1. The maximum speed for running SCL signal is 400 kHz.
2. To obtain multiple temperature values from the sensor, I will create a while loop in my python code such that it will send a go signal every second by using time.sleep().
3. The minimum value it can record is -256 Celsius and the maximum value it can record is 255 Celsius. (However, it will not be accurate when the temperature is above 150 Celsius or below -40 Celsius.
4. The resolution of the sensor is 0.0625 Celsius when operating in 13-bits and the resolution is 0.0078 Celsius when operating in 16-bits.

Milestone1:

JTEG_Test_File.v:

```
`timescale 1ns / 1ps


module JTEG_Test_File(
   output [7:0] led,
   input sys_clkn,
   input sys_clkp,
   output ADT7420_A0,
   output ADT7420_A1,
   output I2C_SCL_0,
   inout I2C_SDA_0,
   input  [4:0] okUH,
   output [2:0] okHU,
   inout  [31:0] okUHU,
   inout  okAA
);
```

```
wire  ILA_Clk, ACK_bit, FSM_Clk, TrigerEvent;
wire [23:0] ClkDivThreshold = 1_000;
wire SCL, SDA;
wire [7:0] State;
wire [31:0] PC_control;

assign TrigerEvent = PC_control[0];

//Instantiate the module that we like to test
I2C_Transmit I2C_Test1 (
    .led(led),
    .sys_clkn(sys_clkn),
    .sys_clkp(sys_clkp),
    .ADT7420_A0(ADT7420_A0),
    .ADT7420_A1(ADT7420_A1),
    .I2C_SCL_0(I2C_SCL_0),
    .I2C_SDA_0(I2C_SDA_0),
    .FSM_Clk_reg(FSM_Clk),
    .ILA_Clk_reg(ILA_Clk),
    .ACK_bit(ACK_bit),
    .SCL(SCL),
    .SDA(SDA),
    .State(State),
    .PC_control(PC_control),
```

```
        .okUH(okUH),
        .okHU(okHU),
        .okUHU(okUHU),
        .okAA(okAA)
        );

    //Instantiate the ILA module
    ila_0 ila_sample12 (
        .clk(ILA_Clk),
        .probe0({State, SDA, SCL, ACK_bit}),
        .probe1({FSM_Clk, TrigerEvent})
        );
endmodule
```

