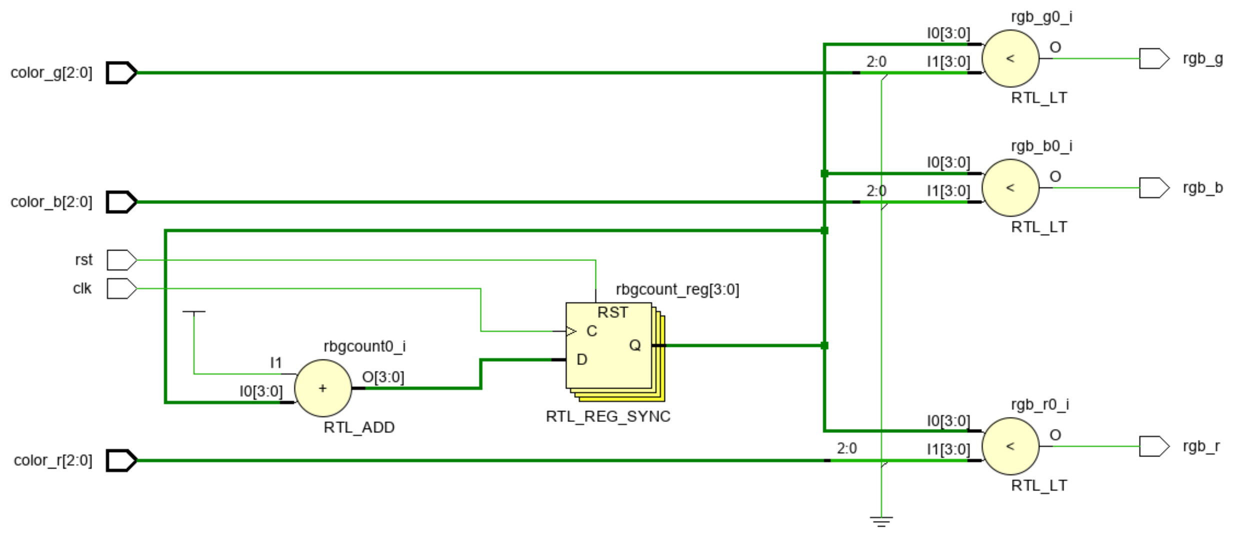
- rgb_r : red light display
- rgb_g-: greenlight display
- rgb_b: blue light display

- The module that implements this design is a counter. It increments an internal signal every clock cycle and compares the inputs to the current value of this signal. So long as the input value is less than that of this signal the LED will be on. In this way intensity is controlled by determining the duty cycle of the lamps.

rgb_pwm.sv

```
`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

// Company: Otienom

// Engineer:

//

// Create Date: 04/22/2023 03:43:15 PM

// Design Name:

// Module Name: rgb_pwm

// Project Name:

// Target Devices:

// Tool Versions:

// Description:

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////



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]  rbgcount;


always_ff @(posedge clk)

    if (rst) rbgcount <= '0;
    else rbgcount <= rbgcount + 4'd1;

always_comb
    begin
        if (rbgcount < color_r) rgb_r = 1; else rgb_r = 0;
        if (rbgcount < color_g) rgb_g = 1; else rgb_g = 0;
        if (rbgcount < color_b) rgb_b = 1; else rgb_b = 0;
    end

endmodule
```

