ECE 437 Final Project Report

Soham Karanjikar

Written: May 11th 2020

1 INTRODUCTION

The purpose of this report is to talk over the procedure of the final project in ECE 437. The project's main goal was to be able to receive frames at a rate of 20FPS from the CMV300 CMOSIS camera on the board as well as 150 readings per second from the ADT temperature sensor. Both of these parts were completed separately in previous labs but not yet implemented together.

Everything done in these two labs done through the use of Vivado to write the Verilog code and Python to process the data and display to console. Code for both of the parts is appended at the end of this document

2 PROCEDURE

2.1 *Acquiring Temperature Readings*

The requirement of this lab was to acquire 150 readings per second from the ADT7420 temperature sensor. This was something we had already completed previously so the communication state machine was available.

The communication with the temperature sensor is done through I2C and the data sheet does explains exactly what the procedure is to acquire a single reading. I just repeated this process in a loop to get multiple readings. The exact waveform to receive data from a specific register is pictured below (Source: ADT7420 Data-sheet):

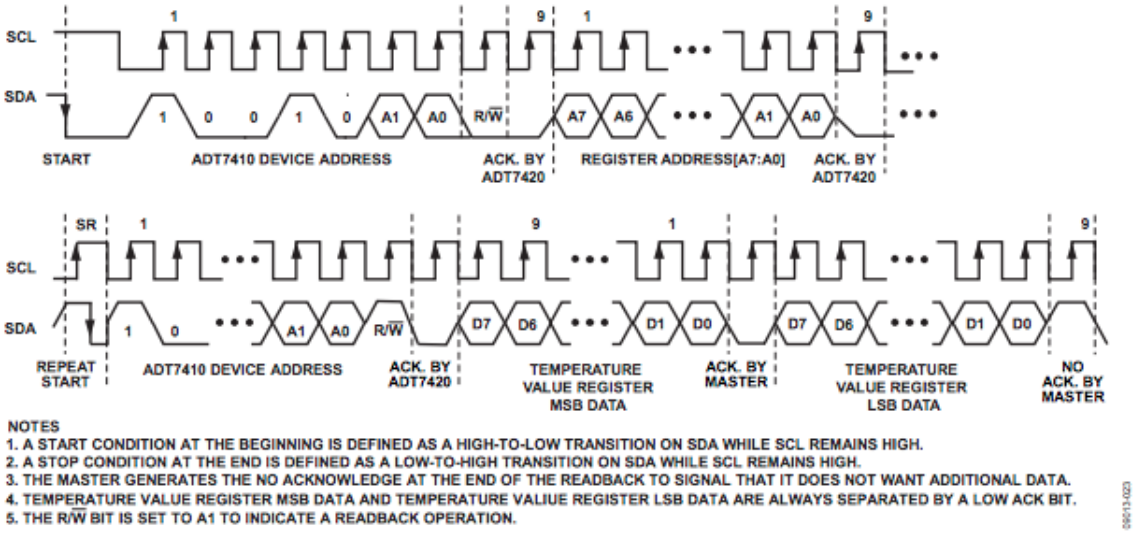


Figure 17. Reading Back Data from the Temperature Value Register

All of these signal were manipulated through a state machine which had 4 states for each 1/4 of a clock cycle so I could hit the timing requirements precisely as the start and restart signals are not exactly on rising and falling edges.

Once the data was received from these registers on the ADT chip, it was stored in a register. This register value could then be sent to a PC using the OpalKelley interface using USB. The OpalKelley interface will be discussed in depth later in this report.

2.2 *Acquiring Images from Image Sensor*

This part was definitely the most challenging part of the lab, not only because of how it was hard to interact with the Image sensor but also because there was so much data coming in and it had to be handled correctly.

The communication with image sensor is done using another serial interface named SPI. This is similar to I2C but instead of having just 1 data line, there is an SPI_IN and SPI_OUT. On the SPI_IN line the master (FPGA in our case) sends data to the slave (sensor in our case) and SPI_OUT is where the slave sends data to the master. This sensor had a built in FSM which did a lot of the starting sequence and signaling for us so the read/write procedures were a lot simpler than the temperature sensor:

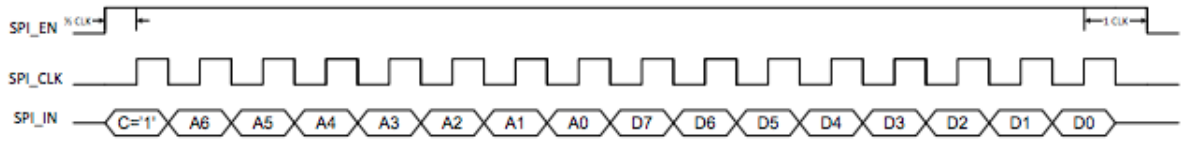


FIGURE 7: SPI WRITE TIMING

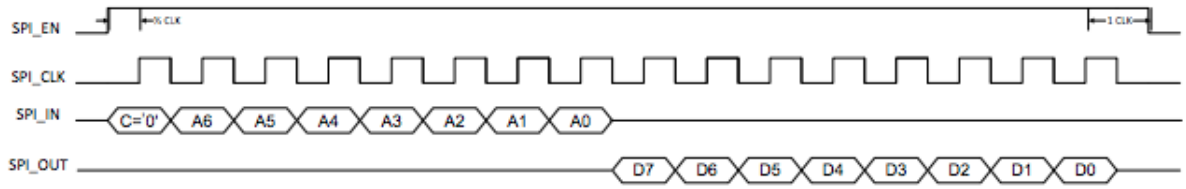


FIGURE 9: SPI READ TIMING

On startup, some of the registers on the sensor had to set to a required value specified by the manufacturer while other configurable ones had to be set depending on which modes we were using. For example we had to set a register to indicate that we wanted to use 8-bit mode not the default 13-bit mode.

We were using the image sensor in its 8-bit parallel output mode so that we could receive the images as fast as possible. This 8-bit mode used 8 of the DATA_OUT pins on the image sensor to output the image. The image resolution was 488x648, and each pixel was 8-bits, so total image size was around .32Mb. This does not sound big but was huge compared to the 9bit temperature readings we were dealing with before.

The state machine for working with the camera was not too difficult, but processing the data was a hard task. We were instructed to use FIFOs. FIFOs are basically a type of memory. We used these because Vivado had an IP that generated these so it was a quick setup. There was not a separate state machine to control this because the signals from the image sensors were very helpful. The Data_Valid signal was only high when there was data output from the parallel output pins. So data was written to FIFO only when necessary, this was very helpful so that we did not have to write a WRITE_ENABLE signal ourselves.

The FIFO write width was set to 8-bits and read width was set to 32-bits. Exactly like how we wanted because the 8-bit output from image sensor and 32-bit input into blockPipe. Further the FIFO also had a full_thresh

value which we used to signal that enough data for 1 frame is in the FIFO, this same signal was used to trigger the start of sending data using blockPipe.

After all the data was in the FIFO, we had to transfer it to the PC to actually see the image. We had to again use the Opal Kelley interface, but a different module of it.

2.3 *Transferring Data to PC*

All the data transfer from FPGA to PC was done through the use of Opal Kelley. The dev board used in this class already has an Opal Kelley module implemented on it which made the transfer very easy. All of the transfers were done through USB, but this Opal Kelley interface gave us the ease of not having to do any low level communication to use the USB interface.

To transfer any data that was relatively small in size, less than 32bits long, we used the okWIRE modules. These were used for the transferring the data we wanted to write to the registers, receiving temperature values form FPGA to PC, and setting any start/stop signals. These okWIRE modules have a capability of 32-bits, so they were perfect for these small transfers. To use them all you had to do was instantiate an okWIRE instance in Verilog, with a name, address, and data register. Then to read/write using these wire, the setWireInValue() or getWireInValue() functions were used on the Python side.