I2C_Transmit.v:

```
`timescale 1ns / 1ps

module I2C_Transmit(
    output [7:0] led,
    input  sys_clkn,
    input  sys_clkp,
    output ADT7420_A0,
    output ADT7420_A1,
    output I2C_SCL_0,
    inout  I2C_SDA_0,
```

```
output reg FSM_Clk_reg,
output reg ILA_Clk_reg,
output reg ACK_bit,
output reg SCL,
output reg SDA,
output reg [7:0] State,
output wire [31:0] PC_control,
input  wire    [4:0] okUH,
output wire    [2:0] okHU,
inout  wire    [31:0] okUHU,
inout wire okAA
);

//Instantiate the ClockGenerator module, where three signals are generate:
//High speed CLK signal, Low speed FSM_Clk signal
wire [23:0] ClkDivThreshold = 100;
wire FSM_Clk, ILA_Clk;
ClockGenerator ClockGenerator1 (  .sys_clkn(sys_clkn),
                                  .sys_clkp(sys_clkp),
                                  .ClkDivThreshold(ClkDivThreshold),
                                  .FSM_Clk(FSM_Clk),
                                  .ILA_Clk(ILA_Clk) );

reg [7:0] SingleByteData = 8'b1001_0001;
```

```
reg error_bit = 1'b1;

localparam STATE_INIT      = 8'd0;
assign led[7] = ACK_bit;
assign led[6] = error_bit;
assign ADT7420_A0 = 1'b0;
assign ADT7420_A1 = 1'b0;
assign I2C_SCL_0 = SCL;
assign I2C_SDA_0 = SDA;

initial  begin
   SCL = 1'b1;
   SDA = 1'b1;
   ACK_bit = 1'b1;
   State = 8'd0;
end

always @(*) begin
   FSM_Clk_reg = FSM_Clk;
   ILA_Clk_reg = ILA_Clk;
end

always @(posedge FSM_Clk) begin
   case (State)
```

```
        // Press Button[3] to start the state machine. Otherwise, stay in the
STATE_INIT state
        STATE_INIT : begin
            if (PC_control[0] == 1'b1) State <= 8'd1;
            else begin
                SCL <= 1'b1;
                SDA <= 1'b1;
                State <= 8'd0;
             end
        end

        // This is the Start sequence
        8'd1 : begin
            SCL <= 1'b1;
            SDA <= 1'b0;
            State <= State + 1'b1;
        end

        8'd2 : begin
            SCL <= 1'b0;
            SDA <= 1'b0;
            State <= State + 1'b1;
        end

        // transmit bit 7
```

```
8'd3 : begin
    SCL <= 1'b0;
    SDA <= SingleByteData[7];
    State <= State + 1'b1;
end

8'd4 : begin
    SCL <= 1'b1;
    State <= State + 1'b1;
end

8'd5 : begin
    SCL <= 1'b1;
    State <= State + 1'b1;
end

8'd6 : begin
    SCL <= 1'b0;
    State <= State + 1'b1;
end

// transmit bit 6
8'd7 : begin
    SCL <= 1'b0;
```

```
    SDA <= SingleByteData[6];
    State <= State + 1'b1;
end

8'd8 : begin
    SCL <= 1'b1;
    State <= State + 1'b1;
end

8'd9 : begin
    SCL <= 1'b1;
    State <= State + 1'b1;
end

8'd10 : begin
    SCL <= 1'b0;
    State <= State + 1'b1;
end

// transmit bit 5
8'd11 : begin
    SCL <= 1'b0;
    SDA <= SingleByteData[5];
    State <= State + 1'b1;
```

```
end

8'd12 : begin
    SCL <= 1'b1;
    State <= State + 1'b1;
end

8'd13 : begin
    SCL <= 1'b1;
    State <= State + 1'b1;
end

8'd14 : begin
    SCL <= 1'b0;
    State <= State + 1'b1;
end

// transmit bit 4
8'd15 : begin
    SCL <= 1'b0;
    SDA <= SingleByteData[4];
    State <= State + 1'b1;
end
```

```
8'd16 : begin
    SCL <= 1'b1;
    State <= State + 1'b1;
end

8'd17 : begin
    SCL <= 1'b1;
    State <= State + 1'b1;
end

8'd18 : begin
    SCL <= 1'b0;
    State <= State + 1'b1;
end

// transmit bit 3
8'd19 : begin
    SCL <= 1'b0;
    SDA <= SingleByteData[3];
    State <= State + 1'b1;
end

8'd20 : begin
    SCL <= 1'b1;
```

```
    State <= State + 1'b1;
end

8'd21 : begin
    SCL <= 1'b1;
    State <= State + 1'b1;
end

8'd22 : begin
    SCL <= 1'b0;
    State <= State + 1'b1;
end

// transmit bit 2
8'd23 : begin
    SCL <= 1'b0;
    SDA <= SingleByteData[2];
    State <= State + 1'b1;
end

8'd24 : begin
    SCL <= 1'b1;
    State <= State + 1'b1;
end
```

```
8'd25 : begin
    SCL <= 1'b1;
    State <= State + 1'b1;
end

8'd26 : begin
    SCL <= 1'b0;
    State <= State + 1'b1;
end

// transmit bit 1
8'd27 : begin
    SCL <= 1'b0;
    SDA <= SingleByteData[1];
    State <= State + 1'b1;
end

8'd28 : begin
    SCL <= 1'b1;
    State <= State + 1'b1;
end

8'd29 : begin
```

```
    SCL <= 1'b1;
    State <= State + 1'b1;
end

8'd30 : begin
    SCL <= 1'b0;
    State <= State + 1'b1;
end

// transmit bit 0
8'd31 : begin
    SCL <= 1'b0;
    SDA <= SingleByteData[0];
    State <= State + 1'b1;
end

8'd32 : begin
    SCL <= 1'b1;
    State <= State + 1'b1;
end

8'd33 : begin
    SCL <= 1'b1;
    State <= State + 1'b1;
```

```
end

8'd34 : begin
    SCL <= 1'b0;
    State <= State + 1'b1;
end

// read the ACK bit from the sensor and display it on LED[7]
8'd35 : begin
    SCL <= 1'b0;
    SDA <= 1'bz;
    State <= State + 1'b1;
end

8'd36 : begin
    SCL <= 1'b1;
    State <= State + 1'b1;
end

8'd37 : begin
    SCL <= 1'b1;
    ACK_bit <= SDA;
    State <= State + 1'b1;
end
```

```
8'd38 : begin
    SCL <= 1'b0;
    State <= State + 1'b1;
end

//stop bit sequence and go back to STATE_INIT
8'd39 : begin
    SCL <= 1'b0;
    SDA <= 1'b0;
    State <= State + 1'b1;
end

8'd40 : begin
    SCL <= 1'b1;
    SDA <= 1'b0;
    State <= State + 1'b1;
end

8'd41 : begin
    SCL <= 1'b1;
    SDA <= 1'b1;
   // State <= STATE_INIT;
end
```

```
        //If the FSM ends up in this state, there was an error in teh FSM code
        //LED[6] will be turned on (signal is active low) in that case.
        default : begin
            error_bit <= 0;
        end
      endcase
   end


   // OK Interface
   wire [112:0]    okHE;  //These are FrontPanel wires needed to IO communication
   wire [64:0]     okEH;  //These are FrontPanel wires needed to IO communication

   //This is the OK host that allows data to be sent or recived
   okHost hostIF (
      .okUH(okUH),
      .okHU(okHU),
      .okUHU(okUHU),
      .okClk(okClk),
      .okAA(okAA),
      .okHE(okHE),
      .okEH(okEH)
   );
```

```
    //  PC_controll is a wire that contains data sent from the PC to FPGA.
    //  The data is communicated via memeory location 0x00
    okWireIn wire10 (   .okHE(okHE),
                        .ep_addr(8'h00),
                        .ep_dataout(PC_control));

endmodule
```

