Hand Gesture Computer Interface (HGCI)

Arash Taheri-Dezfouli
Armin Karami
James Aziz
April 13, 2017

Contents

1 Overview

This section provides a high-level description of the project, providing a background and presenting the designed system.

1.1 Motivation

At the present, the world is looking for new means of interaction with technology. One possible approach is the use of hand gestures, as they are very intuitive and non-intrusive. For example, a presenter would be able to move away from their podium and perhaps scroll through or zoom in and out of their presentation simply by using their hands. Such technology could also be extended to, for example, allow doctors to interact with three-dimensional models of organs in an intuitive manner.

A similar product does exist, the CyberGlove III - however, this application uses a glove with flex sensor technology and bicep cuff [1]. A solution that requires only a minimal hand apparatus is therefore a suitable endeavour.

1.2 Goals

The purpose of this project is the creation of a system capable of gesture tracking based around the Xilinx Nexys Video Artix-7 FPGA. LEDs are placed on a pair of gloves, which indicate the location of the user's hands. An image processing system must be capable of detecting the locations of LEDs on gloves. When the user moves their hands, a camera captures a video stream, which is fed into a custom image processing IP within the FPGA. This IP extracts the locations of the LEDs and supplies this data to a MicroBlaze soft processor, which determines which gesture has occurred. The corresponding actions are sent as keyboard input to a personal computer, to control content on the screen.

1.3 System Level Diagram

The completed project can be summarized simply as a combination of an external hardware input, an FPGA, followed by the output external hardware. Figure 1 illustrates a more detailed breakdown of the architecture, showing all the blocks that were used.

The information flow of the system occurs in 4 stages. The first stage starts with LEDs that are captured on video by the camera and sent to the FPGA through an HDMI cable. The input video stream passes through a DVI to RGB video format converter and then it is decoded into a AXI stream format. Next it is sent to I/O VDMA which stores it into DDR memory, one frame at a time, through the AXI memory bus. The second stage starts with the IP VDMA retrieving the frame from memory and passing it to the LED Detect IP as an input. In the third stage, the IP detects the LEDs and stores their locations in its slave registers. At this point the Microblaze soft processor reads the new coordinates for processing in software. Then the IP outputs the filtered frame back to the IP VDMA so that it can be stored in DDR memory again. In the final stage the I/O VDMA reads the filtered image from memory and passes it through an AXI stream to RGB converter. This is followed by an RGB to DVI video encoder so that the frame can be outputted to the display through another HDMI cable.