*Figure 31 rgb_pwm schematic diagram*

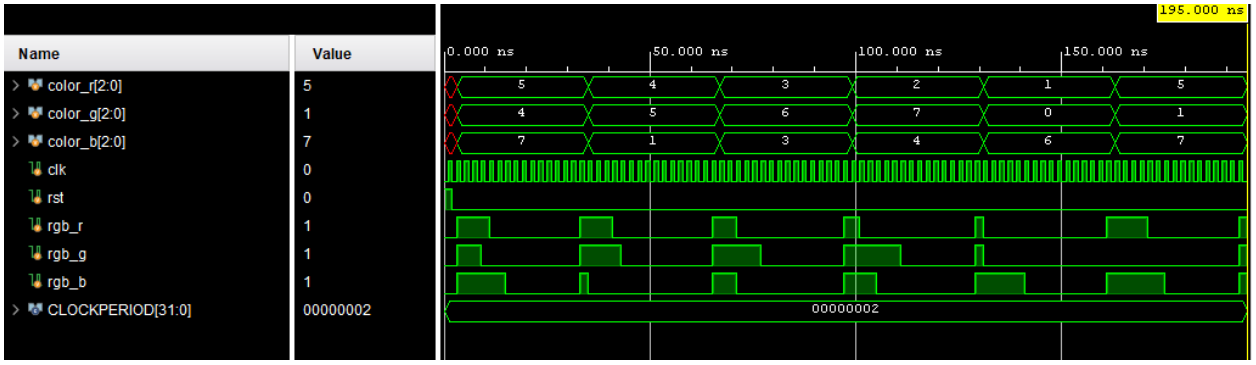


rgb_pwm_tb.sv

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: Lafayette college
// Engineer: Otieno Maurice
//
// Create Date: 04/24/2023 12:05:24 AM
// Design Name:
// Module Name: rgb_pwm_tb
// Project Name:
```

```
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module rgb_pwm_tb;
    logic [2:0] color_r, color_g, color_b;
    logic clk,rst,rgb_r, rgb_g, rgb_b;

    rgb_pwm DUV(.rgb_r,. rgb_g, .rgb_b,.color_r, .color_g, .color_b, .clk, .rst);

    parameter CLOCKPERIOD = 2;

    always begin
         clk = 0; #(CLOCKPERIOD/2);
         clk = 1; #(CLOCKPERIOD/2);
       end
    initial begin
       rst = 1;

        # (CLOCKPERIOD);
        rst = 0;
        # (CLOCKPERIOD/2);
        color_r = 3'd5; color_g =3'd4; color_b = 3'd7; #(CLOCKPERIOD*16);
        color_r = 3'd4; color_g =3'd5; color_b = 3'd1; #(CLOCKPERIOD*16);
        color_r = 3'd3; color_g =3'd6; color_b = 3'd3; #(CLOCKPERIOD*16);
        color_r = 3'd2; color_g =3'd7; color_b = 3'd4; #(CLOCKPERIOD*16);
        color_r = 3'd1; color_g =3'd0; color_b = 3'd6; #(CLOCKPERIOD*16);
        color_r = 3'd5; color_g =3'd1; color_b = 3'd7; #(CLOCKPERIOD*16);
      $stop;
    end
endmodule
```

*FIGURE 32 rgb_pwm_tb simulation scope*



e. Random Wait

- This is the component of the reaction timer that sets the system in waiting for a random amount of time.
  Functionality
  **inputs:**
- clk:
- rst:
- start_rwait: it is the enable of the counter. It starts the waiting for a random number process
  **outputs:**
- wait_done: connected to rwait_done of the finite state machine. Delivers the signal to stop the random wait

- The idea is that there should be a 3-bit counter with enable that is connected to a register with enable. The enable to the registers is the start signal from the user while the enable to the counter is from the finite state machine
- When the start button is pressed, the enable of this register is asserted and the register picks up the current number in the counter and stores it.
- Since the clock is running at 1kHz, this number is between 0.000s and 0.007s.
- This value is first made to be between 0.001s and 0.008s by an incrementor and then made to between 1.000s to 8.000s by a multiplier (essentially shifting to the left by 10 bits).

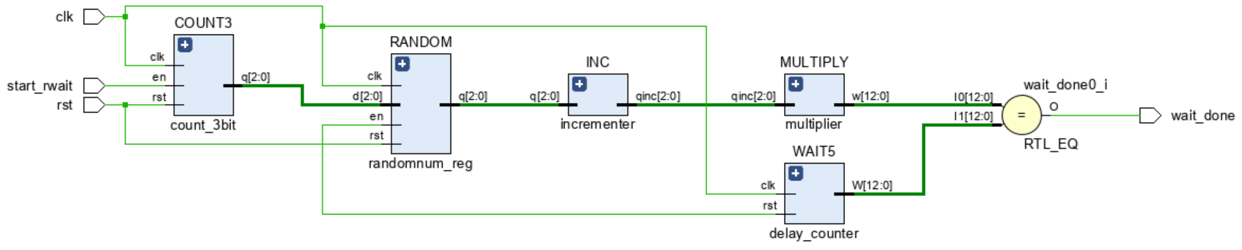
- Finally, there is a delay counter that compares this random generated number to its value. When the value of the delay counter and the random number are equal, the output wait_done is asserted.

random_wait.sv

