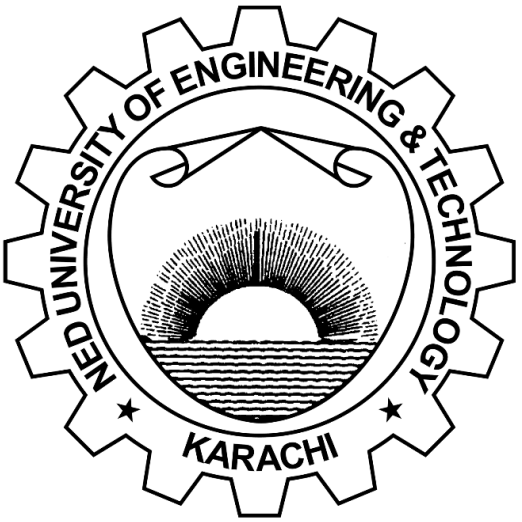


COMPLEX ENGINEERING PROBLEM REPORT

FPGA-Based Accelerometer/Temperature Sensor Data-Based Intensity Control and Color-Coding Design

Course	Digital System Design
Submitted by	Aleena Yameen CS-047 Safia Faiz CS-058 Syed Muhammad Mubashir Rizvi CS-071 Sunyah Faisal CS-082
Submitted to	Ms. Ramish Fatima
Batch	2020
Date of submission	17th January 2024

DEPARTMENT OF COMPUTER & INFORMATION SYSTEMS ENGINEERING

NED UNIVERSITY OF ENGINEERING & TECHNOLOGY

Contents

1. Task Description

Design a digital system incorporating FPGA-based realization for an accelerometer/temperature sensor data-based intensity control and color-code (using tri-color LED's) application. Intensity control shall be designed using Pulse Width Modulation (PWM) technique. [CPA1, CPA2, CPA-3].

You are required to take accelerometer/temperature sensor data as input [CPA-1, CPA-2] and incorporate Pulse Width Modulation (PWM) technique to control the intensity level of tri-color LED (along with color coding) for different levels of values (set any range you want) in the x, y and z-axes for accelerometer or temperature register values for temperature sensor. [CPA1, CPA3]. The accelerometer sensor requires serial communication (SPI protocol) to access its data whereas temperature sensor requires I2C protocol. [CPA 3]. The system shall also display the values on the timemultiplexed seven segment displays interfaced with FPGA on the FPGA board. Alternatively, you can display the values on a monitor interfaced with FPGA through VGA port available on the board [CPA1, CPA2, CPA-3].

You will be required to exhibit simulation of pulse width modulation and accelerometer/temperature sensor data as well [CPA-3]. Maintain a hierarchical structure of your design with separate modules for different functionalities. Your design should be robust enough to handle different values. [CPA2].

The design can be enhanced as per the vision of the design team, however, keeping the basic functionality and requirements in perspective [CPA3].

2. Design Methodology

2.1 System Overview

The project aims to design a digital system using FPGA for intensity control and color-coding based on accelerometer/temperature sensor data. The system uses Pulse Width Modulation (PWM) for intensity control of a tri-color LED and displays the values on seven-segment displays. The accelerometer sensor uses SPI protocol, and the temperature sensor uses I2C protocol.

2.2 High Level Design

The design consists of several modules:

2.2.1 main

The main module integrates all modules and manages the selection of data sources, handling either switch-based temperature values or sensor-based values. It functions as the core control unit for a temperature monitoring and display system. It interfaces with the Nexys clock signal (CLK100MHZ), handles temperature data from switches (TMP_SWTCH) or a sensor (TMP_SDA) using an I2C master, and controls display elements. The module dynamically selects temperature data based on the SELECT

signal, either from switches or the sensor. The seven-segment display (inst_seg) visually represents the temperature, and the Nexys LEDs (LED) display the binary temperature representation. Additionally, a tri-color LED is controlled using the clrcode_intensity module (inst_led), determining color and intensity based on the temperature data. This integrated architecture efficiently realizes a comprehensive temperature monitoring and display system on the Nexys platform.

2.2.2 i2c_master

The i2c_master module serves as an I2C master controller for communication with a temperature sensor. It generates a 10kHz clock signal (SCL) from a 200kHz input clock (clk_200kHz) and handles bidirectional data on the SDA line. The module follows a state machine with 28 states to initiate communication, address the sensor, read temperature data, and handle acknowledgments. It synchronizes the received temperature data and controls the direction of the SDA signal during different phases of the I2C communication. The module efficiently manages the intricacies of I2C communication, providing a robust interface between the master controller and the temperature sensor.

2.2.3 clkgen_200kHz

The clkgen_200kHz module generates a 200kHz clock signal (clk_200kHz) from a 100MHz input clock (clk_100MHz). It utilizes an 8-bit counter to divide the frequency, toggling the output clock signal at half the frequency of the input. The counter resets when it reaches a count of 249, effectively dividing the input clock frequency by 500. This module provides a reliable way to generate a lower-frequency clock signal for applications that require a slower clock rate.

