ECE 494-405 Final Report

Final Report for OV5640 Camera Project

ECE 494-405 Capstone Design II

Submitted by

Jack Codemo, Daniel Edwards, and Mesab Almutairi

Department of Electrical and Computer Engineering

The University of Alabama

11/27/2022

Abstract

A growing and important field in modern society involves wearable technologies. In order to assist in the development of these growing technologies, an OV5640 camera system was developed that functions by connecting an FPGA to a camera, a microcontroller, and to HyperRAM on a custom Printed Circuit Board. The microcontroller was programmed in C to initialize the camera via I2C communication and communicate with the FPGA via SPI. The FPGA was programmed in Verilog to connect to the microcontroller via SPI, communicate with the camera with a parallel interface, and communicate with HyperRAM through the HyperBus communication protocol. In testing, the camera initialization protocol was verified, the SPI interface was verified, the initial HyperRAM design was verified, and the hardware was verified. Though the final product was not completely tested, considerable progress was made in the development and testing phases of the product.

Table of Contents

1. INTRODUCTION

Wearable technologies are a growing field that provides users ease of access in their daily lives. The development of cameras in wearable technologies is needed for the advancement of this field and to increase its value in the general market. Our project over the past two semesters was to create a camera design that can be used in wearable technologies. Our design involves a PCB that connects an FPGA to HyperRAM, a Camera, and to a microcontroller. The microcontroller is also linked to the camera. The camera sends image data to the FPGA through a parallel interface. The FPGA buffers and sends this data to the microcontroller through serial communication. This buffer is between the FPGA and HyperRAM which is accomplished by using a HyperBUS communication method. The microcontroller communicates with the camera via I2C protocol in order to initialize the camera's functionality and trigger the image capture sequence. The block diagram of the system is shown in Figure 1. When designing the system, there were several system requirements that needed to be met. The system was to operate at less than 20 mA during peak data capture, the FPGA was to be smaller than 5mm x 5mm to 7mm x 7mm, the RAM was to be smaller than 10mm x 10mm, and the camera was to operate at a 200 MHz communication interface.

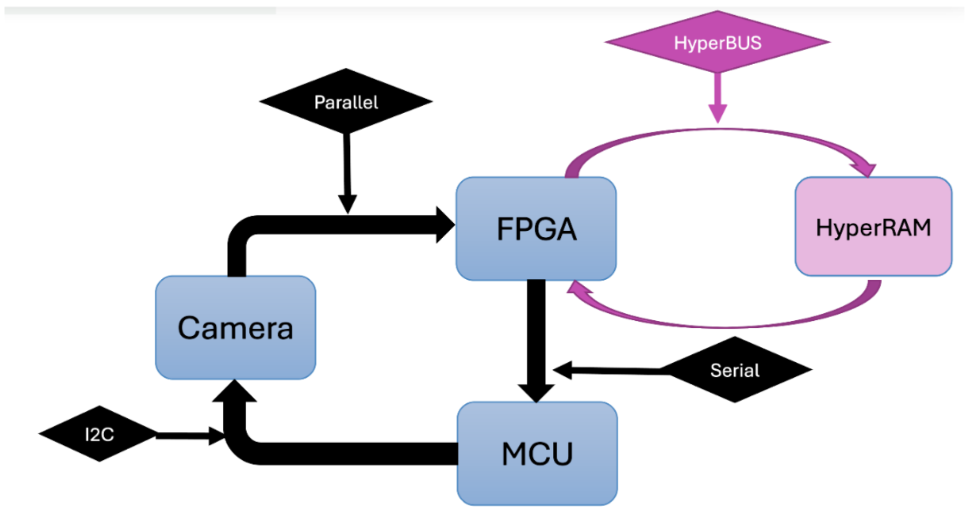


Figure 1. System Block Diagram

2. Design

2.1 Design Procedure

The selection of components within the system involved a number of design decisions and alternatives. The camera that we utilized was the Omnivision OV5640 Camera. The camera was chosen based on design constraints. It supports a parallel interface, operates at 3.3V, and has initialization code readily available. The microcontroller that we used was the Nucleo-64 STM32L476RG. We chose this microcontroller because the camera initialization code provided to the group utilized this family of microcontrollers which allowed for easier microcontroller code design.

There were several options considered when choosing the FPGA and RAM used for the design. For the FPGA, we considered FPGAs from Micro Semiconductors and the Lattice ICE40 family of FPGAs. We chose the Lattice ICE40UP5K-SG48I FPGA due to its size of 7mmx7mm, its active current of 1-10 mA, and its static current of 75µA. When finding what type of RAM to be used, there were several factors to be considered, including pin count and size. Our final decision came down between SRAM and HyperRAM. We chose the HyperRAM chip, W955D8MBYA6I, due to its low pin count of 13 pins, its operating voltage of 1.8V, and its 32 Mb storage size. The SRAM would have been easier to implement into our design, but due to the FPGA's low pin count and the large amount of pins needed for an SRAM chip, HyperRAM had to be used.

2.2 Hardware Design

The PCB for the project connects a Nucleo-64 STM32L476RG microcontroller on the bottom to an Upduino 3.0 FPGA development kit on the top. It also connects an OV5640 camera and HyperRAM to each of the components. The pin headers that connect the microcontroller has connections to both the camera and the FPGA. The schematic for the microcontroller connections is shown in Figure 2. The microcontroller has several IO pins that include FPGA_RESET, CAM_PWREN, MCLK, SPI_CLK, FPGA_PWR_ENABLE, CAPT_READ, READ_CMPLT, and CAPT_CMPLT. The microcontroller also has connections to the camera via I2C with connections to two pull-up resistors that pull the pins to 3.3V. It also contains a connection to the microcontroller via SPI.