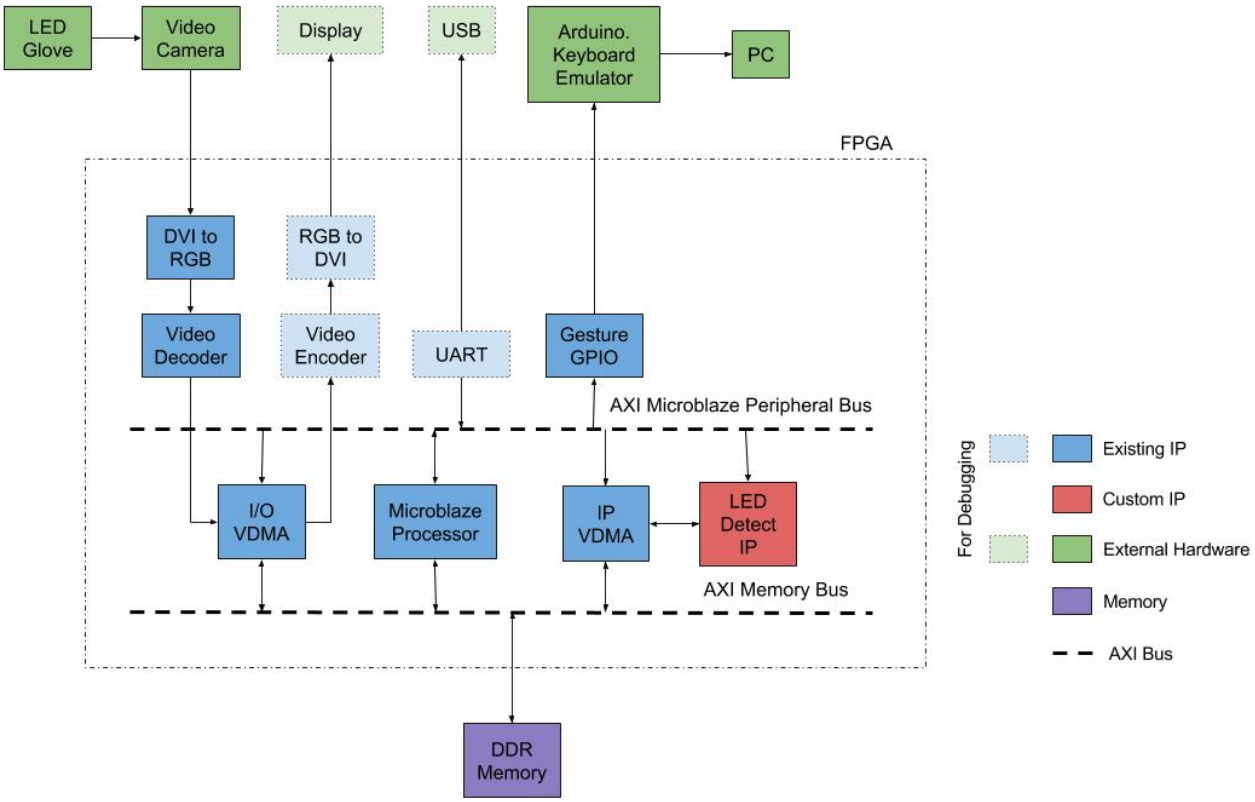

Figure 1: A block diagram of the system level architecture.

1.4 Brief Description of System

The system is comprised of various IP blocks that have been integrated together. Table 1 displays each block used in the system along with a description of its functionality. The table also displays the source of each IP block.

Table 1: Description of System Blocks

IP Instance Name	Description	Block Source
axi_dynclk_0	AXI dynamic clock generator used to sync video output blocks	XILINX
axi_gpio_0	GPIO block used to communicate between Microblaze and Arduino. The block has a 6-bit output that connects to the JA PMOD port. It also has a 1 bit input that signifies an ACK from the Arduino used in the handshaking process.	XILINX
axi_gpio_video	GPIO block used to interface with the the HDMI hot plug detect signal.	XILINX

axi_mem_intercon	Memory interconnect block that allows for different blocks to access DDR memory. This block is essentially an arbiter for the memory.	XILINX
axi_timer_0	Timer block used to generate interrupts.	XILINX
axi_uartlite_0	UART block used to output messages from the Microblaze. In this project, all print statements in software are sent through the UART block and can be displayed on the SDK Terminal.	XILINX
axi_vdma_0	The first instance of the video direct memory access block. This block is responsible for taking in a frame from the video input and storing it into memory. It is also responsible for reading a frame from memory and sending it to the display controller to be displayed on a monitor.	XILINX
axi_vdma_1	The second instance of the video direct memory access block. This block is responsible for reading a frame from memory stored by the first VDMA and passing it to the led_detect_0 block row by row. It is also responsible for reading rows from the led_detect_0 block and writing them to another memory location.	XILINX
dvi2rgb_0	Video decoder that converts hdmi stream information into RGB raw data	XILINX
led_detect_0	This block is responsible for detecting LEDs within a frame. The block takes in one 1280 pixel row of the frame, modifies the row and outputs the modified row. The output of the block is a frame where all pixels are black except for the detected LEDs. Once 720 rows have been given to the block, i.e. one frame, then the block will output the x,y coordinates of the last pixel of the detected LED. If no LED was detected, the coordinates will hold their previous values.	Group 5
mdm_1	Microblaze debug module.	XILINX
microblaze_0	The Microblaze block is the soft processor where all software code is executed. The Microblaze block is responsible for controlling the two VDMA blocks. It is also responsible for syncing the video input and the video output. It also controls the data transfer between the FPGA and the Arduino through the GPIO block.	XILINX/Group 5
microblaze_0_axi_intc	The Microblaze interrupt controller block.	XILINX
microblaze_0_axi_periph	Interconnect to route Microblaze control signals to different blocks.	XILINX
microblaze_0_xlconcat	Interrupt concatination block used by Interrupt Controller.	XILINX
mig_7series_0	Memory Interface generator that provides an interface for reads and writes to DDR memory.	XILINX
rgb2dvi_0	Video encoder block that converts RGB to DVI format used to display through HDMI output.	XILINX
rst_mig_7series_0_100M	Reset block for the whole system.	XILINX
rst_mig_7series_0_pxl	Reset block for the video input modules.	XILINX