```
`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

// Company: LAFAYETTE COLLEGE

// Engineer: Alex Villalba AND OTIENO MAURICE

//

// Create Date: 04/22/2023 04:23:46 PM

// Design Name:  RANDOMW

// Module Name: random_wait

// Project Name: HEALTH MONITOR

// Target Devices: NEXYSA7 100T

// Tool Versions:

// Description: This module outputs a random value using the clock with a 3-bit counter and sampling the
result.

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////



module random_wait(

    input logic clk, rst, start_rwait,

    output logic wait_done

);


    logic [2:0] random_num, q, qinc;

    logic sclk, commonstart;

    logic [12:0]w, W;
```

```
        assign start_wait = commonstart;


        delay_counter WAIT5(.clk, .rst(commonstart),.W);

        count_3bit COUNT3(.clk, .rst, .q(random_num), .en(start_rwait));

        incrementer INC(.q(q),.qinc(qinc));

        randomnum_reg RANDOM(.rst,.clk,.d(random_num),.q(q),.en(commonstart));

        multiplier MULTIPLY(.qinc(qinc),.w(w));


   always_comb

       begin

           if (w == W) wait_done = 1'b1;

           else wait_done = 1'b0;

       end

  endmodule
```

*FIGURE 33 random_wait schematic diagram*

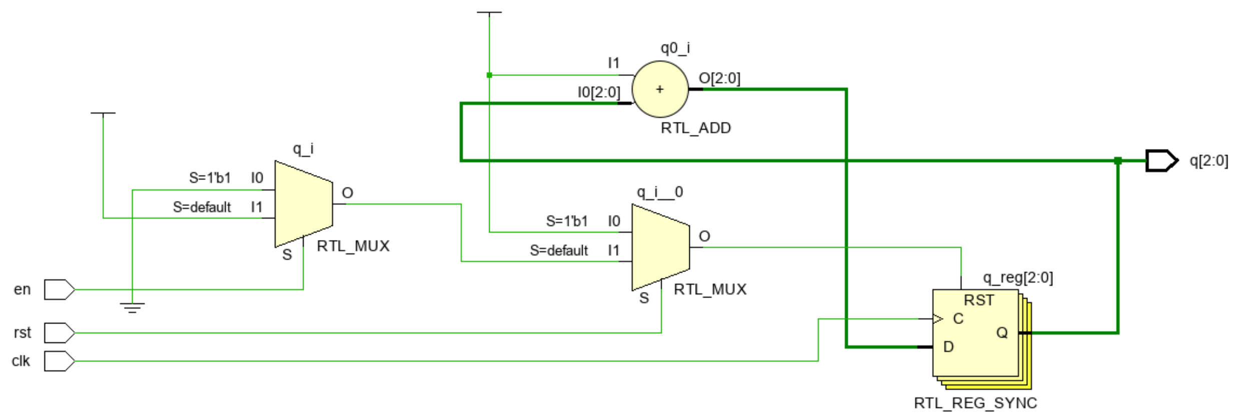


## 1. The 3-bit counter

count_3bit.sv

```
module count_3bit (
 input logic clk, rst, en,
 output logic [2:0] q
 );


 always_ff @(posedge clk)
 if (rst) q <= 3'd0;
 else if (en) q <= q + 3'd1;
 else q <= 0;
endmodule // count
```

*FIGURE 34 count_3bit schematic diagram*

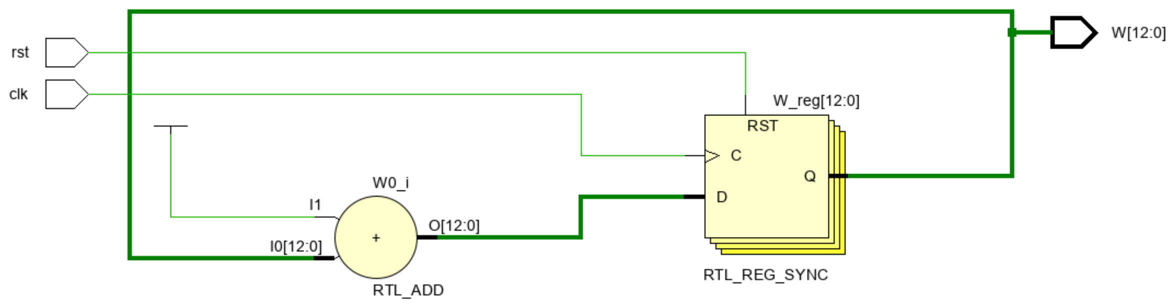


## 2. The Delay Counter

delay_counter.sv

