Group Report: Super Pong^{TM}

ECE532

April 10, 2023

Team 5

Harsimrat, Yash, Aditya, Haoran

1. Overview

This document aims to describe and document the design process for the SuperPong^{TM} game developed by Team 5 of the ECE532 project.

1.1 Project Goals and Motivations

This super pong game console is developed based on the classical pong game running on PC, but with five more interesting features. The first feature is to allow two players to play the pong game using two joysticks, which is the most fundamental function provided by Superpong. The second feature is touchless control with high accuracy and no latency. It allows one player to play the game by moving his or her finger in the air, making the player feel more engaged in the game. The finger movements are captured by a camera and control the game. The third feature is HDMI output that displays the game on a large screen with a super high quality, replacing the fuzzy video quality of the classical game. The fourth feature is a soft processor that is able to execute most software programs. For this project, the processor only executes the program of the pong game. However, this super pong game console can be used to play various other games by running different game programs. The fifth feature is the integrated hardware implementation and the support of plug-in and play. All the hardware modules and the software processing unit are implemented in one single FPGA chip (embedded on a Nexys Video board). The whole system can be easily updated on the FPGA chip and then can be implemented as a small ASIC chip applying 65nm process technology in the future.

1.2 System Architecture

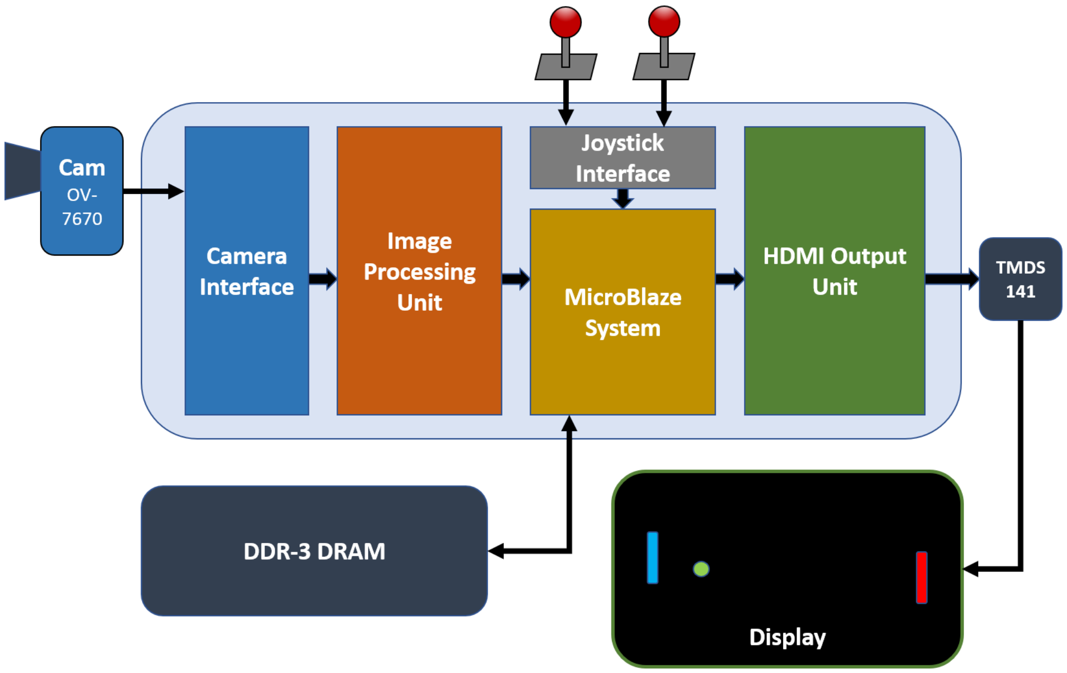


Fig. 1. High level architecture design of the Super Pong System.

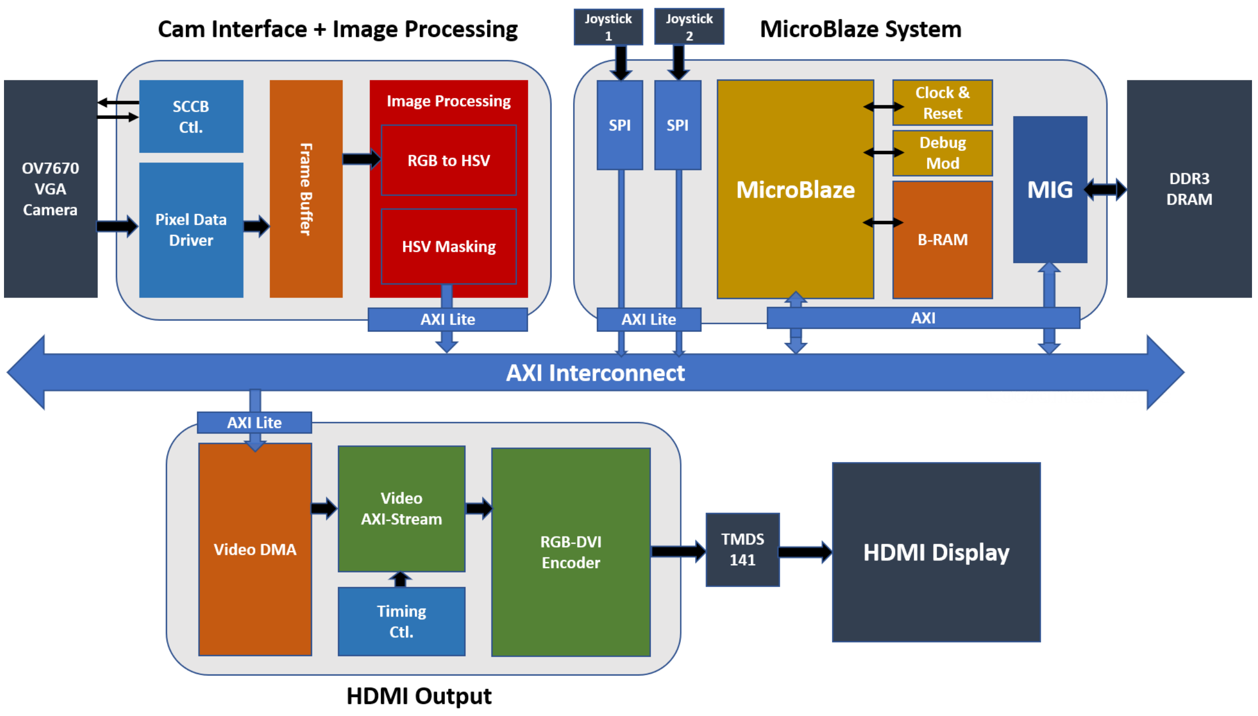


Fig. 2. Architecture Block Diagram of the Super Pong System

1.3 Description of IPs and Reference Designs

1.3.1 IPs in Camera Interface

The Camera Interface unit consists of two parts: *sccb_ctrl* and *ov7670_read*, which is the system interface to the OV7670 VGA camera. The camera interface module together with the image processing unit are integrated in the *ipu_top* module. This top module is wrapped up with AXI-lite interface. There are only two Xilinx IPs used in this part. One IP is for the frame buffer, and another IP is used to generate AXI-lite wrapper. The camera uses 16 bits configuration, where rgb values are sent in 2 bytes (565 configuration of rgb bits) in two cycles.

Table 1: Camera Interface IP

IP	Maintainer	Description
fifo_generator_0	Xilinx	The IP is used to generate the frame buffer that stores image frames and works as an interface between the camera interface unit and the image processing unit.
AXI-lite IP Interface (IPIF)	Xilinx	The IP generates Verilog code for a primitive AXI-lite wrapper that was modified to properly wrap up the top module *ipu_top*.

1.3.2 IPs in Image Processing Unit

Table 2: Image Processing IP

IP	Maintainer	Description
myip1_1.0	Custom developed	The IP consists of two modules *ImageIPU (responsible for masking operation of color objects), and the hsvConverter (responsible to convert rgb2hsv color space)* core module for image processing.
Ram	Xilinx	Block ram IP that is used to store coordinate values and rgb values for display coming out of the image processing unit.

