Final Project Report - Real time Object Tracking using an FPGA and Sensors

Introduction

In this project, I implemented a real-time object tracking system using the Opal Kelly XEM7310 FPGA platform and the ECE 437 sensor board. I was able to achieve a continuous, high-speed flow of data between the FPGA and a host PC, allowing me to capture and process image frames at rates of 25 frames per second (FPS). The core of this setup leveraged the CMV300 imaging sensor, whose SPI-based interface was managed by a custom Verilog state machine. The image data traveled through the FPGA's OkBTPipeOut module to the PC at high throughput and enabled real-time visibility of the captured scene.

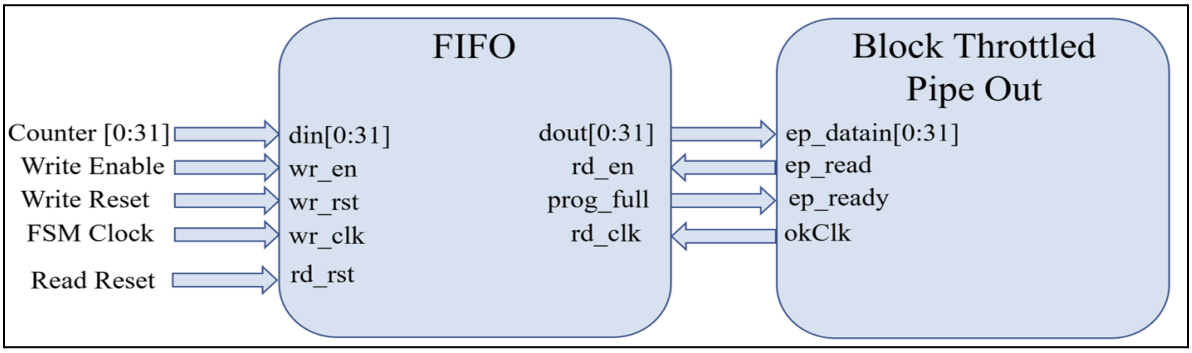
On the software side, I utilized Python to communicate with the FPGA and firmware. Within this environment, I interfaced with the FPGA via the Opal Kelly FrontPanel API to configure the sensor, retrieve image frames, and send motor control commands. Additionally, I employed PyVISA to communicate with various benchtop instruments (specifically, the DC power supply) to measure the motor's electrical characteristics and correlate them with the system's mechanical performance. Through I2C, I integrated the LSM303DLHC accelerometer and magnetometer to collect acceleration and heading data at a rate of at least 100 samples per second. By merging these sensor readings, I could characterize how different voltage levels influenced the motor's acceleration and overall responsiveness.

In parallel with the hardware integration, I incorporated OpenCV's tracking capabilities into the Python environment. Over time, I refined the thresholding, frame differencing, and center-of-mass calculations to ensure accurate positional feedback. Then, I established a control loop that instructed the motor to rotate the camera's platform clockwise or counter-clockwise based on the object's relative location in the image.

Throughout this process, I relied on multiple Verilog modules, to handle the distinct timing requirements of the SPI and I2C transactions, the high-speed image data streaming, and the motor PWM signals. To ensure clean and reliable data transfers between these different timing domains, I used FIFO buffers as a bridge across clock boundaries. For the FIFO module, I made use of the Vivado FIFO IP.

Implementation Specifics

This is a high level block diagram of how the BTPipeOut communicated with the FIFO to transfer frames collected by the CMV300 sensor over to the PC



Python Tracking using OpenCV Library

I used Python to primarily control our FPGA. The Python code sent in values and triggers to the FPGA while the FPGA received the outputs from the sensors and this was overlaid by OpenCV. The values for the frames were received in a numpy array. I did some bit manipulation and added some granularity and the image was displayed. The motor was controlled by sending in bursts of pulses for how long the motor should run for. The I2C was initialized to read 100 readings per second.

Hardware/Firmware part of code (Verilog)

To the Verilog firmware, I integrated several previously developed state machines in the lab - originally designed for I2C communication, motor control, and SPI-based camera interfacing - into one single hardware design. To achieve this, I began by synchronizing multiple clock domains using a dedicated Clock Generator block. The SPI state machine communicating with the CMV300 image sensor was clocked at 20 MHz, ensuring I could reliably configure and read data from the camera at the required data rate. Simultaneously, the I2C state machine managing the LSM303DLHC accelerometer and magnetometer ran at 400 kHz, providing stable, high-frequency communication for retrieving heading and acceleration data at or above 100 samples per second. Additionally, the motor controller operated at a much lower frequency of around 200 Hz, suitable for generating PWM signals to drive the motor and rotate the camera's platform.

To send the data over, I utilized a FIFO buffer in the data path, bridging the camera's pixel output and the high-throughput BTPipeOut interface. This FIFO enabled me to accumulate data from the CMOS sensor efficiently before transferring it in bursts to the PC, ensuring the PC software could process frames at a rate of at least 20 frames per second. By buffering the image data, I reduced jitter and maintained a steady stream of frames to the PC without saturating the communication channel or overburdening the host-side software.

Initially, I experimented with various timing strategies for frame acquisition. After some tuning, I found that eliminating any artificial delay between frame requests helped maintain a steady output of images.