v_axi4s_vid_out_0	Video output converter that converts an AXI video stream to RGB pixels.	XILINX
v_tc_0	Video Timing Controller for video output.	XILINX
v_tc_1	Video Timing Controller for video input.	XILINX
v_vid_in_axi4s_0	Video input converter that convert RGB pixel data to AXI video stream.	XILINX
xlconstant_0	Constant 1 to enable HDMI output.	XILINX
xlconstant_2	Constant to tstrb input of led_detect_0 block.	XILINX

2 Description of System Blocks

2.1 LED Detection IP

The LED detection IP is a critical aspect of this project. It allows the rapid processing of a continuous stream of images supplied by the camera to determine the locations of two coloured LEDs in the frame. Figure 2 illustrates the functionality of the IP, as follows:

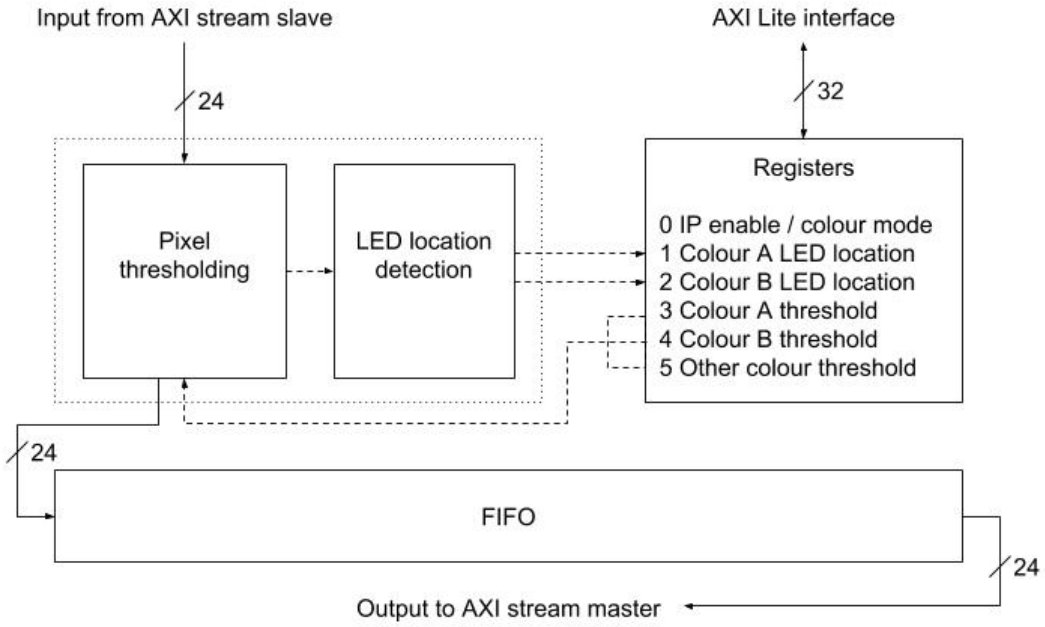

Figure 2: A depiction of the framework of the custom image processing IP.