However, transferring the frames acquired from the image sensor using okWIRE was not viable because it would have taken too long to transfer to achieve a 20FPS video. So we had to use the blockPipe module. This module also had a 32-bit transfer width but the speed is close to 30Mb/s. So far higher than what we needed to achieve 20FPS. Using the blockPipes is very similar to the okWIRE. All you had to do is instantiate a module with an address, data register and name in Verilog, then call the readBlockPipe() function in Python.

2.4 *Processing Data*

Finally, once the hardware level data acquisition and state machines are done, I actually had to process the data. All of the processing was done in Python, using the Spyder IDE.

All the processing was done in 1 loop, this loop printed the temperature data to console, formatted the image data into a proper array to be displayed and then displayed it using openCV. This loop was made so that it would run at around 20 times per second.

To display the temperature values, I just printed them to consoles 8 times in each loop which gave readings of around 160/second, satisfying the 150 per second requirement.

To display the image the received data had to be properly formatted. Since the read and write lengths of the FIFO were different, the image data's endianess was not what I expected at first. In fact of the 32 bits received the bytes were formatted like this:

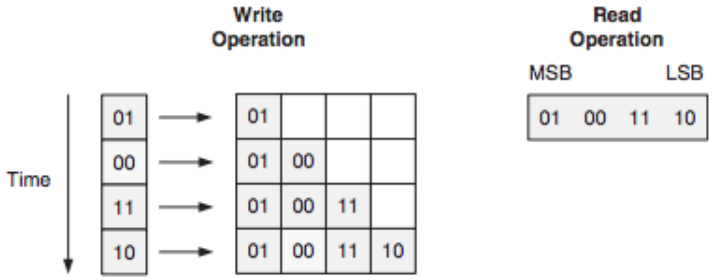


Figure 3-13: **1:4 Aspect Ratio: Data Ordering**

So I had to change the arrangement and make sure pixel 0 was first and pixel 316224 was last.

Once this data was in a reshaped array of size [488,648] dtype=unit8, I could directly send it to imshow(array) to display the image to the screen. This was done each time in the loop achieving 20FPS.

3 NOISE AND LIMITS

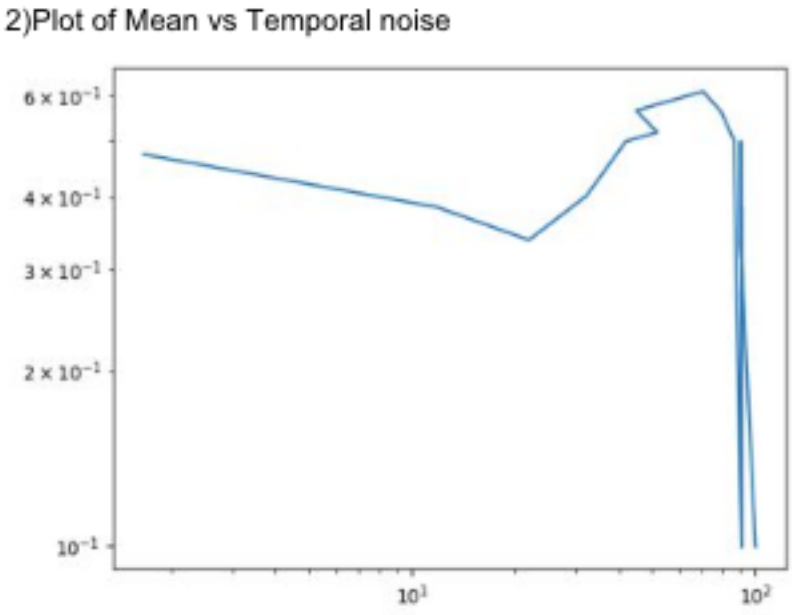
3.1 *Noise in Data*

There was a lot of noise encountered in both the temperature sensor and image sensor however this was not a bad things because we could see the data we are receiving was not static but actually changing over time like expected.

The noise in the temperature sensor was expected random noise. Also this noise was within the temperatures resolution/accuracy so it was not too much of a distinguishing factor. The noise occurred due to the specifications of the temperature sensor and using it on a dev board which had a lot of components heating up and cooling down around it.

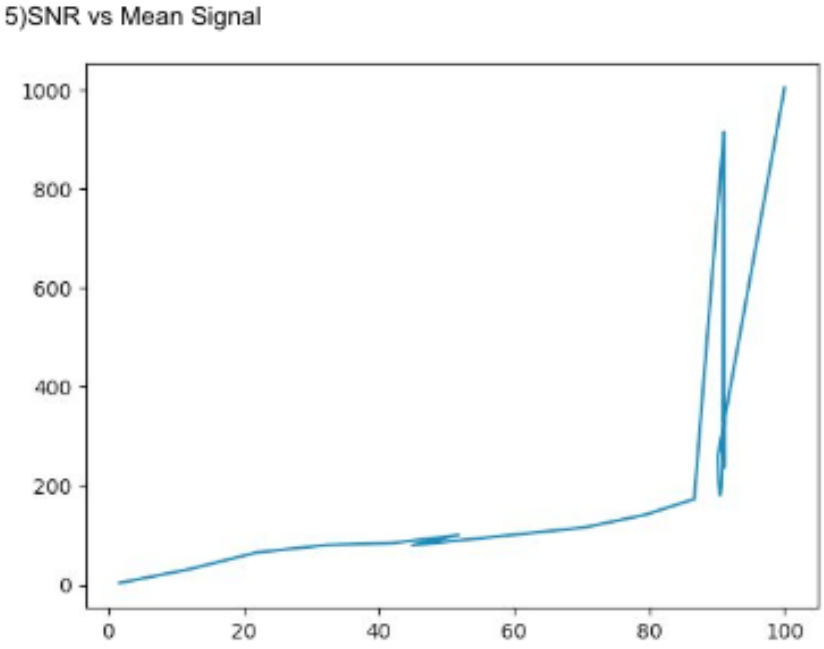
The more noticeable and affecting noise was in the data from image sensor, in fact we had a whole lab where we dived into just the noise of the sensor.

We can see below the mean vs temporal noise plotted on a log scale:



Ignoring the last dip which occurs at around 10^2, which occurs because the sensor reads everything completely white because of extremely high exposure, we can see that there is a very visible trend. The noise increases as mean pixel value (exposure time) increases. This is expected because you are letting more light in which can cause more noise over time. This type of noise causes small pixelated issues and so forth, but does not affect the image to a degree where it is unreadable.

Next, we look at the signal to noise ratio:



This again follows a trend we expect, as long as we ignore that sudden peak which appears for a very long

integration time. The signal to noise ratio increases at an almost steady rate. The noise again increases if we let it run for longer like expected, however it is within an acceptable range as signal values also increase.

Finally, with the data above we can calculate the conversion gain and compare it to the value stated on the data sheet. As seen in the plots above, my dynamic range is roughly 100 which gives a conversion gain from 0-100 of 40dB. This is relatively close to 60dB stated on the data-sheet, however the data-sheet assumes high-dynamic-range mode and has a perfect setup. I am just a student trying to get 20FPS, so cannot achieve such a high conversion gain.

