**ECE 437**

Fall 2023

Final Project

**Real Time Object Tracker**

Ryan Libiano, Noah DuVal

Yifei Jin - Tuesday 4:00 pm

# Introduction

In the final lab for ECE 437: Sensors and Instruments we have built upon our previous work and have brought multiple labs together to create one that encapsulates all that we have learned over the course of the semester. The purpose of this lab was to perform real time object tracking. Through utilizing I2C and SPI protocols in order to communicate with and control sensors across the Opal Kelly Interface. The final product is able to control the H-Bridge Motor to rotate the FPGA in the direction of the object being tracked. In addition, it will read both the acceleration and heading data at one hundred data points per second. While the camera is displayed on the screen at a minimum of 20 frames per second.

# Algorithms Overview

The fully working system consisted of a few pieces of code working together. On the Verilog side we had state machines to run both I2C and SPI protocols to control and communicate with the sensors and camera. The I2C state machine consisted of two portions. One to control the sequence that was to occur such as, reading data, writing data, starting, stopping, and repeat start for multiple reads. The second was much larger and did the actual reading and writing of the data. Controlling and reading the SDA and SCL lines. On the python side we had multiple functions that we created to program the registers with the values that we needed. As well as functions to get the acceleration data and magnetic data. Reading from the camera was done in a very similar style.

For the tracker we had decided to use OpenCV as they compute the frame differences and thresholds for you. The one we chose to use from that library was the MOSSE tracker. While less accurate and consistent it was the fastest and ultimately worked under the right visual conditions.

# Spatial & Temporal Noise of Image Sensor

The spatial noise for the image sensor that we had calculated was 0.987. This number was calculated by finding the standard deviation of the entire image buffer. As for the temporal Noise had received a value of 0. This value was measured by taking the standard deviation of a still image over time for a single set of pixels.

# Signal to Noise Ratio of Sensor

The acceleration signal to noise to ratio that we had calculated is 0.1758. We found that value by taking the standard deviation over the course of one hundred measurements. The magnetometer signal to noise ratio came out to 0.05 being calculated the same way as the acceleration.

# Noise in Acceleration Sensor

According to our calculations, there is no noise within the sensor. Either that or the noise is so minute it is unable to be seen with the values that we are using and would only be visible when using much smaller numbers.

# Acceleration Vs. Applied Voltage

# Accuracy

For the tracking system we had decided to use the algorithms inside of OpenCV. The one we ended up choosing was the MOSSE tracker as it was the fastest and was able to give us the best frame rate. We had initially gone with the KCF tracker, because it was far more accurate, but not fast enough. Thus, with our final decision to use the MOSSE tracker we had forfeited a little bit of the tracking abilities for speed. The tracker was fully able to follow an object to the left and the right, but we would need to make sure that there was a high contrast between the object being tracked and the space behind it. Otherwise it would latch on to a background object such as sunlight on the wall or a monitor in the background. To get around this we took a white piece of paper and shined a flashlight on it to make it bright enough to stand out in front of the rest of the light sources in the image.

# Conclusion

All said and done we have accomplished what we had set out to do in this lab. However, that did not come easy as there were many obstacles that had occurred along the way. In the initial thought we assumed it would be fairly simple to execute considering we had most of the pieces to the puzzle already and all we had to do was just put them all together. Although, we had quickly realized that some of our pieces were not built for conformity and had to be redone and or needed adjustments to accommodate the ability to work together. The I2C protocol was one of the modules that had to be redone to increase efficiency so that they were able to meet the timing requirements for the reading of the acceleration and heading sensors. In that process we also had to adjust the way to send data across the Ok Wires as we started to run out of wires and had to combine some of the output data together. Most of the other problems that we encountered also had to deal with the timing constraints. Nonetheless, it was a great lesson and way to learn not only the importance of speed when building a system, but also what affects the timing. Speed and efficiency is something that is and will always be important in the world and this was a great way to understand in detail the changes needed to be made in order to manipulate it to your liking.

# 

# Appendix

## Python Code

# import various libraries necessary to run your Python code

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

import time

import numpy as np

from PIL import Image

import cv2

import matplotlib.pyplot as plt

#%%

# 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

# 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 ();

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

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

buf = bytearray(648\*480)

#literally the closs common multiple to 486\*648

#%%

#Sensor Stuff

#Addresses for OK

trigger = 0x07;

dev\_addr = 0x08;

sub\_addr = 0x09;

data\_in = 0x10;

d\_out\_1 = 0x21;

d\_out\_2 = 0x22;

d\_out\_3 = 0x23;

d\_out\_4 = 0x24;

d\_out\_5 = 0x25;

d\_out\_6 = 0x26;

#%%

#The functions needed

def get\_accel\_data(dev):

dev.SetWireInValue(dev\_addr, 0x32)

dev.SetWireInValue(sub\_addr, 0xA8)

dev.UpdateWireIns()

dev.SetWireInValue(trigger, 1)

dev.UpdateWireIns()

dev.SetWireInValue(trigger, 0)

dev.UpdateWireIns()

dev.UpdateWireOuts()

X\_L\_A = dev.GetWireOutValue(d\_out\_1)

X\_H\_A = dev.GetWireOutValue(d\_out\_2)

Y\_L\_A = dev.GetWireOutValue(d\_out\_3)

Y\_H\_A = dev.GetWireOutValue(d\_out\_4)

Z\_L\_A = dev.GetWireOutValue(d\_out\_5)

Z\_H\_A = dev.GetWireOutValue(d\_out\_6)

X = twobit(X\_L\_A,X\_H\_A)/16000

Y = twobit(Y\_L\_A,Y\_H\_A)/16000

Z = twobit(Z\_L\_A,Z\_H\_A)/16000

return X,Y,Z

def twobit(m1, m2):

val\_x = (bin(m1 + (m2 << 8)))

if((int(val\_x, 2) >> 15)):

val\_x = int(val\_x, 2) - (1<<16)

else:

val\_x = int(val\_x, 2)

return val\_x

def get\_mag\_data(dev):

dev.SetWireInValue(dev\_addr, 0x3C)

dev.SetWireInValue(sub\_addr, 0b10000011)

dev.UpdateWireIns()

dev.SetWireInValue(trigger, 1)

dev.UpdateWireIns()

dev.SetWireInValue(trigger, 0)

dev.UpdateWireIns()

dev.UpdateWireOuts()

