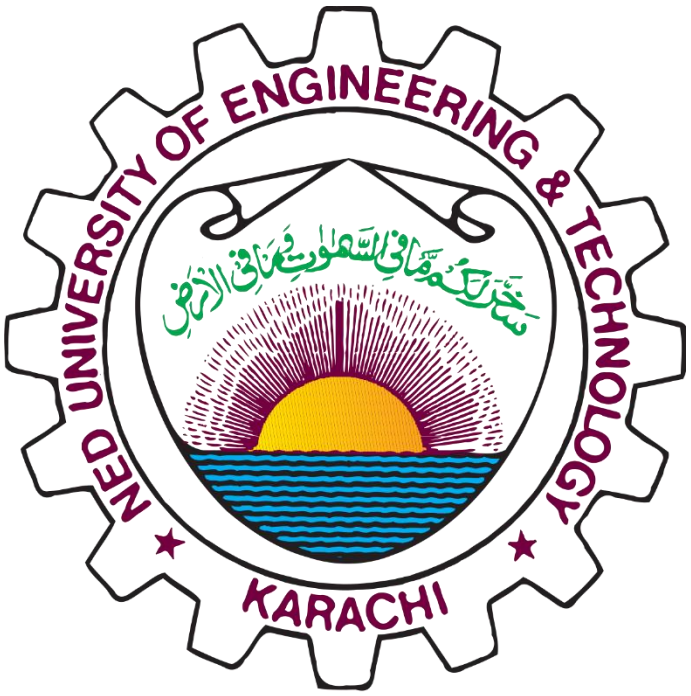
DIGITAL SYSTEM DESIGN

CEP



Temperature Sensor with PWM Report

Aiman	CS-19107
Muhammad Umar	CS-19126
Mir Shahnawaz Khan Abbasi	CS-19131
Shakeel Ahmad	CS-19148
Alauddin Taha Faya	CS-19305

Submitted to: Ms. Syeda Ramish Fatima

Introduction

This report is on a temperature monitoring system that utilizes an **FPGA** with a built-in temperature sensor (**ADT7420**) to access temperature data. The system uses the **I2C** protocol to communicate with the sensor. The system also allows for user input from **switches**, which when asserted, will display temperature data on **7 segment** displays. Additionally, the system incorporates a Pulse Width Modulation (**PWM**) technique to control the intensity of a tri-color **LED**, which changes color based on the temperature register values. If the temperature is greater than 30 degrees, the red light will glow with a 100% duty cycle, if the temperature is between 30 degrees and 0 degrees, the blue light will glow with a 50% duty cycle, and if the temperature is negative, the green light will glow with a 25% duty cycle.

Design Methodology

1. **Requirements**: Field Programmable Gate Array (**FPGA**) board having built-in temperature sensor **(ADT7420)** to access temperature data and the system uses the **I2C** protocol to communicate with the sensor and allows user for input through switches.

2. **System Design**: This part is having a module for separate functionalities like I2C Protocol module, Seven Segment module, Intensity of RGB light, Clock generation module and a Top module having all the modules initiated. The system also incorporates Pulse Width Modulation (**PWM**) technique to control the intensity of a tri-color LED, which changes color based on the temperature register values.

3. **Synthesis of Design**: The system will then synthesize by Vivado built-in Synthesis tool so that if there are any critical warnings related to the system it will indicate and also provide some suggestions to rectify it. The design is analyzed and optimized for the target FPGA device, and the resulting netlist is used to configure the FPGA with the desired logic. This process is done by the synthesis tool in Vivado, which takes the high-level design descriptions as input, and generates a gate-level netlist as output.