2.2.4 seg_Disp

The seg_Dsplay module is designed for driving a seven-segment display and controlling anodes based on a given temperature input. It utilizes multiple registers to store segment values for each digit, a counter (Counter) for multiplexing, and a case statement to convert binary-coded decimal (BCD) values to corresponding seven-segment display configurations.

The module dynamically updates the SEG output, AN (anodes), and dp (decimal point) based on the Counter value, creating a continuous display of the temperature. Additionally, it handles negative temperature values by adjusting the display accordingly.

The temperature data is processed to extract its digits and convert them to corresponding seven-segment patterns. The module utilizes a case statement to determine the segment configurations for each digit. The Counter value is used for multiplexing, cycling through the digits to create the illusion of a continuous display.

2.2.5 clrcode_intensity

The clrcode_intensity module manages the color and intensity of a tri-color LED based on the provided temperature data. Utilizing a PWM counter and temperature information, the module dynamically adjusts the RGB values to represent different temperature ranges. The temperature is converted to a signed 13-bit value, allowing for both positive and negative temperatures.

The module defines specific color and intensity combinations for different temperature ranges, as represented in the table below.

	Temperature ranges	LED Color	LED Intensity
Cold	-20 – 0 degrees	Blue	High
	0 – 10 degrees	Magenta	Low
Warm	10 – 20 degrees	Cyan	Low
	20 – 30 degrees	Green	High
Hot	30 – 40 degrees	Yellow	Low
	40 – 50 degrees	Red	High

The PWM counter is used to control the duty cycle, determining the LED's intensity. The module efficiently handles different temperature ranges, providing a clear and visually informative representation of temperature on the tri-color LED.

2.2.6 pwm

The PWM module generates an 8-bit PWM signal (pwm_count) with a frequency controlled by a 100MHz clock (clk_100MHz). The counter increments until it reaches 100, at which point it resets to zero, creating a PWM signal with a duty cycle that repeats every 100 clock cycles. This module effectively provides a simple PWM functionality for controlling the intensity of devices such as LEDs.

3. Verilog Source Code

3.1 main.v

```
23  module main(
24      input         CLK100MHZ,
25      input         reset,
26      input [12:0]  TMP_SWTCH,
27      input         SELECT,
28      inout         TMP_SDA,
29      output        TMP_SCL,
30      output [6:0]  SEG,
31      output [7:0]  AN,
32      output dp,
33      output [15:0] LED ,
34      output R, G, B
35      );
36
37      wire sda_dir;
38      wire w_200kHz;
39      reg [15:0] w_data;
40      wire [15:0] temperature;
41
42
43      // Instantiate i2c master
44      i2c_master inst_master(
45          .clk_200kHz(w_200kHz),
46          .reset(reset),
47          .temp_data(temperature),
48          .SDA(TMP_SDA),
49          .SDA_dir(sda_dir),
50          .SCL(TMP_SCL)
51      );
52
53      always@(posedge CLK100MHZ)
54
55          begin
56              if (SELECT==1) w_data<=(TMP_SWTCH<<3);
57              else w_data<=temperature;
58          end
59
60      // Instantiate 200kHz clock generator
61      clkgen_200kHz inst_clk(
62          .clk_100MHz(CLK100MHZ),
63          .clk_200kHz(w_200kHz)
64      );
65
66      // Instantiate 7 segment control
67      seg_Dsplay inst_seg(.clk_100MHz(CLK100MHZ),
68          .temp_data(w_data),
69          .SEG(SEG),
70          .AN(AN),
71          .dp(dp));
72      // Set LED value to temp data
73      assign LED = w_data;
74
75      // Instantiate Tri-Color Led Color and Intensity Control
76      clrcode_intensity inst_led(
77          .clk_100MHz(CLK100MHZ),
78          .temp_data(w_data),
79          .R(R),
80          .G(G),
81          .B(B)
82
83      );
84
85  endmodule
```