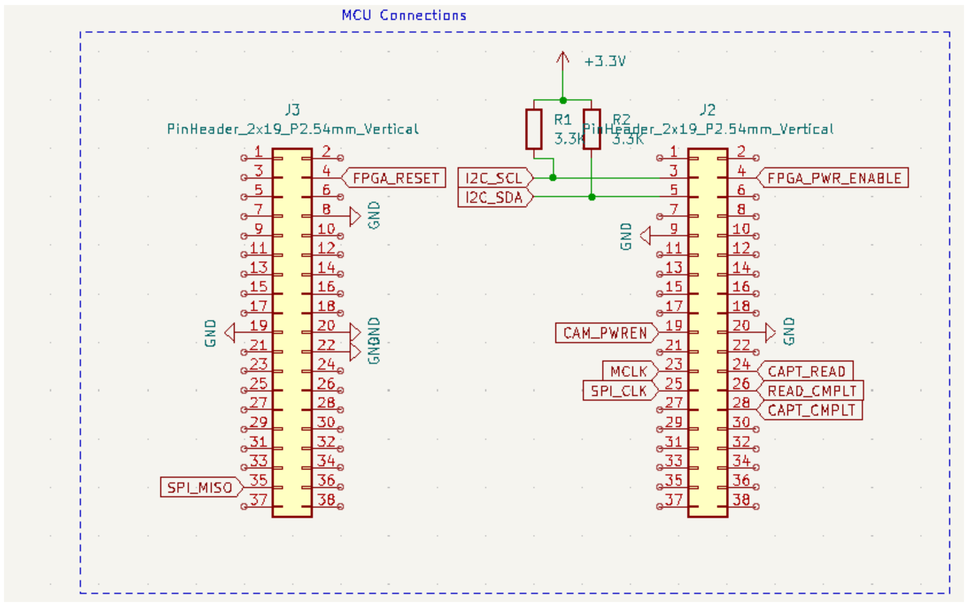


Figure 2. Microcontroller Connections

A crucial part of the project is the FPGA. Its connections are shown in Figure 3. There are a number of connections in the FPGA that connect to the camera, the microcontroller, and HyperRAM. In terms of power, the Upduino provides voltages of 3.3V and 5V. The FPGA is also fed an input voltage of 1.8V in order to interface with the HyperRAM and camera. Of note, the microcontroller has 3.3KΩ pull-up resistors for use in I2C. The microcontroller also controls the clock for the entire system. There are also indicators of the state of the system that are sent to and from the FPGA: the capture complete signal, the read complete pin, the capture read pin, the FPGA Power Enable Pin, and the FPGA Reset Pin. There are also a Master In-Slave Out (MISO) pins which are dedicated to communicating with the FPGA via SPI.

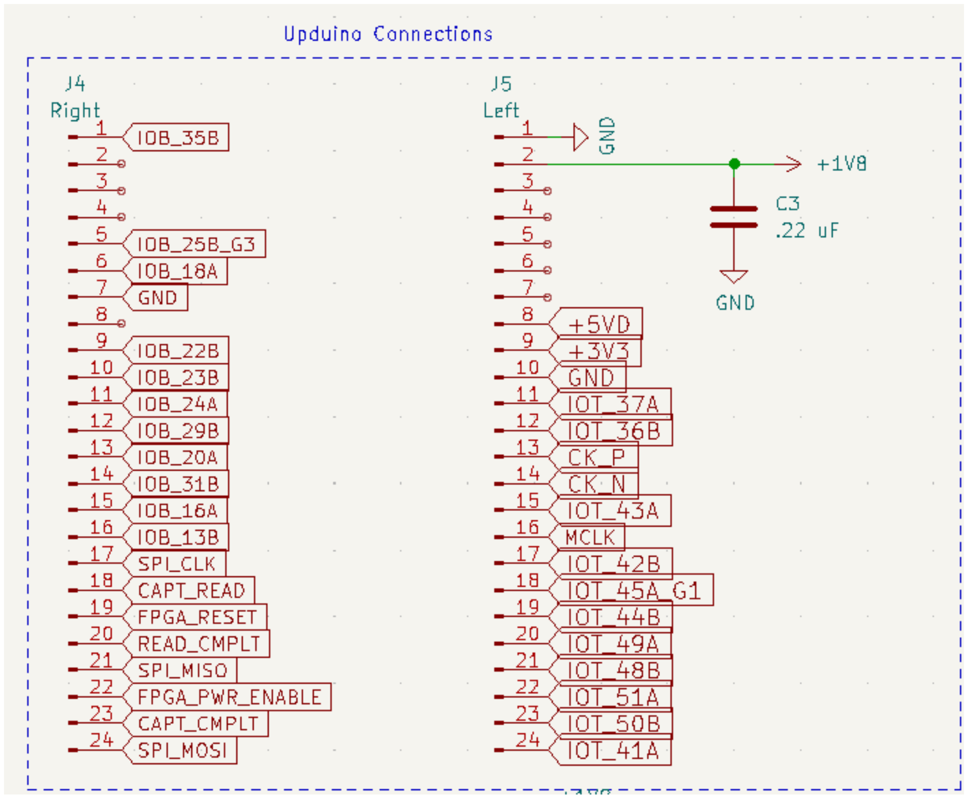


Figure 3. Upduino Connections

Figure 4 provides a schematic for HyperRAM connections and the voltage regulators used in the system, The pin headers on the top left of the figure provide an input voltage of 5V for the PCB. The 5V input is fed through the voltage regulator in the bottom left to provide an Analog voltage for the camera which operates at 3.3V. The voltage regulator on the bottom right inputs a 3.3V input and outputs 1.8V to the HyperRAM and the camera. Each of the voltage regulators utilized decoupling capacitors. The HyperRAM has several connections to the FPGA. Of note, the HyperRAM utilizes a differential clock which required the use of differential pairs.

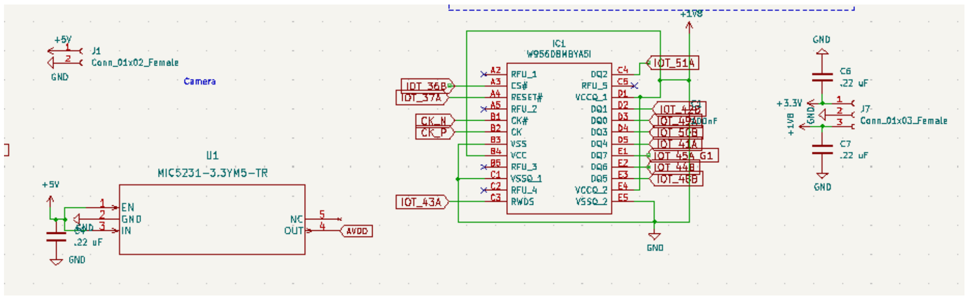


Figure 4. HyperRAM connections and Power Diagram

Figure 5 shows the connections to the camera. Of note, the camera is fed a 1.8V input and a 3.3V input. It also has several connections to the FPGA and the MCU. The signals from the microcontroller include the SCL and SDA pins to the microcontroller, the reset pin, the power down pin, and the master clock pin. The pins that connect to the FPGA are all camera data ports, the Horizontal Reference (HREF) pin, the Vertical Synchronization(VSYNC) pin, and the Pixel Clock(PCLK) pin.