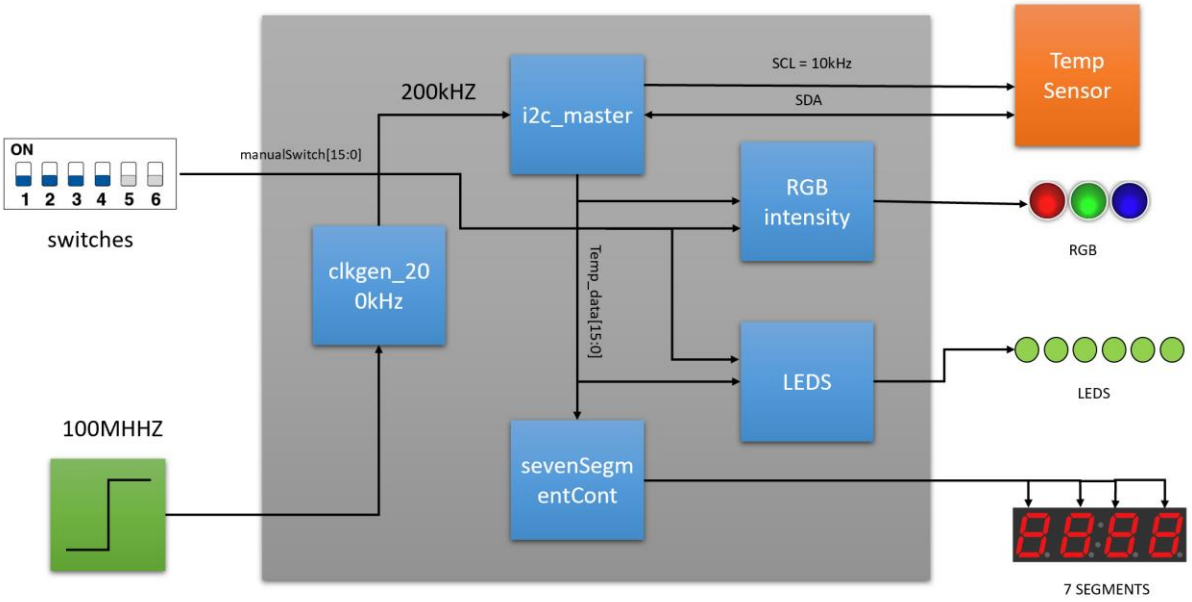
4. **Implementation of Design**: The process of taking the synthesized design and configuring the target FPGA device with the desired logic. This process includes several steps such as:

   Place and Route (P&R): The P&R step is responsible for mapping the logic of the design to the physical resources of the FPGA, including the programmable logic elements (PLEs), input/output (I/O) resources, and routing resources.

   Bitstream Generation: Once the design has been placed and routed, the implementation step generates a bitstream, which is a binary file that configures the FPGA with the desired logic.

   Generating the programming file: Once all the above steps are done, the implementation step generates the programming file that can be used to program the FPGA.

5. **Testing in Real World**: After the FPGA is programmed with .bit file, Now we have to look and observe either it is giving correct result with good accuracy then our Design Methodologies will be achieved.



Resource Utilization

Look Up Tables: (LUTs)

LUTs are digital logic circuits that are used to implement Boolean functions in an FPGA. They are the building blocks of an FPGA and can be used to implement any Boolean function of a fixed number of inputs. They are programmable lookup tables that can be configured to implement different Boolean functions by loading the corresponding truth table into its memory. Temperature sensor system is utilizing 267 LUTs out of 63400 which makes 0.42%.

Flip Flops: (FFs)

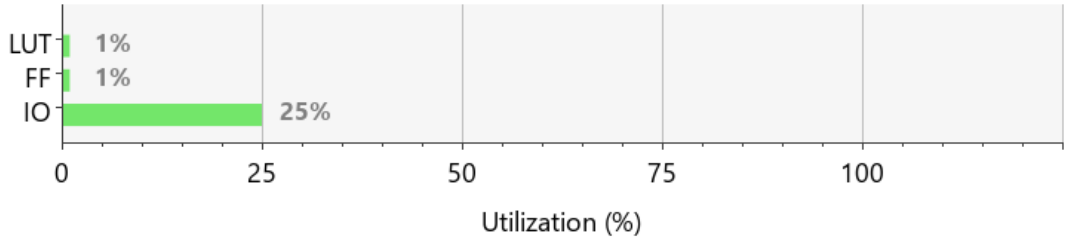
Flip-flops are digital circuits that are used to implement sequential logic in FPGAs. They are used to store one bit of data and have two stable states. They respond to a change in inputs (clocking) to change their state. Our design is utilizing 90 out of 126800 which makes 0.0.7%.