ClockGenerator.v:

```
`timescale 1ns / 1ps

module ClockGenerator(
    input sys_clkn,
    input sys_clkp,
    input [23:0] ClkDivThreshold,
    output reg FSM_Clk,
    output reg ILA_Clk
    );

    //Generate high speed main clock from two differential clock signals
    wire clk;
    reg [23:0] ClkDiv = 24'd0;
    reg [23:0] ClkDivILA = 24'd0;
```

```
IBUFGDS osc_clk(
   .O(clk),
   .I(sys_clkp),
   .IB(sys_clkn)
);

// Initialize the two registers used in this module
initial begin
   FSM_Clk = 1'b0;
   ILA_Clk = 1'b0;
end

// We derive a clock signal that will be used for sampling signals for the ILA
// This clock will be 10 times slower than the system clock.
always @(posedge clk) begin
   if (ClkDivILA == 10) begin
      ILA_Clk <= !ILA_Clk;
      ClkDivILA <= 0;
   end else begin
      ClkDivILA <= ClkDivILA + 1'b1;
   end
end
```

```
   // We will derive a clock signal for the finite state machine from the ILA clock
   // This clock signal will be used to run the finite state machine for the I2C protocol
   always @(posedge ILA_Clk) begin
      if (ClkDiv == ClkDivThreshold) begin
        FSM_Clk <= !FSM_Clk;
        ClkDiv <= 0;
      end else begin
        ClkDiv <= ClkDiv + 1'b1;
      end
   end
endmodule
```

Python code:

```
# -*- coding: utf-8 -*-

#%%
# import various libraries necessary to run your Python code
import time   # time related library
import sys,os    # system related library
ok_sdk_loc = "C:\\Program Files\\Opal Kelly\\FrontPanelUSB\\API\\Python\\x64"
ok_dll_loc = "C:\\Program Files\\Opal Kelly\\FrontPanelUSB\\API\\lib\\x64"

sys.path.append(ok_sdk_loc)   # add the path of the OK library
os.add_dll_directory(ok_dll_loc)
```

```
import ok     # OpalKelly library

#%%

# Define FrontPanel device variable, open USB communication and

# load the bit file in the FPGA

dev = ok.okCFrontPanel()  # define a device for FrontPanel communication

SerialStatus=dev.OpenBySerial("")      # open USB communication with the OK board

# We will NOT load the bit file because it will be loaded using JTAG interface from Vivado


# Check if FrontPanel is initialized correctly and if the bit file is loaded.

# Otherwise terminate the program

print("------------------------------------------------")

if SerialStatus == 0:

   print ("FrontPanel host interface was successfully initialized.")

else:

   print ("FrontPanel host interface not detected. The error code number is:" + str(int(SerialStatus)))

   print("Exiting the program.")

   sys.exit ()



#%%

# Define the two variables that will send data to the FPGA
```