3.2 i2c_master.v

```
module i2c_master(
    input clk_200kHz,               // i_clk
    input reset,                    // btnC on nexys
    inout SDA,                      // i2c standard interface signal
    output [15:0] temp_data,         // 8 bits binary representation of deg C
    output SDA_dir,                 // direction of inout signal on SDA - to/from master
    output SCL                      // i2c standard interface signal - 10KHZ
    );

    // *** GENERATE 10kHz SCL clock from 200kHz ****************************
    // 200 x 10^3 / 10 x 10^3 / 2 = 10
    reg [3:0] counter = 4'b0000;  // count up to 9
    reg clk_reg = 1'b1;

    always @(posedge clk_200kHz or posedge reset)
        if(reset) begin
            counter = 4'b0000;
            clk_reg = 1'b0;
        end
        else
            if(counter == 9) begin
                counter <= 4'b0000;
                clk_reg <= ~clk_reg;    // toggle reg
            end
            else
                counter <= counter + 1;

    // Set value of i2c SCL signal to the sensor - 10kHz
    assign SCL = clk_reg;
    // *********************************************************************

    // Signal Declarations
    parameter [7:0] sensor_address_plus_read = 8'b1001_0111;// 0x97
    reg [7:0] tMSB = 8'b0;                                  // Temp data MSB
    reg [7:0] tLSB = 8'b0;                                  // Temp data LSB
    reg o_bit = 1'b1;                                       // output bit to SDA - starts HIGH
    reg [11:0] count = 12'b0;                               // State Machine Synchronizing Counter
    reg [15:0] temp_data_reg;                               // Temp data buffer register

    // State Declarations - need 28 states
    localparam [4:0] POWER_UP   = 5'h00,
                     START      = 5'h01,
                     SEND_ADDR6 = 5'h02,
                     SEND_ADDR5 = 5'h03,
                     SEND_ADDR4 = 5'h04,
                     SEND_ADDR3 = 5'h05,
                     SEND_ADDR2 = 5'h06,
                     SEND_ADDR1 = 5'h07,
                     SEND_ADDR0 = 5'h08,
                     SEND_RW    = 5'h09,
                     REC_ACK    = 5'h0A,
                     REC_MSB7   = 5'h0B,
                     REC_MSB6   = 5'h0C,
                     REC_MSB5   = 5'h0D,
                     REC_MSB4   = 5'h0E,
                     REC_MSB3   = 5'h0F,
                     REC_MSB2   = 5'h10,
                     REC_MSB1   = 5'h11,
                     REC_MSB0   = 5'h12,
                     SEND_ACK   = 5'h13,
                     REC_LSB7   = 5'h14,
                     REC_LSB6   = 5'h15,
                     REC_LSB5   = 5'h16,
                     REC_LSB4   = 5'h17,
                     REC_LSB3   = 5'h18,
                     REC_LSB2   = 5'h19,
                     REC_LSB1   = 5'h1A,
                     REC_LSB0   = 5'h1B,
                     NACK       = 5'h1C;
```

```
92
93      reg [4:0] state_reg = POWER_UP;                         // state register
94
95      always @(posedge clk_200kHz or posedge reset)
96      begin
97          if(reset) begin
98              state_reg <= START;
99              count <= 12'd2000;
100         end
101         else begin
102             count <= count + 1;
103             case(state_reg)
104                 POWER_UP    : begin
105                                 if(count == 12'd1999)
106                                     state_reg <= START;
107             end
108             START       : begin
109                                 if(count == 12'd2004)
110                                     o_bit <= 1'b0; // send START condition 1/4 clock after SCL goes high
111                                 if(count == 12'd2013)
112                                     state_reg <= SEND_ADDR6;
113             end
114             SEND_ADDR6  : begin
115                                 o_bit <= sensor_address_plus_read[7];
116                                 if(count == 12'd2033)
117                                     state_reg <= SEND_ADDR5;
118             end
119             SEND_ADDR5  : begin
120                                 o_bit <= sensor_address_plus_read[6];
121                                 if(count == 12'd2053)
122                                     state_reg <= SEND_ADDR4;
123             end
124             SEND_ADDR4  : begin
125                                 o_bit <= sensor_address_plus_read[5];
126                                 if(count == 12'd2073)
127                                     state_reg <= SEND_ADDR3;
128             end
129             SEND_ADDR3  : begin
130                                 o_bit <= sensor_address_plus_read[4];
131                                 if(count == 12'd2093)
132                                     state_reg <= SEND_ADDR2;
133             end
134             SEND_ADDR2  : begin
135                                 o_bit <= sensor_address_plus_read[3];
136                                 if(count == 12'd2113)
137                                     state_reg <= SEND_ADDR1;
138             end
139             SEND_ADDR1  : begin
140                                 o_bit <= sensor_address_plus_read[2];
141                                 if(count == 12'd2133)
142                                     state_reg <= SEND_ADDR0;
143             end
144             SEND_ADDR0  : begin
145                                 o_bit <= sensor_address_plus_read[1];
146                                 if(count == 12'd2153)
147                                     state_reg <= SEND_RW;
148             end
149             SEND_RW     : begin
150                                 o_bit <= sensor_address_plus_read[0];
151             if(count == 12'd2169)
152                                     state_reg <= REC_ACK;