I/O Module

I/O module in an FPGA is responsible for communicating with external devices, it provides a interface between the FPGA and the external devices. The I/O module typically includes I/O pads, I/O buffers, and I/O cells. These components are used to connect the FPGA to the external devices, convert the signals from the external devices into a form that can be understood by the FPGA, and implement the I/O functionality. Our design is using 53 out 210 available I/O module which makes a total of 25.24%.

Summary

Resource	Utilization	Available	Utilization %
LUT	267	63400	0.42
FF	90	126800	0.07
IO	53	210	25.24

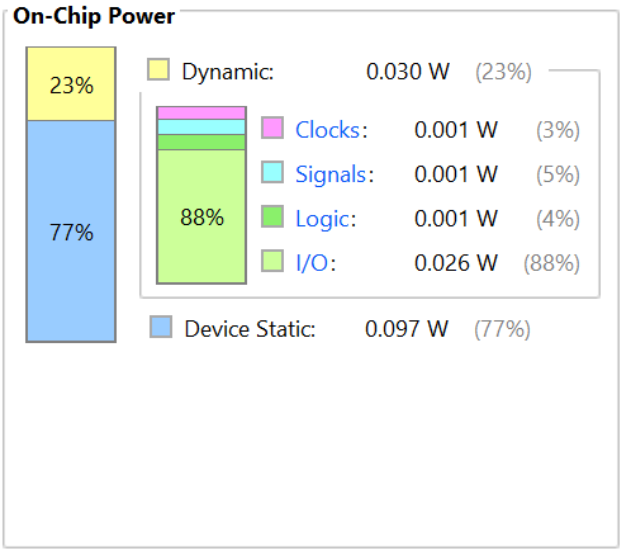


Summary

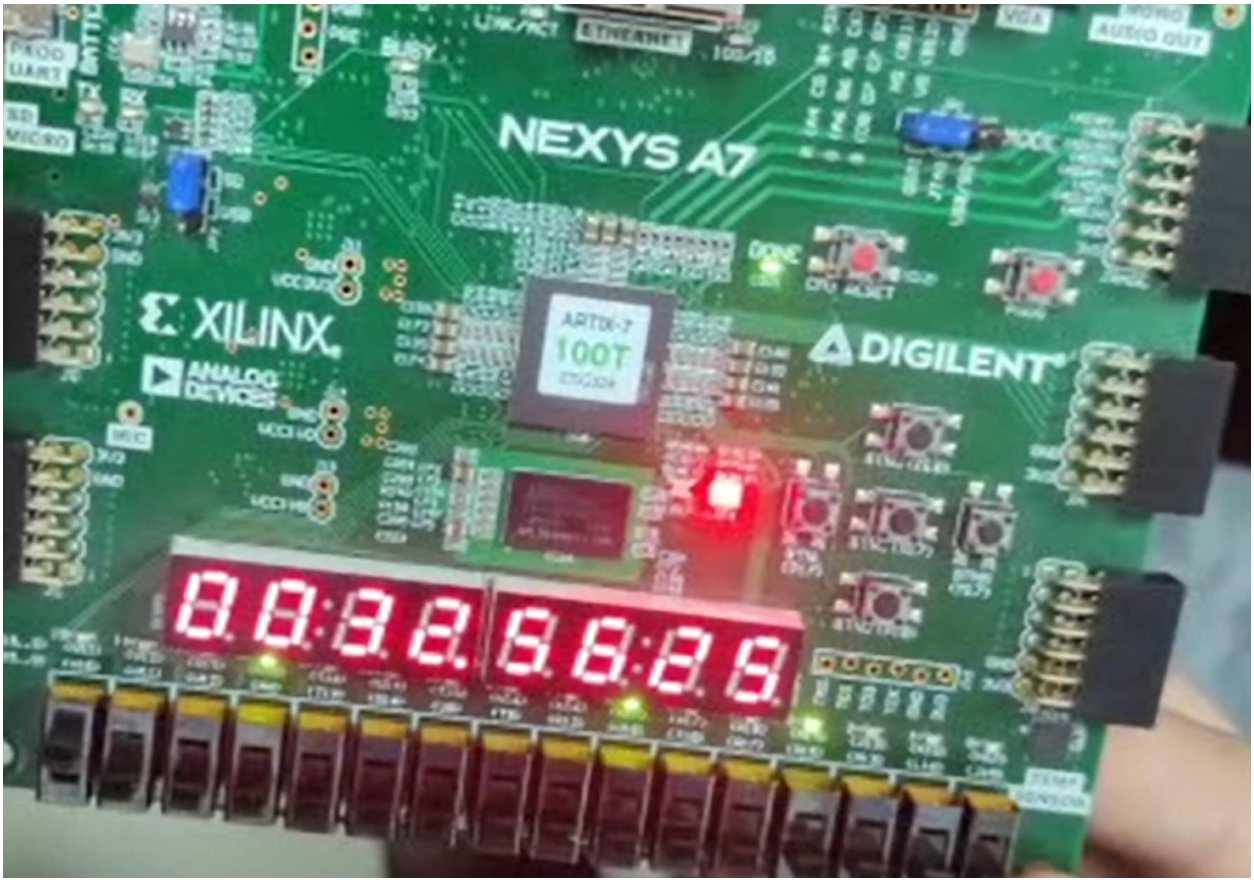
Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.127 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**25.6°C**
Thermal Margin:	59.4°C (12.9 W)
Effective ϑJA:	4.6°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity



Result analysis



Applications

Some of the potential applications include:

1. Industrial settings: The system can be used to monitor temperature in industrial environments, such as factories or power plants, to ensure that the temperature remains within safe levels.
2. Temperature-sensitive environments: The system can be used to monitor temperature in environments where temperature is critical, such as in laboratories, data centers, or museums.
3. Building automation: The system can be used to monitor temperature in buildings, such as offices, hotels, or residential buildings, for the purpose of HVAC control, energy management and also introduced with intensity of tri-color LED so that you can observe the temperature through the defined intensity of LEDs.

4. Medical equipment: The system can be used to monitor temperature in medical equipment, such as incubators or blood refrigerators, to ensure that the temperature remains within safe levels.

Verilog Source Code

Module Top.v

```
module top(
  input       CLK100MHZ,      // nexys clk signal
  input       enable,
  input       [12:0]manualSwitch,
  input       reset,          // btnC on nexys
  inout       TMP_SDA,        // i2c sda on temp sensor - bidirectional
  output      TMP_SCL,        // i2c scl on temp sensor
  output [6:0]  CA,           // 7 segments of each display
  output [7:0]  AN,           // 4 anodes of 4 displays
  output dp,          // 4 anodes always OFF
  output R,G,B,
  output [15:0]  LED          // nexys leds = binary temp in deg C
  );

  wire sda_dir;               // direction of SDA signal - to or from master
  wire w_200kHz;              // 200kHz SCL
  wire [15:0] w_data;         // 8 bits of temperature data

  // Instantiate i2c master
  i2c_master master(
    .clk_200kHz(w_200kHz),
    .reset(reset),
    .temp_data(w_data),
    .SDA(TMP_SDA),
    .SDA_dir(sda_dir),
    .SCL(TMP_SCL)
  );

  // Instantiate 200kHz clock generator
  clkgen_200kHz cgen(
    .clk_100MHz(CLK100MHZ),
    .clk_200kHz(w_200kHz)
  );

  // Instantiate 7 segment control
```

```
   sevenSegmentCont   S1(
   .clk_100MHz(CLK100MHZ),            //100MHZ BUILITIN CLOCK
   .enable(enable),                   //if 0 input will come from tempreture senson otherwise from
switches
   .manualSwitch(manualSwitch),       // 13-bit input from swithches
   .reset(reset),
   .temp_data(w_data),        // 16-bit tempreture data coming from I2C master
   .CA(CA),         // 7 Segments of Displays(cathodes) used to show digits
   .AN(AN),          // for enabling the segments
   .dp(dp)
   );


   // instantiate LED module

   LEDs    l1(
   .enable(enable),
   .manualSwitch(manualSwitch),
   .temp_data(w_data),
   .LED (LED)
   );
   // instantiate RGB INTENSITY MODULE
   RGB_intensity   RGB(
   .clk_100MHz(CLK100MHZ),
   .enable(enable),
   .manualSwitch(manualSwitch),
   .temp_data(w_data),
   .R(R),
   .G(G),
   .B(B)
   );
endmodule
```