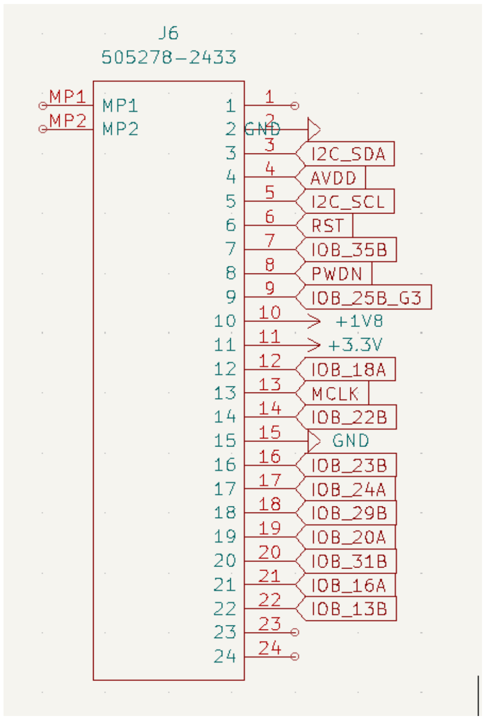


Figure 5. OV5640 Connections

The final PCB layout is shown below in Figure 6. The PCB with the microcontroller, the FPGA, and the camera attached is shown in Figure 7. It passed all design checks, and all connections were verified. The procedure for the hardware verification is explained in more depth in the Hardware Verification section.

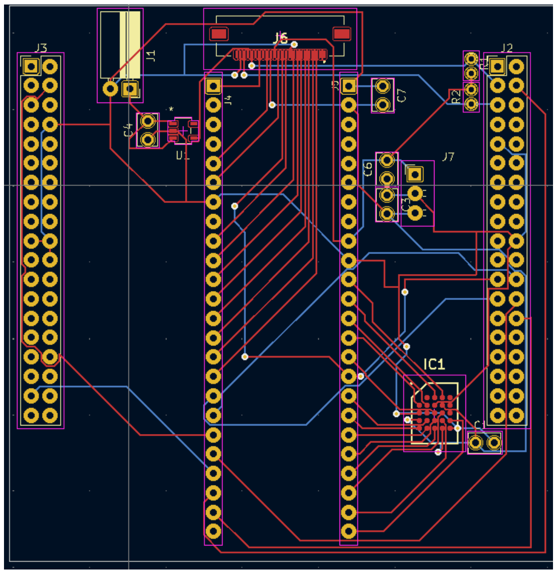


Figure 6. PCB Design

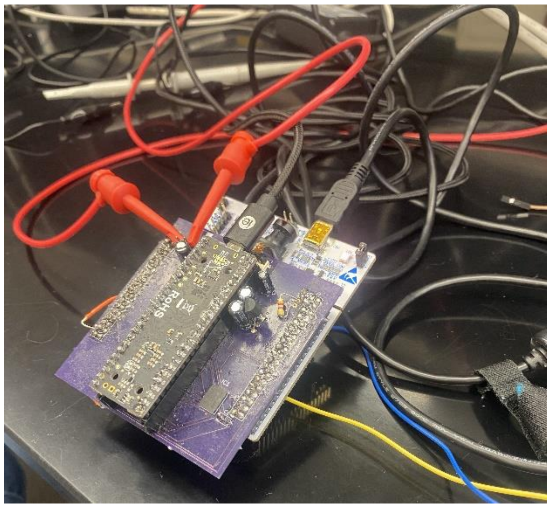


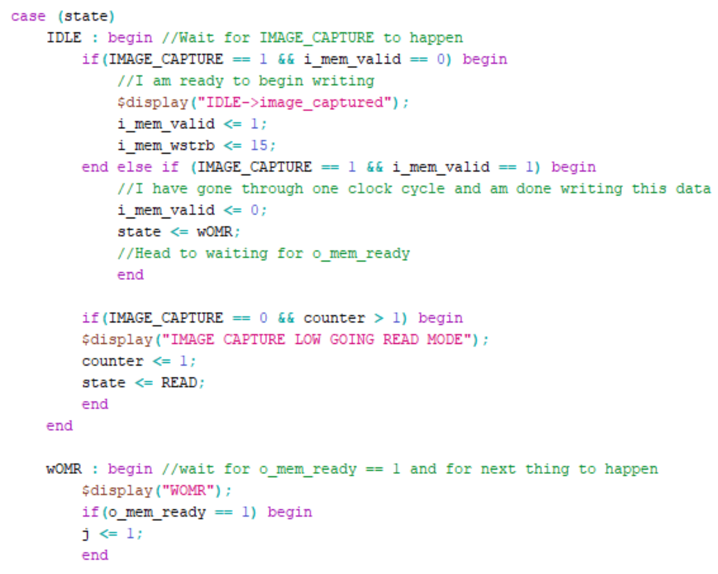
Figure 7. Fully Constructed PCB

2.3 Framebuffer Software Design

The Framebuffer software is loosely based on a similar framebuffer design provided to the group that utilizes SRAM. The difference in RAM communication protocols was cause for a new design to be created. This was also to be done on the Lattice Radiant software as this is the native design suite for iCE40 FPGA's. The original program was completed in Libero software, which has several differences from Lattice Radiant. Lattice utilizes Verilog and VHDL code to synthesize a design into a bin file that can be programmed into the FPGA directly using the Radiant Programmer software that comes with the Lattice Radiant Suite.

The main software is designed to run off of three main sequences, each involving a separate clock cycle. The first sequence is the initialization period for the HyperRAM. This is a period where no data can be transmitted and allows for the HyperRAM to perform a self-setup process using an input clock provided by the microcontroller funneled through the FPGA to help keep track of the current sequence. Once this process is completed, the FPGA sits in an idle state until the microcontroller triggers the camera to take a picture. The camera will then supply the FPGA with a pixel clock that tells the design when to take data from the parallel interface and store it into the HyperRAM. Once the pixel clock stops triggering, the microcontroller will use a SPI Clock to initiate serial data transfer. The FPGA will use this third clock to pull the data out of RAM storage and slowly push out via SPI data transfer.

Due to the complexity of the HyperBUS protocol that allows HyperRAM to have a smaller pin count our group agreed to try and implement a HyperRAM controller found online. There were a couple options but very few of them were user friendly and even less were able to compile after only a few bug fixes. The final Verilog-based design works off of a Case statement function almost like a finite state machine to account for the different sequences and any internal processes between these sequences (a screenshot of a section of this case statement is provided below).