153             end
154             REC_ACK     : begin
155                                 if(count == 12'd2189)
156                                     state_reg <= REC_MSB7;
157             end
158             REC_MSB7     : begin
159                                 tMSB[7] <= i_bit;
160                                 if(count == 12'd2209)
161                                     state_reg <= REC_MSB6;
162
163             end
```

```
164                 REC_MSB6     : begin
165                                   tMSB[6] <= i_bit;
166                                   if(count == 12'd2229)
167                                       state_reg <= REC_MSB5;
168
169                 end
170                 REC_MSB5     : begin
171                                   tMSB[5] <= i_bit;
172                                   if(count == 12'd2249)
173                                       state_reg <= REC_MSB4;
174
175                 end
176                 REC_MSB4     : begin
177                                   tMSB[4] <= i_bit;
178                                   if(count == 12'd2269)
179                                       state_reg <= REC_MSB3;
180
181                 end
182                 REC_MSB3     : begin
183                                   tMSB[3] <= i_bit;
184                                   if(count == 12'd2289)
185                                       state_reg <= REC_MSB2;
186
187                 end
188                 REC_MSB2     : begin
189                                   tMSB[2] <= i_bit;
190                                   if(count == 12'd2309)
191                                       state_reg <= REC_MSB1;
192
193                 end
194                 REC_MSB1     : begin
195                                   tMSB[1] <= i_bit;
196                                   if(count == 12'd2329)
197                                       state_reg <= REC_MSB0;
198
199                 end
200                 REC_MSB0     : begin
201                                   o_bit <= 1'b0;
202                                   tMSB[0] <= i_bit;
203                                   if(count == 12'd2349)
204                                       state_reg <= SEND_ACK;
205
206                 end
207                 SEND_ACK    : begin
208                                   if(count == 12'd2369)
209                                       state_reg <= REC_LSB7;
210                 end
211                 REC_LSB7     : begin
212                                   tLSB[7] <= i_bit;
213                                   if(count == 12'd2389)
214                                       state_reg <= REC_LSB6;
215                 end
216                 REC_LSB6     : begin
217                                   tLSB[6] <= i_bit;
218                                   if(count == 12'd2409)
219                                       state_reg <= REC_LSB5;
220                 end
221                 REC_LSB5     : begin
222                                   tLSB[5] <= i_bit;
223                                   if(count == 12'd2429)
224                                       state_reg <= REC_LSB4;
225                 end
226                 REC_LSB4     : begin
227                                   tLSB[4] <= i_bit;
228                                   if(count == 12'd2449)
229                                       state_reg <= REC_LSB3;
230                 end
```

```
231                 REC_LSB3    : begin
232                                   tLSB[3] <= i_bit;
233                                   if(count == 12'd2469)
234                                       state_reg <= REC_LSB2;
235                 end
236                 REC_LSB2    : begin
237                                   tLSB[2] <= i_bit;
238                                   if(count == 12'd2489)
239                                       state_reg <= REC_LSB1;
240                 end
241                 REC_LSB1    : begin
242                                   tLSB[1] <= i_bit;
243                                   if(count == 12'd2509)
244                                       state_reg <= REC_LSB0;
245                 end
246                 REC_LSB0    : begin
247                                   o_bit <= 1'b1;
248                                   tLSB[0] <= i_bit;
249                                   if(count == 12'd2529)
250                                       state_reg <= NACK;
251                 end
252                 NACK        : begin
253                                   if(count == 12'd2559) begin
254                                       count <= 12'd2000;
255                                       state_reg <= START;
256                                   end
257                 end
258             endcase
259         end
260     end
261
262     // Buffer for temperature data
263     always @(posedge clk_200kHz)
264         if(state_reg == NACK)
265             temp_data_reg <= { tMSB[7:0], tLSB[7:0] };
266
267
268     // Control direction of SDA bidirectional inout signal
269     assign SDA_dir = (state_reg == POWER_UP || state_reg == START || state_reg == SEND_ADDR6 ||
270                       state_reg == SEND_ADDR5 || state_reg == SEND_ADDR4 || state_reg == SEND_ADDR3 ||
271                       state_reg == SEND_ADDR2 || state_reg == SEND_ADDR1 || state_reg == SEND_ADDR0 ||
272                       state_reg == SEND_RW || state_reg == SEND_ACK || state_reg == NACK) ? 1 : 0;
273     // Set the value of SDA for output - from master to sensor
274     assign SDA = SDA_dir ? o_bit : 1'bz;
275     // Set value of input wire when SDA is used as an input - from sensor to master
276     assign i_bit = SDA;
277     // Outputted temperature data
278     assign temp_data = temp_data_reg;
279
280 endmodule
```