1.3.3 IPs in MicroBlaze System

The Microblaze system consists of the Microblaze Processor IP configured for performance with cache control, external DRAM memory for instructions and data, and a debug module. The soft-processor system is configured with interrupts and communicates with various AXI enabled slave interfaces by using the shared AXI interconnect used by the entire digital system. The

Microblaze soft-processor uses C-code to initialize the VDMA driver, the Display controller, interrupt controller and various other peripherals like joysticks, and UART modules. Timer modules, accessible through the AXI interconnect, allow controlling the rendering time to achieve 30 frames per second worth of refresh rates.

1.3.4 IPs in Joystick Control

The joysticks interface with the design through the PMOD ports on the board. The following IPs we were used enable the Joystick control for the board:

Table 3: Joystick Control IP

IP	Maintainer	Description
PmodJSTK2_v1_0	Digilent	This is an IP provided by Digilent to interface the JSTK2 joystick with the board using one of the on-board Pmod ports. The IP implements a SPI communication channel with the joystick and takes a 16Mhz clock as input. This IP also implements a AXI Lite slave interface to connect with the microblaze processor.
Clock Wizard	Xilinx	This is a Xilinx provided IP core that produces various clocks needed in the design from an input clock.

Apart from the above-mentioned IPs the joystick interface also utilized the AXI interconnects and AXI bridges to connect to the microblaze processor.

1.3.5 IPs in HDMI Output

Table 4: HDMI IP

IP	Maintainer	Description
Memory Interface Generator (MIG) 7-Series	Xilinx	The MIG provides a memory controller and a PHY for the MicroBlaze processor and other IPs to communicate with a DDR3 DRAM through an AXI interface.
AXI Video Direct Memory Access	Xilinx	This is a Xilinx provided IP core that provides high-bandwidth direct memory access between memory and AXI4-Stream video type target peripherals. The VDMA in this project reads the DRAM frame buffer and forwards the data as an AXI-Stream to the AXI-Stream to Video Out IP as shown in Fig. 4.
AXI4-Stream to Video Out	Xilinx	This Xilinx IP block converts AXI-4 Stream interface signals to a standard parallel video output interface with timing signals. This core works in conjunction with the Xilinx Video Timing Controller (VTC) core to generate the video format timing signals.
RGB to DVI Video Encoder	Digilent	This module connects to a top level DVI 1.0 source interface consisting of three TMDS data channels and one TMDS clock channel. It includes the necessary clock infrastructure (optional), encoding and serialization logic.
Video Timing Controller	Xilinx	This IP core is a general purpose video timing detector and generator, which automatically detects blanking and active data timing based on the input horizontal and vertical synchronization pulses.

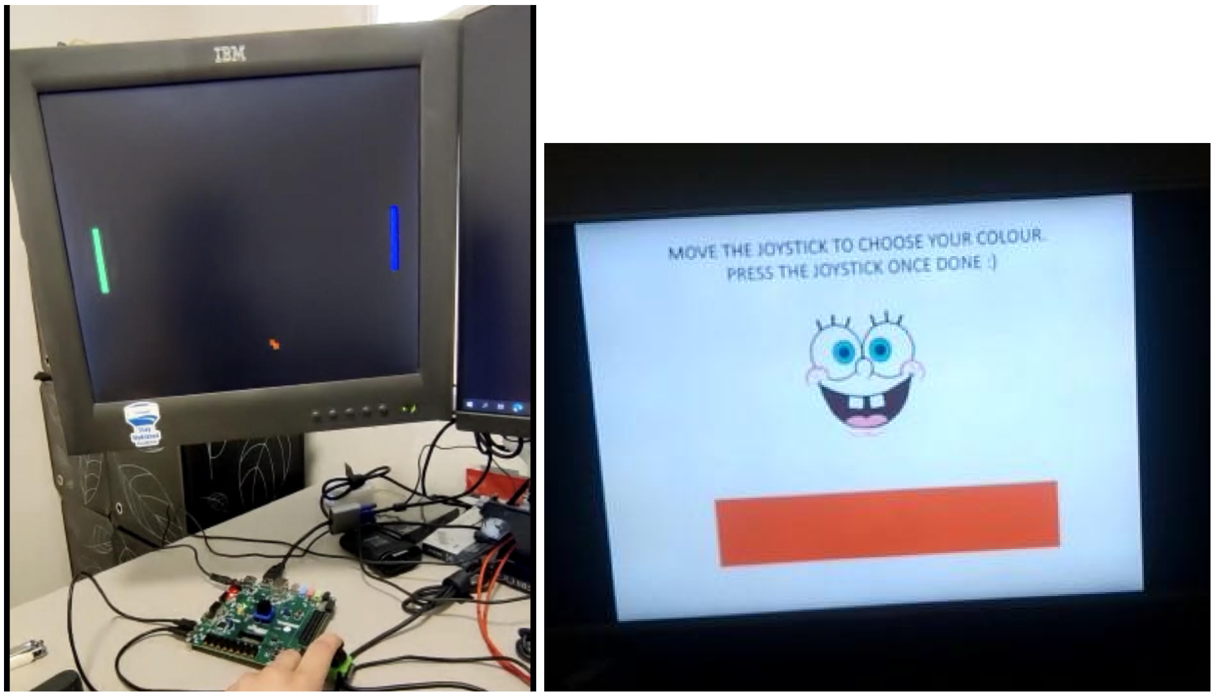
1.3.6 Other IPs

AXI Timer and AXI Uartlite modules from Xilinx are used in the system to determine time passed between frame renders and facilitate communication through a Serial terminal.

2. Outcome

2.1 Results

In this project, we were able to independently develop the HDMI display and image processing units. Unfortunately, due to time constraints, we were unable to integrate the image processing unit together with the HDMI display system. The HDMI display system is able to render the video game screen at 30 frames per second and it allows 2-player and player vs. CPU capabilities. The integration of 2 joysticks with the HDMI Pong system allows a richer, immersive playing experience and highlights the work put in by the team to integrate as many components together. The image processing system is able to detect the coordinates of coloured components in the field of view of the camera; however, the filtering algorithm can be improved to better distinguish coloured components from the background under various lighting conditions. Since the addition of the HLS implementation of IPU was added relatively late during the design process further improvements will greatly improve the accuracy of the image processing system. The figure 3 shows output from pong game running on microblaze and joystick integrated as part of our MVP.


IBM
MOVE THE JOYSTICK TO CHOOSE YOUR COLOUR.
PRESS THE JOYSTICK ONCE DONE :)

a.) b.)

Fig 3. a.) Pong Game + Joystick b.) Pong Game Player Selection

Further, in figure 4 the blue cap is taken as the object to detect and the masked output can be seen accordingly. The image and threshold quality is low in comparison to opencv since the rgb values from camera ov7670 is 5-6-5 r-g-b (16 bits) only. The coordinates of the detected object can be seen as the LEDs light up according to the Y axis numbers of the top point of the detected object.