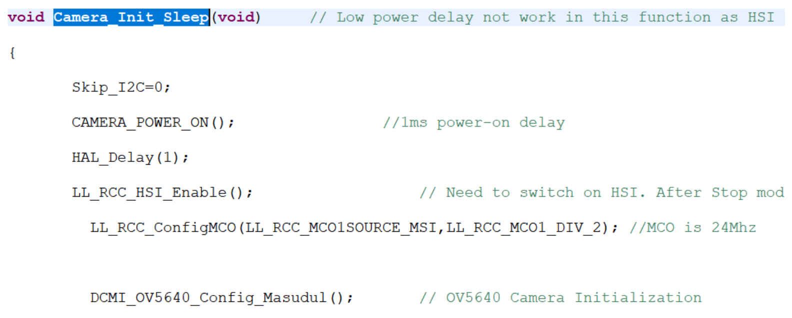
```
case (state)
    IDLE : begin //Wait for IMAGE_CAPTURE to happen
        if(IMAGE_CAPTURE == 1 && i_mem_valid == 0) begin
            //I am ready to begin writing
            $display("IDLE->image_captured");
            i_mem_valid <= 1;
            i_mem_wstrb <= 15;
        end else if (IMAGE_CAPTURE == 1 && i_mem_valid == 1) begin
            //I have gone through one clock cycle and am done writing this data
            i_mem_valid <= 0;
            state <= wOMR;
            //Head to waiting for o_mem_ready
            end

        if(IMAGE_CAPTURE == 0 && counter > 1) begin
        $display("IMAGE CAPTURE LOW GOING READ MODE");
        counter <= 1;
        state <= READ;
        end
    end

    wOMR : begin //wait for o_mem_ready == 1 and for next thing to happen
        $display("WOMR");
        if(o_mem_ready == 1) begin
        j <= 1;
        end
```

Figure 8. Case Statement Snippet

The main file utilizes two other modules to assist with the overall functionality. These two modules are the actual HyperRAM controller found on GitHub created by user gtjennings, and a simple LED module that blinks the onboard RGB LED’s when the enable is triggered high (this is used for debugging to see if certain points are reached and if data is matching with desired values). The testing and simulation of this design was done through the ModelSIM software which is further explained in section *3.3 FPGA Software Verification*.

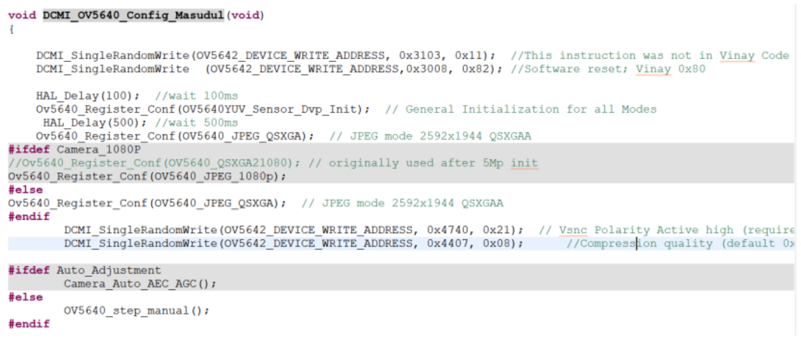
2.4 Microcontroller Software Design

The microcontroller software consisted of two main components: the initialization of the camera to the microcontroller via I2C and the transmission of data from the FPGA to the microcontroller via SPI. The initialization of the camera via I2C is called from the function shown in Figure 10. In this procedure, the camera is powered on, the clock for the system is enabled, and then the camera is initialized using the function shown in Figure 11.



```
void Camera_Init_Sleep(void)      // Low power delay not work in this function as HSI
{
        Skip_I2C=0;
        CAMERA_POWER_ON();                //1ms power-on delay
        HAL_Delay(1);
        LL_RCC_HSI_Enable();                  // Need to switch on HSI. After Stop mod
          LL_RCC_ConfigMCO(LL_RCC_MCO1SOURCE_MSI,LL_RCC_MCO1_DIV_2); //MCO is 24Mhz

          DCMI_OV5640_Config_Masudul();       // OV5640 Camera Initialization
```

Figure 9. Camera Initialization Function



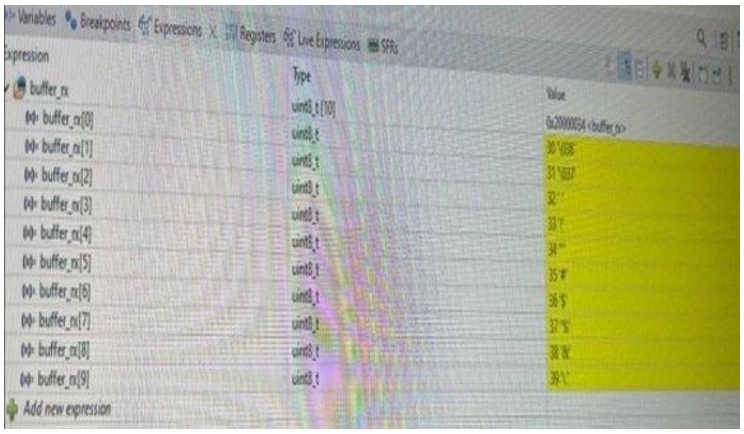
```
void DCMI_OV5640_Config_Masudul(void)
{

    DCMI_SingleRandomWrite(OV5642_DEVICE_WRITE_ADDRESS, 0x3103, 0x11);  //This instruction was not in Vinay Code
    DCMI_SingleRandomWrite  (OV5642_DEVICE_WRITE_ADDRESS,0x3008, 0x82); //Software reset; Vinay 0x80

    HAL_Delay(100);  //wait 100ms
    Ov5640_Register_Conf(OV5640YUV_Sensor_Dvp_Init);  // General Initialization for all Modes
     HAL_Delay(500); //wait 500ms
    Ov5640_Register_Conf(OV5640_JPEG_QSXGA);  // JPEG mode 2592x1944 QSXGAA
#ifdef Camera_1080P
//Ov5640_Register_Conf(OV5640_QSXGA21080); // originally used after 5Mp init
Ov5640_Register_Conf(OV5640_JPEG_1080p);
#else
Ov5640_Register_Conf(OV5640_JPEG_QSXGA);  // JPEG mode 2592x1944 QSXGAA
#endif
        DCMI_SingleRandomWrite(OV5642_DEVICE_WRITE_ADDRESS, 0x4740, 0x21);  // Vsnc Polarity Active high (require
        DCMI_SingleRandomWrite(OV5642_DEVICE_WRITE_ADDRESS, 0x4407, 0x08);      //Compression quality (default 0x

#ifdef Auto_Adjustment
        Camera_Auto_AEC_AGC();
#else
        OV5640_step_manual();
#endif
```