3.2 *Limits*

The physical limits of real-time data acquisition come mainly from how well optimized your state machine is and how fast your PC can receive the data. The FPGA is 100x faster at receiving data and sending it out through its pins than the PC, but that's no use when the processing has to be done on PC. The main bottleneck is the speed of the Opal Kelly interface. Since we are using USB2.0 to connect the boards, we get around 38 MB/s using the blockPipes modules from OpalKelley which is the fastest method of transfer they have. Whereas the okWIRES only have a transfer speed of .032 MB/s

Another limit is also processing the data itself which takes some time in Python and is limited by the processor speed. This happened when we had to use the same Opal Kelley (OK) interface for reading temperature and image data. Since I cannot use the same USB interface to receive 2 different pieces of data in parallel, I had to wait until each one was done before allowing next one to use the interface. I did not optimize this very well.

To achieve the best speed I could have:

-Used the max rated clock speed on all devices

-Optimized state machine to write data until FIFO is completely full rather than clearing it every time prog_full is asserted

-Also used blockPipes for temperature readings with a separate FIFO

-Multi-threaded on Python side and using locks to ensure that the use of the OK interface is never going to waste.

4 CONCLUSION

This project was one of the most interesting and enjoyable projects I have done at UIUC. It was something that is definitely applicable directly to something I may use at a job or even at any DIY project because now I can use an FPGA to get readings at very high speeds and process that data using Python. A skill not many people have.

Further the learning of reading through documentation, tutorials and slides was also very insightful because it is something any electrical engineer has to do on a constant basis.

Finally, I would like to say thanks to the TAs (Anthony and Mebin) and Professor Gruev for providing help whenever needed, whether it be through Piazza or E-mail. This time during COVID-19 was very hard but the

faculty did an extremely amazing job keeping us up to date with information and any help we needed.

5 REFERENCES

ADT7420 Data-Sheet

CMV300 Data-Sheet

FIFO IP Documentation

Opal Kelley Frontpanel Documentation

ECE 437 Website

6 APPENDIX

*All this code is also available on my **Github**. It will be alot easier to read and interact with on there, and you can submit issues or pull if better implementations are found.*

Python Code:

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

#%%
# import various libraries necessery to run your Python code
import time    # time related library
import sys     # system related library
ok_loc = 'C:\\Program Files\\Opal Kelly\\FrontPanelUSB\\API\\Python\\3.6\\x64'
sys.path.append(ok_loc)    # add the path of the OK library
import ok      # OpalKelly library
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
import threading
import cv2

#%%
# 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 communicaiton with the OK board
ConfigStatus=dev.ConfigureFPGA("BTPipeExample.bit"); # Configure the FPGA with this bit file

# 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(Seria
    print("Exiting the program.")
    sys.exit ()

if ConfigStatus == 0:
    print ("Your bit file is successfully loaded in the FPGA.")
else:
    print ("Your bit file did not load. The error code number is:" + str(int(ConfigStatus)))
    print ("Exiting the progam.")
    sys.exit ()
print("------------------------------------------------------------")
print("------------------------------------------------------------")
```

```
#%%
# Define the two variables that will send data to the FPGA
# We will use WireIn instructions to send data to the FPGA
dev.UpdateWireOuts()
regaddress = [1,2,3,4,39,42,43,44,57,58,59,60,68,69,80,83,97,98,100,101,102,103,106,107,108,1
regvalues = [232,1,0,0,1,232,1,0,3,44,240,10,2,9,2,187,240,10,112,98,34,64,94,110,91,82,80,91
R_W = -1
FLAG = 0
ADDRS = -1
REG_Val = -1
START_BIT = -1

dev.SetWireInValue(0x06, 0)

while(dev.GetWireOutValue(0x24) != 1):
    dev.UpdateWireOuts()
    continue
print("Setting register values to required")
for i in range(len(regaddress)):
    R_W = -1
    FLAG = 0
    ADDRS = -1
    REG_Val = -1
    START_BIT = -1
    dev.UpdateWireIns()
    START_BIT = 0
    dev.SetWireInValue(0x03, START_BIT)
    dev.UpdateWireIns()
    time.sleep(.1)
    START_BIT = 1
    REG_ADDR = regaddress[i]
    dev.SetWireInValue(0x00, REG_ADDR) #Input data for Variable 1 using mamoery spacee 0x00
    dev.SetWireInValue(0x01, 1) #Input data for Variable 2 using mamoery spacee 0x01
    REG_VAL = regvalues[i]
    dev.SetWireInValue(0x02, REG_VAL)
    dev.UpdateWireIns()  # Update the WireIns
    time.sleep(.1)
    dev.SetWireInValue(0x03, START_BIT)
    dev.UpdateWireIns()  # Update the WireIns
    time.sleep(.1)
    dev.UpdateWireOuts()
    while(dev.GetWireOutValue(0x21)!=1):
            continue

print('Configuring required registers completed')
```

```
def writeToReg(add, val):
    R_W = -1
    FLAG = 0
    START_BIT = -1
    dev.UpdateWireIns()
    START_BIT =  0
    dev.SetWireInValue(0x03, START_BIT)
    dev.UpdateWireIns()
    time.sleep(.1)
    START_BIT = 1
    dev.SetWireInValue(0x00, add) #Input data for Variable 1 using mamoery spacee 0x00
    dev.SetWireInValue(0x01, 1) #Input data for Variable 2 using mamoery spacee 0x01
    dev.SetWireInValue(0x02, val)
    dev.UpdateWireIns()  # Update the WireIns
    time.sleep(.1)
    dev.SetWireInValue(0x03, START_BIT)
    dev.UpdateWireIns()  # Update the WireIns
    time.sleep(.1)
    dev.UpdateWireOuts()
    while(dev.GetWireOutValue(0x21)!=1):
            continue

def readFromReg(add):
    START_BIT = -1
    dev.UpdateWireIns()
    START_BIT =  0
    dev.SetWireInValue(0x03, START_BIT)
    dev.UpdateWireIns()
    time.sleep(.1)
    START_BIT = 1
    dev.SetWireInValue(0x00, add) #Input data for Variable 1 using mamoery spacee 0x00
    dev.SetWireInValue(0x01, 0) #Input data for Variable 2 using mamoery spacee 0x01
    dev.UpdateWireIns()  # Update the WireIns
    time.sleep(.1)
    dev.SetWireInValue(0x03, START_BIT)
    dev.UpdateWireIns()  # Update the WireIns
    time.sleep(.1)
    dev.UpdateWireOuts()
dev.SetWireInValue(0x05, 1); #Reset FIFOs and counter
dev.UpdateWireIns();  # Update the WireIns

time.sleep(.01)

dev.SetWireInValue(0x05, 0); #Release reset signal
dev.UpdateWireIns();  # Update the WireIns

buf = np.arange(315392).astype('uint8')
```

```
buf12 = np.arange(648*486).astype('uint8')

while(dev.GetWireOutValue(0x24) != 1):
    dev.UpdateWireOuts()
    continue

print("start")
print("done: "+str(dev.ReadFromBlockPipeOut(0xa0, 1024, buf)))

print("done")
buf12 = buf[0:314928]


buf12 = buf12.reshape(648*486)