The crux of this IP is the determination of the location of two LEDs, one of colour A and another of colour B. In this discussion, A is assumed to be red, while B is green, as this was the demonstrated setup. The IP is constituted of three hardware modules described in Verilog HDL: an AXI streaming slave S_00_AXIS, an AXI streaming master M00_AXIS and an AXI-Lite slave S00_AXI. The streaming slave accepts the stream of pixel values supplied by the VDMA through the appropriate RX handshaking. This represents the video data, which is sourced from

its memory-mapped storage in DDR. Each pixel is 24 bits in size (8 for each colour) and thus the AXIS_TDATA input bus has a width of 24. The streaming slave then passes this data stream to the S00_AXI AXI-Lite slave, in addition to a write pointer indicating which pixel in the current line of the frame is associated with this RGB value.

The actual processing of the pixel to detect the location of the LEDs occurs in the AXI-Lite slave. For simplicity, consider the mode that was ultimately demonstrated, mode 2, which assumes red and green coloured LEDs. Upon receiving the raw pixel data stream from the streaming slave, the Lite thresholds this RGB value to determine whether it represents a red or green pixel. The thresholding works as follows - to check whether a pixel is red, check if its R channel is greater than the colour A threshold (slv_reg3) and that both G and B are less than the *other* colour threshold (slv_reg5). Likewise, when checking if a pixel is green, check that G is greater than the colour B threshold (slv_reg4) and that R and B are less than the value in slv_reg5.

After thresholding, a generated pixel is stored in a FIFO queue. This queue has a depth equal to the width of a frame, namely 1280, and thus represents an entire line. The location in the FIFO queue is indicated by the write pointer supplied by the streaming slave. If the thresholding determined that the pixel was sufficiently red, a pure red (0xff0000) pixel is stored at this location in the FIFO. This is the same for green, and if neither the red or green conditions are met, a black pixel is stored. The purpose of this creation of a new frame is debugging - it allows the iterative updating of threshold values quickly to determine which are good for a particular environment.

The next step is to determine the location of the LEDs in the frame. For this application, the absolute position of the LEDs in the frame is not very important. It is more important to correctly characterize the relative position and movement of the LEDs. Therefore, the actual location of the LEDs in the frame can safely be approximated, as long as the localization technique is consistent. Accordingly, the simple method of detecting the last red and green pixel on the screen was used for finding the location of the LED. Assuming the signature of the LED is a circle or oval of coloured pixels in the frame, this will find the bottom center of the blob. Therefore, this provides a sufficiently consistent method to determine the motion of the LEDs. This is implemented by maintaining a count of the current y coordinate, or line number of the frame being processed. Whenever a red pixel is encountered, a register ledr_xy is updated to contain the x index (write_pointer) and y index (y_coord) of the pixel. When the y_coord counter indicates that the IP has processed 720 lines (the height of the frame), it is reset and the current value stored in ledr_xy (which represents the location of the last red pixel in the frame) is written to register 1, the red LED location register.

After the FIFO has been entirely populated, i.e. the whole line of a frame has been processed, the IP switches to transmission mode. Here, the contents of the FIFO are sent from the AXI-Lite slave to the VDMA through the IP's streaming master, M00_AXIS. As the data is output as a stream, the master is responsible for TX handshaking and specifying which pixel value to send at which time. To accomplish this, the master maintains a counter similar to the slave's write pointer, the read pointer. This pointer indicates which pixel in the FIFO, and equivalently which pixel in the line being streamed, is present on the master AXIS_TDATA output bus.

The discussion above assumed operation in the red/green mode, mode 2, which expects a red and green LED to be used. This IP is capable of a total of three modes, 1, 2 and 3. The desired mode is specified by setting the enable/mode register (register 0) through AXI-Lite. The other two

available modes (colour A/colour B) are red/blue (1) and blue/green (3). To disable the IP, set this register to 0. The layout of the registers accessible via the AXI-Lite interface are as follows. Please note that the red/green mode (2) is considered the default, and thus the descriptions here are based on that setup for clarity.