Clock Generation Module clkgen_200KHz.v

```
module clkgen_200kHz(
   input clk_100MHz,
   output clk_200kHz
   );
   // 100 x 10^6 / 200 x 10^3 / 2 = 250 <-- 8 bit counter
   reg [7:0] counter = 8'h00;
   reg clk_reg = 1'b1;

   always @(posedge clk_100MHz) begin
```

```
    if(counter == 249) begin
      counter <= 8'h00;
      clk_reg <= ~clk_reg;
    end
    else
      counter <= counter + 1;
  end

  assign clk_200kHz = clk_reg;

endmodule
```

I2C Protocol Module i2c.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'b0;  // count up to 9
  reg clk_reg = 1'b1;

  always @(posedge clk_200kHz or posedge reset)
    if(reset) begin
      counter = 4'b0;
      clk_reg = 1'b0;
    end
    else
      if(counter == 9) begin
        counter <= 4'b0;
        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;

        reg [4:0] state_reg = POWER_UP;             // state register

        always @(posedge clk_200kHz or posedge reset) begin
          if(reset) begin
            state_reg <= START;
```

```
                        count <= 12'd2000;
    end
    else begin
                        count <= count + 1;
      case(state_reg)
        POWER_UP    : begin
                    if(count == 12'd1999)
                      state_reg <= START;
        end
        START      : begin
                    if(count == 12'd2004)
                      o_bit <= 1'b0;        // send START condition 1/4 clock after SCL goes high
                    if(count == 12'd2013)
                      state_reg <= SEND_ADDR6;
        end
        SEND_ADDR6  : begin
                    o_bit <= sensor_address_plus_read[7];
                    if(count == 12'd2033)
                      state_reg <= SEND_ADDR5;
        end
                                SEND_ADDR5  : begin
                    o_bit <= sensor_address_plus_read[6];
                    if(count == 12'd2053)
                      state_reg <= SEND_ADDR4;
        end
                                SEND_ADDR4  : begin
                    o_bit <= sensor_address_plus_read[5];
                    if(count == 12'd2073)
                      state_reg <= SEND_ADDR3;
        end
                                SEND_ADDR3  : begin
                    o_bit <= sensor_address_plus_read[4];
                    if(count == 12'd2093)
                      state_reg <= SEND_ADDR2;
        end
                                SEND_ADDR2  : begin
                    o_bit <= sensor_address_plus_read[3];
                    if(count == 12'd2113)
                      state_reg <= SEND_ADDR1;
        end
                                SEND_ADDR1  : begin
                    o_bit <= sensor_address_plus_read[2];
                    if(count == 12'd2133)
                      state_reg <= SEND_ADDR0;
```

```
end
                        SEND_ADDR0  : begin
          o_bit <= sensor_address_plus_read[1];
          if(count == 12'd2153)
            state_reg <= SEND_RW;
end
                        SEND_RW     : begin
          o_bit <= sensor_address_plus_read[0];
                        if(count == 12'd2169)
            state_reg <= REC_ACK;
end
REC_ACK     : begin
          if(count == 12'd2189)
            state_reg <= REC_MSB7;
end
REC_MSB7    : begin
          tMSB[7] <= i_bit;
          if(count == 12'd2209)
            state_reg <= REC_MSB6;

end
                        REC_MSB6    : begin
          tMSB[6] <= i_bit;
          if(count == 12'd2229)
            state_reg <= REC_MSB5;

end
                        REC_MSB5    : begin
          tMSB[5] <= i_bit;
          if(count == 12'd2249)
            state_reg <= REC_MSB4;

end
                        REC_MSB4    : begin
          tMSB[4] <= i_bit;
          if(count == 12'd2269)
            state_reg <= REC_MSB3;

end
                        REC_MSB3    : begin
          tMSB[3] <= i_bit;
          if(count == 12'd2289)
            state_reg <= REC_MSB2;
```