X\_H\_A = dev.GetWireOutValue(d\_out\_1)

X\_L\_A = dev.GetWireOutValue(d\_out\_2)

Z\_H\_A = dev.GetWireOutValue(d\_out\_3)

Z\_L\_A = dev.GetWireOutValue(d\_out\_4)

Y\_H\_A = dev.GetWireOutValue(d\_out\_5)

Y\_L\_A = dev.GetWireOutValue(d\_out\_6)

X = twobit(X\_L\_A,X\_H\_A)/1000

Y = twobit(Y\_L\_A,Y\_H\_A)/1000

Z = twobit(Z\_L\_A,Z\_H\_A)/1000

return X,Y,Z

def prog\_i2c(dev):

#program the accelerometer

dev.SetWireInValue(dev\_addr, 0x32)

dev.SetWireInValue(sub\_addr, 0x20)

dev.SetWireInValue(data\_in, 0b01010111)

dev.UpdateWireIns()

dev.SetWireInValue(trigger, 1)

dev.UpdateWireIns()

dev.SetWireInValue(trigger, 0)

dev.UpdateWireIns()

time.sleep(0.75)

dev.SetWireInValue(dev\_addr, 0x32)

dev.SetWireInValue(sub\_addr, 0x20)

dev.SetWireInValue(data\_in, 0b00000000)

dev.UpdateWireIns()

dev.SetWireInValue(trigger, 1)

dev.UpdateWireIns()

dev.SetWireInValue(trigger, 0)

dev.UpdateWireIns()

time.sleep(0.75)

dev.SetWireInValue(dev\_addr, 0x3C)

dev.SetWireInValue(sub\_addr, 0x02)

dev.SetWireInValue(data\_in, 0x00)

dev.UpdateWireIns()

dev.SetWireInValue(trigger, 1)

dev.UpdateWireIns()

dev.SetWireInValue(trigger, 0)

dev.UpdateWireIns()

time.sleep(0.75)

dev.SetWireInValue(dev\_addr, 0x3C)

dev.SetWireInValue(sub\_addr, 0x00)

dev.SetWireInValue(data\_in, 0b00011000)

dev.UpdateWireIns()

dev.SetWireInValue(trigger, 1)

dev.UpdateWireIns()

dev.SetWireInValue(trigger, 0)

dev.UpdateWireIns()

return

prog\_i2c(dev)

#print("X Accell: " + str(X\_A) + " Y Accell: " + str( Y\_A) + " Z Accell: " + str( Z\_A))

#%% Write defaults to through SPI

#%%

dev.SetWireInValue(0x06, 0)

dev.UpdateWireIns()

data = 0x00;

addr = 0x01;

r\_w = 0x02;

start = 0x03;

read\_data = 0x20;

#---------------- Start of sending the default value to the Default Registers ---------------------#

dev.UpdateWireOuts()

Default\_Addr = [1,2,3,4,39,42,43,44,57,58,59,60,68,69,80,83,97,98,100,101,102,103,106,107,108,109,110,117]

Default\_Values = [232,1,0,0,1,150,2,0,3,44,240,10,2,9,2,187,240,10,112,98,34,64,94,110,91,82,80,91]

#----------------- Writes all of the default values -----------------#

for i in range(len(Default\_Addr)):

TEMP\_addr = Default\_Addr[i]

dev.SetWireInValue(addr, TEMP\_addr)

dev.SetWireInValue(r\_w, 1) #------- Sends Address and Value for Address ------#

TEMP\_VALUE = Default\_Values[i]

dev.SetWireInValue(data, TEMP\_VALUE)

dev.UpdateWireIns()

#time.sleep(.01)

dev.SetWireInValue(start, 1)

dev.UpdateWireIns()

dev.SetWireInValue(start, 0) #------ Starts the FSM -----------#

dev.UpdateWireIns()

#time.sleep(.01)

print(str(TEMP\_VALUE) + ' Written To: ' + str(TEMP\_addr))

#time.sleep(.01)

#------------ Reads all of the default values -------------#

print('Write Completed')

for i in range(len(Default\_Addr)):

TEMP\_addr = Default\_Addr[i]

dev.SetWireInValue(addr, TEMP\_addr)

dev.SetWireInValue(r\_w, 0)

dev.UpdateWireIns() #---------- Reads Register Values ------------#

#time.sleep(.01)

dev.SetWireInValue(start, 1)

dev.UpdateWireIns()

dev.SetWireInValue(start, 0)

dev.UpdateWireIns()

#time.sleep(.01)

dev.UpdateWireOuts()

TEMP\_VALUE = dev.GetWireOutValue(read\_data)

print('Register ' + str(TEMP\_addr) + ' value: ' + str(TEMP\_VALUE))

#time.sleep(0.1)

print('Done reading from registers!')

print('Reading all registers')

for i in range(127):

TEMP\_addr = i

dev.SetWireInValue(addr, TEMP\_addr)

dev.SetWireInValue(r\_w, 0)

dev.UpdateWireIns() #---------- Reads Register Values ------------#

#time.sleep(.01)

dev.SetWireInValue(start, 1)

dev.UpdateWireIns()

dev.SetWireInValue(start, 0)

dev.UpdateWireIns()

#time.sleep(.01)

dev.UpdateWireOuts()

TEMP\_VALUE = dev.GetWireOutValue(read\_data)

print('Register ' + str(TEMP\_addr) + ' value: ' + str(TEMP\_VALUE))

#time.sleep(.01)

print('Done reading from registers!')