3.3 clkgen_200kHz.v

```
23  module clkgen_200kHz(
24      input clk_100MHz,    //100MHz clock
25      output clk_200kHz    //200kHz clock
26      );
27
28      // 100 x 10^6 / 200 x 10^3 / 2 = 250 <-- 8 bit counter
29      reg [7:0] counter = 8'b00000000;
30      reg clk_reg = 1'b1;
31
32      always @(posedge clk_100MHz)
33      begin
34          if(counter == 249)
35          begin
36              counter <= 8'b00000000;
37              clk_reg <= ~clk_reg;
38          end
39          else
40              counter <= counter + 1;
41      end
42
43      assign clk_200kHz = clk_reg;
44
45  endmodule
```

3.4 seg_Disp.v

```
23  module seg_Dsplay(input clk_100MHz,         // Nexys A7 clock
24      input [15:0] temp_data,                 // Temp data
25      output reg [6:0] SEG,                   // 7 Segments of Displays
26      output reg [7:0] AN,                    //Anodes
27      output reg dp                           //Decimal point
28
29      );
30
31      reg [3:0] first;  // first seven segment
32      reg [3:0] second; //second seven segment
33      reg [3:0] third; //third seven segment
34      reg [3:0] fourth; // fourth seven segment
35      reg [3:0] fifth; //fifth seven segment
36      reg [3:0] sixth; //sixth seven segment
37      reg [3:0] seventh; //seventh seven segment
38
39      reg [4:0] seg;
40      reg [17:0] Counter=0;
41      reg [12:0] temperature;
42
43
44  always@(posedge clk_100MHz)
45  begin
46      Counter<=Counter+1;
47  end
48
```

```
always @ (*)
begin
    case(seg)
    0 : SEG = 7'b1000000; //0
    1 : SEG = 7'b1111001; //1
    2 : SEG = 7'b0100100; //2
    3 : SEG = 7'b0110000; //3
    4 : SEG = 7'b0011001; //4
    5 : SEG = 7'b0010010; //5
    6 : SEG = 7'b0000010; //6
    7 : SEG = 7'b1111000; //7
    8 : SEG = 7'b0000000; //8
    9 : SEG = 7'b0011000; //9
    10: SEG = 7'b0001000; //A
    11: SEG = 7'b0000011; //B
    12: SEG = 7'b1000110; //C
    13: SEG = 7'b0100001; //D
    14: SEG = 7'b0000110; //E
    15: SEG = 7'b0111111; //-
    16: SEG = 7'b0001110; //F
    default: SEG =7'b1000000;//0
    endcase
    //conerting from 9 digit binary to decimal value
    // Starting at 8 degrees.

//Working on the decimal point values.
    temperature<=temp_data[15:3];
    if (temp_data[15]==1) //For negative temperature
        begin
           temperature<=(~((temp_data[15:3])))+1;
           seventh = 5'b01111;
           sixth = (temperature[12:4] / 10);  // Tens value of temp data
           fifth = (temperature[12:4] % 10);
    end

    else //For positive temperature
        begin
            seventh = 5'b00000;
            sixth = temperature[12:4] / 10;   // Tens value of temp data
            fifth = temperature[12:4] % 10;
        end

case (temperature[3:0])

4'b0000: begin first  <= 5'b00000;
               second <= 5'b00000;
               third  <= 5'b00000;
               fourth <= 5'b00000;end
4'b0001: begin first  <= 5'b00101;
               second <= 5'b00010;
               third  <= 5'b00110;
               fourth <= 5'b00000;end
4'b0010: begin first  <= 5'b00000;
               second <= 5'b00101;
               third  <= 5'b00010;
               fourth <= 5'b00001;end
4'b0011: begin first  <= 5'b00101;
               second <= 5'b00111;
               third  <= 5'b01000;
               fourth <= 5'b00001;end
```

```
4'b0100: begin first  <= 5'b00000;
                second <= 5'b00000;
                third  <= 5'b00101;
                fourth <= 5'b00010;end
4'b0101: begin first  <= 5'b00101;
                second <= 5'b00010;
                third  <= 5'b00001;
                fourth <= 5'b00011;end
4'b0110: begin first  <= 5'b00000;
                second <= 5'b00101;
                third  <= 5'b00111;
                fourth <= 5'b00011;end
4'b0111: begin first  <= 5'b00101;
                second <= 5'b00111;
                third  <= 5'b00011;
                fourth <= 5'b00100;end
4'b1000: begin first  <= 5'b00000;
                second <= 5'b00000;
                third  <= 5'b00000;
                fourth <= 5'b00101;end
4'b1001: begin first  <= 5'b00101;
                second <= 5'b00010;
                third  <= 5'b00110;
                fourth <= 5'b00101;end
4'b1010: begin first  <= 5'b00000;
                second <= 5'b00101;
                third  <= 5'b00010;
                fourth <= 5'b00110;end
4'b1011: begin first  <= 5'b00101;
                second <= 5'b00111;
                third  <= 5'b01000;
                fourth <= 5'b00110;end
4'b1100: begin first  <= 5'b00000;
                second <= 5'b00000;
                third  <= 5'b00101;
                fourth <= 5'b00111;end
4'b1101: begin first  <= 5'b00101;
                second <= 5'b00010;
                third  <= 5'b00001;
                fourth <= 5'b01000;end
4'b1110: begin first  <= 5'b00000;
                second <= 5'b00101;
                third  <= 5'b00111;
                fourth <= 5'b01000;end
4'b1111: begin first  <= 5'b00101;
                second <= 5'b00111;
                third  <= 5'b00011;
                fourth <= 5'b01001;end
default: begin first  <= 5'b00000;
                second <= 5'b00000;
                third  <= 5'b00000;
                fourth <= 5'b00000;end
   endcase


end
```

```
165 
166 always @ (*)
167 begin
168 
169     case (Counter[17:15])
170 
171     3'b000: begin
172                 seg <= first;
173                 AN <= 8'b11111110;
174                 dp <= 1'b1;
175             end
176     3'b001: begin
177                 seg <= second;
178                 AN <= 8'b11111101;
179                 dp <= 1'b1;
180             end
181     3'b010: begin
182                 seg <= third;
183                 AN <= 8'b11111011;
184                 dp <= 1'b1;
185             end
186     3'b011: begin
187                 seg <= fourth;
188                 AN <= 8'b11110111;
189                 dp <= 1'b1;
190             end
191     3'b100: begin
192                 seg <= fifth;
193                 AN <= 8'b11101111;
194                 dp <= 1'b0;
195           end
196     3'b101: begin
197                 seg <= sixth;
198                 AN <= 8'b11011111;
199                 dp <= 1'b1;
200           end
201     3'b110: begin
202                 seg <= seventh;
203                 AN <= 8'b10111111;
204                 dp <= 1'b1;
205           end
206     default: begin AN <=8'b11111111; dp <= 1'b1;end
207 
208     endcase
209 end
210 endmodule
```