```
end
                        REC_MSB2    : begin
        tMSB[2] <= i_bit;
        if(count == 12'd2309)
          state_reg <= REC_MSB1;

end
                        REC_MSB1    : begin
        tMSB[1] <= i_bit;
        if(count == 12'd2329)
          state_reg <= REC_MSB0;

end
                        REC_MSB0    : begin
                                                    o_bit <= 1'b0;
        tMSB[0] <= i_bit;
        if(count == 12'd2349)
          state_reg <= SEND_ACK;

end
SEND_ACK   : begin
        if(count == 12'd2369)
          state_reg <= REC_LSB7;
end
REC_LSB7   : begin
        tLSB[7] <= i_bit;
        if(count == 12'd2389)
                                                state_reg <= REC_LSB6;
end
REC_LSB6   : begin
        tLSB[6] <= i_bit;
        if(count == 12'd2409)
                                                state_reg <= REC_LSB5;
end
                        REC_LSB5    : begin
        tLSB[5] <= i_bit;
        if(count == 12'd2429)
                                                state_reg <= REC_LSB4;
end
                        REC_LSB4    : begin
        tLSB[4] <= i_bit;
        if(count == 12'd2449)
                                                state_reg <= REC_LSB3;
end
```

```
                        REC_LSB3    : begin
                tLSB[3] <= i_bit;
                if(count == 12'd2469)
                                                        state_reg <= REC_LSB2;
            end
                        REC_LSB2    : begin
                tLSB[2] <= i_bit;
                if(count == 12'd2489)
                                                        state_reg <= REC_LSB1;
            end
                        REC_LSB1    : begin
                tLSB[1] <= i_bit;
                if(count == 12'd2509)
                                                        state_reg <= REC_LSB0;
            end
                        REC_LSB0    : begin
                                                    o_bit <= 1'b1;
                tLSB[0] <= i_bit;
                if(count == 12'd2529)
                                                        state_reg <= NACK;
            end
            NACK      : begin
                if(count == 12'd2559) begin
                                                        count <= 12'd2000;
                   state_reg <= START;
                                                    end
            end
          endcase
        end
      end

      // Buffer for temperature data
      always @(posedge clk_200kHz)
        if(state_reg == NACK)
          temp_data_reg <= { tMSB[7:0], tLSB[7:0] };


      // Control direction of SDA bidirectional inout signal
      assign SDA_dir = (state_reg == POWER_UP || state_reg == START || state_reg == SEND_ADDR6
|| state_reg == SEND_ADDR5 ||
                                        state_reg == SEND_ADDR4 || state_reg ==
SEND_ADDR3 || state_reg == SEND_ADDR2 || state_reg == SEND_ADDR1 ||
                state_reg == SEND_ADDR0 || state_reg == SEND_RW || state_reg == SEND_ACK ||
state_reg == NACK) ? 1 : 0;
```

```
    // Set the value of SDA for output - from master to sensor
    assign SDA = SDA_dir ? o_bit : 1'bz;
    // Set value of input wire when SDA is used as an input - from sensor to master
    assign i_bit = SDA;
    // Outputted temperature data
    assign temp_data = temp_data_reg;