Sensor Usage

Data	**Sensor**	**Communication Protocol**
Acceleration/Magnetic Field Heading	LSM303DLHC	I2C
Motor Control	PMOD DHB1	PWM
Imaging	CMV300	SPI

Spatial Noise in the Image Sensor:

Spatial noise was calculated by examining the pixel intensity distribution across the entire sensor array (648x488) while the sensor was completely blocked from receiving external light. By evaluating the standard deviation of the intensities across all pixels in a single "dark" frame, I obtained a spatial noise value of about 5.1654. Spatial noise thus represents the inherent pixel-to-pixel variation within the sensor.

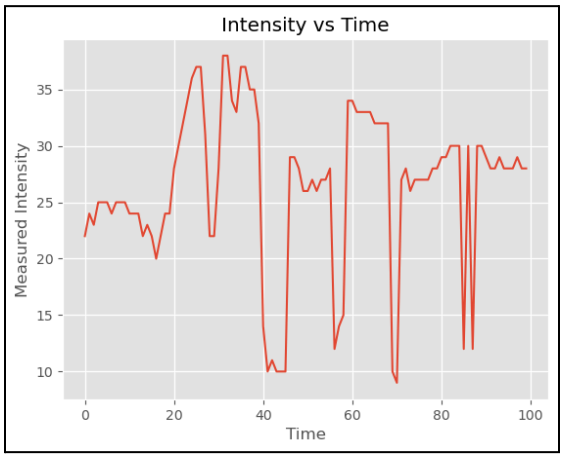
Spatial noise depends the sensor's hardware characteristics and fabrication differences across pixels. While spatial noise cannot be entirely eliminated, it can be reduced by maintaining a high signal-to-noise ratio

Temporal Noise in the Image Sensor:

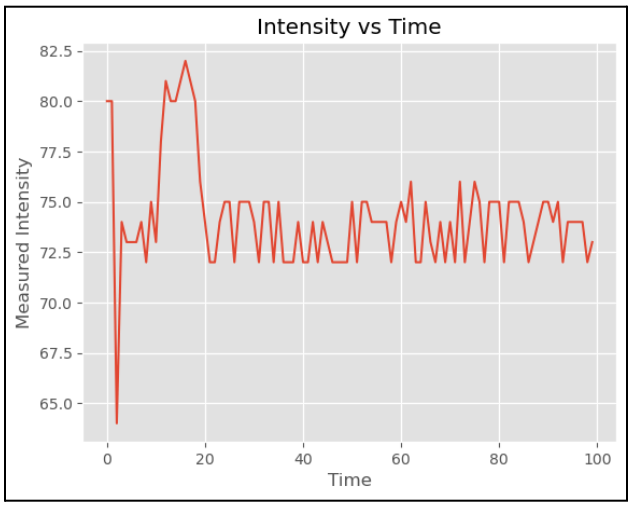
I computed temporal noise by observing the intensity of a single pixel - specifically pixel (50,50) - across a set of 100 consecutive frames. I then calculated the standard deviation of that pixel's intensity values over time. In my measurements, the temporal noise was approximately 14.268. This metric effectively tells us about how much a single pixel's reading fluctuates between frames, given consistent illumination and exposure conditions.

Based on what we discussed in lecture, temporal noise can be reduced through techniques such as increasing exposure time and implementing frame stacking (to average out fluctuations).