Table 2: Details of LED Detection IP Registers

Register	**Read/Write**	**Description**
slv_reg0	read/write	IP enable/mode.
slv_reg1	read-only	Represents the approximate location of the colour A (**red** in this discussion) LED in the frame, or its last known location if it is no longer visible.
slv_reg2	read-only	Represents the approximate location of the colour B (**green** in this discussion) LED in the frame, or its last known location if it is no longer visible.
slv_reg3	read/write	The minimum threshold for colour A, **red** in this discussion.
slv_reg4	read/write	The minimum threshold for colour B, **green** in this discussion.
slv_reg5	read/write	The maximum threshold for the *other* colour, i.e. when checking if a pixel is red, G and B must be less than this limit.

For the detailed layouts of these registers, please refer to Appendix A.

2.2 Gesture Detection Software

The purpose of the gesture detection software is to use the LED coordinates detected by the LED detection IP to determine any occurrence of a set of gestures. To achieve this, it polls registers 1 and 2 of the LED detection IP to get x and y coordinates representing the red and green LED positions respectively. It stores the coordinate information into arrays for further processing. After every new set of coordinates are received a check is done for each gestures that is to be detected, this includes vertical swipes to indicate scrolling, horizontal swipes to indicate page/task switching, and converging/diverging motion of the two LEDs to indicate zooming.

2.2.1 Scrolling and Task Switching

The scroll detection only tracks the motion of the red LED. The x displacement and y displacement between consecutive points in the array are calculated and summed. This continues until a scroll condition is met or until all the coordinates in the array are exhausted. The scroll condition is satisfied when the total y displacement exceeds the y-threshold while the total x displacement is below the x-threshold, and the direction of the scroll is determined by the sign of the total y displacement. This signifies that the swipe motion has reached a certain distance and has a slope close to vertical. The maximum duration(i.e. minimum speed) of the motion is set by the size of the array. Once the gesture is detected, it is sent to the Arduino so that the Arduino can send the corresponding command to the computer. Then the arrays are reset before a new gesture can be detected. The same process is used to detect horizontal swipes for task switching by interchanging the operations done on x and y.

2.2.2 Zooming

The zoom gesture is evaluated based on the distance between the two LEDs. When new coordinates are received from the the LED detection IP, the distance between the two points is calculated and stored in a separate array. The ratio between the newest point and previous points is calculated one at a time until a zoom condition is met or the array of distances is exhausted. The zoom in condition occurs when the ratio of the newest distance and one of the previous distances is greater than the zoom in ratio and a zoom out occurs if the ratio is less than the zoom out ratio. The maximum duration(i.e. minimum speed) of the gesture is set by the size of the array

The Gesture detection software is run on the Microblaze soft processor. The Microblaze version used for this project is 9.6(Rev. 1) from the Xilinx library.

2.3 Video Direct Memory Access

The software executing on the Microblaze is responsible for controlling the transfer of data between the two Video Direct Memory Access(VDMA) blocks. In the software, two $1280 \times 720 \times 3$ global arrays are allocated to hold two frames. Since the .data section of the code is located in DDR memory, it means that these arrays will also be allocated in DDR memory. Through software, a frame is read into the first VDMA which will store this frame in the first array location in DDR. Next, the second VDMA will read 1280×3 bytes from the first array location, which corresponds to one row of the frame. This row is then sent to the LED detection block which will return a modified row. The second VDMA continues to poll the output of the LED detection block until valid data is read. The modified row is then written back to DDR in the second array location. This process is repeated until 720 rows have been processed. At this time, the complete modified frame exists in the second array location in DDR. The first VDMA then proceeds to read the entire second array from DDR and sends it to the display controller to be displayed on the monitor.