3.5 led_intensity.v

```
module clrcode_intensity(
    input clk_100MHz,    // Nexys A7 clock
    input [15:0] temp_data,  // temperature data
    output reg R, //for Tricolor LED
    output reg G, //for Tricolor LED
    output reg B //for Tricolor LED

);

//8 bit pwm counter
wire [7:0] pwm_count;
//3 bit RGB_Color
//reg [2:0] RGB_Color;
//reg [12:0] temperature;
reg signed [12:0] temperature;

always @(posedge clk_100MHz)
begin

    if (temp_data[15]==1) //For negative temperature
        begin
         temperature<=(((temp_data[15:3])-8192)/16);
        end

    else                //For positive temperature
        begin
            temperature<=(temp_data[15:3])/16;
    end

//For Color Coding and variation in Intensity
   if(-13'sd20<=temperature && temperature<=13'sd10) //For range of temperature [-20,10] Cold temperature
               begin
                    if(-13'sd20<=temperature && temperature<=-13'sd0) //High intensity for temperature [-20,0]
                        begin
                            if (pwm_count<90)  //90% duty cycle with blue color
                                begin
                                     R <= 1'b0;
                                     G <= 1'b0;
                                     B <= 1'b1;
                                end
                            else
                                 begin
                                      R<=0;
                                      G<=0;
                                      B<=0;
                                end
                     end
                    else if(13'sd0<temperature && temperature<=13'sd10) //Low intensity for temperature(0,10]
                    begin
                      if (pwm_count<10)  //10% duty cycle with magenta color
                            begin
                                R <= 1'b1;
                                G <= 1'b0;
                                B <= 1'b1;
                            end
                       else
                            begin
                                R<=0;
                                G<=0;
                                B<=0;
                            end
                end
                else
                     begin
                     end
                end
```

```
89
90      else if(10<temperature && temperature<=30) //For range of temperature (10,30] Normal temperature
91          begin
92              if(10<temperature && temperature<=20) //Low intensity for temperature (10,20]
93                  begin
94                       if (pwm_count<10) //10% duty cycle with cyan color
95                           begin
96                             R <= 1'b0;
97                             G <= 1'b1;
98                             B <= 1'b1;
99                           end
100                      else
101                          begin
102                            R<=0;
103                            G<=0;
104                            B<=0;
105                          end
106                 end
107             else if(20<temperature && temperature<=30)//High intensity for temperature (20,30]
108                     begin
109                     if (pwm_count<90) //90% duty cycle with green color
110                        begin
111                            R <= 1'b0;
112                            G <= 1'b1;
113                            B <= 1'b0;
114
115                        end
116                     else
117                       begin
118                           R<=0;
119                           G<=0;
120                           B<=0;
121                       end
122                 end
123         else
124           begin
125           end
126     end
127
128 else if(30<temperature && temperature<=50) //For range of temperature (30,50] Hot temperature with Red color
129     begin
130             if(30<temperature && temperature<=40) //Low intensity for temperature (30,40]
131                begin
132                    if (pwm_count<10) //10% duty cycle with yellow color
133                           begin
134                               R <= 1'b1;
135                               G <= 1'b1;
136                               B <= 1'b0;
137
138                           end
139                    else
140                           begin
141                               R<=0;
142                               G<=0;
143                               B<=0;
144                           end
145                end
146                else if (40<temperature && temperature<=50) //High intensity for temperature (40,50]
147                begin
148                   if (pwm_count<90)//90% duty cycle with red color
149                           begin
150                               R <= 1'b1;
151                               G <= 1'b0;
152                               B <= 1'b0;
153
154                           end
155                    else
156                           begin
157                               R<=0;
158                               G<=0;
159                               B<=0;
160                           end
161                end
162             else
163                begin
164                end
165     end
```