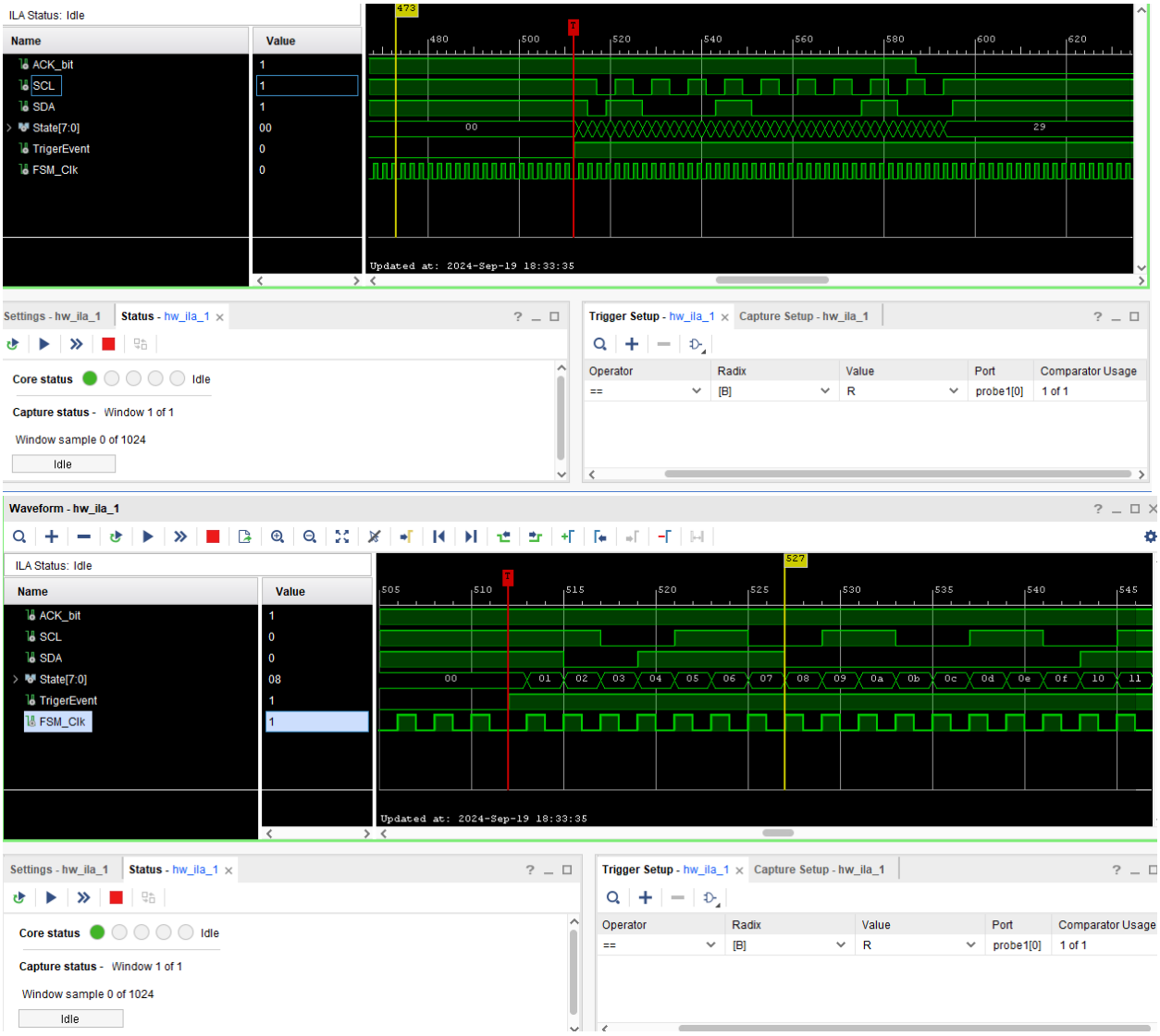
```
# We will use WireIn instructions to send data to the FPGA
PC_Control = 1; # send a "go" signal to the FSM
dev.SetWireInValue(0x00, PC_Control)
dev.UpdateWireIns()  # Update the WireIns
print("Send GO signal to the FSM")
#%%
# Since we are using a slow clock on the FPGA to compute the results
# we need to wait for the result to be computed
time.sleep(0.5)

PC_Control = 0; # send a "stop" signal to the FSM
dev.SetWireInValue(0x00, PC_Control)
dev.UpdateWireIns()  # Update the WireIns
print("Send STOP signal to the FSM")

dev.Close

#%%
```

Milestone1 waveforms:



Milestone2:

I2C.v:

`timescale 1ns / 1ps

//

// Company:

// Engineer:

//

```
// Create Date: 2024/09/22 02:36:21
// Design Name:
// Module Name: I2C
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module I2C_driver(
    output [7:0] led,
    input  clk,
    output ADT7420_A0,
    output ADT7420_A1,
    output I2C_SCL_0,
    inout  I2C_SDA_0,
```

```
output reg ACK,
output reg SCL,
output reg SDA,

output reg [5:0] State,
input  wire [7:0] tx_byte,
output reg [7:0] rx_byte,
input  wire [1:0] next_step,
output reg ready
);

localparam idle_   = 6'b000000;
localparam start_  = 6'b000001;
localparam tx      = 6'b000010;
localparam tx_ack  = 6'b000100;
localparam rx      = 6'b001000;
localparam rx_ack  = 6'b010000;
localparam end_    = 6'b100000;
localparam error_  = 6'b111111;

reg [2:0] bit_counter;
reg [9:0] clk_counter;
```

```
reg [7:0] rx_byte_reg;
reg [7:0] tx_byte_reg;

reg error;

assign led[7] = ACK;
assign led[6] = SCL;
assign led[5] = SDA;
assign led[4:0] = {5{error}};
assign I2C_SCL_0 = SCL;
assign I2C_SDA_0 = SDA;
assign ADT7420_A0 = 1'b0;
assign ADT7420_A1 = 1'b0;

initial begin
   SCL = 1'b1;
   SDA = 1'b1;
   ACK = 1'b1;
   error = 1'b0;
   ready = 1'b1;
   State = idle_;
   rx_byte = 8'b00000000;
   rx_byte_reg = 0;
   tx_byte_reg = 0;
```

```
end

always @(posedge clk) begin
   case (State)
      idle_ : begin
         if (next_step == 2'b01)begin
            State <= start_;
            clk_counter <= 10'd400;
            bit_counter <= 0;
         end
      end
      start_: begin
         case (clk_counter)
            10'd0   : begin
               SCL <= 1'b0;
               SDA <= 1'bz;
               clk_counter <= clk_counter + 1;
            end
            10'd400 : begin
               SCL <= 1'b1;
               clk_counter <= clk_counter + 1;
            end
            10'd600 : begin
               SCL <= 1'b1;
```

```
            SDA <= 1'b0;
            clk_counter <= clk_counter + 1;
        end
        10'd799 : begin
            State <= tx;
            tx_byte_reg <= tx_byte;
            clk_counter <= 10'd0;
        end
        default : begin
            clk_counter <= clk_counter + 1;
        end
    endcase
end

tx: begin
    case (clk_counter)
        10'd0 : begin
            SCL <= 1'b0;
            clk_counter <= clk_counter + 1;
        end
        10'd200 : begin
            SDA <= tx_byte_reg[bit_counter];
            SCL <= 1'b0;
            clk_counter <= clk_counter + 1;
```

```
        end
        10'd400 : begin
          SCL <= 1'b1;
          clk_counter <= clk_counter + 1;
        end
        10'd799 : begin
          if (bit_counter == 3'd7) begin
            rx_byte <= rx_byte_reg;
            State <= tx_ack;
            bit_counter <= 3'd0;
            end
          else begin
            bit_counter <= bit_counter + 1;
          end
          clk_counter <= 10'd0;
        end
        default : begin
          clk_counter <= clk_counter + 1;
        end
    endcase
end

tx_ack : begin
    case (clk_counter)
```

```
10'd0 : begin
   SCL <= 1'b0;
   SDA <= 1'bz;
   clk_counter <= clk_counter + 1;
end
10'd400 : begin
   SCL <= 1'b1;
   ACK <= SDA;
   clk_counter <= clk_counter + 1;
end
10'd799 : begin
   tx_byte_reg <= tx_byte;
   clk_counter <= 10'd0;
   case (next_step)
      2'b00: begin
         State <= end_;
      end
      2'b01: begin
         State <= start_;
      end
      2'b10: begin
         State <= tx;
      end
      2'b11: begin
```

```
                State <= rx;
            end
        endcase
    end
    default : begin
        clk_counter <= clk_counter + 1;
    end
  endcase
end

rx: begin
  case (clk_counter)
    10'd0 : begin
        SCL <= 1'b0;
        SDA <= 1'bz;
        clk_counter <= clk_counter + 1;
    end
    10'd400 : begin
        SCL <= 1'b1;
        clk_counter <= clk_counter + 1;
    end
    10'd500 : begin
        rx_byte_reg[bit_counter] <= SDA;
        clk_counter <= clk_counter + 1;
```

```
        end
        10'd799 : begin
          if (bit_counter == 3'd7) begin
            rx_byte <= rx_byte_reg;
            State <= rx_ack;
            bit_counter <= 3'd0;
          end else begin
            bit_counter <= bit_counter + 1;
          end
          clk_counter <= 10'd0;
        end
        default : begin
          clk_counter <= clk_counter + 1;
        end
      endcase
    end

    rx_ack : begin
      case (clk_counter)
        10'd0 : begin
          SCL <= 1'b0;
          clk_counter <= clk_counter + 1;
        end
        10'd200: begin
```

```
    SDA <= tx_byte_reg[0];
    clk_counter <= clk_counter + 1;
end
10'd400 : begin
    SCL <= 1'b1;
    clk_counter <= clk_counter + 1;
end
10'd799 : begin
    clk_counter <= 10'd0;
    tx_byte_reg <= tx_byte;
    case (next_step)
        2'b00: begin
            State <= end_;
        end
        2'b01: begin
            State <= start_;
        end
        2'b10: begin
            State <= tx;
        end
        2'b11: begin
            State <= rx;
        end
    endcase
```

```
        end
        default : begin
            clk_counter <= clk_counter + 1;
        end
    endcase
end

end_ : begin
    case (clk_counter)
        10'd0: begin
            SCL <= 1'b0;
            SDA <= 1'b0;
            clk_counter <= clk_counter + 1;
        end
        10'd400: begin
            SCL <= 1'b1;
            SDA <= 1'b0;
            clk_counter <= clk_counter + 1;
        end
        10'd600 : begin
            SCL <= 1'b1;
            SDA <= 1'b1;
            clk_counter <= 10'd0;
            State <= idle_;
```

```
                end
                default : begin
                    clk_counter <= clk_counter + 1;
                end
              endcase
            end
          default : begin
            error <= 1'b1;
          end
        endcase
    end

    always @(posedge clk) begin
        case (State)
            tx : ready <= 1'b0;
            rx : ready <= 1'b0;
            default : ready <=1'b1;
        endcase
    end


endmodule
```