#reading temp:
global count,average
count = 0
print(dev.SetWireInValue(0x07,1))
dev.UpdateWireIns()
time.sleep(.01)
def readTemp():
    global count,average
    dev.UpdateWireOuts()
    Final_Temp = dev.GetWireOutValue(0x25)
    if((float(Final_Temp)/128.0) == 0.0):
        return
    elif(count == 0):
        average = (float(Final_Temp)/16.0)
        count = count + 1;
    else:
        print("The Temperature is: "+str((average+(float(Final_Temp)/16.0))/2));

global abc
def updateImage():
    global abc,pixels
    dev.SetWireInValue(0x06, 1)
    dev.UpdateWireIns()
    dev.SetWireInValue(0x06, 0)
    dev.UpdateWireIns()
    print("done: "+str(dev.ReadFromBlockPipeOut(0xa0, 1024, buf)))
    buf12 = buf[0:314928]

    readTemp()
    readTemp()
    readTemp()
    readTemp()
```

```
    readTemp()
    readTemp()
    readTemp()
    readTemp()
    return buf12.reshape(486,648)

framecount = 0
print(time.time())
while True:
    array = updateImage()
    framecount += 1
#     if(framecount == 20):
#         print(time.time())
#         break
    cv2.imshow('test',array)
    cv2.waitKey(1)

dev.Close
```

Verilog Code:

Top Level Module

```
        `timescale 1ns / 1ps
module BTPipeExample(
    input    wire     [4:0] okUH,
    output   wire     [2:0] okHU,
    inout    wire     [31:0] okUHU,
    inout    wire     okAA,
    input [3:0] button,
    output [7:0] led,
    input sys_clkn,
    input sys_clkp,
    output CVM300_SYS_RES_N,
        output CVM300_SPI_EN,
    output CVM300_SPI_IN,
    input CVM300_SPI_OUT,
    output CVM300_SPI_CLK,
    inout [9:0] CVM300_D,
    inout CVM300_CLK_OUT,
    inout CVM300_Line_valid,
    inout CVM300_Data_valid,
    output CVM300_CLK_IN,
    output CVM300_FRAME_REQ,
    output ADT7420_A0,
    output ADT7420_A1,
    output I2C_SCL_0,
    inout I2C_SDA_0
    );
    wire okClk;             //These are FrontPanel wires needed to IO communication
    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)
    );

        //SPI Transmit Modules and code
    wire TrigerEvent;
    wire [23:0] SPIClkDivThreshold = 1_000;
    wire SPI_CLK, SPI_EN, SPI_IN, SPI_OUT;
```

```
assign SPI_OUT = CVM300_SPI_OUT;
wire START_BIT;
wire C;
wire [7:0] DATA_READ;
wire [7:0] DATA_WRITE;
wire [7:0] CURR_STATE;
wire [6:0] ADDR;
wire writecomplete;
wire readcomplete;
reg  SPI_FSM_Clk, CVM_Clk, TEMP_Clk;
wire FRAME_REQ, NEW_REQ;
//Instantiate the module that we like to test
SPI_Transmit I2C_Test1 (
    .FSM_Clk(SPI_FSM_Clk),
    .SPI_CLK_o(CVM300_SPI_CLK),
    .SPI_EN_o(CVM300_SPI_EN),
    .SPI_IN_o(CVM300_SPI_IN),
    .SPI_OUT_o(CVM300_SPI_OUT),
    .CURR_STATE(CURR_STATE),
    .ADDR(ADDR),
    .DATA_WRITE(DATA_WRITE),
    .DATA_READ(DATA_READ),
    .C(C),
    .START_BIT(START_BIT),
    .SPI_CLK(SPI_CLK),
    .SPI_EN(SPI_EN),
    .writecomplete(writecomplete),
    .readcomplete(readcomplete),
    .SPI_IN(SPI_IN)
    );
//Depending on the number of outgoing endpoints, adjust endPt_count accordingly.
//In this example, we have 1 output endpoints, hence endPt_count = 1.
localparam  endPt_count = 7;
wire [endPt_count*65-1:0] okEHx;
okWireOR # (.N(endPt_count)) wireOR (okEH, okEHx);

//Instantiate the ClockGenerator module, where three signals are generate:
//High speed CLK signal, Low speed FSM_Clk signal
wire [23:0] ClkDivThreshold = 2;
wire [23:0] ClkDivThresholdSPI = 5;
wire [23:0] ClkDivThresholdCVM = 1;
wire [23:0] ClkDivThresholdTEMP = 9;
reg [23:0] ClkDivSPI = 24'd0;
reg [23:0] ClkDivCVM = 24'd0;
reg [23:0] ClkDivTEMP = 24'd0;
wire FSM_Clk, ILA_Clk;
assign CVM300_CLK_IN = CVM_Clk;
```

```
ClockGenerator ClockGenerator1 (   .sys_clkn(sys_clkn),
                                   .sys_clkp(sys_clkp),
                                   .ClkDivThreshold(ClkDivThreshold),
                                   .FSM_Clk(FSM_Clk),
                                   .ILA_Clk(ILA_Clk) );
always @(posedge FSM_Clk) begin
    if (ClkDivSPI == ClkDivThresholdSPI) begin
     SPI_FSM_Clk <= !SPI_FSM_Clk;
     ClkDivSPI <= 0;
   end else begin
     ClkDivSPI <= ClkDivSPI + 1'b1;
   end
end

always @(posedge FSM_Clk) begin
    if (ClkDivCVM == ClkDivThresholdCVM) begin
     CVM_Clk <= !CVM_Clk;
     ClkDivCVM <= 0;
   end else begin
     ClkDivCVM <= ClkDivCVM + 1'b1;
   end
end

always @(posedge CVM_Clk) begin
    if (ClkDivTEMP == ClkDivThresholdTEMP) begin
     TEMP_Clk <= !TEMP_Clk;
     ClkDivTEMP <= 0;
   end else begin
     ClkDivTEMP <= ClkDivTEMP + 1'b1;
   end
end

// Temperature:
wire [7:0] CURR_STATE_TEMP;

wire [7:0] MSB_BYTE;
wire [7:0] LSB_BYTE;
wire [15:0] Final_Temp;
wire [12:0] Temptopython;
wire starttempsig;
assign Final_Temp = {3'booo,MSB_BYTE,LSB_BYTE[7],LSB_BYTE[6],LSB_BYTE[5], LSB_BYTE[4], LSB
wire  ILA_Clk, ACK_bit, FSM_Clk, SCL, SDA;

//Instantiate the module that we like to test
TEMP_Transmit TEMP_Test1 (
    .ADT7420_A0(ADT7420_A0),
    .ADT7420_A1(ADT7420_A1),
```

```
    .I2C_SCL_o(I2C_SCL_o),
    .I2C_SDA_o(I2C_SDA_o),
    .ACK_bit(ACK_bit),
    .SCL(SCL),
    .SDA(SDA),
    .Temp_MSByte(MSB_BYTE),
    .Temp_LSByte(LSB_BYTE),
    .starttempsig(starttempsig),
    .FSM_Clk(TEMP_Clk)
    );



//Video:

localparam STATE_INIT                = 8'do;
localparam STATE_RESET               = 8'd1;
localparam STATE_DELAY               = 8'd2;
localparam STATE_RESET_FINISHED      = 8'd3;
localparam STATE_ENABLE_WRITING      = 8'd4;
localparam STATE_COUNT               = 8'd5;
localparam STATE_FINISH              = 8'd6;

reg [31:0] counter = 32'do;
reg [15:0] counter_delay = 16'do;
reg [7:0] State = STATE_INIT;
reg [7:0] led_register = 0;
reg [3:0] button_reg, write_enable_counter;
reg write_reset, read_reset, write_enable;
wire [31:0] Reset_Counter;
wire [31:0] DATA_Counter;
wire FIFO_read_enable, FIFO_BT_BlockSize_Full, FIFO_full, FIFO_empty, BT_Strobe;
wire [31:0] FIFO_data_out;

assign led[0] = ~FIFO_empty;
assign led[1] = ~FIFO_full;
assign led[2] = ~FIFO_BT_BlockSize_Full;
assign led[3] = ~FIFO_read_enable;
assign led[7] = ~read_reset;
assign led[6] = ~write_reset;
initial begin
    write_reset <= 1'bo;
    read_reset <= 1'bo;
    write_enable <= 1'bo;
end
reg [7:0] CVM_State = 8'do;
reg [7:0] CVM_State2 = 8'do;
```