<u>10msec integration time (increased exposure time):</u>

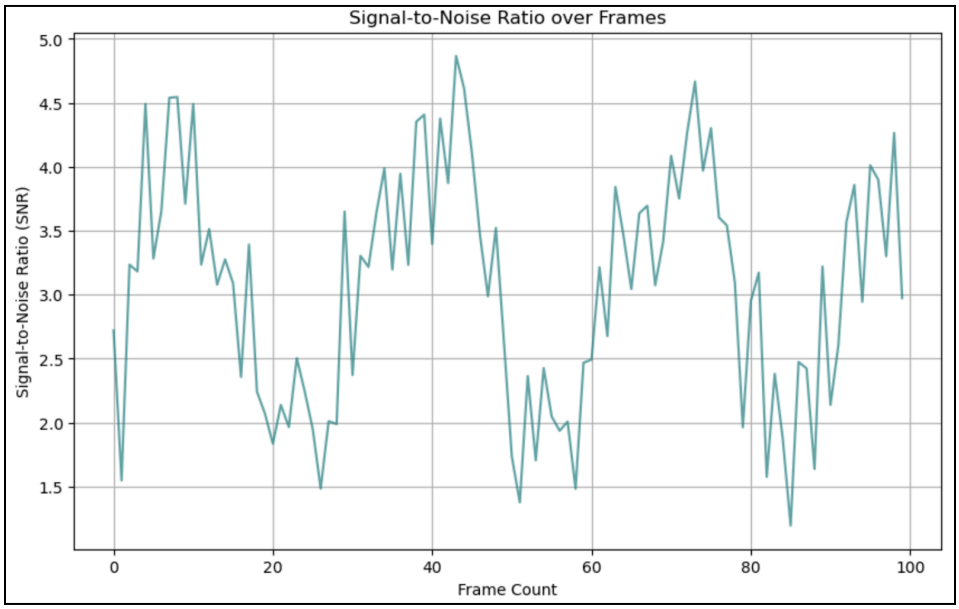


<u>1 msec integration time (reduced exposure time):</u>



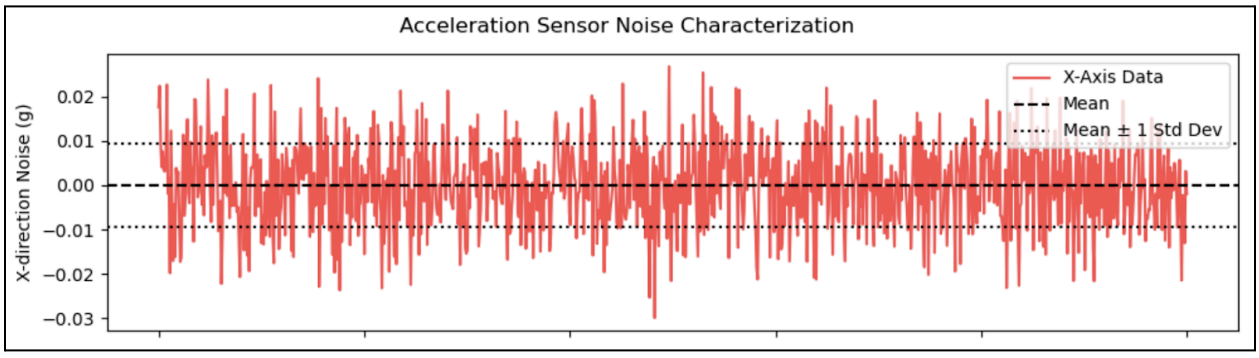
Signal-to-Noise Ratio (SNR)

I computed the signal-to-noise ratio (SNR) by taking the average intensity of pixels (50,50) across the same 100 frames, I obtained an average intensity of approximately 3.54. Comparing this average intensity to the standard deviation (around 0.5) gives me an SNR value, reflecting how "clean" or "noisy" the signal is.



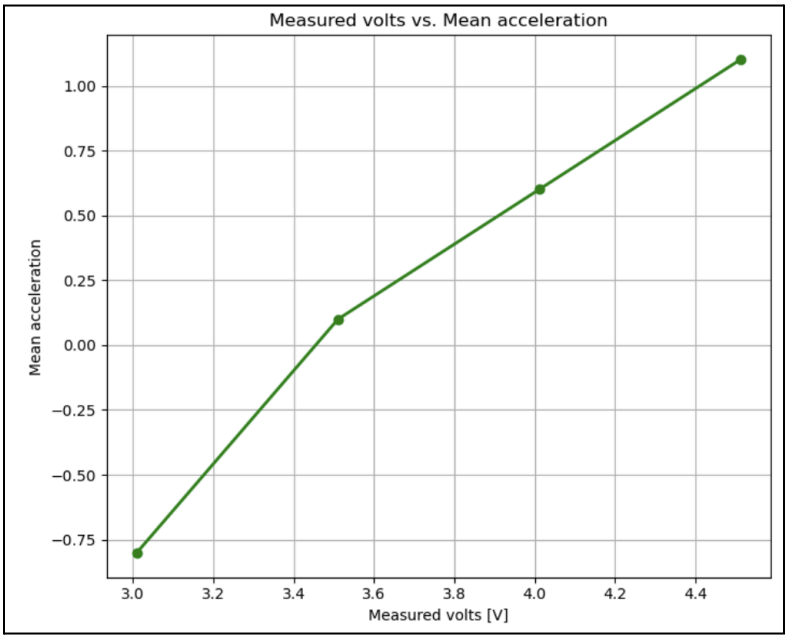
Noise in the Acceleration Sensor

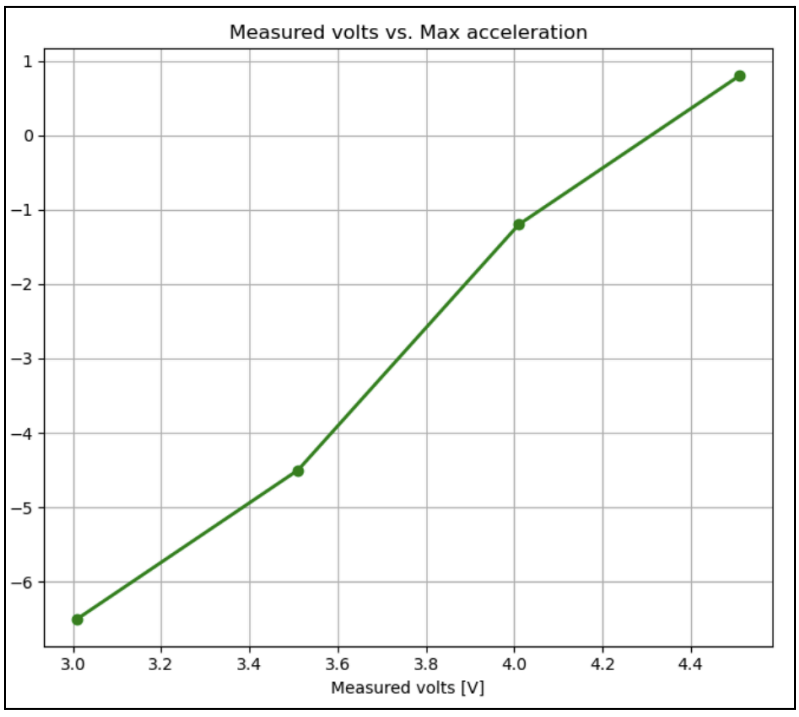
Along with looking into the noise properties of the image sensor, I also took a closer look at the noise in the acceleration sensor data. I used the LSM303DLHC accelerometer, which I read via I2C at 100 samples per second. This sensor, like most, has some built-in noise, caused by things like imperfections in the sensor itself, electromagnetic interference from the environment, and quantization errors from the digital interface. To get a better sense of the noise, I took a series of acceleration readings while keeping the sensor as still as possible. By looking at the data over time, I calculated the standard deviation of the acceleration in each of the X, Y, and Z axes. The noise levels I observed reflect the baseline uncertainty of the accelerometer when the system is stationary.