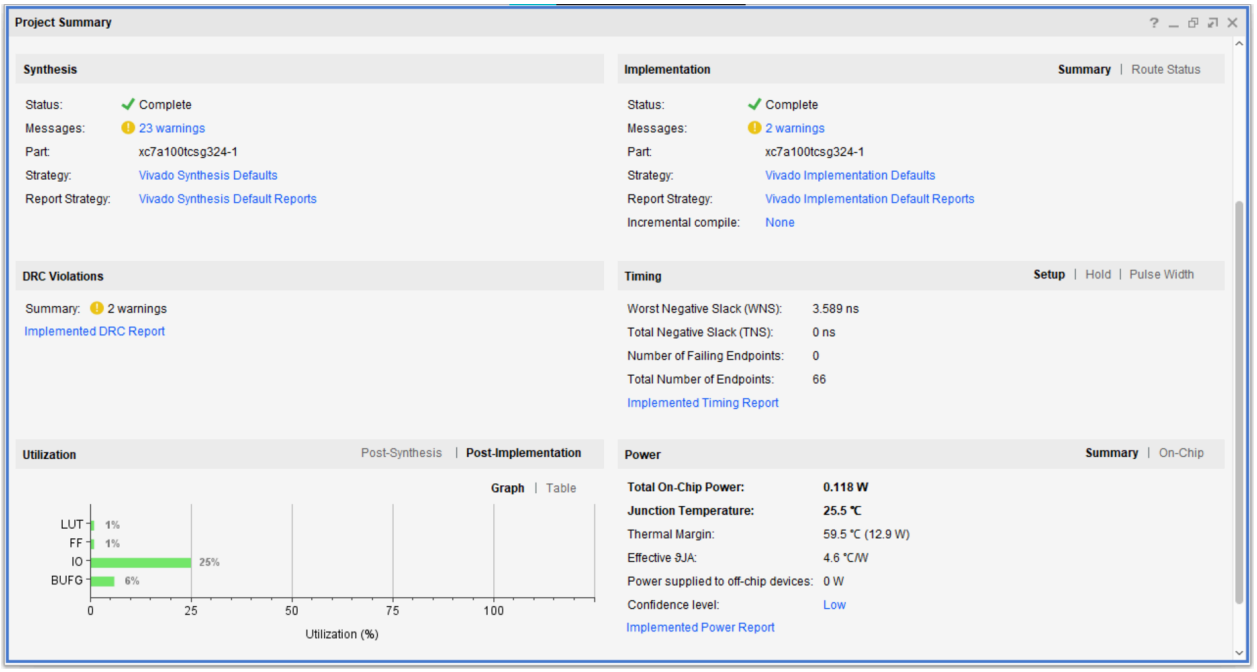
```
166      else
167         begin
168                  R<=0;
169                  G<=0;
170                  B<=0;
171         end
172  end
173
174  //instance for pwm
175  pwm inst_pwm(clk_100MHz,pwm_count);
176
177  endmodule
```

3.6 pwm.v

```
23  module pwm(
24  input clk_100MHz,
25  output reg [7:0] pwm_count
26  );
27
28  reg [7:0] pwm_count=0;
29      always @(posedge clk_100MHz )
30        begin
31              if (pwm_count < 100) pwm_count<=pwm_count+1; //count until 100
32              else pwm_count <=0; //reset counter
33        end
34
35  endmodule
```