```
reg [31:0] CVM_4Pixels = 32'do;
reg imgreadcompleter;
wire imgreadcomplete;
reg enable_write;
reg CVM300_SYS_RES_N_R = 1'bo;
assign CVM300_SYS_RES_N = CVM300_SYS_RES_N_R;
assign imgreadcomplete = imgreadcompleter;

reg CVM300_FRAME_REQ_R =1'bo;
assign CVM300_FRAME_REQ = CVM300_FRAME_REQ_R;

reg spistarttrigr;
wire spistarttrig;
assign spistarttrig = spistarttrigr;

reg [15:0] startdelay = 16'do;
reg triggerila;
//get Image data from Sensor
always @(posedge CVM300_CLK_IN) begin
    if(NEW_REQ == 1'b1) CVM_State = 8'd12;

    case(CVM_State)
        8'do:   begin
            if (startdelay == 16'b1111_1111_1111_1111) begin
                 CVM_State <= 8'd1;
                 CVM300_SYS_RES_N_R <= 1'b1;
            end
            else begin
                startdelay <= startdelay + 1;
                CVM_State <= 8'do;
                CVM300_SYS_RES_N_R <= 1'bo;
            end
        end

        8'd1:   begin
            startdelay <= 16'do;
            CVM_State <= 8'd2;
            spistarttrigr <= 1'bo;
        end

        8'd2:   begin
            if (startdelay == 16'boooo_1111_1111_1111) begin
                write_reset <= 1'b1;
                read_reset <= 1'b1;
                CVM_State <= 8'd3;
                spistarttrigr <= 1'b1;
            end
```

```
      else begin
          startdelay <= startdelay + 1;
          CVM_State <= 8'd2;
      end
 end

 8'd3:    begin
      write_reset <= 1'b1;
      read_reset <= 1'b1;
      counter_delay <= 0;
      CVM300_FRAME_REQ_R <= 1'b0;
      if(FRAME_REQ == 1'b1) begin
          spistarttrigr <= 1'b0;
          CVM_State <= 8'd4;
      end
end

8'd4:     begin
      write_reset <= 1'b0;
      read_reset <= 1'b0;
      if(FRAME_REQ == 1'b0) begin
          CVM_State <= 8'd5;
      end
end

8'd5:    begin
      if (counter_delay == 16'b0000_1111_1111_1111)  CVM_State <= 8'd6;
      else counter_delay <= counter_delay + 1;
 end

 8'd6:    begin
       triggerila <= 1'b1;
       //CVM300_FRAME_REQ_R <= 1'b1;
       CVM_State <= 8'd7;
 end

 8'd7:    begin
      //CVM300_FRAME_REQ_R <= 1'b0;
      CVM_State <= 8'd8;
      counter_delay <= 16'd0;
 end

 8'd8:    begin
       if (counter_delay == 16'b0000_0000_1111_1111) begin
           CVM_State <= 8'd9;
           CVM300_FRAME_REQ_R <= 1'b1;
           spistarttrigr <= 1'b1;
```

```
            end
            else counter_delay <= counter_delay + 1;
        end

        8'd9:   begin
            CVM300_FRAME_REQ_R <= 1'b1;
            CVM_State <= 8'd10;
        end

        8'd10:   begin
            CVM300_FRAME_REQ_R <= 1'bo;
            CVM_State <= 8'd10;
        end

        8'd12: begin
            write_reset <= 1'b1;
            read_reset <= 1'b1;
            if(NEW_REQ == 1'bo) begin
                CVM_State <= 8'd13;
            end
        end

        8'd13: begin
            write_reset <= 1'bo;
            read_reset <= 1'bo;
            CVM300_FRAME_REQ_R <= 1'b1;
            CVM_State <= 8'd14;
        end

        8'd14: begin
            CVM300_FRAME_REQ_R <= 1'b1;
            CVM_State <= 8'd15;
        end

        8'd15: begin
            CVM300_FRAME_REQ_R <= 1'bo;
            CVM_State <= 8'd15;
        end
    endcase
end

reg[18:0] pixelcounter;

initial  begin
    write_enable <= 1'bo;
    imgreadcompleter <= 1'bo;
    CVM300_FRAME_REQ_R <= 1'bo;
```

```
    pixelcounter <= 19'do;
    write_enable_counter <= 0;
end

fifo_generator_0 FIFO_for_Counter_BTPipe_Interface (
    .wr_clk(CVM300_CLK_OUT),
    .wr_rst(write_reset),
    .rd_clk(okClk),
    .rd_rst(read_reset),
    .din(CVM300_D[9:2]),
    .wr_en(CVM300_Data_valid & CVM300_Line_valid),
    .rd_en(FIFO_read_enable),
    .dout(FIFO_data_out),
    .full(FIFO_full),
    .empty(FIFO_empty),
    .prog_full(FIFO_BT_BlockSize_Full)
);

okBTPipeOut CounterToPC (
    .okHE(okHE),
    .okEH(okEHx[ 0*65 +: 65 ]),
    .ep_addr(8'ha0),
    .ep_datain(FIFO_data_out),
    .ep_read(FIFO_read_enable),
    .ep_blockstrobe(BT_Strobe),
    .ep_ready(FIFO_BT_BlockSize_Full)
);

     okWireOut wire20 (  .okHE(okHE),
                         .okEH(okEHx[ 1*65 +: 65 ]),
                         .ep_addr(8'h20),
                         .ep_datain(DATA_READ));

okWireOut wire21 (   .okHE(okHE),
                     .okEH(okEHx[ 2*65 +: 65 ]),
                     .ep_addr(8'h21),
                     .ep_datain(writecomplete));
okWireOut wire22 (   .okHE(okHE),
                     .okEH(okEHx[ 3*65 +: 65 ]),
                     .ep_addr(8'h22),
                     .ep_datain(readcomplete));

okWireOut wire23 (   .okHE(okHE),
                     .okEH(okEHx[ 4*65 +: 65 ]),
                     .ep_addr(8'h23),
                     .ep_datain(imgreadcomplete));
```

```
okWireOut wire24 (   .okHE(okHE),
                     .okEH(okEHx[ 5*65 +: 65 ]),
                     .ep_addr(8'h24),
                     .ep_datain(spistarttrig));

okWireOut wire25 (   .okHE(okHE),
                     .okEH(okEHx[ 6*65 +: 65 ]),
                     .ep_addr(8'h25),
                     .ep_datain(Final_Temp));

 okWireIn wire10 (    .okHE(okHE),
                     .ep_addr(8'h00),
                     .ep_dataout(ADDR));

 okWireIn wire11 (    .okHE(okHE),
                     .ep_addr(8'h01),
                     .ep_dataout(C));

 okWireIn wire12 (    .okHE(okHE),
                     .ep_addr(8'h02),
                     .ep_dataout(DATA_WRITE));

 okWireIn wire13 (    .okHE(okHE),
                     .ep_addr(8'h03),
                     .ep_dataout(START_BIT));

okWireIn wire14 (    .okHE(okHE),
                     .ep_addr(8'h04),
                     .ep_dataout(Reset_Counter));

okWireIn wire15 (   .okHE(okHE),
                     .ep_addr(8'h05),
                     .ep_dataout(FRAME_REQ));
okWireIn wire16 (   .okHE(okHE),
                     .ep_addr(8'h06),
                     .ep_dataout(NEW_REQ));

okWireIn wire17 (    .okHE(okHE),
                     .ep_addr(8'h07),
                     .ep_dataout(starttempsig));