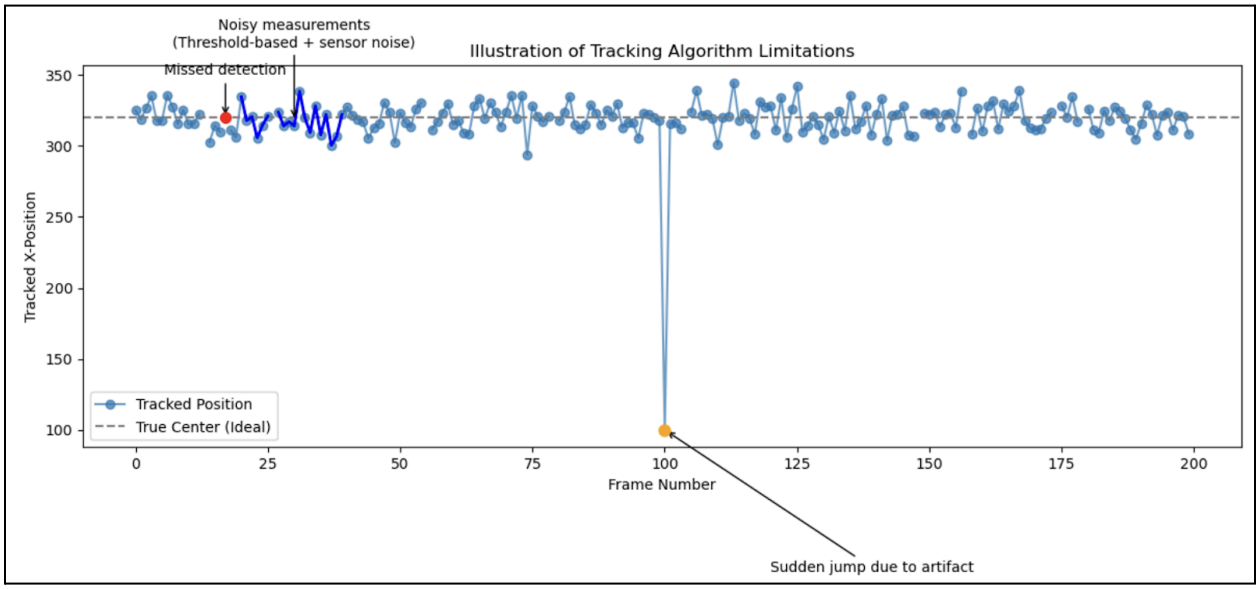
Acceleration of the Sensor Board as a function of different Applied Voltages

In my observations, the relationship between acceleration and applied power voltage starts off as linear at lower voltages. As I increase the voltage, I notice that the acceleration increases proportionally, which makes sense because the motor is producing more torque. However, once I hit around 4V to 5V, things change. The relationship becomes non-linear, likely due to factors like friction and the motor reaching its limits. At that point, I see the acceleration plateau, suggesting that the motor just can't take any more power without risking damage.





Accuracy of Tracking Algorithms



At its core, the tracking logic relies on identifying bright pixels within a single frame using a fixed intensity threshold, then computing a weighted center of mass to determine the object's location. While this approach provides a simple and fast means of estimating the object's position, it has several limitations:

1. **Threshold Sensitivity:**
   The choice of a single brightness threshold is highly sensitive to changes in lighting conditions and object appearance. If the object's intensity is too close to the background or if ambient lighting fluctuates, the threshold might fail to isolate the object reliably. This can lead to erratic detection or the object briefly "disappearing" from the algorithm's perspective.
2. **Noise and Artifacts:**
   Low-level camera noise, hot pixels, or other bright artifacts in the frame can cause false positives that distort the center-of-mass calculation. Even though I apply a weighted average based on pixel intensities, random bright spots or sensor noise can skew the computed position slightly, reducing the algorithm's accuracy.
3. **Resolution and Frame Rate Constraints:**
   The CMV300 sensor provides a fixed resolution, and the system processes discrete pixel intensities. The algorithm's center-of-mass approach inherently rounds positions to pixel coordinates, imposing a limit on spatial accuracy. Although sub-pixel estimation techniques exist, I have not implemented them here. Additionally, since the algorithm runs in real-time at a certain

frame rate, fast-moving objects can “jump” between frames, reducing temporal accuracy and making the tracking appear jerky or lagging behind the object.