'''

#%% KCF tracker

'''

dev.SetWireInValue(0x04, 1); #Reset FIFOs and counter

dev.UpdateWireIns(); # Update the WireIns

dev.SetWireInValue(0x04, 0); #Release reset signal

dev.UpdateWireIns(); # Update the WireIns

dev.ReadFromBlockPipeOut(0xa0, 256, buf)

arr\_buffer=np.frombuffer(buf,dtype=np.uint8)

pixel = arr\_buffer.reshape((480,648))

roi = cv2.selectROI('Select Region of Interest',pixel)

cv2.destroyAllWindows()

font = cv2.FONT\_HERSHEY\_SIMPLEX

bottomLeftCornerOfText = (320,320)

fontScale = 1

fontColor = (0,255,0)

thickness = 1

lineType = 2

tracker = cv2.legacy.TrackerMOSSE.create()

tracker.init(pixel, roi)

#%%

RUN\_LENGTH = 1000

data\_buffer = np.arange(RUN\_LENGTH)

for i in range(RUN\_LENGTH):

start = time.time()

dev.SetWireInValue(0x04, 1); #Reset FIFOs and counter

dev.UpdateWireIns(); # Update the WireIns

dev.SetWireInValue(0x04, 0); #Release reset signal

dev.UpdateWireIns(); # Update the WireIns

dev.ReadFromBlockPipeOut(0xa0, 256, buf)

arr\_buffer=np.frombuffer(buf,dtype=np.uint8)

pixel = arr\_buffer.reshape((480,648))

success, roi = tracker.update(pixel)

time.sleep(.0005)

X\_A, Y\_A, Z\_A = get\_accel\_data(dev)

print("X Accell: " + str(X\_A) + " Y Accell: " + str( Y\_A) + " Z Accell: " + str( Z\_A))

if success:

(x,y,w,h) = tuple(map(int, roi))

cv2.rectangle(pixel,(x, y), (x + w,y+h), (255,0,0),2)

if ( (x+w/2) < 648/2 - 50):

dev.SetWireInValue(0x05, 1)

dev.SetWireInValue(0x06, 1)

dev.UpdateWireIns()

cv2.putText(pixel,'Go Right!',

bottomLeftCornerOfText,

font,

fontScale,

fontColor,

thickness,

lineType)

elif ((x+w/2) > 648/2 + 150):

dev.SetWireInValue(0x05, 0)

dev.SetWireInValue(0x06, 1)

dev.UpdateWireIns()

cv2.putText(pixel,'Go Left!',

bottomLeftCornerOfText,

font,

fontScale,

fontColor,

thickness,

lineType)

else:

dev.SetWireInValue(0x06, 0)

dev.UpdateWireIns()

cv2.putText(pixel,'Locked!',

bottomLeftCornerOfText,

font,

fontScale,

fontColor,

thickness,

lineType)

cv2.imshow('Latest',pixel)

stop = time.time()

print(1/(stop-start))

if cv2.waitKey(1) & 0xFF == ord('q'): # by press 'q' you quit the process

break

print('The Sensor Noise is'+str(np.std(data\_buffer)))

cv2.destroyAllWindows()

'''

'''

img = Image.new('L',(488\*4,648\*4))

img.putdata(buf)

img.show()

#why isnt this working???

#%%

dev.Close

## Verilog Top Level

`timescale 1 ps / 1 ps

module BTPipeExample(

input wire [4:0] okUH,

output wire [2:0] okHU,

inout wire [31:0] okUHU,

inout wire okAA,

//PMOD Stuff

output PMOD\_A1,

output PMOD\_A2,

//SPI signaling

input CVM300\_SPI\_OUT,

output CVM300\_SPI\_EN,

output CVM300\_SPI\_IN,

output CVM300\_SPI\_CLK,

//Parallel CMOS out data

output CVM300\_FRAME\_REQ,

output CVM300\_SYS\_RES\_N,

output CVM300\_CLK\_IN,

input CVM300\_CLK\_OUT,

input [9:0] CVM300\_D,

input CVM300\_Line\_valid,

input CVM300\_Data\_valid,

//stuff for clock

input [3:0] button,

output [7:0] led,

input sys\_clkn,

input sys\_clkp,

//I2C Stuff Here

inout I2C\_SDA\_1,

output I2C\_SCL\_1

);

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)

);

wire [31:0] data\_in, addr\_in, r\_w, start\_fsm;

wire [7:0] data\_out;

wire [31:0] motor\_dir, motor\_en;

reg sysreq, framereq;

assign CVM300\_SYS\_RES\_N = sysreq;

assign CVM300\_FRAME\_REQ = framereq;

assign PMOD\_A2 = motor\_dir;

//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 = 8;

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 = 24'd100;

wire FSM\_Clk, I2C\_Clk;

ClockGenerator ClockGenerator1 ( .sys\_clkn(sys\_clkn),

.sys\_clkp(sys\_clkp),

.ClkDivThreshold(ClkDivThreshold),

.FSM\_Clk(I2C\_Clk),

.ILA\_Clk(FSM\_Clk) );

assign CVM300\_CLK\_IN = FSM\_Clk;

//SPI stuff

SPI\_master SPI\_mod (.FSM\_clk(FSM\_Clk), .ADDR(addr\_in[7:0]),

.D\_IN(data\_in[7:0]), .MISO\_IN(CVM300\_SPI\_OUT), .R\_W(r\_w[0]),

.START\_FSM(start\_fsm[0]), .SPI\_CLK\_OUT(CVM300\_SPI\_CLK),

.MOSI\_OUT(CVM300\_SPI\_IN), .D\_OUT(data\_out),

.SPI\_EN(CVM300\_SPI\_EN), .READ\_DONE(read\_done),

.WRITE\_DONE(write\_done));

//I2C

localparam STATE\_INIT = 8'd0;

localparam STATE\_RESET = 8'd1;

localparam STATE\_DELAY = 8'd2;

localparam STATE\_RESET\_FINISHED = 8'd3;

localparam FRAME\_BEGIN = 8'd4;

localparam FRAME\_END = 8'd5;

localparam STATE\_FINISH = 8'd6;

reg [31:0] counter = 8'd0;

reg [15:0] counter\_delay = 16'd0;

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;

wire [31:0] Reset\_Counter;

wire [31:0] DATA\_Counter;

wire PWM\_Clk;

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[4] = ~PWM\_Clk;

assign led[7] = ~read\_reset;

assign led[6] = ~write\_reset;

initial begin

write\_reset <= 1'b0;

read\_reset <= 1'b0;

end

always @(negedge FSM\_Clk) begin

button\_reg <= ~button; // Grab the values from the button, complement and store them in register

if (Reset\_Counter[0] == 1'b1) State <= STATE\_RESET;

sysreq <= 1'b1;

case (State)

STATE\_INIT: begin

sysreq <= 1'b1;

if (Reset\_Counter[0] == 1'b1)

State <= STATE\_RESET;

end

STATE\_RESET: begin

counter <= 0;

counter\_delay <= 0;

write\_reset <= 1'b1;

read\_reset <= 1'b1;

if (Reset\_Counter[0] == 1'b0) State <= STATE\_RESET\_FINISHED;

end

STATE\_RESET\_FINISHED: begin

write\_reset <= 1'b0;

read\_reset <= 1'b0;