//ILA module for debugging
ila_0 ila_sample12 (
    .clk(ILA_Clk),
    .probe0(CVM_4Pixels),
```

```
        .probe1(CVM300_CLK_OUT),
        .probe2(CVM300_Line_valid),
        .probe3(CVM300_Data_valid),
        .probe4(CVM300_Data_valid),
        .probe5(CVM300_D)
        );
endmodule
```

Clock Generator Module

```
`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
```

I2C Temperature Sensor Communication Module

```
`timescale 1ns / 1ps

module TEMP_Transmit(
    output ADT7420_A0,
    output ADT7420_A1,
    output I2C_SCL_0,
    inout  I2C_SDA_0,
    output reg ACK_bit,
    output reg SCL,
    output reg SDA,
    output reg [7:0] Temp_MSByte,
    output reg [7:0] Temp_LSByte,
    input starttempsig,
    input FSM_Clk
    );

    //Instantiate the ClockGenerator module, where three signals are generate:
    //High speed CLK signal, Low speed FSM_Clk signal
    wire [23:0] ClkDivThreshold = 100;

    reg [7:0] SingleByteData = 8'b1001_0000;
    reg [7:0] State = 8'd0;
    reg error_bit = 1'b1;
    reg repeat_start_bit;
    reg reloop_start_bit;
    reg read_lsb;
    reg loop_done;
    reg [7:0] MSB_ADDR = 8'd0;
    reg [7:0] LSB_ADDR = 8'd1;
    reg [3:0] count;


    localparam STATE_INIT       = 8'd0;
    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;
        read_lsb = 1'b0;
        repeat_start_bit = 1'b0;
        reloop_start_bit = 1'b0;
```

```
    count = 4'd0;
    loop_done = 1'b0;
    State = STATE_INIT;
end

always @(posedge FSM_Clk) begin
    case (State)
        // Press Button[3] to start the state machine. Otherwise, stay in the STATE_INIT
        STATE_INIT : begin
             if (starttempsig == 1'b1) State <= 8'd1;
             else begin
                  SCL <= 1'b1;
                  SDA <= 1'b1;
                  State <= 8'd0;
                  Temp_MSByte <= 8'd0;
                  Temp_LSByte <= 8'd0;
              end
        end

        // This is the Start sequence
        8'd1 : begin
              SCL <= 1'b1;
              SDA <= 1'b0;
              if(loop_done == 1'b1) begin
                  Temp_MSByte <= 8'd0;
                  Temp_LSByte <= 8'd0;
                  loop_done = 1'b0;
              end
              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;
        if(repeat_start_bit == 1)
        begin
           SDA <= 1'b1;
           State <= State + 1'b1;
        end
        else
        begin
            SDA <= SingleByteData[0];
            State <= State + 1'b1;
        end
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;
            if(repeat_start_bit == 1'b1 && read_lsb == 1'b0) begin
                State <= 8'd88;
            end else if (repeat_start_bit == 1 && read_lsb == 1'b1) begin
                State <= 8'd129;
            end else begin
                State <= State + 3'd5;
            end

        end


//          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'b0;
//                State <= State + 1'b1;
//          end

//          //change numbrs
//          8'd42 : begin
//                SCL <= 1'b0;
//                State <= State + 1'b1;
//          end

        8'd43 : begin
            SCL <= 1'b0;
            if(read_lsb == 1'b0) begin
              SDA <= MSB_ADDR[7];
            end else begin
              SDA <= LSB_ADDR[7];
```

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

8'd44 : begin
        SCL <= 1'b1;
        State <= State + 1'b1;
end

8'd45 : begin
        SCL <= 1'b1;
        State <= State + 1'b1;
end

8'd46 : begin
        SCL <= 1'bo;
        State <= State + 1'b1;
end

8'd47 : begin
        SCL <= 1'bo;
        if(read_lsb == 1'bo) begin
          SDA <= MSB_ADDR[6];
        end else begin
          SDA <= LSB_ADDR[6];
        end
        State <= State + 1'b1;
end

8'd48 : begin
        SCL <= 1'b1;
        State <= State + 1'b1;
end

8'd49 : begin
        SCL <= 1'b1;
        State <= State + 1'b1;
end

8'd50 : begin
        SCL <= 1'bo;
        State <= State + 1'b1;
end

8'd51 : begin
        SCL <= 1'bo;
        if(read_lsb == 1'bo) begin
```

```
        SDA <= MSB_ADDR[5];
      end else begin
        SDA <= LSB_ADDR[5];
      end
      State <= State + 1'b1;
end

8'd52 : begin
      SCL <= 1'b1;
      State <= State + 1'b1;
end

8'd53 : begin
      SCL <= 1'b1;
      State <= State + 1'b1;
end

8'd54 : begin
      SCL <= 1'bo;
      State <= State + 1'b1;
end

8'd55 : begin
      SCL <= 1'bo;
      if(read_lsb == 1'bo) begin
        SDA <= MSB_ADDR[4];
      end else begin
        SDA <= LSB_ADDR[4];
      end
      State <= State + 1'b1;
end

8'd56 : begin
      SCL <= 1'b1;
      State <= State + 1'b1;
end

8'd57 : begin
      SCL <= 1'b1;
      State <= State + 1'b1;
end

8'd58 : begin
      SCL <= 1'bo;
      State <= State + 1'b1;
end
```

```
8'd59 : begin
      SCL <= 1'b0;
      if(read_lsb == 1'b0) begin
        SDA <= MSB_ADDR[3];
      end else begin
        SDA <= LSB_ADDR[3];
      end
      State <= State + 1'b1;
end

8'd60 : begin
      SCL <= 1'b1;
      State <= State + 1'b1;
end

8'd61 : begin
      SCL <= 1'b1;
      State <= State + 1'b1;
end

8'd62 : begin
      SCL <= 1'b0;
      State <= State + 1'b1;
end

8'd63 : begin
      SCL <= 1'b0;
      if(read_lsb == 1'b0) begin
        SDA <= MSB_ADDR[2];
      end else begin
        SDA <= LSB_ADDR[2];
      end
      State <= State + 1'b1;
end

8'd64 : begin
      SCL <= 1'b1;
      State <= State + 1'b1;
end

8'd65 : begin
      SCL <= 1'b1;
      State <= State + 1'b1;
end

8'd66 : begin
      SCL <= 1'b0;
```

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

8'd67 : begin
      SCL <= 1'bo;
      if(read_lsb == 1'bo) begin
        SDA <= MSB_ADDR[1];
      end else begin
        SDA <= LSB_ADDR[1];
      end
      State <= State + 1'b1;
end

8'd68 : begin
      SCL <= 1'b1;
      State <= State + 1'b1;
end

8'd69 : begin
      SCL <= 1'b1;
      State <= State + 1'b1;
end

8'd70 : begin
      SCL <= 1'bo;
      State <= State + 8'd5;
end

8'd75 : begin
      SCL <= 1'bo;
      if(read_lsb == 1'bo) begin
        SDA <= MSB_ADDR[0];
      end else begin
        SDA <= LSB_ADDR[0];
      end
      State <= State + 1'b1;
end

8'd76 : begin
      SCL <= 1'b1;
      State <= State + 1'b1;
end

8'd77 : begin
      SCL <= 1'b1;
      State <= State + 1'b1;
end
```

```
8'd78 : begin
      SCL <= 1'b0;
      State <= State + 1'b1;
end

// read the ACK bit from the sensor and display it on LED[7]
8'd79 : begin
      SCL <= 1'b0;
      SDA <= 1'bz;
      State <= State + 1'b1;
end