4. **Smoothing and Latency Trade-Offs:**
   To improve stability, I implemented exponential smoothing of recent positions. While this can reduce jitter, it introduces latency and can cause the tracked position to “lag” behind the object’s true location. If the object moves quickly, the smoothing may prevent the algorithm from keeping the box or indicator precisely centered, effectively lowering spatial accuracy when rapid changes occur.
5. **Lack of Advanced Tracking Features:**
   More sophisticated computer vision algorithms - such as those using optical flow, template matching, or machine learning-based object detectors - could improve robustness and accuracy. By contrast, the current method is intentionally minimalistic, making it more prone to accuracy issues when faced with complex or dynamic scenes.

Conclusion

In conclusion, I successfully integrated the FPGA-based firmware with Python software to enable real-time object tracking, high-speed image acquisition, and reliable sensor data processing. Despite the inherent challenges - such as noise in both the image sensor and the accelerometer, and the sensitivity of the simple threshold-based tracking method - I managed to achieve a system that responds to object movement and adjusts the image sensor’s orientation accordingly. The careful selection of clock domains, the use of FIFOs for stable data transfer, and the employment of I2C and SPI protocols were all critical steps in ensuring smooth operation. Although the tracking algorithm could be improved by more sophisticated techniques, the work I have done here provides accurate tracking capability. This can serve as a starting point for future enhancements, such as more advanced image processing algorithms or improved noise mitigation strategies.