```
`timescale 1ns / 1ps


module delay_counter(

    input logic rst, clk,

    output logic [12:0]W

);


always_ff @(posedge clk)

    begin

       if (rst) W <= '0;

          W <= W + 1;

    end

endmodule
```

*FIGURE 35 delay_counter schematic diagram*



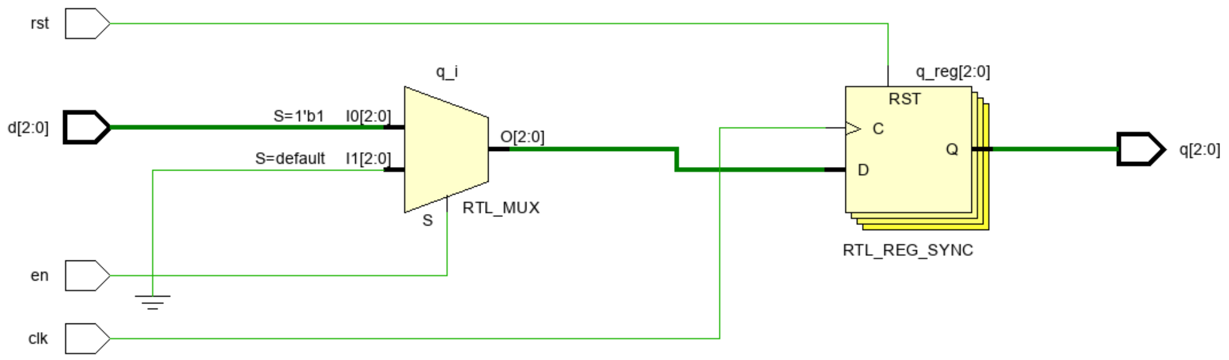
## 3. The random number register

randomnum_reg.sv

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: LAFAYETTE COLLEGE
// Engineer: OTIENO MAURICE
//
// Create Date: 04/22/2023 04:27:47 PM
// Design Name: RANDOM
// Module Name: delay_counter
// Project Name: HEALTH MONITOR
// Target Devices: NEXYSA7 100T
// Tool Versions:
// Description: This module outputs the signal that the 5 seconds delay was completed
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
```

```
module randomnum_reg(

input logic rst, clk,[2:0]d,

output logic [2:0]q,

input logic en

);

always_ff @(posedge clk)

    begin

       if (rst)  q <= '0;

       else if (en) q <= d;

       else q <= '0;

       end

endmodule
```