The VDMA block used was version 6.2(Rev.8) from the Xilinx library. The software library code to read and write to a VDMA was taken from an online source [2]. This code was modified to read and write only a single row. The methodology in which this code was used was completely unique by the team.

2.4 Existing IP

The following IP blocks used in the project have been taken off the shelf with zero or minor modifications.

2.4.1 AXI Dynamic Clock

AXI Dynamic Clock that is used to synchronize the output video blocks within the system. Version 1.0 (Rev. 2) was used from the Xilinx library.

2.4.2 AXI GPIO

The AXI GPIO block is responsible for interfacing AXI signals from the microblaze to the GPIO output signals.

2.4.3 AXI Memory Interconnect

The AXI memory interconnect allows for arbitration between various AXI blocks and the memory controller. Version 2.1(Rev.10) was from the Xilinx Library.

2.4.4 AXI Timer

AXI Timer block that generates interrupts. Version 2.0(Rev.11) was used from the Xilinx library.

2.4.5 AXI Uartlite

The Uart block allows for the Microblaze to communicate with the host computer using a terminal. Version 2.0(Rev. 13) was used from the Xilinx Library.

2.4.6 DVI2RGB

The DVI2RGB converts the HDMI input stream into raw RGB data. Version 1.5(Rev. 7) was used from the Xilinx library.

2.4.7 Microbraze Debug Module

The Microbraze Debug Module allows for software debugging on the Microblaze. Version 3.2(Rev. 6) was used from the Xilinx library.

2.4.8 Microbalze AXI Interrupt Controller

The Microblaze interrupt controller handles all the interrupts to and from the Microblaze. For this system, version 4.1(Rev. 7) was used from the Xilinx library.

2.4.9 Microblaze AXI Peripheral

The Microblaze AXI Peripheral block is an interconnect that routes the AXI output from the Microblaze to the correct AXI block. Version 1.5(Rev. 17) was used from the Xilinx library.

2.4.10 Microblaze Xlconcat

Microblaze Concat block is responsible for concatenating all interrupts into the Microblaze. Version 2.1(Rev. 2) was used from the Xilinx library.

2.4.11 Memory Interface Generator

The Memory Interface Generator is a memory controller for the DDR memory. It allows for interfacing between various blocks and the memory. Version 4.0 was used from the Xilinx library.

2.4.12 RGB2DVI

The RGB2DVI block is a video encoder responsible for converting RGB data to DVI format used by HDMI output. Version 1.2(Rev. 5) was used from the Xilinx library.

2.4.13 Processor System Reset

The Processor System Reset provides the reset signals used by all blocks within the system. Version 5.0(Rev. 9) was used from the Xilinx library.

2.4.14 AXI4-Stream Video Converter

This block is responsible for converting AXI stream video data into a complete RGB frame, This block can also be configured to converted RGB data into AXI stream. Version 4.0(Rev. 3) was used from the Xilinx library.

2.4.15 Video Timing Controller

The Video Timing Controller receives synchronization and timing signals as input. It is then able to output the received video frame. This block allows for synchronization of the video output through the Microblaze. Version 1.5(Rev. 17) was used from the Xilinx library.

2.4.16 Xlconstant

A constant block used to tie off certain signals. Version 1.1(Rev. 2) was used from the Xilinx library.

2.5 Arduino Keyboard Interface

The final piece of this project is the interface connecting the FPGA to the PC. After the LED data is processed by the MicroBlaze, a gesture is determined, encoded and sent out using the 3.3V logic of the Nexys GPIO. These signals pass through a level shifter to bring them to 5V and are supplied to the Arduino input pins. The Arduino decodes the action and sends corresponding keystrokes to a connected PC using the standard Keyboard.h [3] library. After sending the keystroke(s), the Arduino asserts the ACK line high, which again passes through the shifter and is accepted as input by the FPGA, indicating that the transaction is complete.

3 Project Schedule

This section details the proposed project schedule and the actual weekly accomplishments.