Python code

```
################################################################################
######################
# Function to swap every 4 bytes in a buffer
def swap(buf):
    buf_rearranged = bytearray(len(buf))
    for i in range(0, len(buf), 4):
        chunk = buf[i:i + 4]  # Extract 4-byte chunk
```

```
        chunk[:] = chunk[::-1]  # Reverse the entire 4-byte chunk
        buf_rearranged[i:i + 4] = chunk  # Store the modified chunk in the
rearranged array
    return buf_rearranged

# Function to swap upper and lower byte of a 16 bit value passed in
def bits(value):
    # Mask to extract upper 8 bits (most significant bits)
    upper_mask = (value & 0xFF00) >> 8
    # Mask to extract lower 8 bits (least significant bits)
    lower_mask = (value & 0x00FF) << 8
    # Combine the swapped bits
    swapped_value = upper_mask | lower_mask
    if swapped_value & 0x8000:
       signed_value = -((swapped_value ^ 0xFFFF) + 1)
    else:
       signed_value = swapped_value
    return signed_value

##########################################################################
######################
# IMPORTS
import numpy as np
import pyvisa as visa  # You should pip install pyvisa and restart the
kernel.
import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.style.use('ggplot')

from PIL import Image
import cv2
import time
import threading
import sys
import os

# import various libraries necessary to run your Python code
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
device_lock = threading.Lock()  # For thread-safe device access

################################################################################
######################
# FRONTPANEL INITIALIZATION
# Define FrontPanel device variable, open USB communication and
# load the bit file in the FPGA
dev = ok.okCFrontPanel()  # define a device for FrontPanel communication
SerialStatus = dev.OpenBySerial("")  # open USB communication with the OK
board
# We will NOT load the bit file because it will be loaded using JTAG
interface from Vivado
# Check if FrontPanel is initialized correctly and if the bit file is
loaded.
# Otherwise terminate the program
print("------------------------------------------------")

if SerialStatus == 0:
    print("FrontPanel host interface was successfully initialized.")
else:
    print("FrontPanel host interface not detected. The error code number
is:" + str(int(SerialStatus)))
    print("Exiting the program.")
    sys.exit()


################################################################################
############
# SPI INITIALIZATION AND CONFIRMATION FOR CAMERA SENSOR

# dictionary for spi values and addresses
dic =
{1:232,2:1,43:11,55:120,57:3,58:44,59:240,60:10,68:1,69:9,80:2,83:187,97:24
0,98:10,

100:112,101:98,102:34,103:64,106:94,107:110,108:91,109:82,110:80,117:91}

# initializing spi registers
for i in dic:
    with device_lock:
        dev.SetWireInValue(0x01, dic[i])
        dev.UpdateWireIns()
```

```
        dev.SetWireInValue(0x02, i)
        dev.UpdateWireIns()

        SPI_Control = 3  # send a "go" signal to the SPI FSM
        dev.SetWireInValue(0x00, SPI_Control)
        dev.UpdateWireIns()  # Update the WireIns

        SPI_Control = 0  # send a "stop" signal to the FSM
        dev.SetWireInValue(0x00, SPI_Control)
        dev.UpdateWireIns()  # Update the WireIns

# reading spi registers to confirm values
# print("Reading SPI registers to confirm values:")
for i in dic:
    with device_lock:
        dev.SetWireInValue(0x02, i)
        dev.UpdateWireIns()
        SPI_Control = 1  # send a "go" signal to the FSM
        dev.SetWireInValue(0x00, SPI_Control)
        dev.UpdateWireIns()  # Update the WireIns

        SPI_Control = 0  # send a "stop" signal to the FSM
        dev.SetWireInValue(0x00, SPI_Control)
        dev.UpdateWireIns()  # Update the WireIns
        dev.UpdateWireOuts()
        z = dev.GetWireOutValue(0x20)
        # print(f"Register {i}: {z}")

################################################################################
#############
# I2C/MOTOR PREPARATION SECTION
motor_control = 0  # 9

i2c_control = 0  # 4

i2c_acc_slave_address = 0x32  # 5
i2c_acc_ctrl_address = 0x20  # 6
i2c_acc_ctrl_data = 0x57  # 7
i2c_acc_some_value = 0xA8  # 8

# Initialize acceleration sensor (if needed)
with device_lock:
    # setting addresses for acceleration data collection
```

```
    dev.SetWireInValue(0x05, i2c_acc_slave_address)  # slave address
    dev.UpdateWireIns()
    dev.SetWireInValue(0x06, i2c_acc_ctrl_address)  # control register
address
    dev.UpdateWireIns()
    dev.SetWireInValue(0x07, i2c_acc_ctrl_data)  # control register data
    dev.UpdateWireIns()
    dev.SetWireInValue(0x08, i2c_acc_some_value)  # x data with top bit set
as 1 for auto increment
    dev.UpdateWireIns()

##############################################################################
######################
# FUNCTIONS FOR FRAME ACQUISITION AND ACCELERATION READING

def control_motor(direction):
    global dev, device_lock
    with device_lock:
        if direction == 'CW':
            # Set motor to rotate clockwise
            dev.SetWireInValue(0x09, 1)  # Assuming 1 means clockwise
            dev.UpdateWireIns()
            # time.sleep(0.5)
            # dev.SetWireInValue(0x09, 0)  # Assuming 0 stops the motor
            # dev.UpdateWireIns()
        elif direction == 'CCW':
            # Set motor to rotate counter-clockwise
            dev.SetWireInValue(0x09, 3)  # Assuming 3 means
counter-clockwise
            dev.UpdateWireIns()
            # time.sleep(0.5)
            # dev.SetWireInValue(0x09, 0)  # Assuming 0 stops the motor
            # dev.UpdateWireIns()
        elif direction == 'STOP':
            # Stop the motor
            dev.SetWireInValue(0x09, 0)  # Assuming 0 stops the motor
            dev.UpdateWireIns()
            # time.sleep(0.5)
        else:
            # Default action
            dev.SetWireInValue(0x09, 0)
            dev.UpdateWireIns()
            # time.sleep(0.5)
```

```
def frame_acquisition():
    global dev, device_lock
    width, height = 648, 484
    buf = bytearray(314368)
    buf2 = bytearray(height * width)

    # Threshold for bright spots