8'd80 : begin
      SCL <= 1'b1;
      State <= State + 1'b1;
end

8'd81 : begin
      SCL <= 1'b1;
      ACK_bit <= SDA;
      State <= State + 1'b1;
end

8'd82 : begin
      SCL <= 1'b0;
      State <= State + 1'b1;
end

8'd83 : begin
      SCL <= 1'b0;
      SDA <= 1'b1;
      State <= State + 8'd4;
end

8'd87 : begin
      SCL <= 1'b1;
      State <= 8'd1;
      repeat_start_bit <= 1'b1;

end

8'd88 : begin
      SCL <= 1'b0;
      State <= State + 1'b1;
end
```

```
8'd89 : begin
        SCL <= 1'b1;
        State <= State + 1'b1;
end

8'd90 : begin
        SCL <= 1'b1;
        Temp_MSByte[7] <= SDA;
        State <= State + 1'b1;
end

8'd91 : begin
        SCL <= 1'b0;
        State <= State + 1'b1;
end

8'd92 : begin
        SCL <= 1'b0;
        State <= State + 1'b1;
end

8'd93 : begin
        SCL <= 1'b1;
        State <= State + 1'b1;
end

8'd94 : begin
        SCL <= 1'b1;
        Temp_MSByte[6] <= SDA;
        State <= State + 1'b1;
end

8'd95 : begin
        SCL <= 1'b0;
        State <= State + 1'b1;
end

8'd96 : begin
        SCL <= 1'b0;
        State <= State + 1'b1;
end

8'd97 : begin
        SCL <= 1'b1;
        State <= State + 1'b1;
end
```

```
8'd98 : begin
      SCL <= 1'b1;
      Temp_MSByte[5] <= SDA;
      State <= State + 1'b1;
end

8'd99 : begin
      SCL <= 1'b0;
      State <= State + 1'b1;
end

8'd100 : begin
      SCL <= 1'b0;
      State <= State + 1'b1;
end

8'd101 : begin
      SCL <= 1'b1;
      State <= State + 1'b1;
end

8'd102 : begin
      SCL <= 1'b1;
      Temp_MSByte[4] <= SDA;
      State <= State + 1'b1;
end

8'd103 : begin
      SCL <= 1'b0;
      State <= State + 1'b1;
end

8'd104 : begin
      SCL <= 1'b0;
      State <= State + 1'b1;
end

8'd105 : begin
      SCL <= 1'b1;
      State <= State + 1'b1;
end

8'd106 : begin
      SCL <= 1'b1;
      Temp_MSByte[3] <= SDA;
      State <= State + 1'b1;
end
```

```
8'd107 : begin
      SCL <= 1'b0;
      State <= State + 1'b1;
end

8'd108 : begin
      SCL <= 1'b0;
      State <= State + 1'b1;
end

8'd109 : begin
      SCL <= 1'b1;
      State <= State + 1'b1;
end

8'd110 : begin
      SCL <= 1'b1;
      Temp_MSByte[2] <= SDA;
      State <= State + 1'b1;
end

8'd111 : begin
      SCL <= 1'b0;
      State <= State + 1'b1;
end

8'd112 : begin
      SCL <= 1'b0;
      State <= State + 1'b1;
end

8'd113 : begin
      SCL <= 1'b1;
      State <= State + 1'b1;
end

8'd114 : begin
      SCL <= 1'b1;
      Temp_MSByte[1] <= SDA;
      State <= State + 1'b1;
end

8'd115 : begin
      SCL <= 1'b0;
      State <= State + 1'b1;
end
```

```
8'd116 : begin
      SCL <= 1'b0;
      State <= State + 1'b1;
end

8'd117 : begin
      SCL <= 1'b1;
      State <= State + 1'b1;
end

8'd118 : begin
      SCL <= 1'b1;
      Temp_MSByte[0] <= SDA;
      State <= State + 1'b1;
end

8'd119 : begin
      SCL <= 1'b0;
      State <= State + 1'b1;
end

//send NACK
8'd120 : begin
      SCL <= 1'b0;
      SDA <= 1'b1;
      State <= State + 1'b1;
end

8'd121 : begin
      SCL <= 1'b1;
      State <= State + 1'b1;
end

8'd122 : begin
      SCL <= 1'b1;
      State <= State + 1'b1;
end

8'd123 : begin
      SCL <= 1'b0;
      State <= State + 1'b1;
end

//stop and go to state 1
8'd124 : begin
      read_lsb = 1'b1;
```

```
        repeat_start_bit = 1'b0;
        SCL <= 1'b0;
        SDA <= 1'b0;
        State <= State + 1'b1;
end

8'd125 : begin
        SCL <= 1'b1;
        State <= State + 1'b1;
end

8'd126 : begin
        SCL <= 1'b1;
        SDA <= 1'b1;
        State <= State + 1'b1;
end

8'd127 : begin
        SCL <= 1'b1;
        SDA <= 1'b1;
        State <= State + 1'b1;
end

8'd128 : begin
        SCL <= 1'b1;
        SDA <= 1'b1;
        State <= 1'd1;
end

//Read LSB.
8'd129 : begin
        SCL <= 1'b0;
        State <= State + 1'b1;
end

8'd130 : begin
        SCL <= 1'b1;
        State <= State + 1'b1;
end

8'd131 : begin
        SCL <= 1'b1;
        Temp_LSByte[7] <= SDA;
        State <= State + 1'b1;
end

8'd132 : begin
```

```
        SCL <= 1'b0;
        State <= State + 1'b1;
end

8'd133 : begin
        SCL <= 1'b0;
        State <= State + 1'b1;
end

8'd134 : begin
        SCL <= 1'b1;
        State <= State + 1'b1;
end

8'd135 : begin
        SCL <= 1'b1;
        Temp_LSByte[6] <= SDA;
        State <= State + 1'b1;
end

8'd136 : begin
        SCL <= 1'b0;
        State <= State + 1'b1;
end

8'd137 : begin
        SCL <= 1'b0;
        State <= State + 1'b1;
end

8'd138 : begin
        SCL <= 1'b1;
        State <= State + 1'b1;
end

8'd139 : begin
        SCL <= 1'b1;
        Temp_LSByte[5] <= SDA;
        State <= State + 1'b1;
end

8'd140 : begin
        SCL <= 1'b0;
        State <= State + 1'b1;
end

8'd141 : begin
```

```
        SCL <= 1'b0;
        State <= State + 1'b1;
end

8'd142 : begin
        SCL <= 1'b1;
        State <= State + 1'b1;
end

8'd143 : begin
        SCL <= 1'b1;
        Temp_LSByte[4] <= SDA;
        State <= State + 1'b1;
end

8'd144 : begin
        SCL <= 1'b0;
        State <= State + 1'b1;
end

8'd145 : begin
        SCL <= 1'b0;
        State <= State + 1'b1;
end

8'd146 : begin
        SCL <= 1'b1;
        State <= State + 1'b1;
end

8'd147 : begin
        SCL <= 1'b1;
        Temp_LSByte[3] <= SDA;
        State <= State + 1'b1;
end