Table 3: Project Schedule

Milestone	Date	Proposed	Delivered	Notes
1	Feb 10	Setup camera to be able to read video data into a video decoder. Store decoded video in DDR memory.	Setup of HDMI demo system. HDMI video input stream sent to DDR and displayed on HDMI display.	A demonstrative system was found, so focus was on making it work on Vivado 2016.2 and the FPGA.
2|original testbench demo	Feb 17	Setup block level testbench to be able to test custom IP image processing block. Display video frame from DDR into VGA display. Setup MicroBlaze.	Image processing algorithm prototyped in software. IP core master IP block created. Investigated IR and coloured LED video capture and OpenCV processing.	As the date of the testbench demo was pushed back a week, the focus shifted to implementation.

3|actual testbench demo	Mar 3	Completion of video converter.	Initial prototype of IP core implemented. Simulation testbench implemented, including output stream analysis, golden value comparison. Initial project set up for ILA testbench.	Testbench was shifted to this week. Video converter implementation was unnecessary. IP was started earlier than originally anticipated.
4	Mar 10	Completion of image processing custom IP block.	IP updated with prototype of LED location detection and writing to AXI slave registers. Thresholding successful in testbench. Integration of image processing IP into HDMI demo system. Initial base code for controlling VDMA written.	IP development longer than expected, though successful progress was being made. Integration of IP into system ahead of schedule.
5|mid-project demo	Mar 17	Completion of an initial version of gesture recognition software. Demo: display the detection of the LEDs onto a monitor	Integration of image thresholding version of IP with overall video demo system, successfully able to threshold red and blue areas in input stream and send new stream to display. Updated IP (with LED location detection) successfully tested with one frame in testbench. VDMA code completed.	Gesture recognition software delayed. However, satisfied requirements of initially proposed demo.
6	Mar 24	Completion of Arduino keyboard emulator.	Setup of Arduino as keyboard emulator, text typed into PC. Preliminary gesture detection software implemented. IP updated to allow different combinations of LED colours to be used, thresholds can be written in software.	Arduino keyboard emulator slightly behind schedule, but base setup works.
7|final demo	Mar 31	Integration of all components.	Components successfully integrated, demonstrated. IP correctly supplies locations of LEDs which are read and analyzed by software on the MicroBlaze.	Project completed on time.

4 Outcome

4.1 Results

The goals set out to be completed for this project were all achieved with only a few minor variations. An LED glove was successfully created to indicate the locations of the user's hands. HDMI camera input was used as the input to the system and a custom created LED Detection IP was able to detect the locations of the LEDs. These locations were used by gesture detection software which ran on the Microblaze soft processor. Then the corresponding gesture keyboard input was sent to the personal computer through an Arduino, thus allowing a user to interact with the screen. Six gestures were included and demonstrated: scroll up/down, switch task left/right and zoom in/out. The system worked reliably when used in the real practical scenario of presenting the final demonstration slideshow.

4.2 Possible Further Improvements

Although the goals of the project were met, there are still many ways to further advance this design. Since the bottleneck of this project was time to completion rather than resources, the software written for gesture detection was written as quickly as possible with the main concern being readability rather than optimization. For example there are more arrays used that can be optimized out of the code. Further optimization of the gesture detection function can be done by implementing it in hardware. There can also be an unlimited number of additional gestures added to the system to allow even more remote interactivity with the computer. One of the more complex possibilities would be to control the mouse. One possible implementation of this would allow the user to go into mouse mode by flashing the red LED twice, after which point the motion of the green LED would translate into motion of the mouse pointer and single flashes of the red LED would indicate left mouse clicks. Another improvement would be to make the LED detection IP thresholds automatically adjust so that they don't need to be manually tweaked for each different room that the system is used in. This can be achieved by adding a calibration step during which the LEDs are turned on at the location where they will be used, and the system makes the threshold values increasingly conservative until only a single cluster of red and green pixels are detected. Alternatively, the whole LED detection system can be replaced if hand detection is implemented so that the coordinates of two hands can be used instead of LEDs which would mean that no glove would be required to operate the system.