    brightness_threshold = 20  # Adjust based on your flashlight brightness

    # Define thirds of the frame
    left_third = width // 3
    right_third = (width * 2) // 3

    # Position smoothing
    position_history = []
    history_size = 10  # Number of past positions to average

    # Start time for FPS calculation
    start_time = time.time()
    frame_count = 0

    while True:
        # Acquire frame
        with device_lock:
            dev.ActivateTriggerIn(0x40, 0)
            dev.ReadFromBlockPipeOut(0xa0, 1024, buf)

        # Process buffer into frame
        buf2 = buf[0:height * width]
        image = np.frombuffer(buf2, dtype=np.uint8)
        reshaped_array = image.reshape(int(len(buf2) / 4), 4)
        reversed_array = np.flip(reshaped_array, axis=1)
        result_array = reversed_array.flatten()
        frame = np.reshape(result_array, (height, width))

        motor_direction = 'No Movement'

        # Threshold to find bright spots (flashlight)
        _, thresh = cv2.threshold(frame, brightness_threshold, 255,
cv2.THRESH_BINARY)

        # Find coordinates of bright pixels
```

```
        y_coords, x_coords = np.nonzero(thresh)

        if len(x_coords) > 0:
            # Calculate center of mass of bright region
            # Weight pixels by their intensity for more accurate center
            intensities = frame[y_coords, x_coords]
            x_center = int(np.average(x_coords, weights=intensities))
            y_center = int(np.average(y_coords, weights=intensities))

            # Add to position history
            position_history.append((x_center, y_center))
            if len(position_history) > history_size:
                position_history.pop(0)

            # Calculate smoothed position
            if len(position_history) > 0:
                # Use exponential moving average for smooth tracking
                smoothed_x = int(sum([pos[0] * (0.8 ** i) for i, pos in
enumerate(reversed(position_history))]) /
                                 sum([0.8 ** i for i in
range(len(position_history))]))
                smoothed_y = int(sum([pos[1] * (0.8 ** i) for i, pos in
enumerate(reversed(position_history))]) /
                                 sum([0.8 ** i for i in
range(len(position_history))]))

                # Draw tracking point
                cv2.circle(frame, (smoothed_x, smoothed_y), 5, (255), -1)

                # Determine motor direction based on smoothed position
                if smoothed_x < left_third:
                    motor_direction = 'Counter Clockwise'
                    control_motor('CCW')
                elif smoothed_x > right_third:
                    motor_direction = 'Clockwise'
                    control_motor('CW')
                else:
                    motor_direction = 'Centered'
                    control_motor('STOP')

        # Draw frame divisions
        cv2.line(frame, (left_third, 0), (left_third, height), (128), 1)
        cv2.line(frame, (right_third, 0), (right_third, height), (128), 1)
```

```
        # Calculate and display FPS
        frame_count += 1
        elapsed_time = time.time() - start_time
        fps = frame_count / elapsed_time

        # Display information
        cv2.putText(frame, f"Motor Direction: {motor_direction}", (10, 30),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.7, (255), 2)
        cv2.putText(frame, f"FPS: {fps:.2f}", (10, 50),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.7, (255), 2)

        # Display the frame
        cv2.imshow('Frame', frame)

        if cv2.waitKey(1) & 0xFF == ord('q'):
            break

    cv2.destroyAllWindows()




def read_acceleration():
    global dev, device_lock
    # Function to read acceleration data at 100 samples per second
    sample_interval = 1 / 100  # 100 samples per second
    while True:
        start_time = time.time()

        # Read acceleration data
        with device_lock:
            # setting addresses for acceleration data collection
            dev.SetWireInValue(0x05, i2c_acc_slave_address)  # slave
address
            dev.UpdateWireIns()
            dev.SetWireInValue(0x06, i2c_acc_ctrl_address)  # control
register address
            dev.UpdateWireIns()
            dev.SetWireInValue(0x07, i2c_acc_ctrl_data)  # control register
data
            dev.UpdateWireIns()
```

```
            dev.SetWireInValue(0x08, i2c_acc_some_value)  # x data with top
bit set as 1 for auto increment
            dev.UpdateWireIns()

            i2c_control = 1  # send a "go" signal to the FSM
            dev.SetWireInValue(0x04, i2c_control)
            dev.UpdateWireIns()  # Update the WireIns

            time.sleep(0.005)  # Wait for 5ms, adjust as needed

            i2c_control = 0  # send a "stop" signal to the FSM
            dev.SetWireInValue(0x04, i2c_control)
            dev.UpdateWireIns()  # Update the WireIns

            dev.UpdateWireOuts()

            x = dev.GetWireOutValue(0x21)
            y = dev.GetWireOutValue(0x22)
            z = dev.GetWireOutValue(0x23)

        x_acc = bits(x)/16200.5333333*9.8
        y_acc = bits(y)/16142.93333333*9.8
        z_acc = bits(z)/16200.5333333*9.8

        # Display the acceleration data
        print(f"Acceleration: X={x_acc:.2f} m/s^2, Y={y_acc:.2f} m/s^2,
Z={z_acc:.2f} m/s^2")

        # Wait until next sample time
        elapsed_time = time.time() - start_time
        time_to_wait = sample_interval - elapsed_time
        if time_to_wait > 0:
            time.sleep(time_to_wait)

##############################################################################
######################
# START THREADS FOR FRAME ACQUISITION AND ACCELERATION READING

# Start threads
dev.SetWireInValue(0x10, 50)
dev.UpdateWireIns()

frame_thread = threading.Thread(target=frame_acquisition)
```

```
acc_thread = threading.Thread(target=read_acceleration)

frame_thread.start()
acc_thread.start()

# Wait for threads to finish (they won't unless interrupted)
try:
    while True:
        time.sleep(1)
except KeyboardInterrupt:
    print("Program interrupted by user.")

# Clean up
# power_supply.write("OUTPUT OFF")
# power_supply.close()
dev.Close()
print("Program terminated.")
```