State <= STATE\_DELAY;

end

STATE\_DELAY: begin

if (counter\_delay == 16'b0000\_0000\_0001\_1111) State <= FRAME\_BEGIN;

else counter\_delay <= counter\_delay + 1;

end

FRAME\_BEGIN: begin

framereq <= 1'b1;

State <= FRAME\_END;

end

FRAME\_END: begin

framereq <= 1'b0;

State <= STATE\_FINISH;

end

STATE\_FINISH: begin

State <= STATE\_INIT;

end

endcase

end

//PWM STUFF FOR CONTROLLER

PWM\_Controller PWM\_cntrl(.PWM\_Clk (I2C\_Clk), .PWM\_Out(PWM\_Clk), .DUTY\_CYC(16'd200), .pulse\_cnt(8'd1), .en(motor\_en));

assign PMOD\_A1 = PWM\_Clk;

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),

.rd\_en(FIFO\_read\_enable),

.dout(FIFO\_data\_out),

.full(FIFO\_full),

.empty(FIFO\_empty),

.prog\_full(FIFO\_BT\_BlockSize\_Full)

);

//I2C Stuff the module that we like to test

wire [31:0] PC\_control, dev\_addr, sub\_addr, data\_in\_I2C, d\_out\_1, d\_out\_2, d\_out\_3;

wire SDA, SCL, ACK\_bit;

wire [9:0] State\_I2C;

I2C\_Master I2C\_Trans(

.FSM\_Clk(I2C\_Clk),

.trigger(PC\_control[0]),

.I2C\_SCL\_1(I2C\_SCL\_1),

.I2C\_SDA\_1(I2C\_SDA\_1),

.dev\_addr(dev\_addr[7:0]),

.sub\_addr(sub\_addr[7:0]),

.data\_in(data\_in\_I2C[7:0]),

.Data\_Out\_1(d\_out\_1),

.Data\_Out\_2(d\_out\_2),

.Data\_Out\_3(d\_out\_3),

.State(State\_I2C),

.SCL(SCL),

.SDA(SDA),

.ACK\_bit(ACK\_bit));

//ILA for debugging the system

//Opal Kelly Stuff

okBTPipeOut CounterToPC (

.okHE(okHE),

.okEH(okEHx[ 7\*65 +: 65 ]),

.ep\_addr(8'ha0),

.ep\_datain({FIFO\_data\_out[7:0],FIFO\_data\_out[15:8],FIFO\_data\_out[23:16],FIFO\_data\_out[31:24]}),

.ep\_read(FIFO\_read\_enable),

.ep\_blockstrobe(BT\_Strobe),

.ep\_ready(FIFO\_BT\_BlockSize\_Full)

);

okWireIn wire10 ( .okHE(okHE),

.ep\_addr(8'h00),

.ep\_dataout(data\_in));

okWireIn wire11 ( .okHE(okHE),

.ep\_addr(8'h01),

.ep\_dataout(addr\_in));

okWireIn wire12 ( .okHE(okHE),

.ep\_addr(8'h02),

.ep\_dataout(r\_w));

okWireIn wire13 ( .okHE(okHE),

.ep\_addr(8'h03),

.ep\_dataout(start\_fsm));

okWireIn wire14 ( .okHE(okHE),

.ep\_addr(8'h04),

.ep\_dataout(Reset\_Counter));

okWireIn wire15 ( .okHE(okHE),

.ep\_addr(8'h05),

.ep\_dataout(motor\_dir));

okWireIn wire16 ( .okHE(okHE),

.ep\_addr(8'h06),

.ep\_dataout(motor\_en));

okWireIn wire17 ( .okHE(okHE),

.ep\_addr(8'h07),

.ep\_dataout(PC\_control));

okWireIn wire18 ( .okHE(okHE),

.ep\_addr(8'h08),

.ep\_dataout(dev\_addr));

okWireIn wire19 ( .okHE(okHE),

.ep\_addr(8'h09),

.ep\_dataout(sub\_addr));

okWireIn wire20 ( .okHE(okHE),

.ep\_addr(8'h0a),

.ep\_dataout(data\_in\_I2C));

//wire out stuff

okWireOut wire21 ( .okHE(okHE),

.okEH(okEHx[ 0\*65 +: 65 ]),

.ep\_addr(8'h20),

.ep\_datain({24'd0,data\_out}));

okWireOut wire23 ( .okHE(okHE),

.okEH(okEHx[ 1\*65 +: 65 ]),

.ep\_addr(8'h21),

.ep\_datain({24'd0,d\_out\_1[7:0]}));

okWireOut wire24 ( .okHE(okHE),

.okEH(okEHx[ 2\*65 +: 65 ]),

.ep\_addr(8'h22),

.ep\_datain({24'd0,d\_out\_1[15:8]}));

okWireOut wire25 ( .okHE(okHE),

.okEH(okEHx[ 3\*65 +: 65 ]),

.ep\_addr(8'h23),

.ep\_datain({24'd0,d\_out\_2[7:0]}));

okWireOut wire26 ( .okHE(okHE),

.okEH(okEHx[ 4\*65 +: 65 ]),

.ep\_addr(8'h24),

.ep\_datain({24'd0,d\_out\_2[15:8]}));

okWireOut wire27 ( .okHE(okHE),

.okEH(okEHx[ 5\*65 +: 65 ]),

.ep\_addr(8'h25),

.ep\_datain({24'd0,d\_out\_3[7:0]}));

okWireOut wire28 ( .okHE(okHE),

.okEH(okEHx[ 6\*65 +: 65 ]),

.ep\_addr(8'h26),

.ep\_datain({24'd0,d\_out\_3[15:8]}));

endmodule

## SPI Module

//This program is a SPI module for performing the operations using the SPI control interface on the camera

module SPI\_master(input wire FSM\_clk,

input wire [6:0] ADDR,

input wire [7:0] D\_IN,

input wire MISO\_IN,

input wire R\_W,

input wire START\_FSM,

output wire SPI\_CLK\_OUT,

output wire MOSI\_OUT,

output reg [7:0] D\_OUT,

output reg SPI\_EN,

output reg READ\_DONE,

output reg WRITE\_DONE,

output reg [7:0] STATE);