*FIGURE 36 randomnum_reg schematic diagram*



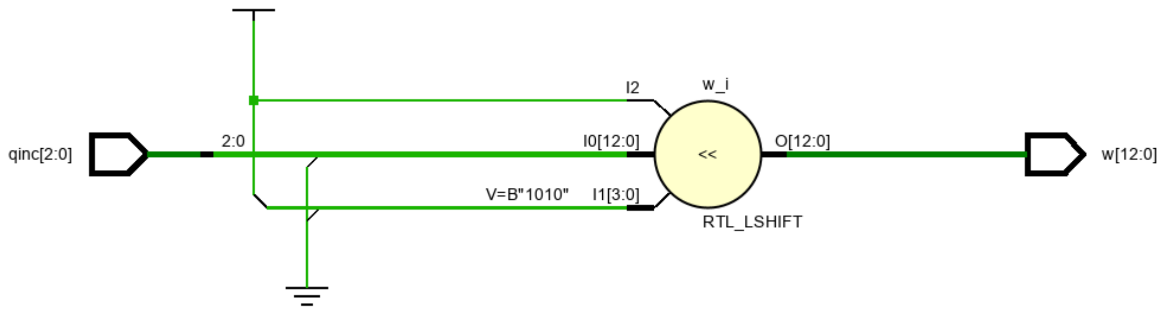
## 4. Incrementor

incrementor.sv

```
`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

// Company: Lafayette college

// Engineer: Otieno Maurice

//

// Create Date: 04/23/2023 10:52:00 PM

// Design Name:

// Module Name: incrementor
```

```
module incrementor(
  input logic [2:0] q,
  output logic [2:0] qinc
);
 assign qinc = q + 3'd1;
endmodule
```

*FIGURE 37 incrementor schematic diagram*

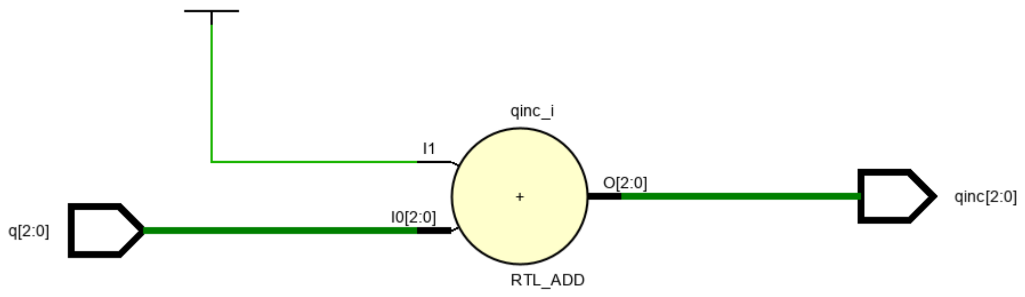


## 5. Multiplier

multiplier.sv

```
`timescale 1ns / 1ps
module multiplier (
input logic [2:0]qinc,
output logic [12:0]w
    );
   assign w = qinc << 10;
endmodule
```