Verilog code (Imaging module):

```
`timescale 1ns / 1ps

module finaltoplevel(


   output CVM300_CLK_IN,
   input CVM300_CLK_OUT,
   output CVM300_SYS_RES_N,
   output CVM300_FRAME_REQ,
   input CVM300_Enable_LVDS,
   input CVM300_Line_valid,
   input CVM300_Data_valid,
   input [9:0] CVM300_D,

   output CVM300_SPI_EN,
   input CVM300_SPI_OUT,
   output CVM300_SPI_CLK,
   output  CVM300_SPI_IN,
```

```
    output [7:0] led,
    input sys_clkn,
    input sys_clkp,

    input  [4:0] okUH,
    output [2:0] okHU,
    inout  [31:0] okUHU,
    inout  okAA
);


    ////////////////////////////////////////////////////////////////
GLOBAL CONTROL VARIABLES
    wire [31:0] SPI_control, I2C_control, motor_control;
    wire TrigerEvent;


    // MAIN: WRITTEN TO BY CLOCK GENERATOR
    wire ILA_Clk, ACK_bit, FSM_Clk;




////////////////////////////////////////////////////////////////////
I2C STUFF
    // monitoring signal from module
    wire SCL, SDA;


    // monitoring signal from module
    wire [8:0] I2C_State;


    // coming in from okwirein
    wire [31:0] tempregadd;
    wire [31:0] regadd;
    wire [31:0] regval;
    wire [31:0] i2c_slave_address;
```

```
//////////////////////////////////////////////////////////////////
/////// VALUES FOR FIFO STUFF TO WORK

   localparam STATE_INIT               = 8'd0;
   localparam STATE_RESET              = 8'd1;
   localparam STATE_DELAY              = 8'd2;
   localparam STATE_RESET_FINISHED     = 8'd3;
   localparam STATE_FRAME_REQ          = 8'd4;
   localparam STATE_FINISH             = 8'd5;

   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, write_enable = 0; // initialize
without initial block??

   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;

   wire clkout;
   assign clkout = CVM300_CLK_OUT;

   reg frame_req = 0;
   wire [7:0] dat;
   assign dat = CVM300_D[9:2];
   assign CVM300_FRAME_REQ = frame_req;

//////////////////////////////////////////////////////////////////
/////// SPI STUFF
   wire miso;
   wire mosi;
```

```
   wire en;
   wire sclk;
   wire [31:0] val;
   wire [31:0] add;
   wire [31:0] data;

   spi_toplevel spi(
   .led(),
   .FSM_Clk(FSM_Clk),
   .ILA_Clk(ILA_Clk),
   .sys_clkn(sys_clkn),
   .sys_clkp(sys_clkp),
   .State(SPI_STATE),
   .PC_control(SPI_control),
   .CVM300_SPI_EN(CVM300_SPI_EN),
   .CVM300_SPI_OUT(CVM300_SPI_OUT),
   .CVM300_SPI_CLK(CVM300_SPI_CLK),
   .CVM300_SPI_IN(CVM300_SPI_IN),
   .miso(miso),
   .mosi(mosi),
   .sclk(sclk),
   .en(en),
   .data(data[7:0]),
   .val(val),
   .add(add[7:0])
   );



////////////////////////////////////////////////////////////////////
////////////////// OKHOST STUFF

   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)
   );


////////////////////////////////////////////////////////////////////////
////////////////////// FIFO MODULE

    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), // write to
FIFO when line valid signal goes high... what about data valid?
       .rd_en(FIFO_read_enable),
       .dout(FIFO_data_out),
       .full(FIFO_full),
       .empty(FIFO_empty),
       .prog_full(FIFO_BT_BlockSize_Full)
   );



    ////////////////////////////////////////////////////// OK
WIRE INS FOR CONTROLLING SPI AND CAMERA

    okWireIn wire0 (  .okHE(okHE),
                      .ep_addr(8'h00),
                      .ep_dataout(SPI_control));
```

```
    okWireIn wire1 (   .okHE(okHE),
                       .ep_addr(8'h01),
                       .ep_dataout(data));

    okWireIn wire2 (   .okHE(okHE),
                       .ep_addr(8'h02),
                       .ep_dataout(add));

   okWireIn wire4 (    .okHE(okHE),
                       .ep_addr(8'h04),
                       .ep_dataout(I2C_control));
   okWireIn wire5 (    .okHE(okHE),
                       .ep_addr(8'h05),
                       .ep_dataout(i2c_slave_address));
   okWireIn wire6 (    .okHE(okHE),
                       .ep_addr(8'h06),
                       .ep_dataout(tempregadd));
   okWireIn wire7 (    .okHE(okHE),
                       .ep_addr(8'h07),
                       .ep_dataout(regval));
   okWireIn wire8 (    .okHE(okHE),
                       .ep_addr(8'h08),
                       .ep_dataout(regadd));


   // motor wireins
   okWireIn wire9 (    .okHE(okHE),
                       .ep_addr(8'h09),
                       .ep_dataout(motor_control));


   okWireIn wire10 (    .okHE(okHE),
                       .ep_addr(8'h10),
                       .ep_dataout(cycle_count));
//   reg [8:0]cnt;
//   reg clk;
//   always @(posedge FSM_Clk) begin
//   if (cnt >= 10) begin
//       cnt <= 0;
```

```
//       clk <= ~clk; // Toggle the output clock
//   end else begin
//       cnt <= cnt + 1;
//   end
//end
   okTriggerIn trigIn53 (.okHE(okHE),.ep_addr(8'h40),
.ep_clk(FSM_Clk), .ep_trigger(Reset_Counter));
//    okTriggerIn trigIn44 (.okHE(okHE),.ep_addr(8'h44),
.ep_clk(clk), .ep_trigger(I2C_control));
   ////////////////////////////////////////////////////// OK
WIRE OUTS FOR SPI CONFIRMATION

   localparam endPt_count = 2 + 3;
   wire [endPt_count*65-1:0] okEHx;
   okWireOR # (.N(endPt_count)) wireOR (okEH, okEHx);

   okWireOut wire20 ( .okHE(okHE),
   .okEH(okEHx[ 0*65 +: 65 ]),
   .ep_addr(8'h20),
   .ep_datain(val));

   okBTPipeOut CounterToPC (
       .okHE(okHE),
       .okEH(okEHx[ 1*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)
   );

   ////////////////////////////////////////////////////// CAMERA
READING CODE -> CLOCKS CONTROLLED INSIDE SPI MODULE
//    reg [8:0]cnt=0;
//    reg clk;
   assign CVM300_CLK_IN = FSM_Clk;

   always @(posedge FSM_Clk) begin
```

```
       if (Reset_Counter[0] == 1'b1) State <= STATE_RESET;

       case (State)
           STATE_INIT:   begin
               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 // does this state also give us the
necessary wait time of 1 us?
               if (counter_delay == 16'b0000_0000_0000_0000)  State
<= STATE_FRAME_REQ;
               else counter_delay <= counter_delay + 1;
           end

            STATE_FRAME_REQ: begin
               frame_req <= 1;
               State <= STATE_FINISH;
            end

            STATE_FINISH:   begin
               frame_req <= 0;
               State <= STATE_INIT; // you just sit here, the DVAL
will handle the write enable as pixels get written
            end
```

```
        endcase
    end

endmodule
```