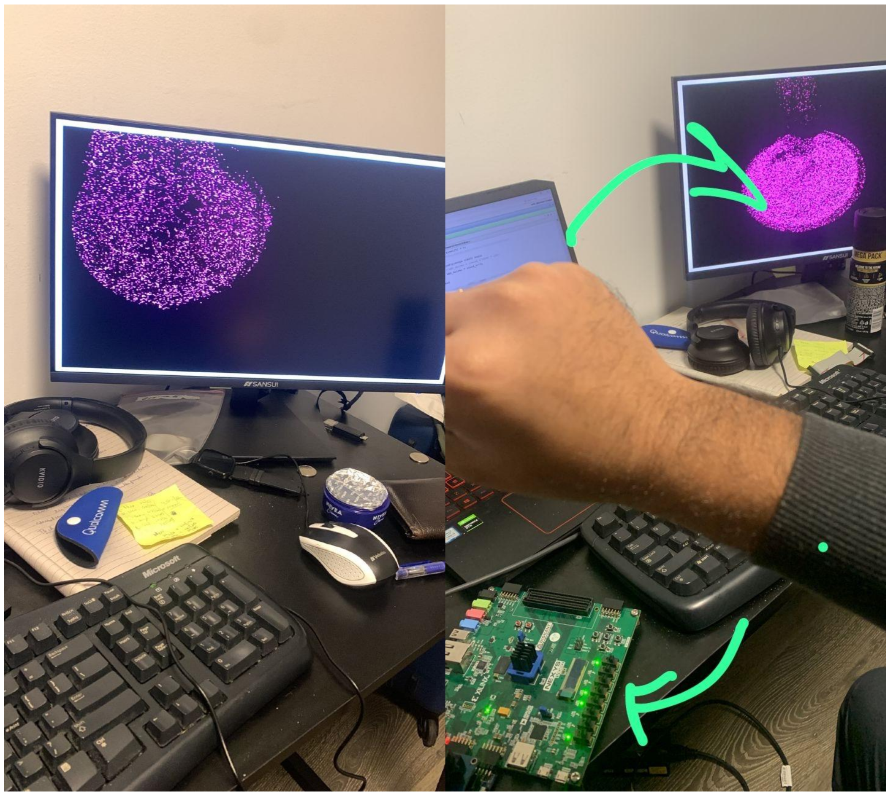


Fig. 4. Detection of Blue color cap masked video stream and coordinates in the left image can be seen on LEDs on the board.

As mentioned earlier, the final integration of passing these coordinate values through UART to microblaze running was unfinished due to time constraints.

2.2 Possible Further Improvements

If we could start this project over, we would completely redesign the HDMI subsystem as it was adapted from a prebuilt project and consumed too many BRAM resources because of an HDMI stream ingress system incorporated into it. This was a major bottleneck when integrating the image processing system with the HDMI system as both independently consumed 35% and 80% of the onboard BRAM resources, respectively. Therefore, a recommendation for someone taking over the project would be to spend more time thinking about the integration of the individual components in the system before rushing into their implementation.

3. Project Schedule

Table 5: Milestones

Milestone #1			
Team	**Plan**	**Execution**	**Challenges**
Aditya	Basic Pong game in C code	Same as the plan	Output game GUI through HDMI port
Harsimrat	Pong game logic emulation	Same as the plan	Go through HDMI tutorials and understand HDMI interface
Yash	Basic interface for joysticks	Same as the plan	Understand hardware requirements and go through the tutorials
Haoran	Basic architecture design	Same as the plan	Evaluate compute resource, memory resource, and bus bandwidth required by the system

Milestone #2			
Team	**Plan**	**Execution**	**Challenges**
Aditya	Pong game starter project on PC	Complete the Pong game running on PC	Switch between 2 scenes for 2 player and CPU options
Harsimrat	Basic HDMI output module	Follow HDMI tutorial and try to display test images	Pick reference modules and adapt them to the project needs
Yash	Hardware interface for one joystick	Complete the joystick interface	Compose software code to that enables MicroBlaze to read joystick positions
Haoran	Basic MicroBlaze system	Complete basic MicroBlaze system	Test the MicroBlaze system using C code

Milestone #3			
Team	**Plan**	**Execution**	**Challenges**
Aditya	Image processing unit implementation	Implementing line buffers for MAC. Completed researching/reading blogs about FPGA implementation on image masking.	understanding the image streaming process, AXI stream/DMA to communicate with IPU.
Harsimrat	Implement an HDMI tutorial on the FPGA and show some images on screen if possible.	Updating HDMI code to display images from RAM. Draw rectangles. Porting Aditya’s code to Microblaze.	Understanding memory addressing for the images and how to write to the correct pixel coordinates. Adapting the Pong game to run on the Microblaze processor (currently it is for PCs using SDL)
Yash	Software support for joysticks	Implemented Software Support for Joysticks. Searching/Started going through the HDMI tutorials.	HDMI tutorial not available for Vivado version 2018.3. The Laptop does not have enough memory for 2 versions of Vivado.
Haoran	Try to make a more complicated MicroBlaze system that provides better performance	Successfully implement and verify the new proposed MicroBlaze system.	Figuring out the interfaces to Image Processing module and HDMI Output module respectively.

Milestone #4			
Team	**Plan**	**Execution**	**Challenges**
Aditya	Complete processing stages for kernels and test on sample images.	Implemented Custom Image processing IP. Test IPU HSV and masking logic on software and calculated mask values.	IPU: Vivado 2016.4 throws error when integrated with DD3 (component UI missing) hence using 2018.4 to test the functionality, however backbone resource bitstream error persists.
Harsimrat	Fully integrate game code on the soft processor and allow the ability to play the game using on-board switches.	The game can be played on the HDMI screen using push buttons and joysticks.	Updating game logic to interface with HDMI while refreshing at a consistent speed, i.e. using timers to create precise delay before each frame is drawn.
Yash	Ramp-up on HDMI/go through the HDMI tutorials	Integrated the Joystick with the complete design. Playing games with the joysticks.	Updating the H/W platform for sdk was challenging. Updating the block diagram and importing the project onto the new hardware was challenging. The Vivado SDK does not handle updates to the hardware very well.
Haoran	Try to run the game on the new MicroBlaze system. Start to implement the camera interface.	Implement the VGA camera interface, which includes the camera controller and the input buffer that will be directly connected to the	Implementing an asynchronous FIFO buffer that is clocked from two different clock domains, and then testing it via testbench.

		Image Processing Unit.	Understanding the SCCB protocol and how to configure the camera. Figuring out the format of video data generated by the camera and how to organize it and write it to the buffer.

Milestone #5			
Team	**Plan**	**Execution**	**Challenges**
Aditya	Once the error is resolved, integrate RGB to HSV convert logic in existing custom IP and detect coordinates of objects (fingers).	Implemented Custom Image processing IP RGV2HSV and mask and connected AXI stream DMA to MM DDR3 for test image. Integrated IPU to main project.	IPU: Vivado 2016.4 throws error when integrated with DD3 (component UI missing) still trying to debug the issue.
Harsimrat	Update game code to clear any bugs with racquets and implement pausing functionality. Add images and animations to make the game more aesthetically pleasing.	Game implemented with Joysticks. The color of the joysticks and the player pads can be changed. The game resets if the ball goes out of bounds	No major challenges encountered other than being faced with an issue of sharing the hw/sw project using GitHub and co-ordinating with other team members. Trying to understand how to store sound files so that they can be played on the FPGA.
Yash	Ramp-up on the code/work already done on the image processor and	Ramped up on the image processing unit design. Could not contribute much this	Balancing the workloads of other courses this week made it difficult to

	contribute to the next steps.	week due to other workloads.	contribute much to the project this week.
Haoran	Complete the VGA camera interface, and then display live videos via HDMI.	Interface controller and an asynchronous input buffer are wrapped up as a SCCB interface module, which has been successfully verified by testbench.	Implementing and debugging the I2C protocol and its control logic in Verilog. Manually configuring the VGA camera. Debugging the wrapped-up module that integrates the controller and input buffer. Making testbench to verify the interface module independently from the rest system.

Milestone #6			
Team	**Plan**	**Execution**	**Challenges**
Aditya	RGB2HSV module is built and integrated with the main project, once the error is resolved, detect coordinates of objects and get image streams (FIFO) from camera integration.	Integration of IPU to main project (right now implemented IPU as a standalone project).	We scrapped the idea of getting images from DDR as it was showing errors while processing. Integrate camera design and IPU streaming.
Harsimrat	No major challenges encountered other than being faced with an issue of sharing the hw/sw project using GitHub and co-ordinating with other team members. Trying to understand how to store sound	Game completed with 2 Joysticks, pausing, color selection, score bars. Images added to indicate game over. Camera module tested and image shown on screen.	Understanding how to reverse engineer the project for the camera obtained from the internet. How to interface the image processing module with the AXI Lite interface to talk with MicroBlaze.

	files so that they can be played on the FPGA.		
Yash	Finish and contribute to the image processing unit.	Assisted with the creation of the Image Processing unit IP. Now the IP is packed in an AXI wrapper and ready to be integrated in the main project.	Although we are able to generate the masks now for the input images, we still have to get the coordinates from the masked output of the IPU. Given the time constraints we might get that done in the SW rather than hardware.
Haoran	Wrap up the module with AXI-lite interface, and integrate it to the whole system.	Complete the camera SCCB interface is done. Harsimrat. The AXI-lite wrapper for the whole IPU module is almost ready.	Figuring out the AXI Lite protocol, and how to correctly build internal connections between the blocks of IPU and the AXI Lite ports.

4. Description of System Components

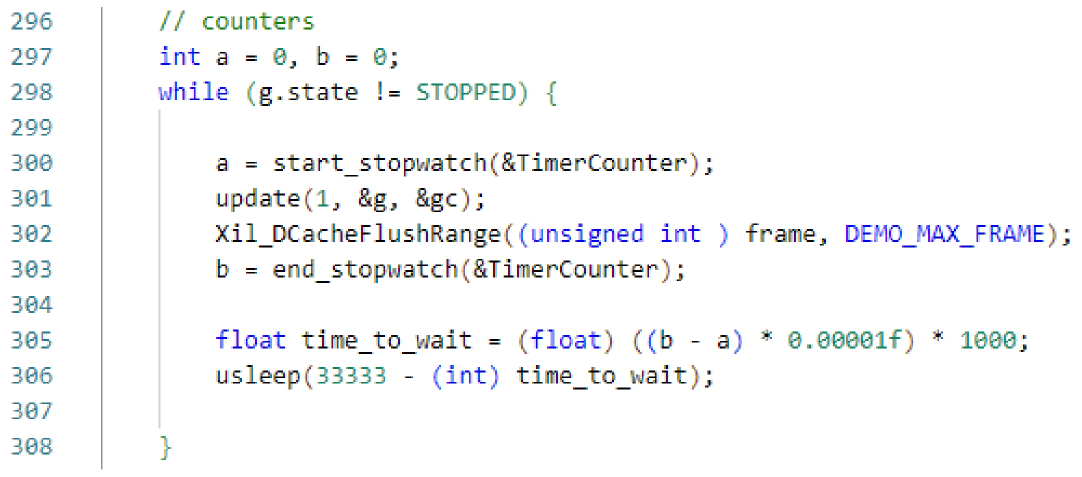
4.1 MicroBlaze Processor

The MicroBlaze processor has been configured under the "Performance" option inside the Vivado designer. This option uses instruction and data caches to improve instruction performance. The MicroBlaze processor acts as an AXI-Master to initiate transactions with the DRAM, Joystick IPs, Video DMAs, GPIOs, and timers accessible through the AXI interconnect. The MicroBlaze processor's instruction and data memory is primarily the DRAM as the BRAM resources are being used by the HDMI display system. The C program initializes the HDMI display IP, the Video DMA and interrupts and then proceeds with the main game logic

During gameplay, the processor is responsible for updating the HDMI frame buffer once every 30 seconds according to the location of the ball and paddle pixel locations. The location of an individual pixel, **(x,y)**, maps to a specific address in the DRAM buffer denoted by Eqn. 1 where **stride** is a constant offset defined in the source code. For a 1920 x 1080 resolution image, the value of stride would correspond to 1920.

`int iPixelAddr = x + (stride * y);` (Eqn. 1)

Achieving rendering updates every 30 seconds is facilitated by polling hardware timers accessible to the MicroBlaze processor through the AXI bus as discussed previously. The AXI timers are configured in capture mode to obtain values of the timer counters at different intervals and to measure the time difference between them. The programming loop that establishes the update rate can be seen in Fig. 3.



```
296    // counters
297    int a = 0, b = 0;
298    while (g.state != STOPPED) {
299
300        a = start_stopwatch(&TimerCounter);
301        update(1, &g, &gc);
302        Xil_DCacheFlushRange((unsigned int ) frame, DEMO_MAX_FRAME);
303        b = end_stopwatch(&TimerCounter);
304
305        float time_to_wait = (float) ((b - a) * 0.00001f) * 1000;
306        usleep(33333 - (int) time_to_wait);
307
308    }
```

Fig. 5. Listing for the main game loop. Timer values are captured on lines 300 and 303. The time required for the processor to pause execution is calculated on line 305.

4.2 Joystick Control

The JSTK2 joystick supports Serial Peripheral Interconnect (*SPI*) to communicate the coordinates of the joystick to the board. To implement the support for the Joystick in the project PmodJSTK2_v1_0 IP, developed by digilent, was utilized. There are 2 instances of this IP in the design which are connected to the *ja* and *jxadc* Pmod port. The IP requires a 16Mhz clock for the SPI interface. To create the 16 Mhz clock, a Clocking Wizard IP was added to the original HDMI design (which did not use a Clock Wizard).

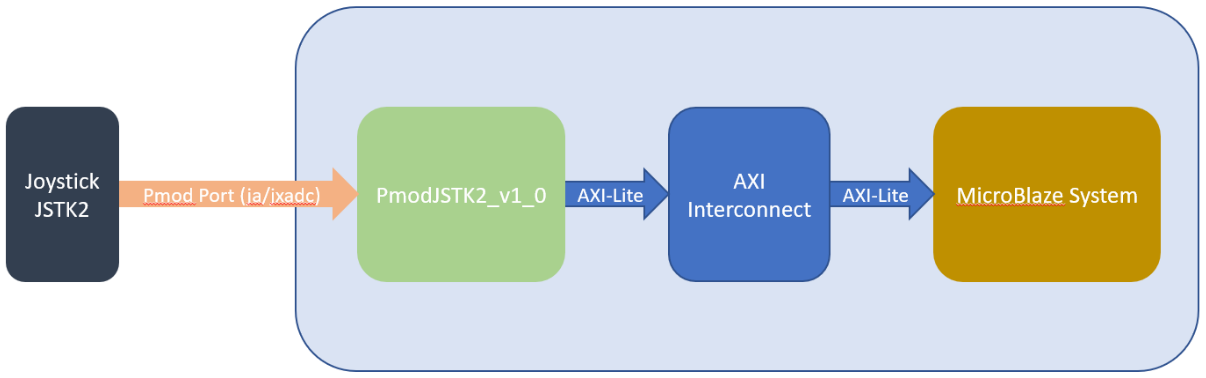


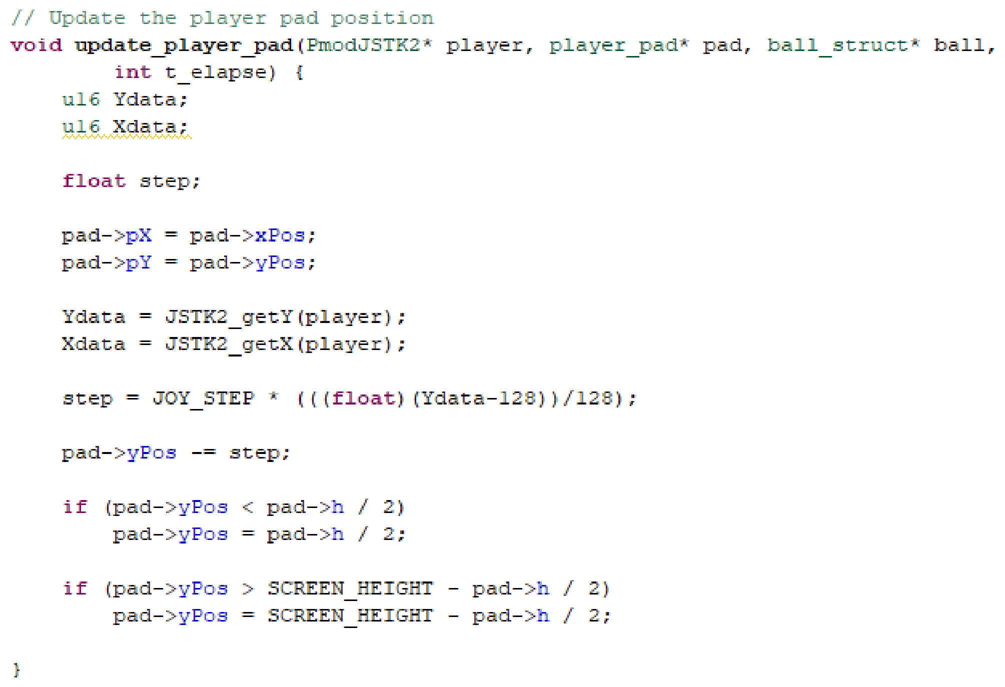
Fig. 6 Architecture Block Diagram for Joystick-MicroBlaze interface

The PmodJSTK2_v1_0 IP also implements a AXI-Lite slave interface which allows for the IP to connect to the MicroBlaze processor through an AXI interconnect.

The game logic retrieves the coordinates for the joystick before updating a frame using the following logic:

step = JOY_STEP * (((float)(Ydata-128))/128)   (Eqn. 2)

The maximum distance that the Joystick could move between 2 consecutive frames is JOY_STEP number of pixels. At each iteration (frame generation cycle) the microblaze processor calculates the position of the joystick relative to the center position of the joystick and then updates the position of the pad before rendering the frame.



```
// Update the player pad position
void update_player_pad(PmodJSTK2* player, player_pad* pad, ball_struct* ball,
        int t_elapse) {
    u16 Ydata;
    u16 Xdata;

    float step;

    pad->pX = pad->xPos;
    pad->pY = pad->yPos;

    Ydata = JSTK2_getY(player);
    Xdata = JSTK2_getX(player);

    step = JOY_STEP * (((float)(Ydata-128))/128);

    pad->yPos -= step;

    if (pad->yPos < pad->h / 2)
        pad->yPos = pad->h / 2;

    if (pad->yPos > SCREEN_HEIGHT - pad->h / 2)
        pad->yPos = SCREEN_HEIGHT - pad->h / 2;

}
```

Fig 7. Function to update player pad position based on joystick position.

The JSTK2 joysticks also contains 2 pushbuttons and a RGB LED which were used to implement features to improve game quality:

- Pushbuttons: By pressing the push buttons the game can be paused.
- RGB LED: When played in player-vs-player mode using the 2 joysticks, the player can choose the color of their Pad on screen and the RGB LED on the corresponding Joystick will have the same color.

4.3 Camera Interface

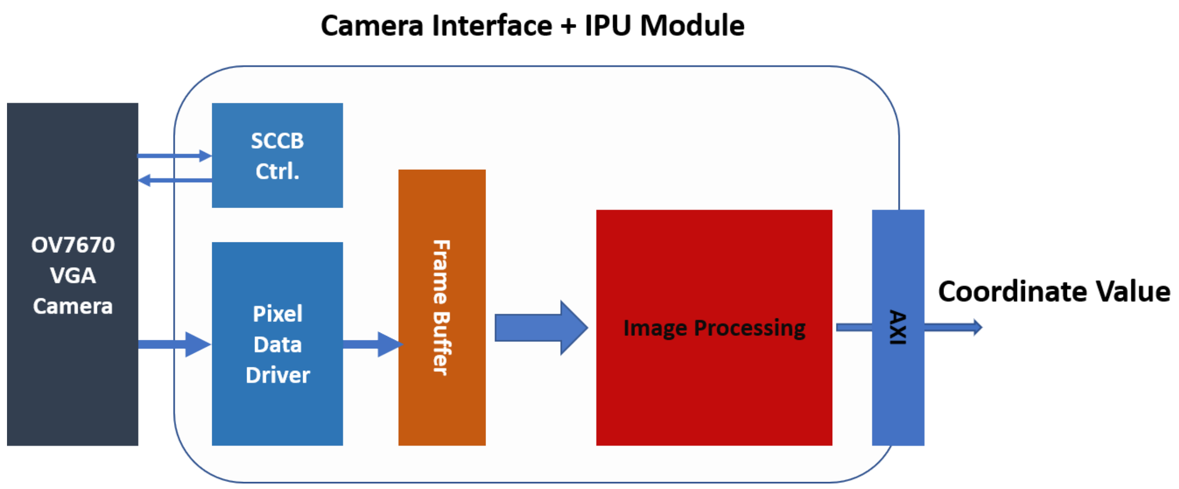
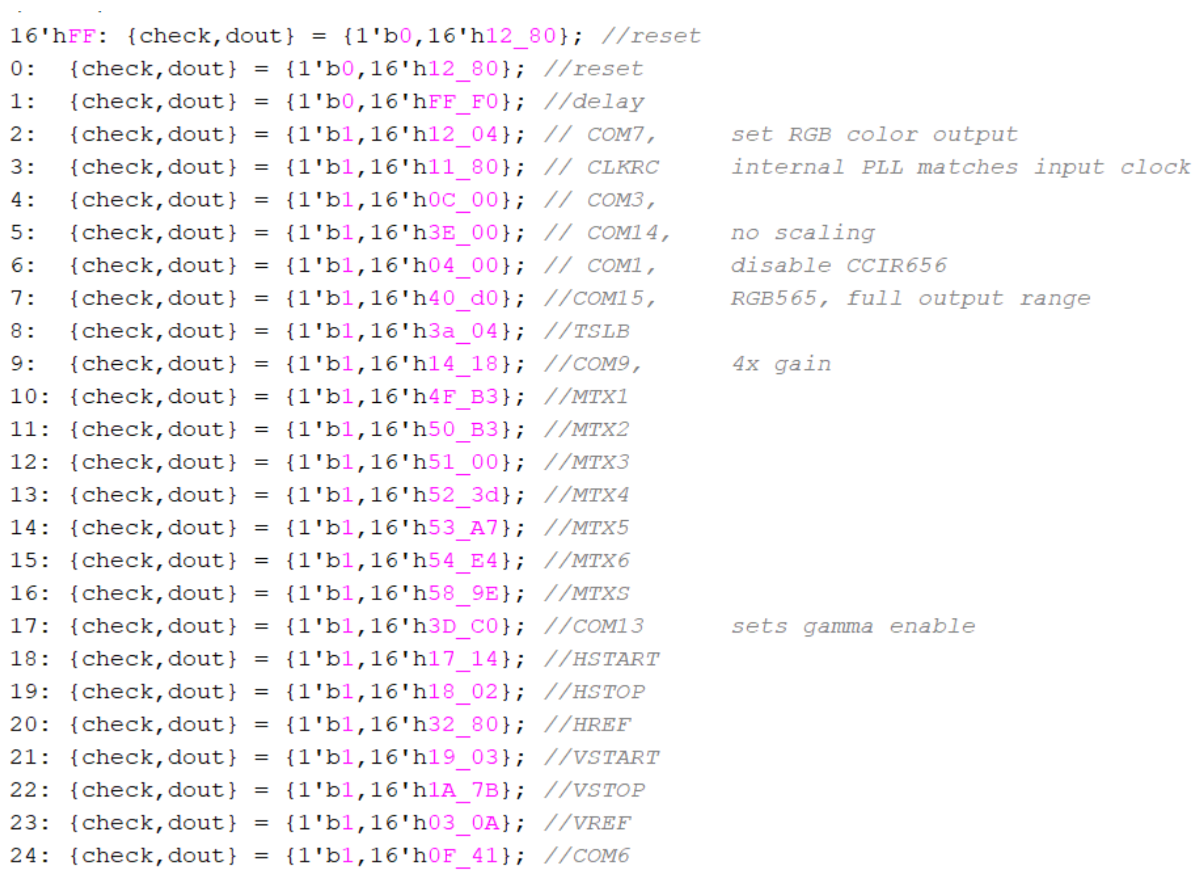


Fig 8. Block Diagram that demonstrates the Camera Interface

As shown in Fig 6, the Camera Interface unit consists of two parts: SCCB controller (implemented as *sccb_ctrl*) and pixel data driver (implemented as *ov7670_read*), which is the system interface to the OV7670 VGA camera. The SCCB controller implements the SCCB protocol, a subset of I2C protocol, consisting of three parts. The *sccb_write* and *sccb_read* send control signals to the camera and receive feedback, following the SCCB protocol. As shown in Fig 7, The *init_regs* initializes the camera configurations by writing specific values to the internal registers in camera. Some of these values are from the reference code on Github [7].



```
16'hFF: {check,dout} = {1'b0,16'h12_80}; //reset
0:  {check,dout} = {1'b0,16'h12_80}; //reset
1:  {check,dout} = {1'b0,16'hFF_F0}; //delay
2:  {check,dout} = {1'b1,16'h12_04}; // COM7,     set RGB color output
3:  {check,dout} = {1'b1,16'h11_80}; // CLKRC     internal PLL matches input clock
4:  {check,dout} = {1'b1,16'h0C_00}; // COM3,
5:  {check,dout} = {1'b1,16'h3E_00}; // COM14,    no scaling
6:  {check,dout} = {1'b1,16'h04_00}; // COM1,     disable CCIR656
7:  {check,dout} = {1'b1,16'h40_d0}; //COM15,     RGB565, full output range
8:  {check,dout} = {1'b1,16'h3a_04}; //TSLB
9:  {check,dout} = {1'b1,16'h14_18}; //COM9,      4x gain
10: {check,dout} = {1'b1,16'h4F_B3}; //MTX1
11: {check,dout} = {1'b1,16'h50_B3}; //MTX2
12: {check,dout} = {1'b1,16'h51_00}; //MTX3
13: {check,dout} = {1'b1,16'h52_3d}; //MTX4
14: {check,dout} = {1'b1,16'h53_A7}; //MTX5
15: {check,dout} = {1'b1,16'h54_E4}; //MTX6
16: {check,dout} = {1'b1,16'h58_9E}; //MTXS
17: {check,dout} = {1'b1,16'h3D_C0}; //COM13      sets gamma enable
18: {check,dout} = {1'b1,16'h17_14}; //HSTART
19: {check,dout} = {1'b1,16'h18_02}; //HSTOP
20: {check,dout} = {1'b1,16'h32_80}; //HREF
21: {check,dout} = {1'b1,16'h19_03}; //VSTART
22: {check,dout} = {1'b1,16'h1A_7B}; //VSTOP
23: {check,dout} = {1'b1,16'h03_0A}; //VREF
24: {check,dout} = {1'b1,16'h0F_41}; //COM6
```

Fig 9. Small part of values specified for camera configuration, included in *init_regs* module

The pixel data driver (ov7670_read) is a separate module that reads pixel data from the OV7670 camera in the specified data format, RGB565. The details of the whole process are explained in the OV7670 VGA camera datasheet [5]. The pixel data is extracted from the camera as a stream, synchronous to the pixel clock signal, and passed to the frame buffer. This FIFO buffer works as an interface between the camera interface unit and the image processing unit. Pixel data streams are recovered as image frames and temporarily stored in the buffer, waiting for the image processing unit to read.

The camera interface module together with the image processing unit are integrated in a top module (implemented as *ipu_top*). This top module is wrapped up with AXI-lite interface, so that it can be easily connected to the AXI interconnect bus and supplies the coordinate values to other modules in the system.

4.4 Image Processing Unit (IPU)

Our IP relies on a simple algorithm that is hardware friendly for identifying objects with some color and getting the coordinate values. The IPU has two modules performing gb2hsv based on algorithm as shown in figure X below and mask values of the interested colored object based on upper and lower bound of HSV values, i.e, set pixel values to high if in between the range and set to low if lies outside. In this way the IPU outputs HSV values and we further devise a simple algorithm to detect object coordinates which is further passed to AXI BRAM that pong can access running on microblaze to move.

$$c = v * s$$
$$m = v - c$$
$$x = c * \left(1 - \left|\frac{h}{60} \bmod 2 - 1\right|\right)$$
$$(r, g, b) = \begin{cases} (c+m, x+m, m) & h \in [0, 60) \\ (x+m, c+m, m) & h \in [60, 120) \\ (m, c+m, x+m) & h \in [120, 180) \\ (m, x+m, c+m) & h \in [180, 240) \\ (x+m, m, c+m) & h \in [240, 300) \\ (c+m, m, x+m) & h \in [300, 360) \\ (m, m, m) & \text{otherwise} \end{cases}$$

Fig 10. RGB2HSV conversion where v, s & h are hue saturation and value respectively

Once masked output is generated, the coordinates of the Player Pad are calculated using the following algorithm:

1. Create a buffer of size COL_NUM, where COL_NUM is the number columns in a frame:

   reg column_det [COL_NUM-1:0];

   The idea is to detect all columns which contain the color we are trying to detect (let this color be *mcolor*). If a column contains *mcolor* detect then at least 1 pixel in the column will have a non-zero pixel value. On the other hand if the color is not present in that column, all pixels will have a zero value. If we detect a column that has pixel(s) having a non-zero value, we set the corresponding bit in column_det.
2. Once all columns having non-zero pixels have been detected, find the first column having a non-zero value from the right by traversing the column_det array from the right. Let this position be *last*.
3. Perform the same traversal from the left to detect the first column that has a non-zero value. Let this position be *first*.
4. The current position of the pad is given by *first* - *last.*

The intuition behind the algorithm is that, provided the IPU works perfectly, only *mcolor* (selected by the *Hue, Saturation* and *Value)* will be detected by the camera and only the pixel corresponding to *mcolor* will have a non-zero value. If there is only one object having the *mcolor* in the frame, then the mid-point of the detected object can be calculated using the above-mentioned algorithm. Only the midpoint along a single axis is detected since the pad in the game has a single degree of freedom.

The Image processing unit is developed in 3 phases.

1. Calculate mask values for blue (randomly chosen) color (since we need to move pong, we need to correctly detect blue color and move pong along coordinates of the detected blue color).
2. Test the IP which includes RGB2HSV + Mask on 2D Images.
3. Integration of camera and streaming through our IP.

In the first phase, we use the OpenCV library to convert rgb to HSV and identify mask values for blue color. The HSV threshold turned out to be

Table 6: Calibrated HSV for blue color

Color channel space	High	Low
Hue	157	110
Saturation	255	47
Value	255	111

Point to note here is, these values are Unsigned 8 bits, we have to scale down to 5-6-5 RGB format since the camera OV7670 PMOD uses 16 bits total configuration for RGB.

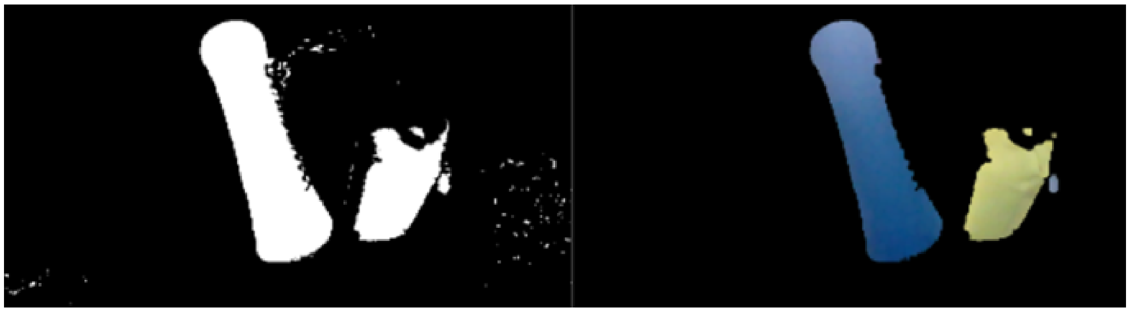


Fig. 11. Mask output sample image from openCV.

In the first Second we test the IPU (RGB to HSV and masking on a 2d image). The images are read through UART_Lite from a PC x86 host and stored in DDR3 running at 200MHz clock through the AXI4-lite interface. Below figure X shows an overview of our IP testing.

We stream the images stored in DDR3 through MM2S DMA port into our designed IP, process it and send it back through S2MM port of DMA to DDR and read images to PC.

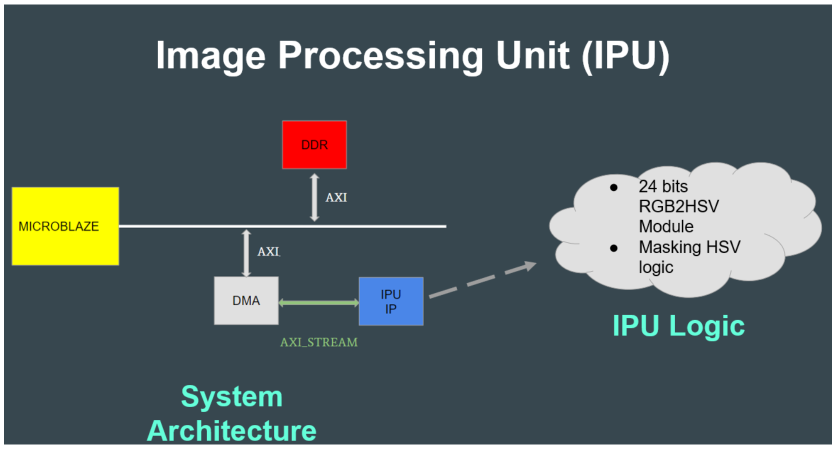


Fig. 12. IPU system diagram.

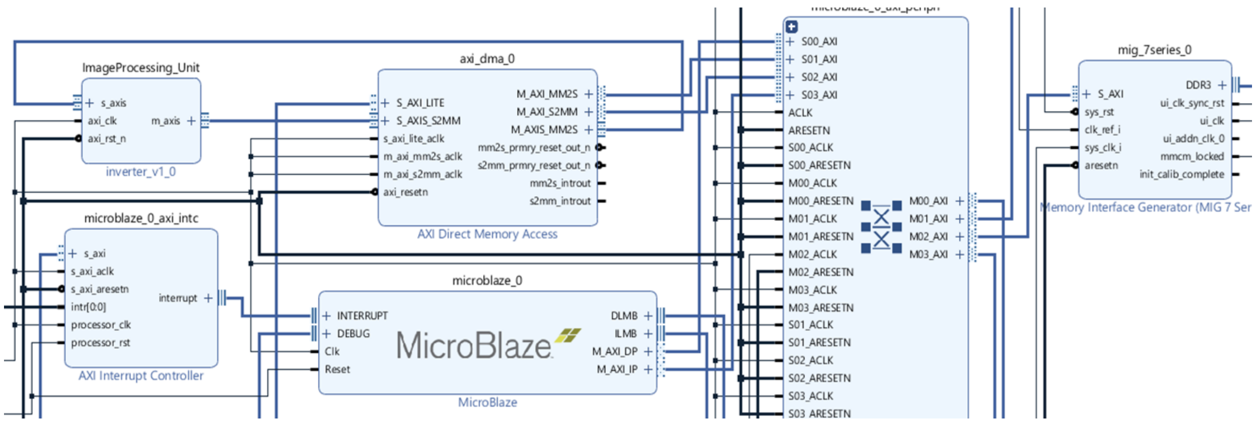


Fig. 13. Image Processing Unit, MIG and DMA integration block diagram.

Fig. 9 shows the overall block diagram for testing IP with MIG & DMA. Unfortunately, the above testing worked for us in the 2018 version of Vivado, but did not work on the 2016 version (our base version of the project that uses HDMI+Pong), due to MIG UI component failure error, which was later found out that it is the version error of DDR3 in 2016 Vivado. Our aim is to continue the project in the 2016 version since all the other components of the project,especially HDMI, are incompatible with 2018.

The third phase of the IPU consists of integrating image frames coming from the camera module and our Image IPU. Similar to the earlier Vivado version challenges we faced, the IPU could not run on the 2016 version of Vivado with a camera interface. We decided to implement camera + Image processing IPU within a single AXI4-Lite Slave and generate the custom Image processing IP developed above using Vivado HLS. The block diagram for the modified implementation of IPU is shown in figure 10.

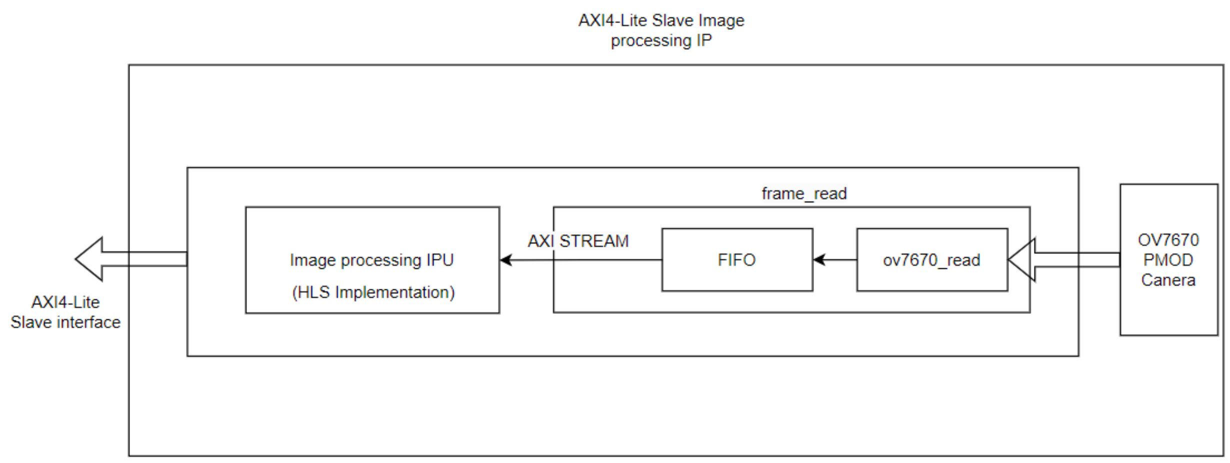


Fig. 14. Modified Custom IP developed in HLS and integration of camera modules within a single AXI4-Lite slave interface.

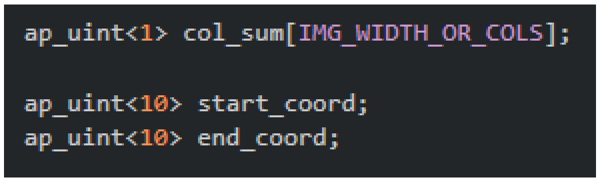
The output from AXI slave IP is the 9 bits coordinate value (only Y-axis, since our video out size is 640x480, hence we take log2(640) ~9 bits). The bits are stored in BRam for microblaze running pong.

HLS Implementation

The IPU has the following interface:

- AXI-Stream Input: Input frame, received from the camera.
- AXI-Stream output: IPU output frame, stored in RAM.
- *coord* output: Coordinates of the detected object.

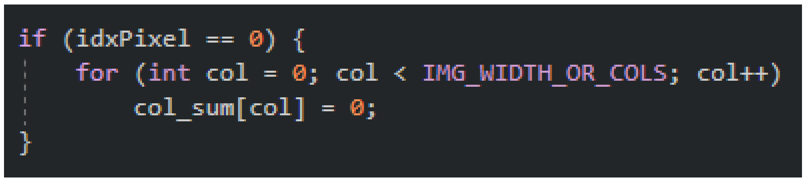
The IPU contains a buffer (let this be column_det) and 2 registers to implement the coordinate detection algorithm discussed previously and as seen in Fig. 15. At the beginning of every frame the column_det is reset to 0 as seen in Fig. 16.



```
ap_uint<1> col_sum[IMG_WIDTH_OR_COLS];

ap_uint<10> start_coord;
ap_uint<10> end_coord;
```

Fig. 15. Registers for implementing coordinate detection algorithm

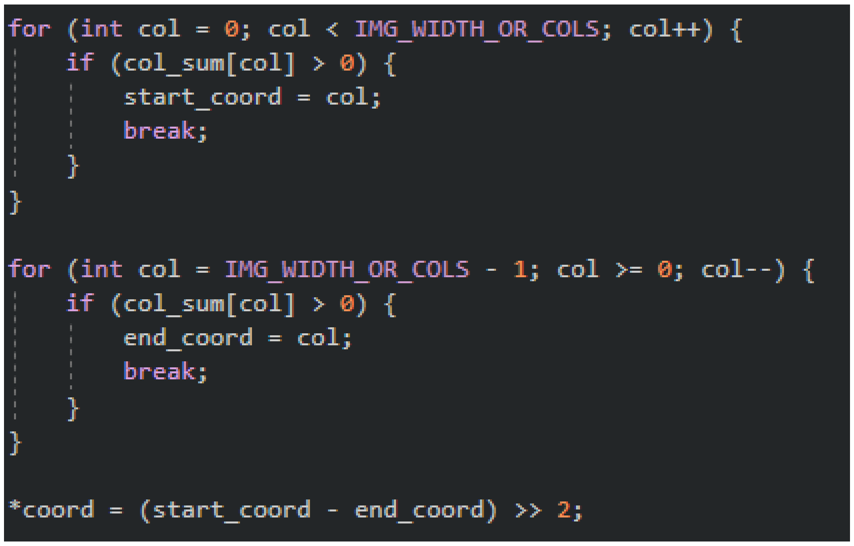


```
if (idxPixel == 0) {
    for (int col = 0; col < IMG_WIDTH_OR_COLS; col++)
        col_sum[col] = 0;
}
```

Fig 16. Reset for column_det array

The IPU works by processing pixels one at a time as they are received through the AXI-Stream input interface. The camera transmits pixel data in 5-6-5 RGB format (2 bytes of data for 1 pixel), hence the IPU first unpacks the color values from pixel data and scales them up to 8 bits. This is done because the HSV mask algorithm expects the color values to be from 0 to 255.

The IPU then implements the HSV mask based on the algorithm discussed previously and converts RGB data to HSV. If the input pixel is of the color to be detected, at the output of the IPU it is assigned a value of RGB = (255, 255, 255), that is, it will appear white at the output. If a pixel is non-zero at the output of the IPU, then the column corresponding to that pixel is recorded to implement the coordinate detection algorithm.



```
for (int col = 0; col < IMG_WIDTH_OR_COLS; col++) {
    if (col_sum[col] > 0) {
        start_coord = col;
        break;
    }
}

for (int col = IMG_WIDTH_OR_COLS - 1; col >= 0; col--) {
    if (col_sum[col] > 0) {
        end_coord = col;
        break;
    }
}

*coord = (start_coord - end_coord) >> 2;
```

Fig.17 Coordinate detection algorithm in HLS

4.5 HDMI Output

The block diagram for the HDMI system is shown in Fig. 11. The Video DMA block is responsible for reading from DRAM buffers and streaming RGB data to the AXI-Stream to Video Out blocks. The AXI-Stream to Video Out bridge IP block converts the AXI stream formatted DRAM data into a custom format that is acceptable by the **RGB2DVI** encoded IP block. The bridge is used to convert AXI4-Stream input to native video by synchronizing to the video timing generator input signals. The inputs to the **RGB2DVI** block consist of a 24-bit RGB signal bus, and video synchronization and timing signals. The outputs of this block consist of TMDS clock and data signals.

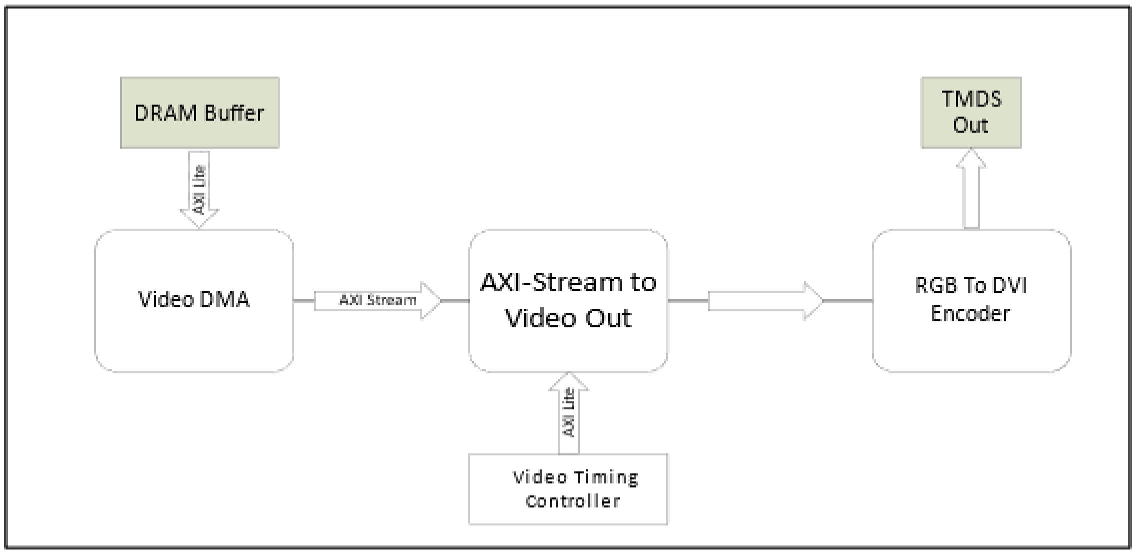


Fig. 18. Block Diagram of the HDMI Display system.

5. Tips and Tricks

Static images can be stored as integer arrays by using a tool developed by Notisrac to convert images into C arrays [3]. This is a quick method to store static images for processing and display purposes. Additionally, it is possible to directly link binary objects by using the MicroBlaze linker tool in Little Endian mode.This technique is useful for directly accessing large binary objects into the executable's data section without the need for converting the file into C-style arrays as explained by CodePlea [4].

6. Acknowledgements

We would like to thank Professor Jason Anderson for his contribution in course content development, our TA Osama Zeeshan Ahmad, and the course staff's constructive feedback on our project proposals, presentation and implementation.

Hackster.io Community - For the project CMOS Sensor camera system from which we took inspiration for our image processing unit desing [2] .

Digilent Reference Tutorials - These tutorials allowed us to adapt the pre-existing MicroBlaze/HDMI systems to suit our needs [8].

7. Description of Design Tree

7.1 URL

https://github.com/aditya-167/SuperPong-FPGA

7.2 Brief Overview

Software stored in the sw directory. Hardware stored in the hw directory. Image Processing project stored in the img directory.

7.3 Structure

- main/root folder
  - **assets/**: Images and video demos
  - **docs/**: Project reports and presentations.
  - **game_source_pc/main.c:** Early stage PC version of Pong game logic
  - **hw/:** HDMI system hardware project.
  - **img/:** Image Processing Project
    - **hls/:** - HLS C++ codes for image processing and coordinate detection.
    - **ip_repo/myip_1.0/src/**: Custom Image Processing IPU developed.
  - **src/:** IPU project source code
  - **repo/:** Directory used by Vivado for caching and databasing.
  - **src/:** Main HDMI system and game source code.
  - **.gitignore/:** Contains various ignored files to stop this repository from being populated with large synthesized caches.
  - **sw/:** hardware platform file for sdk

8. References

1. Joystick Tutorial: https://digilent.com/reference/learn/programmable-logic/tutorials/pmod-ips/start?redirect=1
2. IPU Tutorial: https://www.hackster.io/cc-ad/cmos-sensor-camera-system-9f74f8
3. Notisrac's Image to C Array Converter: https://notisrac.github.io/FileToCArray
4. How to Embed an Arbitrary File in a C Program: https://codeplea.com/embedding-files-in-c-programs
5. OV7670 VGA Camera Data Sheet: http://web.mit.edu/6.111/www/f2016/tools/OV7670_2006.pdf
6. Reference Modules for Camera Interface, CMOS Sensor Camera System: https://www.hackster.io/cc-ad/cmos-sensor-camera-system-9f74f8
7. Reference of OV7670 Camera Configurations from Github: https://github.com/jonlwowski012/OV7670_NEXYS4_Verilog/blob/master/ov7670_registers_verilog.v
8. Nexys Video HDMI Demo: https://digilent.com/reference/learn/programmable-logic/tutorials/nexys-video-hdmi-demo/start