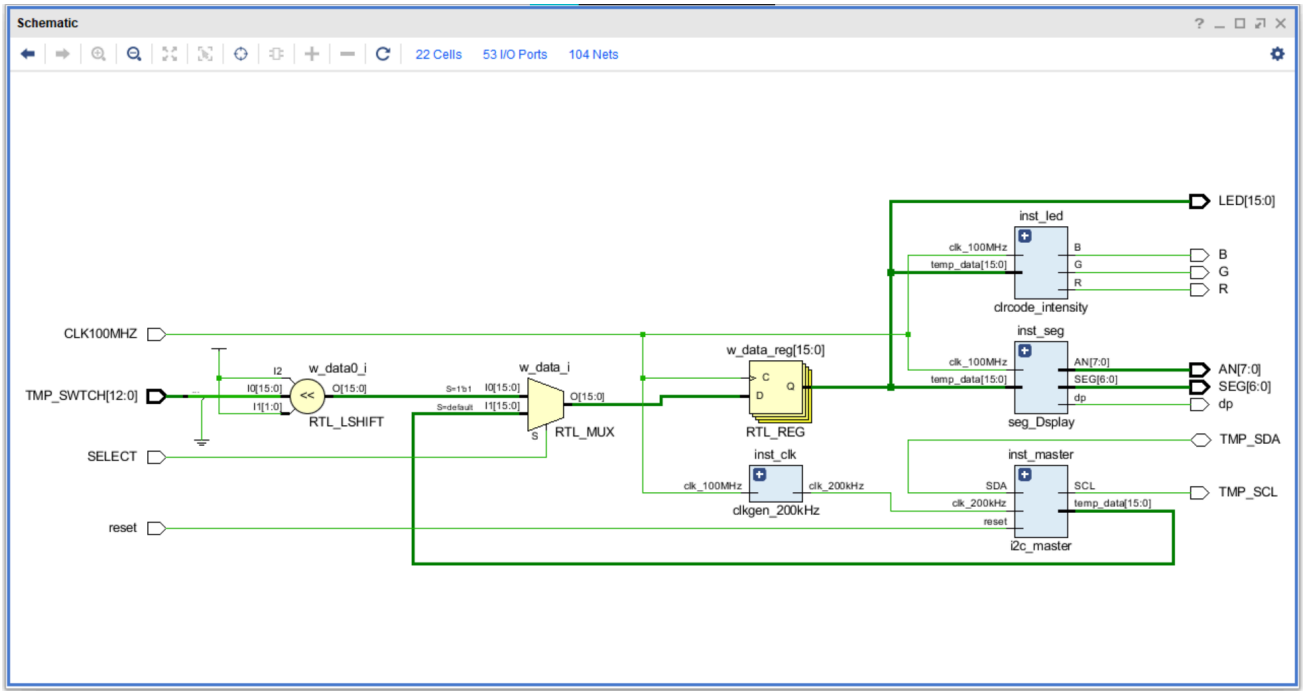
4. Resource Utilization

The resource utilization for each module is dependent on the Nexys A7. It includes utilization of LUTs, Flip Flops, and other resources. The Vivado synthesis report for detailed resource utilization is as follows:



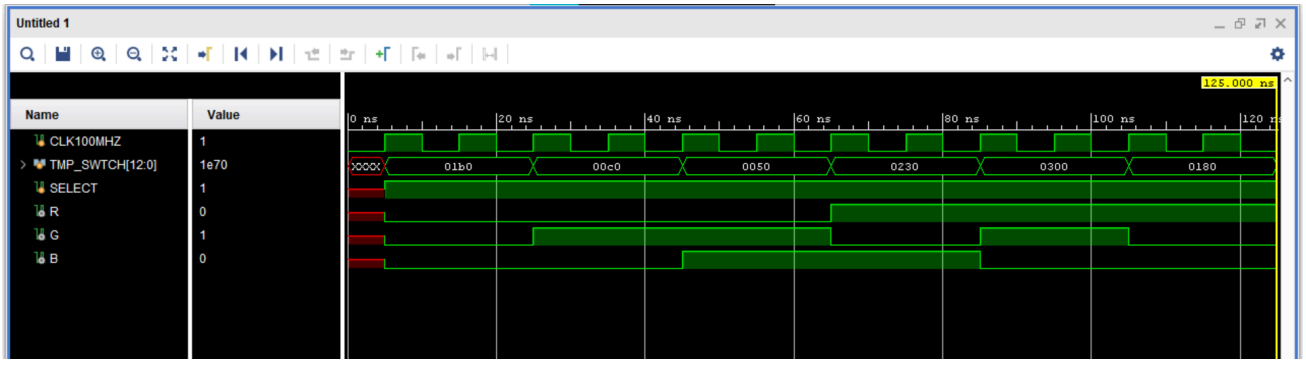
5. RTL Schematic

The RTL schematic for each module is generated during the synthesis process using Vivado. The synthesized RTL schematic for a detailed visualization of the design is as follows:



6. Simulation Results

Waveforms depicting correct communication, LED intensity changes, and seven-segment display values should be included.



7. Result Analysis

7.1 Simulation Results Overview

The simulation results indicate successful functionality of the designed system. The main module orchestrates the integration of accelerometer and temperature sensor data, successfully driving the seven-segment displays and tri-color LED based on the specified conditions. During simulations, the PWM-based intensity control accurately reflects temperature variations, and the seven-segment displays correctly visualize the temperature values.

7.2 Performance Metrics

Timing Analysis: The design meets timing requirements, ensuring proper synchronization of signals and reliable operation.

Resource Utilization: The Vivado synthesis report demonstrates efficient utilization of FPGA resources, meeting the design constraints.

7.3 Issues and Debugging

No critical issues were identified during simulation. The design also worked accurately on the FPGA.

8. Applications

8.1 Environmental Monitoring

The designed system is well-suited for environmental monitoring applications. By incorporating both accelerometer and temperature sensor data, it can be deployed in settings where changes in motion and temperature need to be visually communicated, such as in remote weather stations or environmental monitoring systems.

8.2 Industrial Automation

In industrial automation scenarios, the system can be integrated into machinery or equipment to provide real-time feedback on both temperature conditions and any unusual motion or vibrations. This can contribute to predictive maintenance strategies, enhancing overall system reliability.

8.3 Embedded Systems

For embedded systems requiring intuitive visual feedback, the design offers a compact solution. It can be integrated into embedded platforms for applications such as robotics, where the tri-color LED can signify critical operating states based on temperature and motion data.

8.4 Educational Use

The system can serve as an educational tool for students studying digital systems and FPGA-based designs. It provides a tangible example of integrating sensors, communication protocols, and visual feedback components into a cohesive digital system.