*FIGURE 38 multiplier schematic diagram*



## <u>REALIZATION ON THE ON NEXYSA7 100T PRINTED CIRQUIT BOARD</u>

Lab10top.xdc

```
## Clock signal

set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 } [get_ports { clock }]; #IO_L12P_T1_MRCC_35
Sch=clk100mhz

create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {clock}];




## RGB LEDs

#set_property -dict { PACKAGE_PIN R12   IOSTANDARD LVCMOS33 } [get_ports { rs_en }]; #IO_L5P_T0_D06_14
Sch=led16_b

#set_property -dict { PACKAGE_PIN M16   IOSTANDARD LVCMOS33 } [get_ports { time_clr }]; #IO_L10P_T1_D14_14
Sch=led16_g

set_property -dict { PACKAGE_PIN N15   IOSTANDARD LVCMOS33 } [get_ports { pulse_led }];
#IO_L11P_T1_SRCC_14 Sch=led16_r




set_property -dict { PACKAGE_PIN G14   IOSTANDARD LVCMOS33 } [get_ports { led_b }];
#IO_L15N_T2_DQS_ADV_B_15 Sch=led17_b

set_property -dict { PACKAGE_PIN R11   IOSTANDARD LVCMOS33 } [get_ports { led_g }]; #IO_0_14 Sch=led17_g

set_property -dict { PACKAGE_PIN N16   IOSTANDARD LVCMOS33 } [get_ports { led_r }]; #IO_L11N_T1_SRCC_14
Sch=led17_r


##Switches

set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports { switch }]; #IO_L24N_T3_RS0_15
Sch=sw[0]
```

```
## LEDs

set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports {  led[0] }]; #IO_L18P_T2_A24_15
Sch=led[0]

set_property -dict { PACKAGE_PIN K15   IOSTANDARD LVCMOS33 } [get_ports {  led[1] }]; #IO_L17N_T2_A25_15
Sch=led[2]

set_property -dict { PACKAGE_PIN J13   IOSTANDARD LVCMOS33 } [get_ports {  led[2] }]; #IO_L17N_T2_A25_15
Sch=led[2]

set_property -dict { PACKAGE_PIN N14   IOSTANDARD LVCMOS33 } [get_ports {  led[3] }]; #IO_L17N_T2_A25_15
Sch=led[2]

set_property -dict { PACKAGE_PIN R18   IOSTANDARD LVCMOS33 } [get_ports {  led[4] }]; #IO_L7P_T1_D09_14
Sch=led[4]

set_property -dict { PACKAGE_PIN V17   IOSTANDARD LVCMOS33 } [get_ports {  led[5] }];
#IO_L18N_T2_A11_D27_14 Sch=led[5]

set_property -dict { PACKAGE_PIN U17   IOSTANDARD LVCMOS33 } [get_ports {  led[6] }];
#IO_L17P_T2_A14_D30_14 Sch=led[6]

set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports {  led[7] }];
#IO_L18P_T2_A12_D28_14 Sch=led[7]

set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 } [get_ports {  led[8] }];
#IO_L16N_T2_A15_D31_14 Sch=led[8]

set_property -dict { PACKAGE_PIN T15   IOSTANDARD LVCMOS33 } [get_ports {  led[9] }]; #IO_L14N_T2_SRCC_14
Sch=led[9]

set_property -dict { PACKAGE_PIN U14   IOSTANDARD LVCMOS33 } [get_ports {  led[10] }];
#IO_L22P_T3_A05_D21_14 Sch=led[10]

set_property -dict { PACKAGE_PIN T16   IOSTANDARD LVCMOS33 } [get_ports {  led[11] }];
#IO_L15N_T2_DQS_DOUT_CSO_B_14 Sch=led[11]

set_property -dict { PACKAGE_PIN V15   IOSTANDARD LVCMOS33 } [get_ports {  led[12]  }];
#IO_L16P_T2_CSI_B_14 Sch=led[12]

set_property -dict { PACKAGE_PIN V14   IOSTANDARD LVCMOS33 } [get_ports {  led[13] }];
#IO_L22N_T3_A04_D20_14 Sch=led[13]

set_property -dict { PACKAGE_PIN V12   IOSTANDARD LVCMOS33 } [get_ports {  led[14] }];
#IO_L20N_T3_A07_D23_14 Sch=led[14]

set_property -dict { PACKAGE_PIN V11   IOSTANDARD LVCMOS33 } [get_ports {  led[15] }];
#IO_L21N_T3_DQS_A06_D22_14 Sch=led[15]


#SEVEN SEGMENT DISPLAY

set_property -dict { PACKAGE_PIN T10 IOSTANDARD LVCMOS33 } [get_ports { segs_n[6] }];

#IO_L24N_T3_A00_D16_14 Sch=ca

set_property -dict { PACKAGE_PIN R10 IOSTANDARD LVCMOS33 } [get_ports { segs_n[5] }];

#IO_25_14 Sch=cb

set_property -dict { PACKAGE_PIN K16 IOSTANDARD LVCMOS33 } [get_ports { segs_n[4] }];

#IO_25_15 Sch=cc
```

```
set_property -dict { PACKAGE_PIN K13 IOSTANDARD LVCMOS33 } [get_ports { segs_n[3] }];

#IO_L17P_T2_A26_15 Sch=cd

set_property -dict { PACKAGE_PIN P15 IOSTANDARD LVCMOS33 } [get_ports { segs_n[2] }];

#IO_L13P_T2_MRCC_14 Sch=ce

set_property -dict { PACKAGE_PIN T11 IOSTANDARD LVCMOS33 } [get_ports { segs_n[1] }];

#IO_L19P_T3_A10_D26_14 Sch=cf

set_property -dict { PACKAGE_PIN L18 IOSTANDARD LVCMOS33 } [get_ports { segs_n[0] }];

#IO_L4P_T0_D04_14 Sch=cg

set_property -dict { PACKAGE_PIN H15 IOSTANDARD LVCMOS33 } [get_ports { dot }];

#IO_L19N_T3_A21_VREF_15 Sch=dp

set_property -dict { PACKAGE_PIN J17 IOSTANDARD LVCMOS33 } [get_ports { a_n[0] }];

#IO_L23P_T3_FOE_B_15 Sch=an[0]

set_property -dict { PACKAGE_PIN J18 IOSTANDARD LVCMOS33 } [get_ports { a_n[1] }];

#IO_L23N_T3_FWE_B_15 Sch=an[1]

set_property -dict { PACKAGE_PIN T9 IOSTANDARD LVCMOS33 } [get_ports { a_n[2] }];

#IO_L24P_T3_A01_D17_14 Sch=an[2]

set_property -dict { PACKAGE_PIN J14 IOSTANDARD LVCMOS33 } [get_ports { a_n[3] }];

#IO_L19P_T3_A22_15 Sch=an[3]

set_property -dict { PACKAGE_PIN P14 IOSTANDARD LVCMOS33 } [get_ports { a_n[4] }];

#IO_L8N_T1_D12_14 Sch=an[4]

set_property -dict { PACKAGE_PIN T14 IOSTANDARD LVCMOS33 } [get_ports { a_n[5] }];

#IO_L14P_T2_SRCC_14 Sch=an[5]

set_property -dict { PACKAGE_PIN K2 IOSTANDARD LVCMOS33 } [get_ports { a_n[6] }];

#IO_L23P_T3_35 Sch=an[6]

set_property -dict { PACKAGE_PIN U13 IOSTANDARD LVCMOS33 } [get_ports { a_n[7] }];

#IO_L23N_T3_A02_D18_14 Sch=an[7]


##Buttons

#set_property -dict { PACKAGE_PIN C12   IOSTANDARD LVCMOS33 } [get_ports { CPU_RESETN }];
#IO_L3P_T0_DQS_AD1P_15 Sch=cpu_resetn

set_property -dict { PACKAGE_PIN N17   IOSTANDARD LVCMOS33 } [get_ports { reset }]; #IO_L9P_T1_DQS_14
Sch=btnc

set_property -dict { PACKAGE_PIN M18   IOSTANDARD LVCMOS33 } [get_ports { start }]; #IO_L4N_T0_D05_14
Sch=btnu

set_property -dict { PACKAGE_PIN P17   IOSTANDARD LVCMOS33 } [get_ports { enter }]; #IO_L12P_T1_MRCC_14
Sch=btnl
```

```
##Pmod Header JXADC

set_property -dict { PACKAGE_PIN A14   IOSTANDARD LVCMOS33 } [get_ports { analog_neg_in }];
#IO_L9N_T1_DQS_AD3N_15 Sch=xa_n[1]

set_property -dict { PACKAGE_PIN A13   IOSTANDARD LVCMOS33 } [get_ports { analog_pos_in }];
#IO_L9P_T1_DQS_AD3P_15 Sch=xa_p[1]
```