endmodule
```

Seven Segment Module sevenSegmentCont.V

```
module sevenSegmentCont(input clk_100MHz,              //100MHZ BUILITIN CLOCK
    input enable,
//if 0 input will come from tempreture sensor otherwise from switches
    input [12:0]manualSwitch,          // 13-bit input from swithches
    input reset,
    input [15:0] temp_data,         // 16-bit tempreture data coming from I2C master
    output reg [6:0] CA,           // 7 Segments of Displays(cathodes) used to show digits
    output reg [7:0] AN,           // for enabling the segments
    output reg dp
    );
    reg [4:0] first=0;  // first seven segment
    reg [4:0] second=0; //second seven segment
    reg [4:0] third=0; //third seven segment
    reg [4:0] fourth=0; // fourth seven segment
    reg [4:0] fifth=0; //fifth seven segment
    reg [4:0] sixth=0; //sixth seven segment
    reg [4:0] seventh=0; //seventh seven segment
    reg [4:0] eighth=0; //eighth seven segment
    reg[4:0]switch=0;    // temporary register for storing the input values
    reg [17:0] Counter=0; //18 bit counter
    reg [15:0] temp_register=0


    always@(posedge clk_100MHz or posedge reset)
    begin
    if(reset)
       Counter <= 0;
    else
    Counter<=Counter+1;
    end

    always@(*)
     begin
     if(enable)
```

```
            begin
              temp_register[15:0] <= temp_data[15:0];
              if(temp_register[15] == 0)
              begin                          // logic for display positive tempreture
                sixth <= temp_register[14:7] / 10;          // Tens value of temp data
                fifth <= temp_register[14:7] % 10;          // Ones value of temp data
              end
              else if(temp_register[15] == 1)
              begin
                temp_register[15:3] =  (~(temp_data[15:3])+1);
            //logic for display negative temperature
                sixth <= (temp_register[14:7]) / 10;          // Tens value of temp data
                fifth <= (temp_register[14:7]) % 10;          // Ones value of temp data
                seventh <= 5'b10000;
              end

            end
            else if(enable == 0)
            begin
              temp_register[15:3] = manualSwitch[12:0];
              if(temp_register[15] == 0)
              begin                                  //logic for display positive dummy tempreture coming
from switches
                seventh = (temp_register[14:7] >= 100)?(temp_register[14:7] / 100): 5'bxxxxx;              //
hundreds value of dummy tempreture coming from switches
                sixth = (temp_register[14:7] >= 10)?((temp_register[14:7] % 100) / 10):5'bxxxxx;          //
Tens value of dummy tempreture coming from switches
                fifth = temp_register[14:7] % 10;          // Ones value of dummy tempreture coming from
switches
              end
              else if(temp_register[15] == 1)
              begin
                temp_register[15:3] =  (~(manualSwitch[12:0])+1);                                    //logic
for display negative dummy tempreture coming from switches
                seventh       =       ((temp_register[14:7])       >=       100)?((temp_register[14:7])       /
100):(((temp_register[14:7]) >= 10)?5'b10000:5'bxxxxx);          // hundreds value of dummy tempreture
coming from switches
                sixth       =       ((temp_register[14:7])       >=       10)?(((temp_register[14:7])%100)       /
10):(((temp_register[14:7]) >= 0)?5'b10000:5'bxxxxx);        // Tens value of temp data
                fifth = ((temp_register[14:7])%100) % 10;          // Ones value of temp data
                eighth = ((temp_register[14:7])>= 100)?5'b10000:5'bxxxxx;
              end

            end
```

```
 case(switch)
0 : CA = 7'b1000000; //0
1 : CA = 7'b1111001; //1
2 : CA = 7'b0100100; //2
3 : CA = 7'b0110000; //3
4 : CA = 7'b0011001; //4
5 : CA = 7'b0010010; //5
6 : CA = 7'b0000010; //6
7 : CA = 7'b1111000; //7
8 : CA = 7'b0000000; //8
9 : CA = 7'b0011000; //9
10: CA = 7'b0001000; //A
11: CA = 7'b0000011; //B
12: CA = 7'b1000110; //C
13: CA = 7'b0100001; //D
14: CA = 7'b0000110; //E
15: CA = 7'b0001110; //F
16: CA = 7'b0111111; // DASH (NEGATIVE SIGN)
default: CA = 7'b1111111;
endcase