5 Description of Design Tree

The code for this project is hosted on a GitHub repository. Figure 3 shows the directory breakdown for the repository. The arduino/ folder contains the code run on the Arduino that will convert signals from the FPGA to keyboard strokes. The docs/ folder contains this report, the presentation slides as well as a video. The img_processing_ip/ folder contains the LED detection IP blocl. The subfolder ip_repo/ is the folder that should be included if the IP is going to be used in other projects. The src/ folder contains the main project with the IP integrated into it. Under the hdmi.sdk/ folder, hdmi_wrapper_hw_platform_6/ is the latest exported hardware and works with standalone_bsp_1/ board support package. The subfolder videodemo/ contains

all the software code used in this project. The tb/ folder contains the testbench used to test the LED detection blocks.

```
.
|____ arduino/
|____ docs/
|       |____ ECE532_Group5_Presentation.pdf
|       |____ ECE532_Group5_HGCI_Documentation.pdf
|       |____ Group5_HGCI.mp4
|____ img_processing_ip/
|       |____ ip_repo/
|       |____ managed_ip_project/
|____ README.md
|____ src/
|       |____ hdmi.sdk/
|       |       |____ hdmi_wrapper_hw_platform_6/
|       |       |____ standalone_bsp_1/
|       |       |____ videodemo/
|       |       |       |____ src/
|       |       |       |____ vdma/
|       |       |       |____ video_demo.c
|       |       |       |____ video_demo.h
|       |____ hdmi.xpr
|       |____ NexysVideo_Master.xdc
|       |____ repo/
|____ tb/
```

Figure 3: Directory Tree of Repository

6 Tips and Tricks

One debugging technique used by the group to debug the hardware and software together was to combine the use of ILA blocks along with the SDK's debugging option. Once the ILAs have been placed in the design and the bitstream has been created, the board should be programed through Vivado to enable the ILAs. Then the software should be run from the SDK in debugging mode, without programming the board again. As the code is stepped through, it will trigger the ILAs in the Vivado GUI. This method can be used to debug hardware in a large system.

Obstacles faced during filtering the input image based on the colour of objects lead to testing of different thresholding methods. The simplest and best working method found was using two thresholds per colour to be identified. If, for example, red was to be identified then one threshold would define how red a pixel needs to be, while a second threshold would set a lower limit for the maximum magnitude of blue and green. When only one threshold was used, for example, by simply checking the ratio of red to the green and blue there were many incorrectly identified spots especially around the edges of all objects. (This may have an application for general edge detection, but it was problematic for this project)

Appendix A LED Detection IP Register Layouts

The layouts of the registers are presented in the following tables:

Table 4: Layout of control register, slv_reg0.

Range	Description	Possible Values
31:0	Disable/mode	0 disable | 1 red/blue mode | 2 red/green mode | 3 blue/green

Table 5: Layout of LED A and B location registers, slv_reg1 and slv_reg2.

Range	Description	Possible Values
31:16	x-coordinate	A value between 0 and 1279 (the frame width) representing the horizontal position of the LED in the frame.
15:0	y-coordinate	A value between 0 and 719 (the frame height) representing the vertical position of the LED.

Table 6: Layout of threshold registers, slv_reg3, slv_reg4 and slv_reg5.

Range	Description	Possible Values
31:0	Threshold value	Any 32 bit value.

References

[1] "CyberGlove III", CyberGlove Systems LLC, 2017. [Online]. Available: `http://www.cyberglovesystems.com/cyberglove-iii/`

[2] Safavi F., et al. "Interactive Studio", 2017. [Online]. Available: `https://github.com/marmot1234/G9_InteractiveStudio`

[3] "Mouse and Keyboard libraries", Arduino.cc, 2017. [Online]. Available: `https://www.arduino.cc/en/Reference/MouseKeyboard`