Figure 10. Camera Initialization Second Function

3 Verification

3.1 Microcontroller Verification

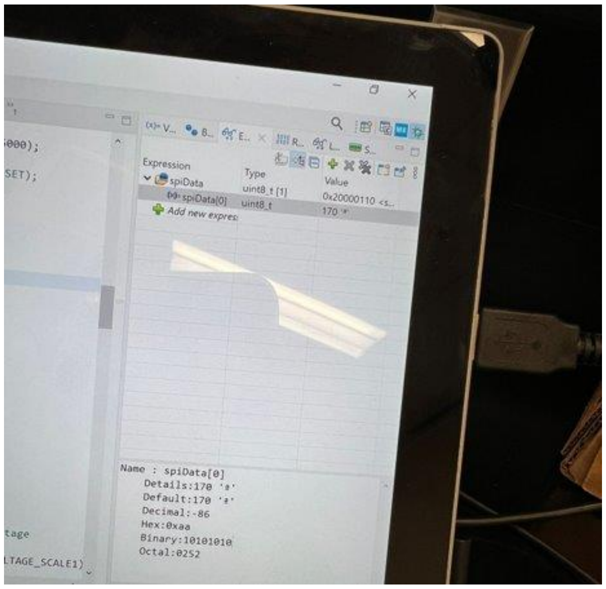
SPI interface testing was required to confirm that both the transmitting and receiving sections of the MCU were functioning. For this purpose, Master In-Slave Out(MISO) and Master Out Slave In(MOSI) pins are utilized. First, the MOSI and MISO pins were shorted physically using wires. Then, a known set of bytes was sent and received by the microcontroller. Then the transmitted data and the received data was compared. The data that was transmitted and received was confirmed to be matching. The matching data verified that the SPI interface utilized by the microcontroller was functioning properly. To re-check SPI Communication, the wire that shorted the MISO and MOSI pins was removed. After removing the jumper wire, the same data was sent by the program. When the program ran with the wire removed, only 0's appeared at receiver side which means that data transfer stopped and that there was no false data being received. Figure 12 displays the data received via SPI.



```
Expression          Type          Value
buffer_rx           uint8_t [10]  0x20000034 <buffer_rx>
  buffer_rx[0]      uint8_t       30 '\036'
  buffer_rx[1]      uint8_t       31 '\037'
  buffer_rx[2]      uint8_t       32 ' '
  buffer_rx[3]      uint8_t       33 '!'
  buffer_rx[4]      uint8_t       34 '"'
  buffer_rx[5]      uint8_t       35 '#'
  buffer_rx[6]      uint8_t       36 '$'
  buffer_rx[7]      uint8_t       37 '%'
  buffer_rx[8]      uint8_t       38 '&'
  buffer_rx[9]      uint8_t       39 '\''
```

Figure 11. Output from shorting MISO and MOSI pins

After testing MISO and MOSI pins on the microcontroller, the communication between the FPGA and the microcontroller was tested. On the microcontroller, the MISO pin, the SPI CLK pin, and the grounds were connected. The FPGA was programmed to shift out a set of known bytes to the microcontroller. By comparing the data that was sent and received, it was confirmed that the communication between FPGA and MCU was functioning properly. While testing, it was imperative that the SPI mode on the microcontroller matched the SPI mode utilized by the FPGA in order to transmit and receive correct data. A value of "10101010" was sent from the FPGA to the microcontroller. The microcontroller correctly received the data. The received data is shown in Figure 13.

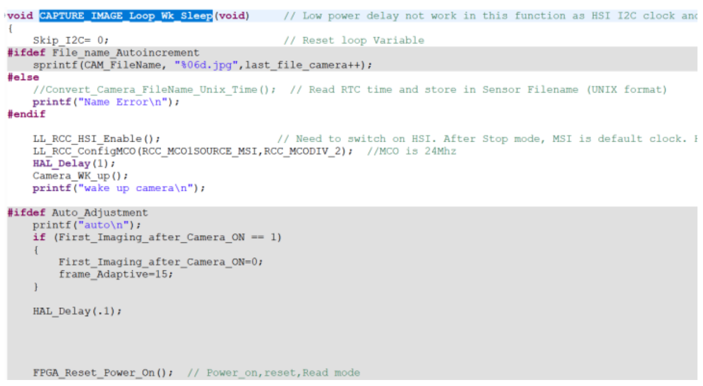


Expression	Type	Value
spiData	uint8_t [1]	0x20000110 <s...
spiData[0]	uint8_t	170 'ª'
Add new expres		

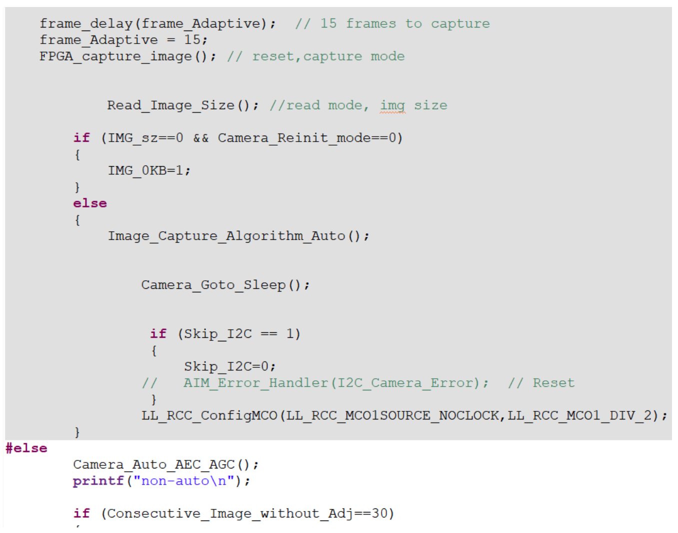
```
Name : spiData[0]
    Details:170 'ª'
    Default:170 'ª'
    Decimal:-86
    Hex:0xaa
    Binary:10101010
    Octal:0252
```