## System Validation and Performance

### Reaction Timer

| Test | PROCEDURE | RESULTS | PASS/ FAIL | COMMENTS |
|---|---|---|---|---|
| 1 | SW0 on | Seven seg off<br>Led is green | P | N/A |
| 2 | Press Start Button | LED turns white after 6 seconds of pressing the button | P | |
| 3 | Press Start Button and then enter in quick succession press | LED TURNS red five seconds then turns green | P | N/A |
| 4 | Press Start Button<br>wait for 7 seconds | LED turns yellow for 5 seconds then turns green<br>Seven Segment remains off | P | N/A |
| 5 | Press the reset while LED is on | Seven Segment turns off | P | N/A |
| 6 | Press the enter button when the light is white | Seven Segment displays 0.881<br>LED turns green | P | N/A |

## Pulse Monitor

| Test | PROCEDURE | RESULT | PASS/ FAIL | COMMENTS |
|---|---|---|---|---|
| 1 | SW0 off | Seven segment display on, all digit are 0<br>LEDS blinking | P | N/A |
| 2 | Finger in Pulse Monitor | Seven Segment displays the pulse in a range of 36-68,<br>Pulse LED blinking | P | We have seen complications with dark skin color as the sensor was displaying 000 |
| 3 | o finger is placed on the sensor | Seven segment displays 000 | P | N/A |
| 4 | Press reset button | Seven segment displays 000 | P | N/A |

## APPENDIX

1. BPM – Beats Per Minute
2. FPGA – Field Programmable Gate Array
3. PCB – Printed Circuit Board
4. PWM – Pulse width modulation