case (temp_register[6:3])            // logic for fractional part using lookup tables
4'b0000: begin first  <= 4'b00000;
          second <= 4'b00000;
          third  <= 4'b00000;
          fourth <= 4'b00000;end
4'b0001: begin first  <= 4'b00101;
          second <= 4'b00010;
          third  <= 4'b00110;
          fourth <= 4'b00000;end
4'b0010: begin first  <= 4'b00000;
          second <= 4'b00101;
          third  <= 4'b00010;
          fourth <= 4'b00001;end
4'b0011: begin first  <= 4'b00101;
          second <= 4'b00111;
          third  <= 4'b01000;
          fourth <= 4'b00001;end
4'b0100: begin first  <= 4'b00000;
          second <= 4'b00000;
          third  <= 4'b00101;
          fourth <= 4'b00010;end
4'b0101: begin first  <= 4'b00101;
```

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

```
          second <= 4'b00000;
          third  <= 4'b00000;
          fourth <= 4'b00000;end
endcase

case (Counter[17:15])

 3'b000: begin
          switch <= first;
          AN <= 8'b11111110;
          dp <= 1'b1;
        end
 3'b001: begin
          switch <= second;
          AN <= 8'b11111101;
          dp <= 1'b1;
        end
 3'b010: begin
          switch <= third;
          AN <= 8'b11111011;
          dp <= 1'b1;
        end
 3'b011: begin
          switch <= fourth;
          AN <= 8'b11110111;
          dp <= 1'b1;
        end
 3'b100: begin
          switch <= fifth;
          AN <= 8'b11101111;
          dp <= 1'b0;
       end
 3'b101: begin
          switch <= sixth;
          AN <= 8'b11011111;
          dp <= 1'b1;
       end
 3'b110: begin
          switch <= seventh;
          AN <= 8'b10111111;
          dp <= 1'b1;
       end
 3'b111: begin
          switch <= eighth;
```

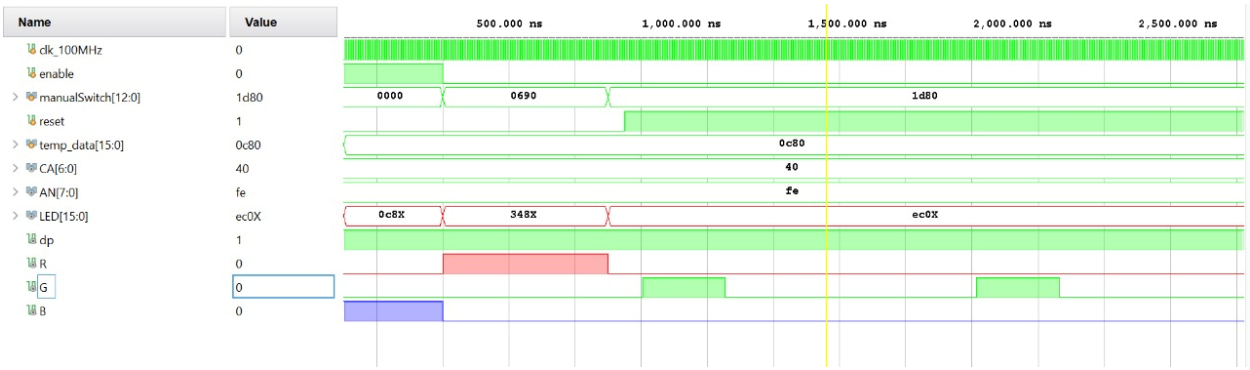
```
                AN <= 8'b10111111;
                dp <= 1'b1;
            end
        default: begin AN <=8'b11111111; dp <= 1'b1;end
        endcase
    end
endmodule
```

LED Module

```
module LEDs(
  input enable,
  input [12:0]manualSwitch,
  input [15:0] temp_data,
  output reg[15:0]LED );

  always@(*)begin
  if(enable)
     LED[15:3] = temp_data[15:3];
  else if(enable == 0)
     LED[15:3] = manualSwitch[12:0];
  end
endmodule
```

Simulation Results



RTL Schematic