Figure 12. Received SPI Value Verification

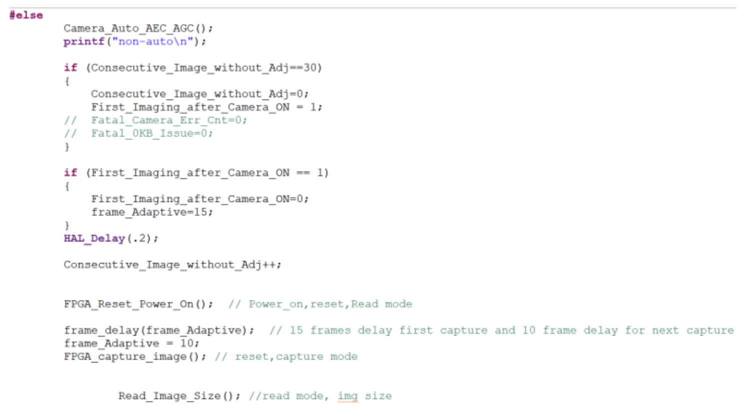
Because the microcontroller is only receiving data from the FPGA, only the MISO pin is required. Every trigger of the SPI clock shifts data from the FPGA pin to the Microcontroller MISO pin. The image data is received by the microcontroller through a file system management function that is shown in Figure 14.



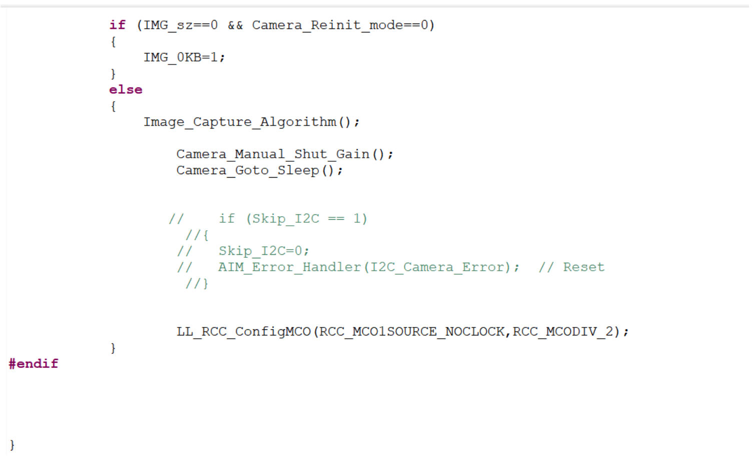
```
void CAPTURE_IMAGE_Loop_Wk_Sleep(void)      // Low power delay not work in this function as HSI I2C clock anc
{
    Skip_I2C= 0;                            // Reset loop Variable
#ifdef File_name_Autoincrement
    sprintf(CAM_FileName, "%06d.jpg",last_file_camera++);
#else
    //Convert_Camera_FileName_Unix_Time();  // Read RTC time and store in Sensor Filename (UNIX format)
    printf("Name Error\n");
#endif

    LL_RCC_HSI_Enable();                   // Need to switch on HSI. After Stop mode, MSI is default clock. I
    LL_RCC_ConfigMCO(RCC_MCO1SOURCE_MSI,RCC_MCODIV_2);  //MCO is 24Mhz
    HAL_Delay(1);
    Camera_WK_up();
    printf("wake up camera\n");

#ifdef Auto_Adjustment
    printf("auto\n");
    if (First_Imaging_after_Camera_ON == 1)
    {
        First_Imaging_after_Camera_ON=0;
        frame_Adaptive=15;
    }

    HAL_Delay(.1);



    FPGA_Reset_Power_On();  // Power_on,reset,Read mode
```



```
    frame_delay(frame_Adaptive);  // 15 frames to capture
    frame_Adaptive = 15;
    FPGA_capture_image(); // reset,capture mode

        Read_Image_Size(); //read mode, img size

        if (IMG_sz==0 && Camera_Reinit_mode==0)
        {
            IMG_0KB=1;
        }
        else
        {
            Image_Capture_Algorithm_Auto();

                Camera_Goto_Sleep();

                 if (Skip_I2C == 1)
                 {
                     Skip_I2C=0;
                //   AIM_Error_Handler(I2C_Camera_Error);  // Reset
                 }
                LL_RCC_ConfigMCO(LL_RCC_MCO1SOURCE_NOCLOCK,LL_RCC_MCO1_DIV_2);
        }
#else
        Camera_Auto_AEC_AGC();
        printf("non-auto\n");

        if (Consecutive_Image_without_Adj==30)
```



```
#else
        Camera_Auto_AEC_AGC();
        printf("non-auto\n");

        if (Consecutive_Image_without_Adj==30)
        {
            Consecutive_Image_without_Adj=0;
            First_Imaging_after_Camera_ON = 1;
        //  Fatal_Camera_Err_Cnt=0;
        //  Fatal_0KB_Issue=0;
        }

        if (First_Imaging_after_Camera_ON == 1)
        {
            First_Imaging_after_Camera_ON=0;
            frame_Adaptive=15;
        }
        HAL_Delay(.2);

        Consecutive_Image_without_Adj++;


        FPGA_Reset_Power_On();  // Power_on,reset,Read mode

        frame_delay(frame_Adaptive);  // 15 frames delay first capture and 10 frame delay for next capture
        frame_Adaptive = 10;
        FPGA_capture_image(); // reset,capture mode


            Read_Image_Size(); //read mode, img size
```



```
        if (IMG_sz==0 && Camera_Reinit_mode==0)
        {
            IMG_0KB=1;
        }
        else
        {
            Image_Capture_Algorithm();

                Camera_Manual_Shut_Gain();
                Camera_Goto_Sleep();

               //    if (Skip_I2C == 1)
                 //{
                //   Skip_I2C=0;
                //   AIM_Error_Handler(I2C_Camera_Error);  // Reset
                 //}


                LL_RCC_ConfigMCO(RCC_MCO1SOURCE_NOCLOCK,RCC_MCODIV_2);
        }
#endif




}
```

Figure 13. SPI Receiver Function

After verifying SPI communication between the microcontroller and the FPGA, the initialization protocol that was sent from the microcontroller to the camera was verified. The first step required verifying that the camera was correctly receiving and sending I2C data by connecting the SCL and SDA pins to a logic analyzer and verifying that the data being read and written was being acknowledged. A snippet of the logic analyzer displaying the I2C data was being acknowledged is shown in Figure 15. The final line of the waveforms shown below display each of the addresses and pieces of data being written to the camera. Each pink "A" next to the packets of data displays the acknowledge bit being sent from the camera, showing that the camera is accepting the camera data being written.