//define some registers

reg CLK, MOSI;

assign MOSI\_OUT = MOSI;

assign SPI\_CLK\_OUT = CLK;

//define a local param

localparam STATE\_INIT = 8'd0;

//define some initial values

initial begin

CLK = 1'b0;

MOSI = 1'b0;

SPI\_EN = 1'b0;

STATE = 8'd0;

READ\_DONE = 1'b1;

WRITE\_DONE =1'b1;

D\_OUT = 8'd0;

end

always @(posedge FSM\_clk) begin

case (STATE)

STATE\_INIT : begin

if (START\_FSM == 1'b1) STATE <= 8'd1;

else if (START\_FSM == 1'b0) begin

STATE <= STATE\_INIT;

SPI\_EN <= 1'b0;

CLK <= 1'b0;

WRITE\_DONE <= 1'b0;

READ\_DONE <= 1'b0;

end

end

//begin start sequence

8'd1 : begin

WRITE\_DONE <= 1'b0;

READ\_DONE <= 1'b0;

SPI\_EN <= 1'b1;

CLK <= 1'b0;

MOSI <= R\_W;

STATE <= STATE + 1'b1;

end

//define if register addr is to read or write

8'd2 : begin

CLK <= 1'b0;

STATE <= STATE + 1'b1;

end

8'd3 : begin

CLK <= 1'b1;

STATE <= STATE + 1'b1;

end

8'd4 : begin

CLK <= 1'b1;

STATE <= STATE + 1'b1;

end

//transmit ADDR[6]

8'd5 : begin

CLK <= 1'b0;

MOSI <= ADDR[6];

STATE <= STATE + 1'b1;

end

8'd6 : begin

CLK <= 1'b0;

STATE <= STATE + 1'b1;

end

8'd7 : begin

CLK <= 1'b1;

STATE <= STATE + 1'b1;

end

8'd8 : begin

CLK <= 1'b1;

STATE <= STATE + 1'b1;

end

//transmit ADDR[5]

8'd9 : begin

CLK <= 1'b0;

MOSI <= ADDR[5];

STATE <= STATE + 1'b1;

end

8'd10 : begin

CLK <= 1'b0;

STATE <= STATE + 1'b1;

end

8'd11 : begin

CLK <= 1'b1;

STATE <= STATE + 1'b1;

end

8'd12 : begin

CLK <= 1'b1;

STATE <= STATE + 1'b1;

end

//transmit ADDR[4]

8'd13 : begin

CLK <= 1'b0;

MOSI <= ADDR[4];

STATE <= STATE + 1'b1;

end

8'd14 : begin

CLK <= 1'b0;

STATE <= STATE + 1'b1;

end

8'd15 : begin

CLK <= 1'b1;

STATE <= STATE + 1'b1;

end

8'd16 : begin

CLK <= 1'b1;

STATE <= STATE + 1'b1;

end

//transmit ADDR[3]

8'd17 : begin

CLK <= 1'b0;

MOSI <= ADDR[3];

STATE <= STATE + 1'b1;

end

8'd18 : begin

CLK <= 1'b0;

STATE <= STATE + 1'b1;

end

8'd19 : begin

CLK <= 1'b1;

STATE <= STATE + 1'b1;

end

8'd20 : begin

CLK <= 1'b1;

STATE <= STATE + 1'b1;

end

//transmit ADDR[2]

8'd21 : begin

CLK <= 1'b0;

MOSI <= ADDR[2];

STATE <= STATE + 1'b1;

end

8'd22 : begin

CLK <= 1'b0;

STATE <= STATE + 1'b1;

end

8'd23 : begin

CLK <= 1'b1;

STATE <= STATE + 1'b1;

end

8'd24 : begin

CLK <= 1'b1;

STATE <= STATE + 1'b1;

end

//transmit ADDR[1]

8'd25 : begin

CLK <= 1'b0;

MOSI <= ADDR[1];

STATE <= STATE + 1'b1;

end

8'd26 : begin

CLK <= 1'b0;

STATE <= STATE + 1'b1;

end

8'd27 : begin

CLK <= 1'b1;

STATE <= STATE + 1'b1;

end

8'd28 : begin

CLK <= 1'b1;

STATE <= STATE + 1'b1;

end

//transmit ADDR[0]

8'd29 : begin

CLK <= 1'b0;

MOSI <= ADDR[0];

STATE <= STATE + 1'b1;

end

8'd30 : begin

CLK <= 1'b0;

STATE <= STATE + 1'b1;

end

8'd31 : begin

CLK <= 1'b1;

STATE <= STATE + 1'b1;

end

8'd32 : begin

CLK <= 1'b1;

STATE <= STATE + 1'b1;

end

//begin to decide it is a read or write

//data should be read on the falling edge, data should be sent on the rising edge

//send data in [7]

8'd33 : begin

CLK <= 1'b0;

if (R\_W == 1'b1) MOSI <= D\_IN[7];

STATE <= STATE + 1'b1;

end

8'd34 : begin

CLK <= 1'b0;

STATE <= STATE + 1'b1;

end

//recieve data in [7]

8'd35 : begin

CLK <= 1'b1;

if (R\_W == 1'b0) D\_OUT[7] <= MISO\_IN;

STATE <= STATE + 1'b1;

end

8'd36 : begin

CLK <= 1'b1;

STATE <= STATE + 1'b1;

end

//send data in [6]

8'd37 : begin

CLK <= 1'b0;

if (R\_W == 1'b1) MOSI <= D\_IN[6];

STATE <= STATE + 1'b1;

end

8'd38 : begin

CLK <= 1'b0;

STATE <= STATE + 1'b1;

end

//recieve data in [6]

8'd39 : begin

CLK <= 1'b1;

if (R\_W == 1'b0) D\_OUT[6] <= MISO\_IN;

STATE <= STATE + 1'b1;

end

8'd40 : begin

CLK <= 1'b1;

STATE <= STATE + 1'b1;

end

//send data in [5]

8'd41 : begin

CLK <= 1'b0;

if (R\_W == 1'b1) MOSI <= D\_IN[5];

STATE <= STATE + 1'b1;

end

8'd42 : begin

CLK <= 1'b0;

STATE <= STATE + 1'b1;

end

//recieve data in [5]