8'd148 : begin
        SCL <= 1'b0;
        State <= State + 1'b1;
end

8'd149 : begin
        SCL <= 1'b0;
        State <= State + 1'b1;
end

8'd150 : begin
```

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

8'd151 : begin
        SCL <= 1'b1;
        Temp_LSByte[2] <= SDA;
        State <= State + 1'b1;
end

8'd152 : begin
        SCL <= 1'bo;
        State <= State + 1'b1;
end

8'd153 : begin
        SCL <= 1'bo;
        State <= State + 1'b1;
end

8'd154 : begin
        SCL <= 1'b1;
        State <= State + 1'b1;
end

8'd155 : begin
        SCL <= 1'b1;
        Temp_LSByte[1] <= SDA;
        State <= State + 1'b1;
end

8'd156 : begin
        SCL <= 1'bo;
        State <= State + 1'b1;
end

8'd157 : begin
        SCL <= 1'bo;
        State <= State + 1'b1;
end

8'd158 : begin
        SCL <= 1'b1;
        State <= State + 1'b1;
end

8'd159 : begin
```

```
        SCL <= 1'b1;
        Temp_LSByte[0] <= SDA;
        State <= State + 1'b1;
end

8'd160 : begin
        SCL <= 1'b0;
        State <= State + 1'b1;
end

8'd161 : begin
        SCL <= 1'b0;
        SDA <= 1'b1;
        State <= State + 1'b1;
end

8'd162 : begin
        SCL <= 1'b1;
        State <= State + 1'b1;
end

8'd163 : begin
        SCL <= 1'b1;
        State <= State + 1'b1;
end

8'd164 : begin
        SCL <= 1'b0;
        State <= State + 1'b1;
end

//stop and go to state 1
8'd165 : begin
        read_lsb = 1'b0;
        SCL <= 1'b0;
        SDA <= 1'b0;
        State <= State + 1'b1;
end

8'd166 : begin
        SCL <= 1'b1;
        State <= State + 1'b1;
end

8'd167 : begin
        SCL <= 1'b1;
        SDA <= 1'b1;
```

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

            8'd168 : begin
                SCL <= 1'b1;
                SDA <= 1'b1;
                State <= State + 1'b1;
            end

            8'd169 : begin
                SCL <= 1'b1;
                State <= 1'd1;
                repeat_start_bit <= 1'bo;
                loop_done = 1'b1;
            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 <= o;
            end
        endcase
    end
endmodule
```

```
8'd19 : begin
        SPI_CLK <= 1'b1;
        State <= State + 1'b1;
end

8'd20 : begin
        SPI_CLK <= 1'b1;
        State <= State + 1'b1;
end

8'd21 : begin
        SPI_CLK <= 1'b0;
        SPI_IN <= ADDR[2];
        State <= State + 1'b1;
end

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

8'd23 : begin
        SPI_CLK <= 1'b1;
        State <= State + 1'b1;
end

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

8'd25 : begin
        SPI_CLK <= 1'b0;
        SPI_IN <= ADDR[1];
        State <= State + 1'b1;
end

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

8'd27 : begin
        SPI_CLK <= 1'b1;
        State <= State + 1'b1;
```

```
end

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

8'd29 : begin
      SPI_CLK <= 1'b0;
      SPI_IN <= ADDR[0];
      State <= State + 1'b1;
end

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

8'd31 : begin
      SPI_CLK <= 1'b1;
      State <= State + 1'b1;
end

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

//write or read data depending on "C"
8'd33 : begin
      SPI_CLK <= 1'b0;
      if(C == 1'b1) begin
            SPI_IN <= DATA_WRITE[7];
      end
      State <= State + 1'b1;
end

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

8'd35 : begin
      SPI_CLK <= 1'b1;
      if(C == 1'b0) begin
            DATA_READ[7] <= SPI_OUT_0;
      end
```

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

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

8'd37 : begin
        SPI_CLK <= 1'b0;
        if(C == 1'b1) begin
                SPI_IN <= DATA_WRITE[6];
        end
        State <= State + 1'b1;
end

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

8'd39 : begin
        SPI_CLK <= 1'b1;
        if(C == 1'b0) begin
                DATA_READ[6] <= SPI_OUT_0;
        end
        State <= State + 1'b1;
end

8'd40 : begin
        SPI_CLK <= 1'b1;
        State <= State + 1'b1;
end

8'd41 : begin
        SPI_CLK <= 1'b0;
        if(C == 1'b1) begin
                SPI_IN <= DATA_WRITE[5];
        end
        State <= State + 1'b1;
end

8'd42 : begin
        SPI_CLK <= 1'b0;
        State <= State + 1'b1;
end
```

```
8'd43 : begin
      SPI_CLK <= 1'b1;
      if(C == 1'b0) begin
            DATA_READ[5] <= SPI_OUT_o;
      end
      State <= State + 1'b1;
end

8'd44 : begin
      SPI_CLK <= 1'b1;
      State <= State + 1'b1;
end

8'd45 : begin
      SPI_CLK <= 1'b0;
      if(C == 1'b1) begin
            SPI_IN <= DATA_WRITE[4];
      end
      State <= State + 1'b1;
end

8'd46 : begin
      SPI_CLK <= 1'b0;
      State <= State + 1'b1;
end

8'd47 : begin
      SPI_CLK <= 1'b1;
      if(C == 1'b0) begin
            DATA_READ[4] <= SPI_OUT_o;
      end
      State <= State + 1'b1;
end

8'd48 : begin
      SPI_CLK <= 1'b1;
      State <= State + 1'b1;
end

8'd49 : begin
      SPI_CLK <= 1'b0;
      if(C == 1'b1) begin
            SPI_IN <= DATA_WRITE[3];
      end
      State <= State + 1'b1;
end
```

```
8'd50 : begin
      SPI_CLK <= 1'b0;
      State <= State + 1'b1;
end

8'd51 : begin
      SPI_CLK <= 1'b1;
      if(C == 1'b0) begin
            DATA_READ[3] <= SPI_OUT_0;
      end
      State <= State + 1'b1;
end

8'd52 : begin
      SPI_CLK <= 1'b1;
      State <= State + 1'b1;
end

8'd53 : begin
      SPI_CLK <= 1'b0;
      if(C == 1'b1) begin
            SPI_IN <= DATA_WRITE[2];
      end
      State <= State + 1'b1;
end

8'd54 : begin
      SPI_CLK <= 1'b0;
      State <= State + 1'b1;
end

8'd55 : begin
      SPI_CLK <= 1'b1;
      if(C == 1'b0) begin
            DATA_READ[2] <= SPI_OUT_0;
      end
      State <= State + 1'b1;
end

8'd56 : begin
      SPI_CLK <= 1'b1;
      State <= State + 1'b1;
end

8'd57 : begin
      SPI_CLK <= 1'b0;
      if(C == 1'b1) begin
```

```
            SPI_IN <= DATA_WRITE[1];
      end
      State <= State + 1'b1;
end

8'd58 : begin
      SPI_CLK <= 1'b0;
      State <= State + 1'b1;
end

8'd59 : begin
      SPI_CLK <= 1'b1;
      if(C == 1'b0) begin
            DATA_READ[1] <= SPI_OUT_0;
      end
      State <= State + 1'b1;
end

8'd60 : begin
      SPI_CLK <= 1'b1;
      State <= State + 1'b1;
end

8'd61 : begin
      SPI_CLK <= 1'b0;
      if(C == 1'b1) begin
            SPI_IN <= DATA_WRITE[0];
            writecomplete <= 1'b1;
      end
      State <= State + 1'b1;
end

8'd62 : begin
      SPI_CLK <= 1'b0;
      State <= State + 1'b1;
end

8'd63 : begin
      SPI_CLK <= 1'b1;
      if(C == 1'b0) begin
            DATA_READ[0] <= SPI_OUT_0;
            readcomplete <= 1'b1;
      end
      State <= State + 1'b1;
end

8'd64 : begin
```

```
                SPI_CLK <= 1'b1;
                if(START_BIT == 1'b0) begin
                        State <= STATE_INIT;
                end
                else begin
                        State <= State;
                end
        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

        end
    endcase
  end
endmodule
```