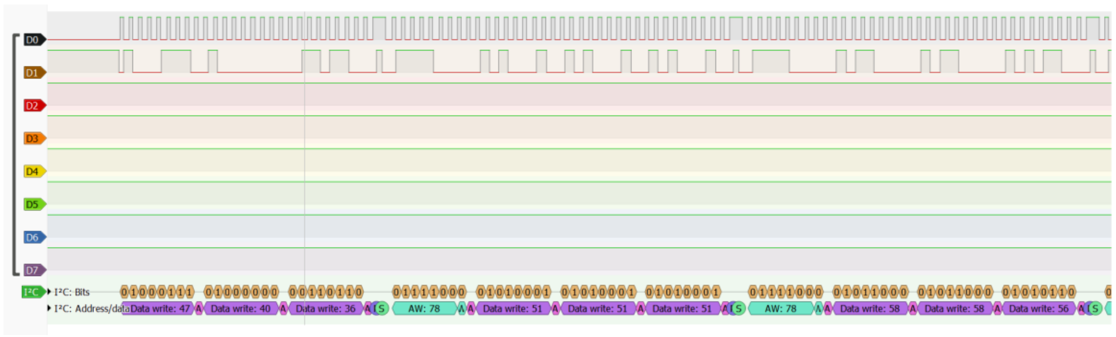


Figure 14. I2C Read and Write Logic Analyzer Results

Following the verification of the I2C messages that were being written and read, the camera data that was being sent to the FPGA needed to be verified. In order to check the data being sent, the compression mode waveform was verified by connecting the horizontal reference (HREF) pin and the vertical synchronization (VREF) pin to an oscilloscope and observing the waveforms. The camera operates in compression mode 3. The waveform from the OV5640 datasheet[1] is shown in Figure 16. The waveform generated from the oscilloscope is shown in Figure 17. The green signal represents the HREF signal and the yellow signal represents the

VSYNC signal. The signals from the camera matched the signals from the timing diagram, displaying that the camera was emitting data correctly.

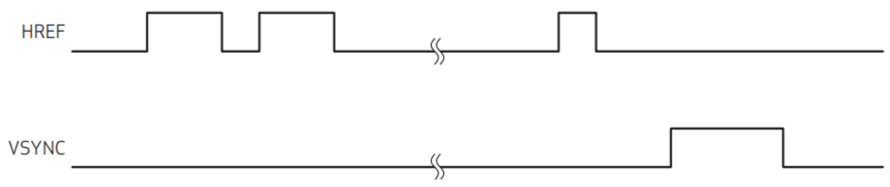


Figure 15. OV5640 Compression Mode 3 Timing Diagram[1]

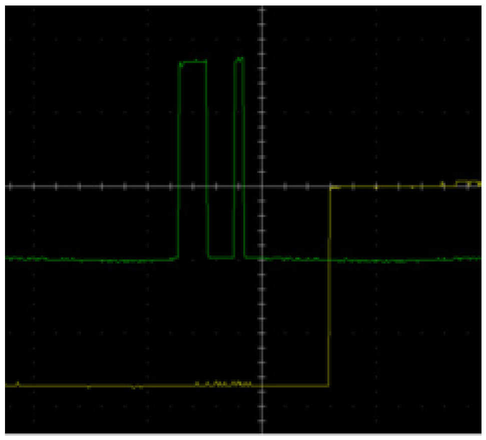


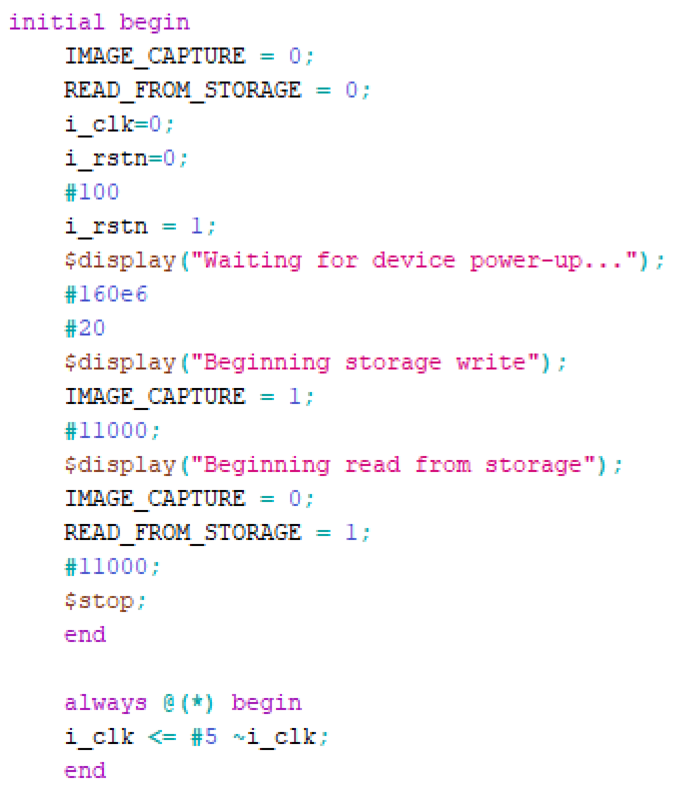
Figure 16. Oscilloscope Measurements of VSYNC and HREF pins

3.2 Hardware Verification

The hardware verification process was mostly intended on verifying power connections and PCB construction. Each of the voltage regulators was tested to verify that it was emitting the correct voltage by connecting the regulators to a multimeter. The HyperRAM chip is a Ball Grid Array chip, which you cannot verify through visual inspection, so the board was inspected under an X-Ray by a third-party vendor. Also, because the HyperRAM chip has CMOS input circuit protection, the chip was also verified with the diode setting on a multimeter by verifying that the values shown on the multimeter were the same for each of the inputs on the chip. Each of the other components was verified by visually inspecting solder joints and checking for continuity with a multimeter.