Main.v:

```
`timescale 1ns / 1ps


module Main(
    output [7:0] led,
    input sys_clkn,
    input sys_clkp,
    output ADT7420_A0,
    output ADT7420_A1,
    output I2C_SCL_0,
    inout I2C_SDA_0,
    input  [4:0] okUH,
    output [2:0] okHU,
    inout  [31:0] okUHU,
    inout  okAA
);




    // Clock
generation////////////////////////////////////////////////////////////////
    reg ILA_Clk;
    wire clk;
    reg [23:0] ClkDivILA = 24'd0;
    IBUFGDS osc_clk(
```

```
        .O(clk),
        .I(sys_clkp),
        .IB(sys_clkn)
    );
    always @(posedge clk) begin
        if (ClkDivILA == 10) begin
            ILA_Clk <= !ILA_Clk;
            ClkDivILA <= 0;
        end else begin
            ClkDivILA <= ClkDivILA + 1'b1;
        end
    end
    // Clock generation;
////////////////////////////////////////////////////////////


    //PC
communication/////////////////////////////////////////////////////////////
    // TODO verify OK communication function
    wire [31:0]     PC_rx;
    wire [31:0]     PC_tx;
    wire [112:0]    okHE;
    wire [64:0]     okEH;
    localparam  endPt_count = 2;
    wire [endPt_count*65-1:0] okEHx;
    okWireOR # (.N(endPt_count)) wireOR (okEH, okEHx);
```

```
   okHost hostIF (
      .okUH(okUH),
      .okHU(okHU),
      .okUHU(okUHU),
      .okClk(okClk),
      .okAA(okAA),
      .okHE(okHE),
      .okEH(okEH)
   );
   okWireIn wire10 (   .okHE(okHE),
                       .ep_addr(8'h00),
                       .ep_dataout(PC_rx));
   okWireOut wire20 (  .okHE(okHE),
                       .okEH(okEHx[ 0*65 +: 65 ]),
                       .ep_addr(8'h20),
                       .ep_datain(PC_tx));


   // PC
communication//////////////////////////////////////////////////////////////////


   //I2C
SERDES/////////////////////////////////////////////////////////////////////////
   wire SCL, SDA,ACK;
```

```
    wire [5:0] State;
    wire [7:0] tx_byte,rx_byte;
    wire [1:0] next_step;
    wire ready;
    I2C_driver I2C_SERDES (
        .led(led),
        .clk(clk),
        .ADT7420_A0(ADT7420_A0),
        .ADT7420_A1(ADT7420_A1),
        .I2C_SCL_0(I2C_SCL_0),
        .I2C_SDA_0(I2C_SDA_0),

        .ACK(ACK),
        .SCL(SCL),
        .SDA(SDA),
        .State(State),

        .tx_byte(tx_byte),
        .rx_byte(rx_byte),
        .next_step(next_step),
        .ready(ready)
        );
    // I2C SERDES
////////////////////////////////////////////////////////////////////////
```

```
   //Sensor
Controller////////////////////////////////////////////////////////////////
   TS_controller TS_controller(
      .clk(clk),


      .PC_rx(PC_rx),
      .PC_tx(PC_tx),


      .next_step(next_step),
      .tx_byte(tx_byte),
      .rx_byte(rx_byte),
      .ready(ready)
   );
   //Sensor
Controller////////////////////////////////////////////////////////////////


   //Instantiate the ILA module
   ila_0 ila_sample12 (
      .clk(clk),
      .probe0({State, SDA, SCL, ACK}),
      .probe1(next_step));
endmodule
```