8'd43 : begin

CLK <= 1'b1;

if (R\_W == 1'b0) D\_OUT[5] <= MISO\_IN;

STATE <= STATE + 1'b1;

end

8'd44 : begin

CLK <= 1'b1;

STATE <= STATE + 1'b1;

end

//send data in [4]

8'd45 : begin

CLK <= 1'b0;

if (R\_W == 1'b1) MOSI <= D\_IN[4];

STATE <= STATE + 1'b1;

end

8'd46 : begin

CLK <= 1'b0;

STATE <= STATE + 1'b1;

end

//recieve data in [4]

8'd47 : begin

CLK <= 1'b1;

if (R\_W == 1'b0) D\_OUT[4] <= MISO\_IN;

STATE <= STATE + 1'b1;

end

8'd48 : begin

CLK <= 1'b1;

STATE <= STATE + 1'b1;

end

//send data in [3]

8'd49 : begin

CLK <= 1'b0;

if (R\_W == 1'b1) MOSI <= D\_IN[3];

STATE <= STATE + 1'b1;

end

8'd50 : begin

CLK <= 1'b0;

STATE <= STATE + 1'b1;

end

//recieve data in [3]

8'd51 : begin

CLK <= 1'b1;

if (R\_W == 1'b0) D\_OUT[3] <= MISO\_IN;

STATE <= STATE + 1'b1;

end

8'd52 : begin

CLK <= 1'b1;

STATE <= STATE + 1'b1;

end

//send data in [2]

8'd53 : begin

CLK <= 1'b0;

if (R\_W == 1'b1) MOSI <= D\_IN[2];

STATE <= STATE + 1'b1;

end

8'd54 : begin

CLK <= 1'b0;

STATE <= STATE + 1'b1;

end

//recieve data in [2]

8'd55 : begin

CLK <= 1'b1;

if (R\_W == 1'b0) D\_OUT[2] <= MISO\_IN;

STATE <= STATE + 1'b1;

end

8'd56 : begin

CLK <= 1'b1;

STATE <= STATE + 1'b1;

end

//send data in [1]

8'd57 : begin

CLK <= 1'b0;

if (R\_W == 1'b1) MOSI <= D\_IN[1];

STATE <= STATE + 1'b1;

end

8'd58 : begin

CLK <= 1'b0;

STATE <= STATE + 1'b1;

end

//recieve data in [1]

8'd59 : begin

CLK <= 1'b1;

if (R\_W == 1'b0) D\_OUT[1] <= MISO\_IN;

STATE <= STATE + 1'b1;

end

8'd60 : begin

CLK <= 1'b1;

STATE <= STATE + 1'b1;

end

//send data in [0]

8'd61 : begin

CLK <= 1'b0;

if (R\_W == 1'b1) begin

MOSI <= D\_IN[0];

WRITE\_DONE <= 1'b1;

end

STATE <= STATE + 1'b1;

end

8'd62 : begin

CLK <= 1'b0;

STATE <= STATE + 1'b1;

end

//recieve data in [0]

8'd63 : begin

CLK <= 1'b1;

if (R\_W == 1'b0) begin

D\_OUT[0] <= MISO\_IN;

READ\_DONE <= 1'b1;

end

STATE <= STATE + 1'b1;

end

8'd64 : begin

CLK <= 1'b1;

STATE <= STATE + 1'b1;

end

default : STATE <= STATE\_INIT;

endcase

end

endmodule

## I2C Master Module

`timescale 1ns / 1ps

module I2C\_Master(

input FSM\_Clk,

input trigger,

output I2C\_SCL\_1,

inout I2C\_SDA\_1,

input [7:0] dev\_addr,

input [7:0] sub\_addr,

input [7:0] data\_in,

output reg [15:0] Data\_Out\_1,

output reg [15:0] Data\_Out\_2,

output reg [15:0] Data\_Out\_3,

output reg ACK\_bit,

output reg SCL,

output reg SDA,

output reg [9:0] State);

////////////////// Creating Variables /////////////////

reg [7:0] Data\_Counter;

reg [7:0] Next\_State;

reg error\_bit = 1'b1;

reg [7:0] Data\_In;

reg RWflag;

reg [7:0] Data\_1, Data\_2, Data\_3, Data\_4, Data\_5, Data\_6;

reg [7:0] Data\_Out;

wire clkp;

////////////// Creating Variables for I2C Sequences //////////////

localparam STATE\_INIT = 10'd0;

parameter start = 8'd1;

parameter stop = 8'd8;

parameter end\_i2c = 8'd88;

parameter Restart = 8'd3;

parameter S\_ACK = 8'd12;

parameter M\_ACK = 8'd16;

parameter N\_ACK = 8'd20;

parameter Send\_Data = 8'd24;

parameter Get\_Data = 8'd56;

initial begin

SCL = 1'b1;

SDA = 1'b1;

ACK\_bit = 1'b1;

State = 10'd0;

RWflag = 1'b0;

Data\_In = 8'd0;

end

assign I2C\_SDA\_1 = SDA;

assign I2C\_SCL\_1 = SCL;

//////////// I2C State Machine ///////////////////

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 (trigger == 1'b1) State <= 10'd1;

else begin

SCL <= 1'b1;

SDA <= 1'b1;

State <= 10'd0;

end

end

///////// Start Sequence ///////////////

10'd1 : begin

Data\_Counter <= 8'd0;

SCL <= 1'b1;

SDA <= 1'b0;

State <= State + 1'b1;

end

10'd2 : begin

SCL <= 1'b0;

SDA <= 1'b0;

Data\_Counter <= Data\_Counter + 1'b1;

State <= Next\_State;

end

////// Start Sequence ////////

10'd3 : begin

SCL <= 1'b0;

SDA <= 1'b0;

State <= State + 1'b1;

end

10'd4 : begin

SCL <= 1'b0;

SDA <= 1'b1;

State <= State + 1'b1;

end

10'd5 : begin

SCL <= 1'b1;

SDA <= 1'b1;

State <= State + 1'b1;

end

10'd6 : begin

SCL <= 1'b1;

SDA <= 1'b0;

State <= State + 1'b1;

end

10'd7 : begin

SCL <= 1'b0;

SDA <= 1'b0;

Data\_Counter <= Data\_Counter + 1'b1;

State <= Next\_State;

end

/////// Stop Sequence //////////

10'd8 : begin

SCL <= 1'b0;

SDA <= 1'b0;

State <= State + 1'b1;

end

10'd9 : begin

SCL <= 1'b1;

SDA <= 1'b0;

State <= State + 1'b1;

end

10'd10 : begin

SCL <= 1'b1;

SDA <= 1'b1;

State <= State + 1'b1;

end

10'd11 : begin

SCL <= 1'b1;

SDA <= 1'b1;

State <= end\_i2c;

end

/////// Slave Ack //////////

10'd12 : begin

SCL <= 1'b0;

SDA <= 1'bz;

State <= State + 1'b1;

end

10'd13 : begin

SCL <= 1'b1;

State <= State + 1'b1;

end

10'd14 : begin

SCL <= 1'b1;

ACK\_bit <= SDA;

State <= State + 1'b1;

end

10'd15 : begin

SCL <= 1'b0;

Data\_Counter <= Data\_Counter + 1'b1;

State <= Next\_State;

end

///////// Master ACK .//////////

10'd16 : begin

SCL <= 1'b0;

SDA <= 1'b0;

State <= State + 1'b1;

end

10'd17 : begin

SCL <= 1'b1;

SDA <= 1'b0;

State <= State + 1'b1;

end

10'd18 : begin

SCL <= 1'b1;

SDA <= 1'b0;

State <= State + 1'b1;

end

10'd19 : begin

SCL <= 1'b0;

SDA <= 1'b0;

Data\_Counter <= Data\_Counter + 1'b1;

State <= Next\_State;

end

//////// Master No ACK ///////////

10'd20 : begin

SCL <= 1'b0;

SDA <= 1'b1;

State <= State + 1'b1;

end

10'd21 : begin

SCL <= 1'b1;

SDA <= 1'b1;

State <= State + 1'b1;

end

10'd22 : begin

SCL <= 1'b1;

SDA <= 1'b1;

State <= State + 1'b1;

end

10'd23 : begin

SCL <= 1'b0;

SDA <= 1'b1;

Data\_Counter <= Data\_Counter + 1'b1;

State <= Next\_State;

end

//////// Transmit Data In ///////////

10'd24 : begin

SCL <= 1'b0;

SDA <= Data\_In[7];

State <= State + 1'b1;

end

10'd25 : begin

SCL <= 1'b1;

State <= State + 1'b1;

end

10'd26 : begin

SCL <= 1'b1;

State <= State + 1'b1;

end

10'd27 : begin

SCL <= 1'b0;

State <= State + 1'b1;

end

10'd28 : begin

SCL <= 1'b0;

SDA <= Data\_In[6];

State <= State + 1'b1;

end

10'd29 : begin

SCL <= 1'b1;

State <= State + 1'b1;

end

10'd30 : begin

SCL <= 1'b1;

State <= State + 1'b1;

end

10'd31 : begin

SCL <= 1'b0;

State <= State + 1'b1;

end

10'd32 : begin

SCL <= 1'b0;

SDA <= Data\_In[5];

State <= State + 1'b1;

end

10'd33 : begin

SCL <= 1'b1;

State <= State + 1'b1;

end

10'd34 : begin

SCL <= 1'b1;

State <= State + 1'b1;

end

10'd35 : begin

SCL <= 1'b0;

State <= State + 1'b1;

end

10'd36 : begin

SCL <= 1'b0;

SDA <= Data\_In[4];

State <= State + 1'b1;

end

10'd37 : begin

SCL <= 1'b1;

State <= State + 1'b1;

end

10'd38 : begin

SCL <= 1'b1;

State <= State + 1'b1;

end

10'd39 : begin

SCL <= 1'b0;

State <= State + 1'b1;

end

10'd40 : begin

SCL <= 1'b0;

SDA <= Data\_In[3];

State <= State + 1'b1;

end

10'd41 : begin

SCL <= 1'b1;

State <= State + 1'b1;

end

10'd42 : begin

SCL <= 1'b1;

State <= State + 1'b1;

end

10'd43 : begin

SCL <= 1'b0;

State <= State + 1'b1;

end

10'd44 : begin

SCL <= 1'b0;

SDA <= Data\_In[2];

State <= State + 1'b1;

end

10'd45 : begin

SCL <= 1'b1;

State <= State + 1'b1;

end

10'd46 : begin

SCL <= 1'b1;

State <= State + 1'b1;

end

10'd47 : begin

SCL <= 1'b0;

State <= State + 1'b1;

end

10'd48 : begin

SCL <= 1'b0;

SDA <= Data\_In[1];

State <= State + 1'b1;

end

10'd49 : begin

SCL <= 1'b1;

State <= State + 1'b1;

end

10'd50 : begin

SCL <= 1'b1;

State <= State + 1'b1;

end

10'd51 : begin

SCL <= 1'b0;

State <= State + 1'b1;

end

10'd52 : begin

SCL <= 1'b0;

SDA <= Data\_In[0];

State <= State + 1'b1;

end

10'd53 : begin

SCL <= 1'b1;

State <= State + 1'b1;

end

10'd54 : begin

SCL <= 1'b1;

State <= State + 1'b1;

end

10'd55 : begin

SCL <= 1'b0;

Data\_Counter <= Data\_Counter + 1'b1;

State <= Next\_State;

end

////// Reading DATA In ////////////

10'd56 : begin

SCL <= 1'b0;

SDA <= 1'bz;

State <= State + 1'b1;

end

10'd57 : begin

SCL <= 1'b1;

State <= State + 1'b1;

end

10'd58 : begin

SCL <= 1'b1;

Data\_Out[7] <= SDA;

State <= State + 1'b1;

end

10'd59 : begin

SCL <= 1'b0;

State <= State + 1'b1;

end

10'd60 : begin

SCL <= 1'b0;

SDA <= 1'bz;

State <= State + 1'b1;

end

10'd61 : begin

SCL <= 1'b1;

State <= State + 1'b1;

end

10'd62 : begin

SCL <= 1'b1;

Data\_Out[6] <= SDA;

State <= State + 1'b1;

end

10'd63 : begin

SCL <= 1'b0;

State <= State + 1'b1;

end

10'd64 : begin

SCL <= 1'b0;

SDA <= 1'bz;

State <= State + 1'b1;

end

10'd65 : begin

SCL <= 1'b1;

State <= State + 1'b1;

end

10'd66 : begin

SCL <= 1'b1;

Data\_Out[5] <= SDA;

State <= State + 1'b1;

end

10'd67 : begin

SCL <= 1'b0;

State <= State + 1'b1;

end

10'd68 : begin

SCL <= 1'b0;

SDA <= 1'bz;

State <= State + 1'b1;

end

10'd69 : begin

SCL <= 1'b1;

State <= State + 1'b1;

end

10'd70 : begin

SCL <= 1'b1;

Data\_Out[4] <= SDA;

State <= State + 1'b1;

end

10'd71 : begin

SCL <= 1'b0;

State <= State + 1'b1;

end

10'd72 : begin

SCL <= 1'b0;

SDA <= 1'bz;

State <= State + 1'b1;

end

10'd73 : begin

SCL <= 1'b1;

State <= State + 1'b1;

end

10'd74 : begin

SCL <= 1'b1;

Data\_Out[3] <= SDA;

State <= State + 1'b1;

end

10'd75 : begin

SCL <= 1'b0;

State <= State + 1'b1;

end

10'd76 : begin

SCL <= 1'b0;

SDA <= 1'bz;

State <= State + 1'b1;

end

10'd77 : begin

SCL <= 1'b1;

State <= State + 1'b1;

end

10'd78 : begin

SCL <= 1'b1;

Data\_Out[2] <= SDA;

State <= State + 1'b1;

end

10'd79 : begin

SCL <= 1'b0;

State <= State + 1'b1;

end

10'd80 : begin

SCL <= 1'b0;

SDA <= 1'bz;

State <= State + 1'b1;

end

10'd81 : begin

SCL <= 1'b1;

State <= State + 1'b1;

end

10'd82 : begin

SCL <= 1'b1;

Data\_Out[1] <= SDA;

State <= State + 1'b1;

end

10'd83 : begin

SCL <= 1'b0;

State <= State + 1'b1;

end

10'd84 : begin

SCL <= 1'b0;

SDA <= 1'bz;

State <= State + 1'b1;

end

10'd85 : begin

SCL <= 1'b1;

State <= State + 1'b1;

end

10'd86 : begin

SCL <= 1'b1;

Data\_Out[0] <= SDA;

State <= State + 1'b1;

end

///// Adjust for Incremental Reading ///////////

10'd87 : begin

SCL <= 1'b0;

case(Data\_Counter)

8'd8 : Data\_1 <= Data\_Out;

8'd10 : Data\_2 <= Data\_Out;

8'd12 : Data\_3 <= Data\_Out;

8'd14 : Data\_4 <= Data\_Out;

8'd16 : Data\_5 <= Data\_Out;

8'd18 : Data\_6 <= Data\_Out;

default : Data\_6 <= Data\_Out;

endcase

if (Data\_Counter == 8'd18) begin

Data\_Out\_1 <= {Data\_2, Data\_1};

Data\_Out\_2 <= {Data\_4, Data\_3};

Data\_Out\_3 <= {Data\_6, Data\_5};

end

Data\_Counter <= Data\_Counter + 1'b1;

State <= Next\_State;

end

10'd88 : begin

if(trigger == 1'b0) begin

State <= STATE\_INIT;

end

end

default : begin

error\_bit <= 0;

end

endcase

end

///////// Case Statement to decide on which I2C sequence to run //////////

always @(\*) begin

Data\_In = 1'bz;

case (Data\_Counter)

8'd0 : begin

Next\_State = Send\_Data;

Data\_In = {dev\_addr[7:1], 1'b0};

end

8'd1 : begin

Next\_State = S\_ACK;

Data\_In = {dev\_addr[7:1], 1'b0};

end

8'd2 : begin

Next\_State = Send\_Data;

Data\_In = sub\_addr;

end

8'd3 : begin

Next\_State = S\_ACK;

Data\_In = sub\_addr;

end

8'd4 : begin

if (sub\_addr[7] == 1'b0) begin

Next\_State = Send\_Data;

end

else begin

Next\_State = Restart;

Data\_In = data\_in;

end

end

8'd5 : begin

if (sub\_addr[7] == 1'b0) begin

Next\_State = S\_ACK;

Data\_In = data\_in;

end

else begin

Next\_State = Send\_Data;

Data\_In = {dev\_addr[7:1], 1'b1};

end

end

8'd6 : begin

if (sub\_addr[7] == 1'b0) begin

Next\_State = stop;

end

else begin

Next\_State = S\_ACK;

Data\_In = {dev\_addr[7:1], 1'b1};

end

end

8'd7 : begin

Next\_State = Get\_Data;

end

8'd8 : begin

Next\_State = M\_ACK;

end

8'd9 : begin

Next\_State = Get\_Data;

end

8'd10 : begin

Next\_State = M\_ACK;

end

8'd11 : begin

Next\_State = Get\_Data;

end

8'd12 : begin

Next\_State = M\_ACK;

end

8'd13 : begin

Next\_State = Get\_Data;

end

8'd14 : begin

Next\_State = M\_ACK;

end

8'd15 : begin

Next\_State = Get\_Data;

end

8'd16 : begin

Next\_State = M\_ACK;

end

8'd17 : begin

Next\_State = Get\_Data;

end

8'd18 : begin

Next\_State = N\_ACK;

## PWM Controller Module

module PWM\_Controller(

input PWM\_Clk,

output PWM\_Out,

input [15:0] DUTY\_CYC,

input [15:0] pulse\_cnt,

input en

);

reg [16:0] counter, pulse\_counter;

reg PWM\_OUT\_REG, cnt;

assign PWM\_Out = PWM\_OUT\_REG;

initial begin

counter = 16'd0;

pulse\_counter = 16'd0;

cnt = 0;

end

always @(posedge PWM\_Clk) begin

counter <= counter + cnt;

if (en && (pulse\_counter < pulse\_cnt)) begin

cnt <= 1;

if (counter >= DUTY\_CYC)

PWM\_OUT\_REG <= 1'd0;

else if (counter == 16'd1000) begin

counter <= 16'd0;

PWM\_OUT\_REG <= 1'd0;

pulse\_counter <= pulse\_counter + 1;

end

else

PWM\_OUT\_REG <= 1'd1;

end

else begin

counter <= 8'd0;

pulse\_counter <= 'd0;

cnt <= 0;

PWM\_OUT\_REG <= 1'd0;

end

end

endmodule

## 