3.3 FPGA Software Verification

To verify the verilog software is working correctly, the Model-Sim tool was used to simulate how the actual device should operate when given certain inputs. These inputs are generated via testbench files created for the specific purpose of driving pins and generating clock cycles. A screenshot below shows an example of the most recent testbench code used to drive the framebuffer design:



```
initial begin
    IMAGE_CAPTURE = 0;
    READ_FROM_STORAGE = 0;
    i_clk=0;
    i_rstn=0;
    #100
    i_rstn = 1;
    $display("Waiting for device power-up...");
    #160e6
    #20
    $display("Beginning storage write");
    IMAGE_CAPTURE = 1;
    #11000;
    $display("Beginning read from storage");
    IMAGE_CAPTURE = 0;
    READ_FROM_STORAGE = 1;
    #11000;
    $stop;
    end

    always @(*) begin
    i_clk <= #5 ~i_clk;
    end
```

Figure 17. Testbench File Code for FPGA Design

The code for this testbench file simply drives the Image_capture input that would normally be triggered by the MCU and drives the clock input for the main design and allows for ample time to view the waveforms in modelsim with the many “wait # of milliseconds” statements (#number). With modelsim, different variables, inputs, outputs, and exact timing of signal changes can easily be seen allowing for precise visualization of the program’s behavior. A screenshot of a waveform used is given below:

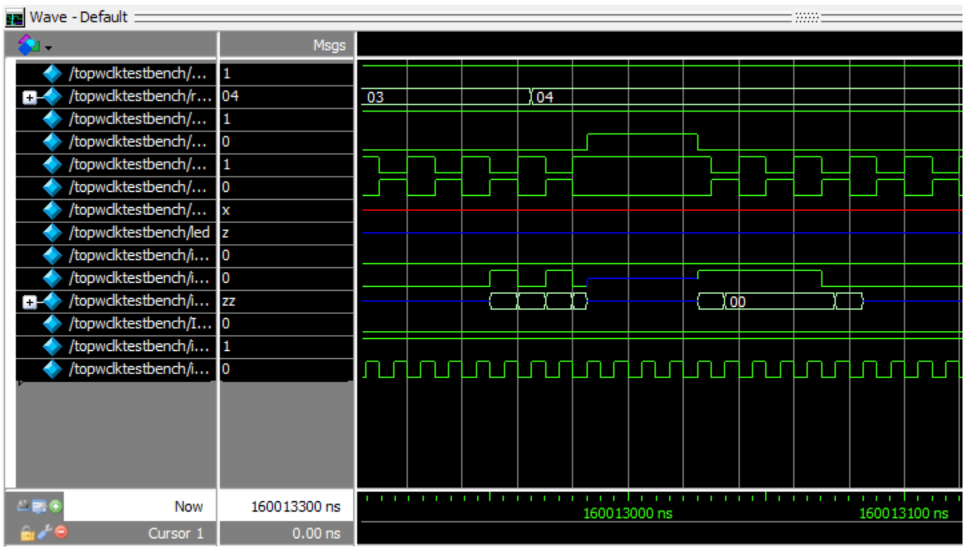


Figure 18. ModelSIM Waveform Example

With this waveform we were able to easily view the behaviors of the signals being affected by our testbench driven inputs to see what needed to be done in order to get the desired outcome. The reason this simulation is also useful is due to the HyperRAM module file that can be integrated into the simulation file that allows us to feed our simulated connections to a Verilog model that behaves exactly like its HyperRAM counterpart. This way we can simulate stored data as well as bidirectional pinouts being driven from the external HyperRAM hardware to further perfect the design.

To finally verify the real Framebuffer design works on the hardware, the use of the LED verilog module we created comes into play. By simply changing the enable parameter of this verilog module to ‘1’, the Upduino will begin flashing its RGB LED. We can use this to verify certain values match with a simple “if equals” statement and flash the light if it is true. A useful example would be to check and see if the data we wrote into the HyperRAM address matches the data we read out of the same HyperRAM address to verify functionality of data storage.

4 Cost

For the project, a $300 budget was provided by the ECE department. A breakdown of how the money was spent is shown in Figure 20. $37.95 was spent on the camera, $50.79 was spent on voltage regulators and the FPGA, $20.95 was spent on HyperRAM, $14.83 was spent on camera connectors, $80 was spent on the PCB. There was $95 remaining in the budget.

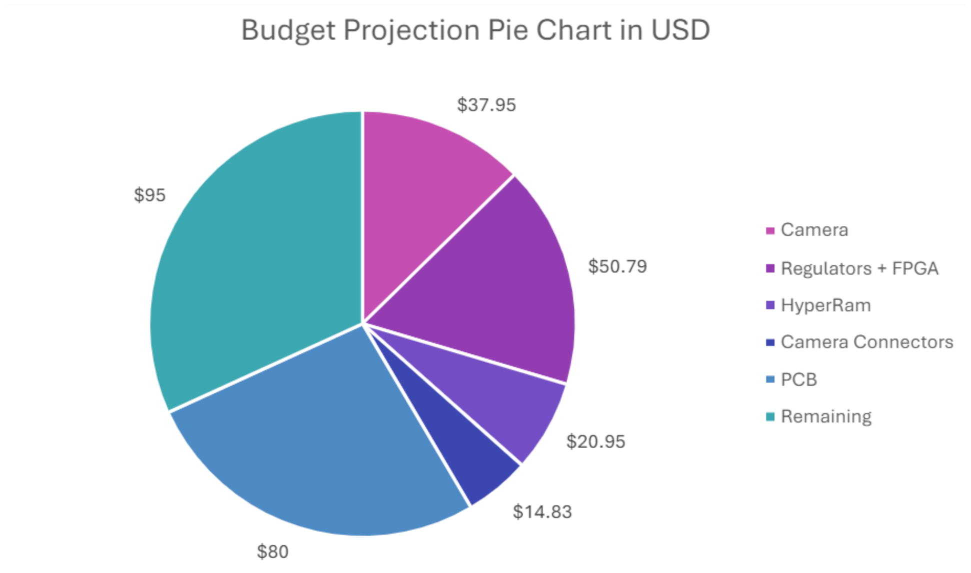


Figure 19. Pie Chart of Budget

CONCLUSION

Over the past two semesters, our project was to create a camera design that can be used in wearable technologies utilizing a microcontroller, an FPGA, an OV5640 camera, and HyperRAM. Through testing and verification, we were able to have a functional camera initialization program, a functional PCB, a functional SPI receiver, and a functional HyperRAM controller. Uncertainties remain in implementation of HyperRAM with the framebuffer design given the HyperRAM's complexity. SRAM, an alternative with a much simpler controller, could be used with a different FPGA chip, but could not be used in an FPGA with a limited number of inputs and outputs. Several small FPGA chips that fit the size constraints given in the project requirements exist, but the FPGAs are in Ball Grid Array packaging that would require a very small trace in order to access pins that are needed for device implementation, which would make the PCB too expensive to fabricate. To address this issue, it would be imperative to modify the design specifications to allow for a larger FPGA with more accessible IO Ports. Though we were unable to fully implement the framebuffer design, we made considerable progress towards our final goal and gained considerable experience in various fields of hardware and software design.

REFERENCES

[1] OV5640 Datasheet Product Specification, Omnivision, Sparkfun, 2022