main_TB.v:

```
`timescale 1ns / 1ps


module Main_TB();


   //I2C SERDES////////////////////////////////////////////////////////////////
   wire SCL, SDA, ACK;
   wire [5:0] State;
   wire [7:0] tx_byte,rx_byte;
   wire [1:0] next_step;
   wire ready;
   wire ADT7420_A0;
   wire ADT7420_A1;
   wire I2C_SCL_0;
   wire I2C_SDA_0;
   reg [31:0]     PC_rx;
   wire [31:0]     PC_tx;
   reg clk = 1;
   I2C_driver I2C_SERDES (
      .led(led),
      .clk(clk),
      .ADT7420_A0(ADT7420_A0),
      .ADT7420_A1(ADT7420_A1),
      .I2C_SCL_0(I2C_SCL_0),
      .I2C_SDA_0(I2C_SDA_0),
```

```
        .ACK(ACK),
        .SCL(SCL),
        .SDA(SDA),
        .State(State),

        .tx_byte(tx_byte),
        .rx_byte(rx_byte),
        .next_step(next_step),
        .ready(ready)
        );
    // I2C SERDES
////////////////////////////////////////////////////////////////////////


    //Sensor
Controller////////////////////////////////////////////////////////////////
    TS_controller TS_controller(
        .clk(clk),

        .PC_rx(PC_rx),
        .PC_tx(PC_tx),

        .next_step(next_step),
        .tx_byte(tx_byte),
```

```
        .rx_byte(rx_byte),
        .ready(ready)
    );
    //Sensor
Controller//////////////////////////////////////////////////////////////////


    always begin
        #5 clk = ~clk;
    end

    initial begin
        #0 PC_rx <= 0;
        #400 PC_rx <= 1;
    end
endmodule
```

TS_controller.v:

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 2024/09/22 14:34:42
// Design Name:
```

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


module TS_controller(
    input clk,

    input wire [31:0] PC_rx,
    output reg [31:0] PC_tx,

    output reg [1:0] next_step,
    output reg [7:0] tx_byte,
    input wire [7:0] rx_byte,
```

```
input wire ready
);

reg ready_reg;
reg [7:0] tx_byte_reg;
reg [7:0] rx_byte_reg;
reg [5:0] cur_state;
reg [31:0] PC_rx_reg;
reg [31:0] PC_tx_reg;

reg byte2_flag;

localparam idle_     = 6'b000000;
localparam start_rt  = 6'b000001;
localparam tx_rt     = 6'b000010;
localparam rstart_rt = 6'b000100;
localparam rx_rt     = 6'b001000;
localparam end_rt    = 6'b010000;

localparam ns_start  = 2'b01;
localparam ns_tx     = 2'b10;
localparam ns_rx     = 2'b11;
localparam ns_end    = 2'b00;
```

```
localparam device_addr_wr = 8'b10010000;
localparam device_addr_rd = 8'b10010001;
localparam temp_reg_addr  = 8'b00000000;

initial begin
   cur_state <= idle_;
   next_step <= ns_end;
   PC_rx_reg <= 0;
   PC_tx_reg <= 0;
   tx_byte_reg <= 0;
   rx_byte_reg <= 0;
   byte2_flag  <= 1'b0;
   ready_reg <= 1'b1;
end

integer i;
always @(posedge clk) begin
   for (i=0; i<8; i=i+1) begin
      tx_byte[i] <= tx_byte_reg[7-i];
      rx_byte_reg[i] <= rx_byte[7-i];
   end
end
```

```
always @(posedge clk) begin
   case (cur_state)
      idle_ : begin
         PC_rx_reg <= PC_rx;
         if (PC_rx_reg != PC_rx) begin
            cur_state <= start_rt;
         end
      end
      start_rt: begin
         ready_reg <= ready;
         tx_byte_reg <= device_addr_wr;
         next_step <= ns_start;
         if (ready_reg == 1'b0 && ready == 1'b1) begin
            cur_state <= tx_rt;
         end
      end
      tx_rt: begin
         ready_reg <= ready;
         tx_byte_reg <= temp_reg_addr;
         next_step <= ns_tx;
         if(ready_reg == 1'b0 && ready == 1'b1) begin
            cur_state <= rstart_rt;
         end
      end
```

```
rstart_rt : begin
   ready_reg <= ready;
   tx_byte_reg <= device_addr_rd;
   next_step <= ns_start;
   if (ready_reg == 1'b0 && ready == 1'b1) begin
      cur_state <= rx_rt;
   end
end
rx_rt : begin
   ready_reg <= ready;
   tx_byte_reg <= {8{byte2_flag}};
   next_step <= ns_rx;
   if (ready_reg == 1'b0 && ready == 1'b1) begin
      case(byte2_flag)
         1'b0: begin
            byte2_flag <= 1'b1;
            for (i=0; i<8; i =i+1) begin
               PC_tx_reg[12-i] <= rx_byte_reg[7-i];
            end
         end
         1'b1: begin
            byte2_flag <= 1'b0;
            for (i=0; i<5; i = i+1) begin
               PC_tx_reg[4-i] <= rx_byte_reg[7-i];
```

```
                end
                cur_state <= end_rt;
              end
            endcase
          end
        end
        end_rt : begin
          tx_byte_reg <= {8{1'b0}};
          next_step <= ns_end;
          cur_state <= idle_;
          PC_tx <= PC_tx_reg;
        end
        default : begin
          tx_byte_reg <= {8{1'b0}};
          next_step <= ns_end;
          cur_state <= idle_;
        end
      endcase
    end

endmodule
```

Python code:

```
# -*- coding: utf-8 -*-
```

```
#%%
# import various libraries necessary to run your Python code
import time   # time related library
import sys,os    # system related library
ok_sdk_loc = "C:\\Program Files\\Opal Kelly\\FrontPanelUSB\\API\\Python\\x64"
ok_dll_loc = "C:\\Program Files\\Opal Kelly\\FrontPanelUSB\\API\\lib\\x64"

sys.path.append(ok_sdk_loc)   # add the path of the OK library
os.add_dll_directory(ok_dll_loc)

import ok     # OpalKelly library
#%%
# Define FrontPanel device variable, open USB communication and
# load the bit file in the FPGA
dev = ok.okCFrontPanel()  # define a device for FrontPanel communication
SerialStatus=dev.OpenBySerial("")      # open USB communication with the OK board
# We will NOT load the bit file because it will be loaded using JTAG interface from Vivado

# Check if FrontPanel is initialized correctly and if the bit file is loaded.
# Otherwise terminate the program
print("------------------------------------------------")
if SerialStatus == 0:
```

```
    print ("FrontPanel host interface was successfully initialized.")
else:
    print ("FrontPanel host interface not detected. The error code number is:" +
str(int(SerialStatus)))
    print("Exiting the program.")
    sys.exit ()



#%%
# Define the two variables that will send data to the FPGA
# We will use WireIn instructions to send data to the FPGA
dev.SetWireInValue(0x00, 0)
dev.UpdateWireIns()  # Update the WireIns
time.sleep(1)
dev.SetWireInValue(0x00, 1)
dev.UpdateWireIns()  # Update the WireIns
print("Send GO signal to the FSM")
#%%
# Since we are using a slow clock on the FPGA to compute the results
# we need to wait for the result to be computed
time.sleep(1)


dev.UpdateWireOuts()
temp_read = dev.GetWireOutValue(0x20)
```

```
print("temp read is " + str(temp_read/16))
#PC_Control = 0; # send a "stop" signal to the FSM
#dev.SetWireInValue(0x00, PC_Control)
#dev.UpdateWireIns()  # Update the WireIns
#print("Send STOP signal to the FSM")


dev.Close


#%%
```

Milestone2 waveforms: