ECE532 Final Report

Air Hockey Computer Opponent

Submission Date: April 13th, 2017

Professor Paul Chow [Nariman @ 2:30pm]

Group #3:

Richard Lin	richard.lin@mail.utoronto.ca
Emil Karimov	emil.karimov@mail.utoronto.ca
Jashva Rafique	j.rafique@mail.utoronto.ca
Prabhnoor Kainth	prabhnoor.kainth@mail.utoronto.ca

Project Files Available:
https://github.com/emilkarimov/G3_airhockey

Video Available: https://youtu.be/aC1zL6u4oJQ

Table of Contents

Overview

Motivation & Goals

The motivation for this project is to create a system that showcases our knowledge from our time at the University of Toronto. This project incorporates digital design, programming, electric motors, and power supply design, as well as building a physical air hockey table.

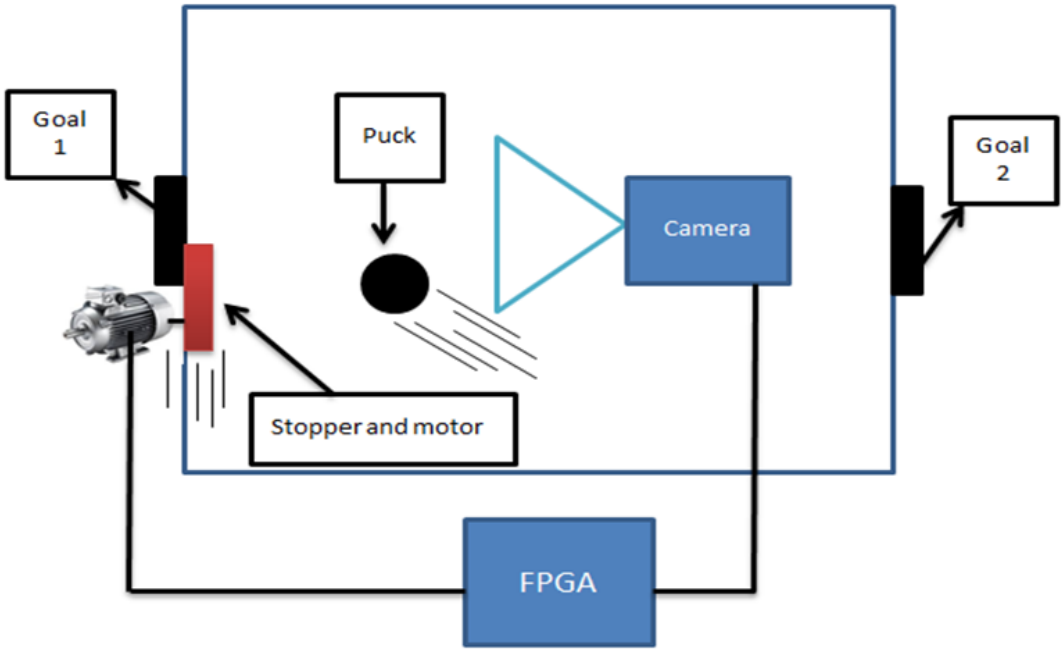

Figure 1: Initial System Level Concept Diagram

The goal of this project is to design an automated air hockey opponent. The system uses the Nexys Video board, a camera, two motors, and a PWM encoder. The FPGA processes an overhead video stream to find the puck and estimate its trajectory; a goalie piece is actuated by a motor system to block the human player's shot. For players of different skill levels, the operator can adjust the difficulty through a software interface, which affects the puck detection and tracking algorithm speed. Additionally, the system can also be controlled by a second human, using a PMOD joystick. In the background, the FPGA loops a user-inputted audio track and displays the filtered video on an HDMI monitor.

Block Diagrams

Figure 2 illustrates the main blocks in the design. Figure 3 provides a highly detailed block diagram from Vivado and is available in higher resolution here: https://github.com/emilkarimov/G3_airhockey/blob/master/docs/hdmi.pdf.

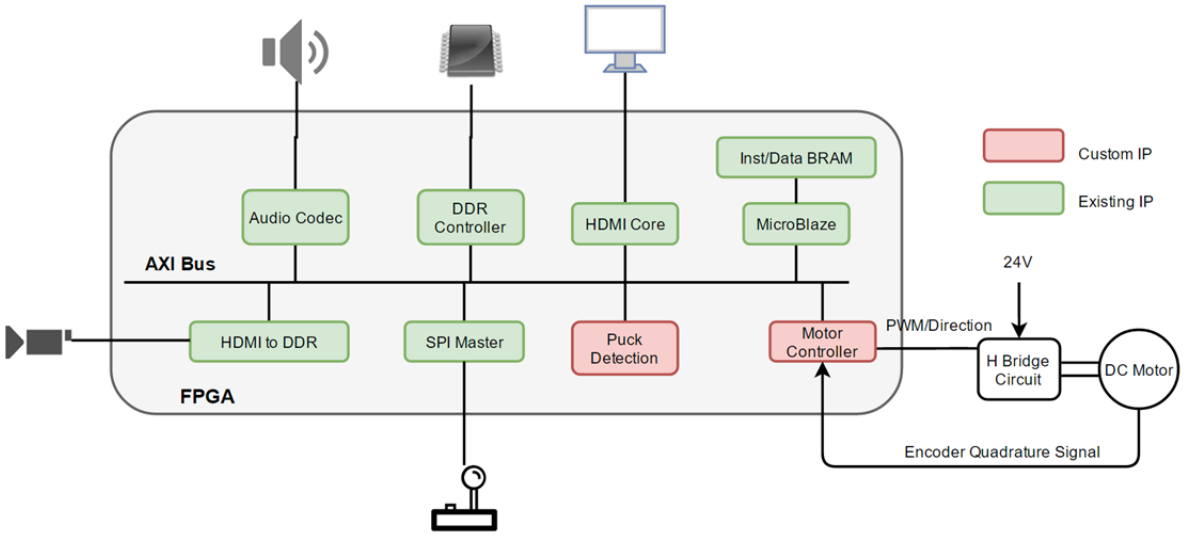

Figure 2: Simplified Block Diagram

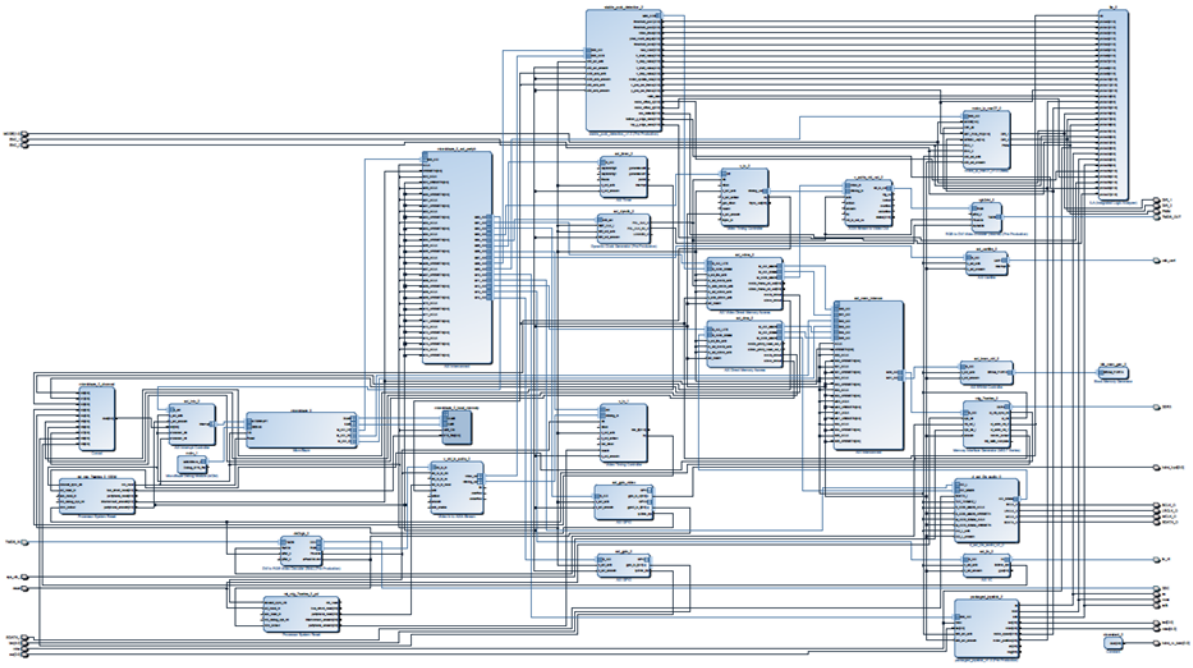

Figure 3: Block Diagram Generated in Vivado

Brief Description of IP

For more detailed descriptions, as well as supporting images for the following projects and IP cores, please refer to the section, “Description of Blocks”.

HDMI Project (Digilent)

The HDMI demo project from Digilent was the starting point for this project for two reasons: First, this demo provided the resolution, frame rate, and video functionality that was required for inputting and outputting video streams with the Nexys Video boards. This project converted a TMDS input video stream from the camera to RGB AXI-4 Stream protocol for the Video Direct Memory Access (VDMA) to store in the DDR memory. Outputting video uses the opposite process, retrieving video frames through the VDMA and converting from AXI-4 Stream to TMDS DVI for the display output.

Secondly, this project also included a MicroBlaze, DDR memory, and SDK code that would not need to be heavily modified for the final demonstration. From the project proposal, the team knew that most of the project would be developing the custom IP blocks for the motor and image processing. The reference project is available here: https://reference.digilentinc.com/learn/programmable-logic/tutorials/nexys-video-hdmi-demo/start

Motor Actuation Block (Custom)

The motor actuation block was written as a custom AXI-Lite IP block to control the two motors and PWM encoder. This block connects to the air hockey table’s electronics through a custom connector with JB PMOD header (outputs PWM, DIR_1, and DIR_2, inputs ENC_1 and ENC_2). With these connections, the Nexys Video board can read the encoder status through ENC_1 and ENC_2, as well as direct the motors. The IP block sends a position between 0-2000 and a speed between 0-255, encoding this information on the PWM, DIR_1, and DIR_2 outputs.

The motor actuation block receives data from the joystick IP block and the puck detection block. The puck detection block provides a continuously updating 32-bit position input (SET_POS_PD[31:0]) between 0 and 2000, for where the motor should go in order to block the puck. The joystick IP block provides an 8-bit speed value (SPEED_JS[7:0]) between 0 and 255, as well as a 1-bit direction (DIR_JS) for left or right. Additionally, the IP block has a two bit MODE input (connected to SW5, SW6), which controls which data is sent to the motor. When set to zero, the motor block connects the puck detection position directly through hardware. When set to one or two, the motor block is connected in hardware to the joystick direction and speed, or the AXI registers respectively.

Joystick Demo (Instructables)
The Pmod JSTK2: Two-axis Joystick demo from Instructables was used as a starting point for this IP core. The demo was originally for the BASYS 3 board, outputting debug information to the HEX displays and the LEDs. This core was migrated to the Nexys 4 DDR board, using the HEX displays to show the position (0-1024) outputted by the block, as well as using the LEDs to show a directional running counter. The LED counter was used in the final design (led[2:0], mled[4:0]), but there is no HEX display (an[3:0], seg[6:0]) on the Nexys Video board. The design also uses three switches to change debug modes (sw[2:0]).

This project was packaged into an AXI-Lite peripheral, using SPI (miso, mosi, ss, sclk) to communicate with the JSTK2, while the motor actuation block receives motor_position and motor_speed[31:0] (only lower 8 bits are used). The original demo is available here: http://www.instructables.com/id/How-to-Use-the-PmodJSTK-With-the-Basys3-FPGA/?ALLSTEPS

Puck Detection & Trajectory Block (Custom)
This custom AXI4-Stream IP block filters 24-bit RGB into a 1-bit B/W image and finds the x-y position of the puck within the video frame. The block was inserted into the HDMI project on the input stream, after the *Video In to AXI4-Stream* block, but before the VDMA. The video data enters the IP block through the S00_AXIS port and through software configured register thresholds, the image is filtered to become black and white. This image is then outputted on the M00_AXIS port to the VDMA, DDR memory, and eventually the HDMI display output. This block finds the x-y location of the black puck sixty times a second and outputs this on a 0-2000 scale for the motor actuation IP block (y_pos_per_frame[31:0]). The other outputs of this block are routed to an integrated logic analyzer (ILA) for debugging purposes since this design supports software customization without re-synthesis. The ILA trigger is a 1-bit output valid_data, which pulses when the x-y position of the puck has been found. The other signals are 32-bit and provide debugging information like the frame borders, the number of pixels to count, the number of video lines counted, and many more to ensure the block's functionality.

DMA Audio Demo Project (Digilent)
The DMA Audio Demo Project from Digilent was used to loop background music for the air hockey table. This project uses push button controls to record and loop music from the audio in port to the speaker out port. The main components of this demo project are an AXI GPIO block for the push buttons, an AXI DMA to store the loop, an IIC audio codec for input and output interfacing, and block RAM (the original project used DDR, which conflicts with the HDMI project). This demo also uses a significant amount of

MicroBlaze resources due to the software to continually probing and starting the audio playback. The original demo project is available here: https://reference.digilentinc.com/learn/programmable-logic/tutorials/nexys-video-dma-audio-demo/start

Outcome

Results

For this project, the team was successful in meeting the project proposal goals of creating an automated air hockey opponent. Additionally, the team was able to provide an entertaining and rewarding final demonstration. Figure 4 below provides images for the final design, I/O, and utilization. A video demonstration is available here: https://youtu.be/aC1zL6u4oJQ

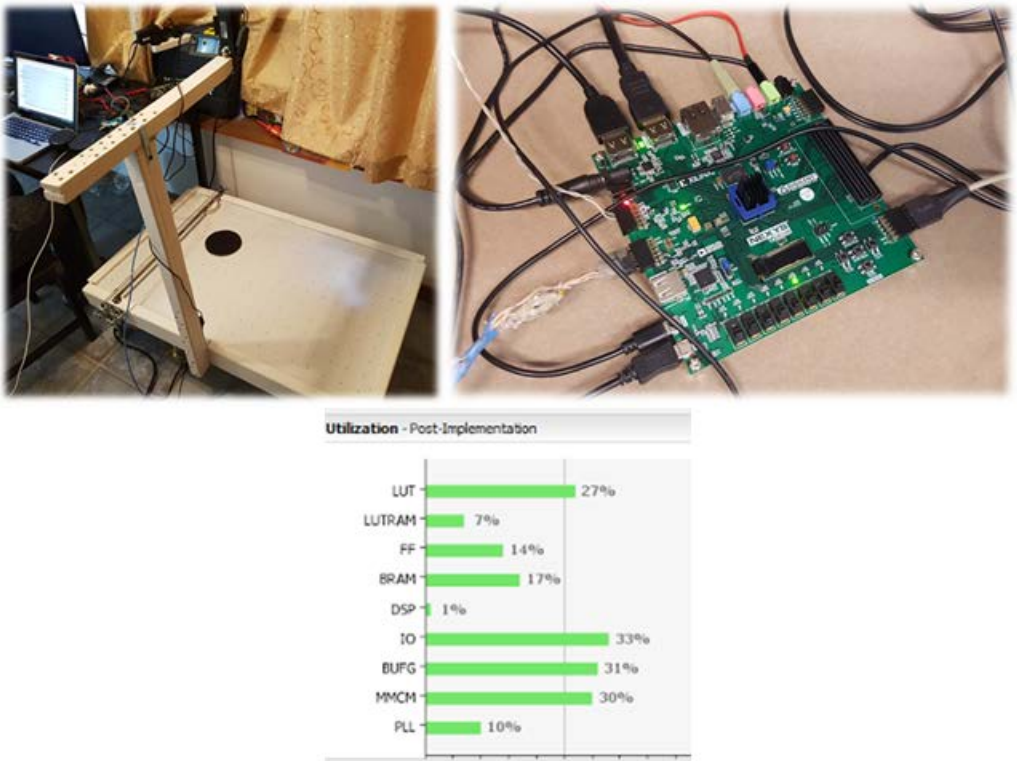

Figure 4: Final Design for the air hockey table, Nexys Video I/O, and the Utilization

Potential Improvements

Throughout the project, the team discussed many improvements to the system, mostly relating to the goalie control and attack capabilities. One major improvement could be the software algorithms used to control the goalie. The current system implements a tracking algorithm, with the ability to support multiple algorithms. This algorithm detects

the puck and moves the goalie to block, even if the puck is still far away. Another algorithm that was started but not finished would calculate where the puck was heading and would ram into it to increase momentum rather than track and block. Additional software profiles such as attack and defend could be developed for this system.

Another potential improvement for this project would be adding more capabilities to the goalie. One idea was to use a two-dimensional rail system to allow the goalie to travel in x and y directions. This would result in greater range of motion and strategies for the automated goalie and the joystick-controlled human as well. The other capability discussed was to add an inductive bumper on the goalie, rather than an elastic bumper. The current system has springs to bounce the puck back to the human player, but a potential improvement could be adding inductive coils to allow the FPGA to trigger a reactive force, actively pushing the puck away.

Project Schedule

The following section compares the initial proposed schedule with the milestones achieved by the end of the project. The proposed milestones are shown in black, while the final project schedule is displayed below in blue on a week-to-week basis.

Milestone 1:

- Show working frame storage block and start working on the object detection block.
- Have the game surface and goalie assembly built.
- Start implementing camera functionality.
- HDMI demo project ported from an older version of Vivado. The project was running properly on the Nexys Video board with an HDMI camera input and HDMI monitor output. Begin understanding the HDMI project components to get a better idea of which blocks must be modified (specifically the VDMA, DDR memory, and the SDK code).
- Begin work on building the air system, set up the motor and encoder.
- Begin work on the image filtering and processing IP block

Milestone 2: Testbench Demo.

- Demonstrate testbench to test frame flow from camera input over HDMI interface stored in DDR and processed by the object detection block.
- Show basic trajectory prediction block functionality.
- Start working on the motor control block.
- Completed the table and goalie, needed to add a second motor to increase speed and power of the goalie.
- Begin work on the motor control block, using an Altera DE2 board.

- Completed the image filtering and processing IP block. This block was connected to the integrated logic analyzer (ILA) for the demo. This was useful in seeing if the block was working correctly before integrating into the main HDMI project.

Milestone 3:

- Fine tune ODB and TPB.
- Implement display output block to project information about trajectory on an HDMI display.
- Demonstrate motor control block with a corresponding testbench.
- Upgrades to the table (painting it white, sanding surfaces, and adding heatsinks and fans to the motor controller).
- Upgrades to the puck detection and trajectory block (making the design color pixels differently after x pixels of the puck have been found, etc.)
- Work on Digilent audio looper demos to produce an IP block for the main project. The Nexys 4 DDR demo was not used in the end because the DDR board takes audio input through a PMOD rather than through the audio codec. The Nexys Video demo was not used in the end because this project used too much DDR and could not be easily integrated in the HDMI project with BRAM. For more details about audio, see Milestone 6.

Milestone 4:

- Integrate all blocks built so far together and prepare for Mid-Project Demo.
- Time for bug fixes.
- Begin integrating the HDMI project, puck detection block, and the motor controller block. The puck detection block was added after the MicroBlaze on the video output side. From this choice, the design was not working correctly. This required more debugging and would be solved in Milestone 6. The motor controller however, was integrated without issues and could be issued position/speed commands.
- Begin converting the joystick demo into an IP block to control the motors.

Milestone 5: Mid-Project Demo.

- Demonstrate working FSB, ODB, and TPB together with the motor control block.
- Have data representing trajectory and expected goalie signals on external display.
- Completed joystick IP block and integrated into the main HDMI project. For the Mid-Project Demo, the motor block was connected to the joystick IP block as the control source.
- The remaining pieces are to work on the audio playback, debug the puck detection block, and work on the software algorithms for motor control.

Milestone 6:

- Lots of ECE496 Work - Not much progress.
- Finely tune system level integration.
- Start working on alternative control mode using joystick.
- Implement audio block.
- Moved the puck detection block to the input stream rather than the output stream, the block is now fast enough to operate and on the correct clock domain. The design was still connected to an ILA for quick debugging and tweaks.
- Begin work on the audio playback feature. Unfortunately, the previous looper projects could not be used for the reasons found in Milestone 3. The final design for audio was from the direct memory access (DMA) demo from Digilent. Integrating this project was difficult since both projects have MicroBlazes, two interconnects, and DDR memory. This demo was edited to remove these conflicts (the second MicroBlaze, two additional interconnects, and the DDR memory). This shared the HDMI demo's MicroBlaze and interconnects, but used BRAM to store the audio data.

Milestone 7: Final Demo.

- Done! It all works. All parts have been put together and are working seamlessly. There is background music and alternative joystick mode.
- Complete audio playback feature integration with the HDMI demo.
- Begin work on second strategy in SDK (goalie attacking the puck, rather than just tracking the puck).
- Done! For details on all features implemented, please see the *Results* section.

During this design project, the integration of blocks and audio playback took longer than projected in the initial proposed milestones. From Milestones 4 & 6, the camera control over the goalie was not implemented until much later than initially planned. One reason was the team focused on controlling the motor with the joystick first to see if the motors were responsive and functioning. Another reason was the placement of the puck detection block was critical to the system timing and would only work on the input stream. As mentioned in Milestones 3 & 6, two of the three audio demos could not be integrated into the HDMI project without significant changes to either project. The looper audio demos from Digilent required too much DDR memory and MicroBlaze resources, which was in use for the HDMI demo from Digilent. Additionally, the air hockey table and the Nexys Video board could only be at one person's house, making testing and tweaking difficult towards the end of the project.

Description of Blocks

The following section provides a more detailed description of each IP block from the design. The sub-section headers are broken into the demo projects and custom IP cores used in the final design.

HDMI Demo Project (Digilent)

Design Hierarchy & Description of Functionality

The main purpose of the HDMI demo is to take an HDMI camera input, apply effects, and output to an HDMI display. The effects applied were changes to resolution, inverting colors, and applying test patterns. To achieve this, the demo project used a MicroBlaze with c code, DDR memory (MIG 7 Series), and several other IP blocks from Xilinx. This demo served as the starting point for the project due to its features and IP blocks used. Figure 5 below provides the starting block diagram:

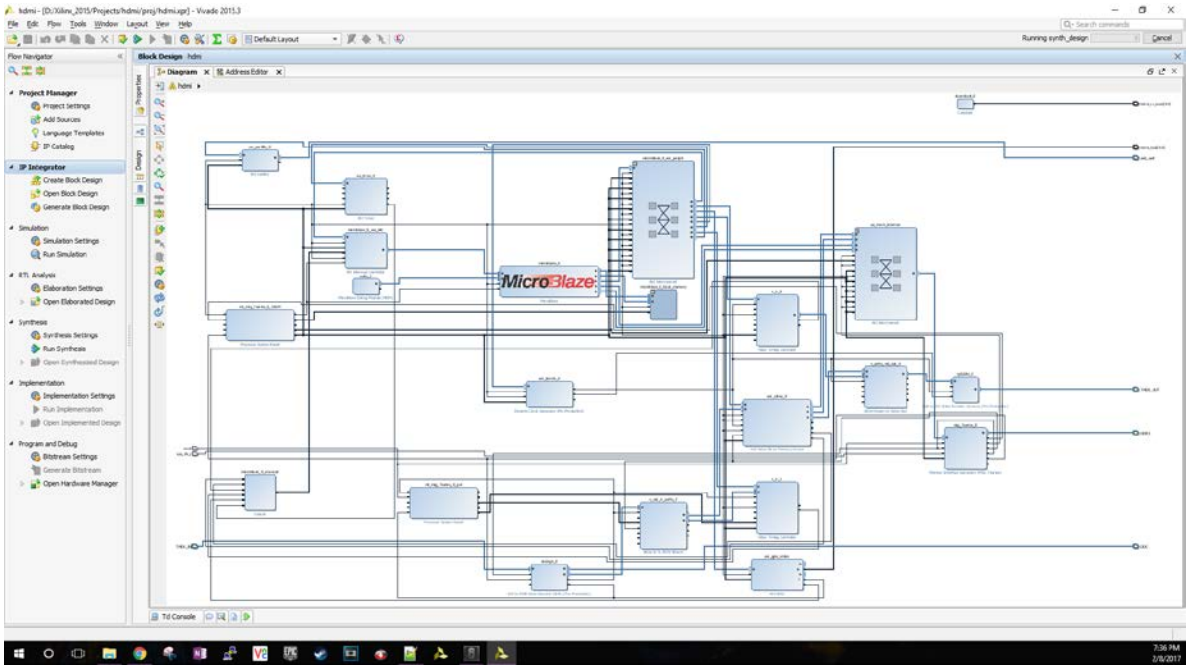

Figure 5: HDMI Demo Project Block Diagram

The top level file for the HDMI demo project is hdmi.vhdl, which instantiates the following via the Vivado block diagram generator:

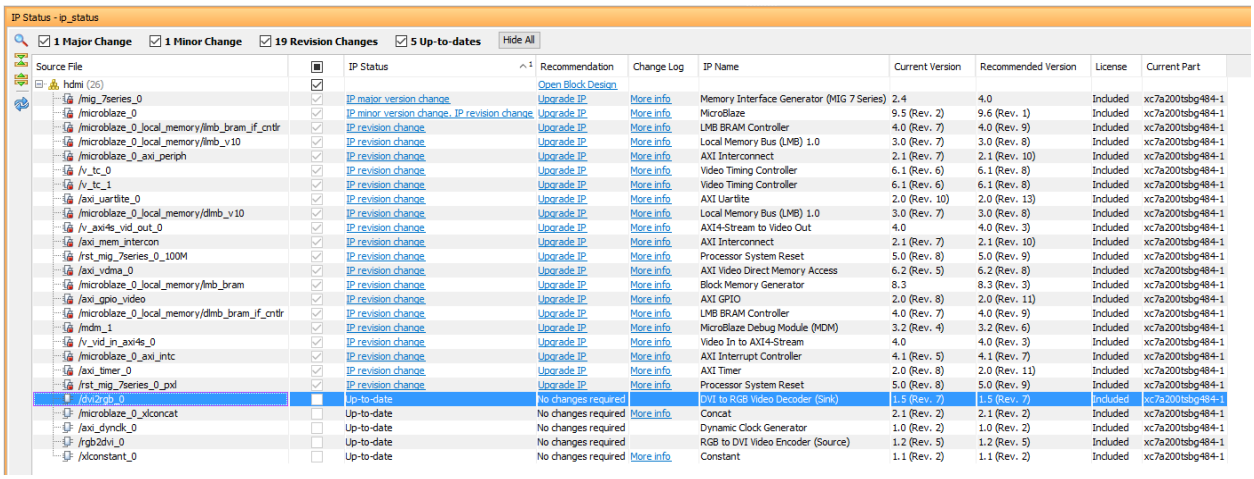

Figure 6: HDMI Demo Project IP Blocks and Versions Used.

For a more detailed description of each IP block used in the Digilent HDMI Demo Project, the following table has been provided (all IP are sourced from Xilinx/Digilent):

IP Block	Version	Description & Reference Material
MIG 7 Series	4.0	Memory Interface Generator for 7 Series Devices for interfacing with DDR3 modules. Available from Xilinx: https://www.xilinx.com/support/documentation/ip_documentation/mig_7series/v4_0/ug586_7Series_MIS.pdf
MicroBlaze	9.6 r1	Xilinx MicroBlaze Soft Processor Reference Guide: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_1/ug984-vivado-MicroBlaze-ref.pdf
AXI Interconnect	2.1 r10	Xilinx AXI Interconnect v2.1 Product Guide: https://www.xilinx.com/support/documentation/ip_documentation/axi_interconnect/v2_1/pg059-axi-interconnect.pdf
Video Timing Controller	6.1 r8	Xilinx Video Timing Controller v6.1 Product Guide: https://www.xilinx.com/support/documentation/ip_documentation/v_tc/v6_1/pg016_v_tc.pdf
AXI Uartlite	2.0 r13	Xilinx AXI UART Lite v2.0 Product Guide: https://www.xilinx.com/support/documentation/ip_documentation/axi_uartlite/v2_0/pg142-axi-uartlite.pdf
AXI4-Stream to Video Out	4.0 r3	Xilinx AXI4-Stream to Video Out v4.0 Product Guide: https://www.xilinx.com/support/documentation/ip_documentation/v_axi4s_vid_out/v4_0/pg044_v_axis_vid_out.pdf

AXI GPIO	2.0 r11	Xilinx AXI GPIO v2.0 Product Guide: https://www.xilinx.com/support/documentation/ip_documentation/axi_gpio/v2_0/pg144-axi-gpio.pdf
Video In to AXI4-Stream	4.0 r3	Xilinx Video In to AXI4-Stream v4.0 Product Guide: https://www.xilinx.com/support/documentation/ip_documentation/v_vid_in_axi4s/v4_0/pg043_v_vid_in_axi4s.pdf
AXI Interrupt Controller	4.1 r7	Xilinx AXI Interrupt Controller v4.1 Product Guide: https://www.xilinx.com/support/documentation/ip_documentation/axi_intc/v4_1/pg099-axi-intc.pdf
AXI Timer	2.0 r11	Xilinx AXI Timer v2.0 Product Guide: https://www.xilinx.com/support/documentation/ip_documentation/axi_timer/v2_0/pg079-axi-timer.pdf
AXI VDMA	6.2 r8	Xilinx AXI VDMA v6.2 Product Guide: https://www.xilinx.com/support/documentation/ip_documentation/axi_vdma/v6_2/pg020_axi_vdma.pdf
dvi2rgb	1.5 r7	Digilent dvi2rgb IP Block Reference Manual: https://github.com/Xilinx/PYNQ/blob/master/Pynq-Z1/vivado/ip/dvi2rgb_v1_6/docs/dvi2rgb.pdf
axi_dynclk	1.0 r2	The Digilent AXI Dynamic Clock IP Block does not have a reference manual or product guide. This IP Block is used to create a dynamic pixel clock based on the output video resolution. The top level file for this IP is available: https://github.com/Digilent/ZYBO/blob/master/Projects/hdmi_in/repo/digilent/ip/axi_dynclk_v1_0/src/axi_dynclk_S00_AXI.vhd
rgb2dvi	1.2 r5	Digilent rgb2dvi IP Block Reference Manual: https://github.com/Xilinx/PYNQ/blob/master/Pynq-Z1/vivado/ip/rgb2dvi_v1_2/docs/rgb2dvi_v1_2.pdf
constant	1.1 r2	Xilinx Constant v1.1 Product Guide: https://www.xilinx.com/support/documentation/ip_documentation/xilinx_com_ip_xlconstant/v1_1/pb040-xilinx-com-ip-xlconstant.pdf

The input data path in this demo project stores video from the camera in DDR memory. The input video stream is digital visual interface (DVI) transition-minimized differential signaling (TMDS) from the HDMI camera. The HDMI demo project uses the *dvi2rgb* and *Video In to AXI4-Stream* IP blocks to convert to a 24-bit RGB AXI Stream data format.

This data is then sent to the *AXI Video Direct Memory Access* (VDMA) IP block, which stores in DDR memory via the MicroBlaze, AXI Interconnect, and MIG 7 Series.

The output data path follows the exact opposite procedure, beginning with the VDMA, MicroBlaze, AXI interconnect, and MIG 7 Series fetching video data from the DDR memory. The 24-bit RGB AXI Stream video data is then converted by the *AXI4-Stream to Video Out* and *rgb2dvi* IP blocks to DVI TMDS, which can be displayed on via HDMI.

This demo project uses the generated MicroBlaze and MIG 7 Series. These designs require supporting IP blocks, such as BRAM, interrupt controllers, AXI Timers, and the Processor System Reset. Beyond these generated blocks, the video stream supports a dynamic pixel clock, which changes with the video resolution. This creates four clock domains in the main project:

1. sys_clk_i: MicroBlaze, MIG 7 Series, and AXI Interconnect
2. Dynamic Clock (axi_dynclk): Generate the output pixel clock based on the output video stream resolution
3. Video Timing Controller 1: Generate the video timing required for *AXI4-Stream to Video Out* conversion.
4. Video Timing Controller 2: Generate the video timing required for *Video In to AXI4-Stream* conversion.

Additionally, the HDMI protocal has a few more output ports that must be driven. The *dvi2rgb* block also outputs the display data channel (DDC), which is copied from the HDMI camera to the monitor to determine the audio visual format. A constant block is also used to drive the HDMI RX/TX enable signal, enabling the display output. The final HDMI output signal is Hot Plug Detection (HPD), allowing HDMI monitor connections without restarting the system. The HDMI HPG is connected to an AXI GPIO IP block.

Through the AXI Uartlite, the user can send commands to change the output resolution, invert the colors, or set a test pattern as seen below in Figure 7.

Figure 7: HDMI Demo Project output (left) with commands (right)

I/O

Signal Name	Signal Type	Description
TMDS_IN	Inout Bus	HDMI Video Input Stream Pins
reset	Input	System reset
sys_clk_i	Clock	System Clock (100MHz)
TMDS_OUT	Inout Bus	HDMI Video Output Stream Pins
DDC	Inout Bus	HDMI Display Data Channel
hdmi_hpd	Output	HDMI Hot-Plug Detection
hdmi_rx_txen	Output	HDMI Receiver/Transmitter Enable
DDR3	Inout Bus	Interface for DDR3 Memory
usb_uart	Inout Bus	Interface for USB UART Protocol

Motor Actuation Block (Custom)

The motor actuation block controls the position of the goalie. It takes input from either MicroBlaze, joystick block or puck detection block directly and moves the goalie to the requested position.

Design Hierarchy

The file *motor_ip_mar27_v1_0.v* is wrapper for the motor ip block. It instantiates motor_ip_mar27_v1_0_S00_AXI.v. The *motor_ip_mar27_v1_0_S00_AXI.v* file implements AXI lite slave with read and write registers to control the main motor_controller which is implemented in *motor_controller.v*.

Files pwm.v, optical_quad_encoder.v and divider.v are used by motor_controller to implement the motor control and goalie feedback system.
Figure 8 below shows the motor ip hierarchy.

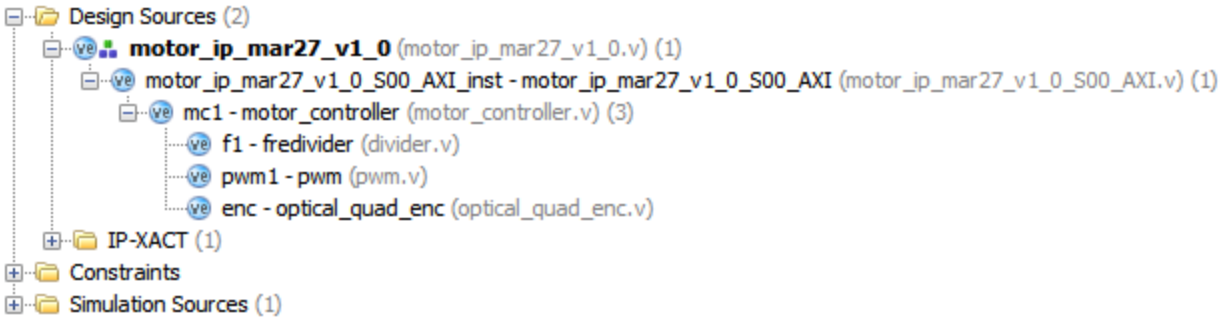

Figure 8: Motor Controller IP hierarchy

I/O & Slave Register Descriptions

The motor control block takes the goalie speed and desired position input from MicroBlaze, joystick block or the puck detection block. It also receives feedback from the goalie which it uses to keep track of goalie location. In addition several registers allow full control using MicroBlaze. Figure 9 and following table list all of the I/O and their description.

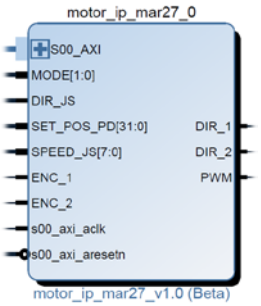

Figure 9: Motor Control block I/O

Signal name	Dir	Description
MODE[1:0]	Input	Selects whether joystick, MicroBlaze or puck detection block control the goalie. Connected to DIP switches.
DIR_JS	Input	Goalie direction select from the joystick block
SET_POS_PD[31:0]	input	Desired goalie position from puck detection block
SPEED_JS[7:0]	input	Desired goalie speed from the joystick block. This sets the motor PWM duty cycle.
ENC_1,	input	Motor feedback input from quadrature optical encoder.

ENC_2		Connected to PMOD header.
s00_axi_aclk	input	AXI lite interface 100 MHz clock
s00_axi_resetn	input	AXI lite interface active low reset signal
DIR_1, DIR_2	output	Controls direction in which the goalie moves. They are complementary to each other. These signals are connected to PMOD header.
PWM	output	PWM signal used to control the speed of the motor. This signal is connected to PMOD header.

The controller contains 6 read/write registers which provide control to MicroBlaze. The table below lists all of the registers and their functionalities.

Register	Type	Description
slv_reg0	read/write	Bit 0: **Enable**. When high, the controller sends the goalie to the set location. Bit 1: **Resetn**. When low, current position of the goalie is set as the 0 position. Must be high for normal operation.
slv_reg1	read/write	Bit 7:0: **Speed**. Speed sets et the duty cycle of the motor. 255 sets the duty cycle to 100%. 127 sets the duty cycle to 50%.
slv_reg2	read/write	Bit[31:0]: **Set_pos.** This sets the desired position of the goalie. If Enable signal is high, the controller moves the goalie to the set position.
slv_reg3	read/write	Bit[31:0]: **Tolerance.** This sets the tolerance of the final goalie position to the set position. This is used to avoid unwanted oscillations since the goalie can overshoot the set position.
slv_reg4	Read only	Bit[31:0]: **Curr_pos.** Current position of the goalie.
slv_reg5	Read only	Bit[31:0]: **Revision.** Used to keep track of changes made to the code.

Description of Functionality

The motor controller outputs DIR and PWM signals to motor driver circuit and receives motor movement feedback from ENC signals. Figure 10 below shows the encoder waveform. Optical_quad_encoder.v uses this information to count the number of steps the goalie has taken thus keeping track of its position at all times.

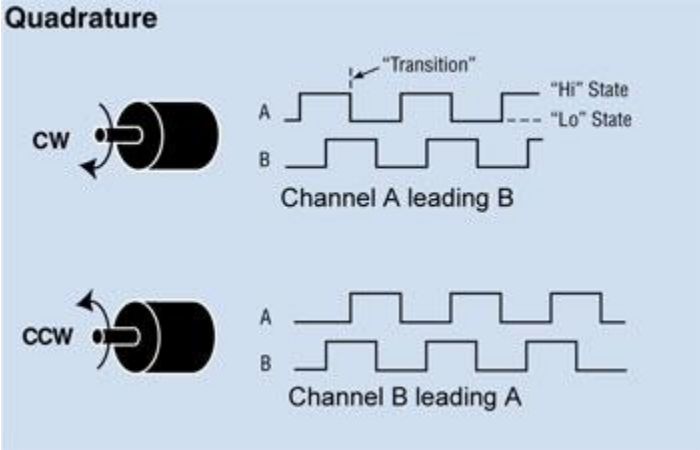

Figure 10: Encoder waveform received from quadrature optical encoder[]

For moving the motor to the set position, simple sequential logic is used to which compares the current and the set position and asserts DIR and PWM signals accordingly. If the set position is greater than current position, DIR_1 is set to 1 DIR_2 to 0 and vice versa. Once the current position is within the tolerance to the set position, PWM is set to zero which stops the motor.

Initially, we used an acceleration/deceleration profile to provide smooth motion of the goalie and prevent overshoot that caused goalie oscillation. During experimentation, we discovered that acceleration slowed down the goalie too much. We switched to using no acceleration (directly apply the set speed) and tolerance to avoid oscillations in case of overshoots. This is acceptable as the encoder provides a very fine resolution of ~0.5 mm per count which is much more than what is needed in our application. We used tolerance of 20 which equates to around $\pm$1cm of positional accuracy. This simplifies the design as well as improve the responsiveness of the goalie.

Since the set position and speed can be controlled from MicroBlaze, joystick or the puck detection block, appropriate muxes are added to switch between different sources using MODE[1:0] input.

The schematics of the motor driver along with PMOD header connections is shown below in Figure 11. The circuit is constructed on a protoboard and housed in an old computer power supply case.

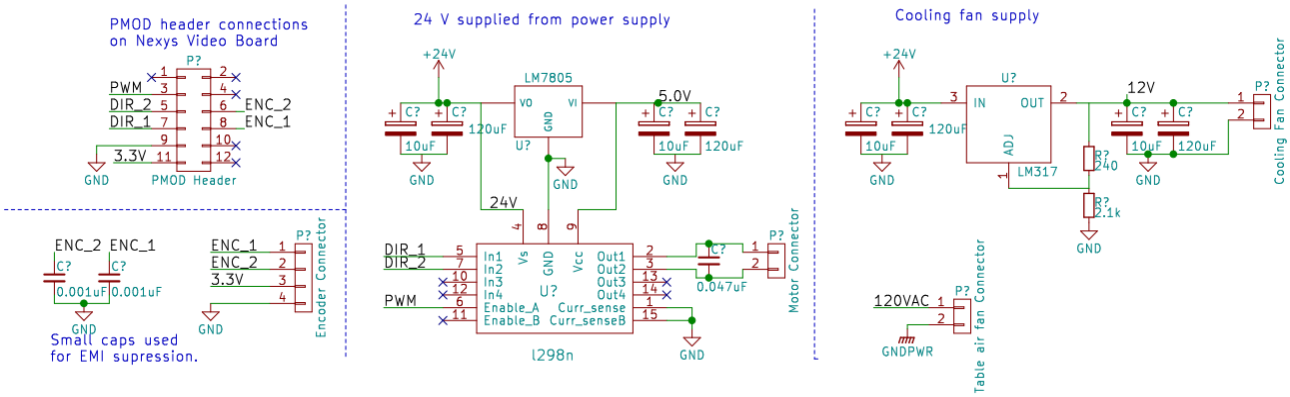

Figure 11. Motor driver and encoder schematics

Joystick Demo (Instructables)

The main purpose of the Joystick demo is to allow user control over the motor speed and position if two human players want to play against each other.

The final joystick packaged IP contains the following hdl source files:

Figure 12: Joystick Demo Design Source Hierarchy

Design Hierarchy

The original BASYS 3 project was not designed to be packaged, and so it did not include the *packaged_joystick_v1_0.v* or *packaged_joystick_v1_0_S00_AXI.v* verilog files. The *packaged_joystick_v1_0_S00_AXI.v* contains 4-slave registers that are not designed to impact the functionality of the joystick core. The idea is that future design changes will allow people the flexibility to extend this core to be programmed via software to serve other purposes.

The *spiCtrl.v*, *SPImode0.v*, *ClkDiv_66_67KHz.v, and ClkDiv_5Hz.v* files support the SPI FSM to ensure proper communication between the FPGA and the joystick peripheral.

The *ssdCtrl.v* and *Binary_To_BCD.v* files were included in the final joystick core, but they did not function because these verilog files required output to HEX display, which was was not a necessity for the project.

I/O

Signal name	**Signal type**	**Description**
sw[2:0]	input	Contains data to turn on joystick LEDs (not used in design)
miso[0:0]	input	Master-in-slave-out (SPI protocol)
mosi[0:0]	output to J4 PMOD pin	Master-out-slave-in (SPI protocol)

ss[0:0]	output to J4 PMOD pin	Chip select (SPI protocol)
sclk[0:0]	input/output to/from J4 PMOD pin	Serial clock (66.67 KHz) (SPI protocol)
led[2:0]	output to LED pins	Indicates joystick push button status (LEDs on Nexys Video Board turn on => button pressed)
mled[4:0]	output to LED pins	Indicates speed and direction of joystick control. (1-bit loaded into LED[2], moves left or right across 5 Nexys Video board LEDs to indicate direction). Position value of joystick changes LED movement speed)
motor_speed[31:0]	output to motor-ip-core	Outputs 0-255 speed value to custom motor-ip-core.
motor_position[0:0]	output to motor-ip-core	Outputs 0 or 1 to custom motor-ip-core to indicate direction that motor should move (left or right)
an[3:0]	Not used	Not used
seg[6:0]	Not used	Not used
S00_AXI	input/output to/from MicroBlaze	Facilitates the possibility to control joystick via software for future project enhancement.

Description of Functionality

Since data will not be sent back to the joystick peripheral, changes were made to only handle input data. Data from the joystick peripheral is sent in 23 bits within 40 clock cycles. The first 2 bytes of data consist of the x-position of the joystick, and the next 2 bytes consist of the y-position of the joystick. Both x and y positions are given in 10 bit values. The first 8 bits of this value is sent in the first byte, and the 2 MSB are sent in the second byte. After the two 10-bit values are sent in 4 bytes, the last byte is sent containing indicators for the 3 push buttons on the joystick peripheral (this served no purpose in the project). The data is represented in 3 bits stored in the LSB of the fifth byte sent.

1	2	3	4	5
X (low)	X (high)	Y (low)	Y (high)	Buttons

Figure 13: PMOD Joystick data representation

Since the motor can only move in a one direction back and forth, only the x-position data of the joystick was considered. The value ranges between 0 and 1024. If the value is greater than 550, the core outputs *motor_position == 1,* which tells the motor to go right, otherwise *motor_position == 0,* which tells the motor to go left. The speed of the motor is controlled through the *motor_speed* output. The more the x-position data of the joystick deviates from the center value of 512, then the higher the speed value sent to the motor-ip-core.

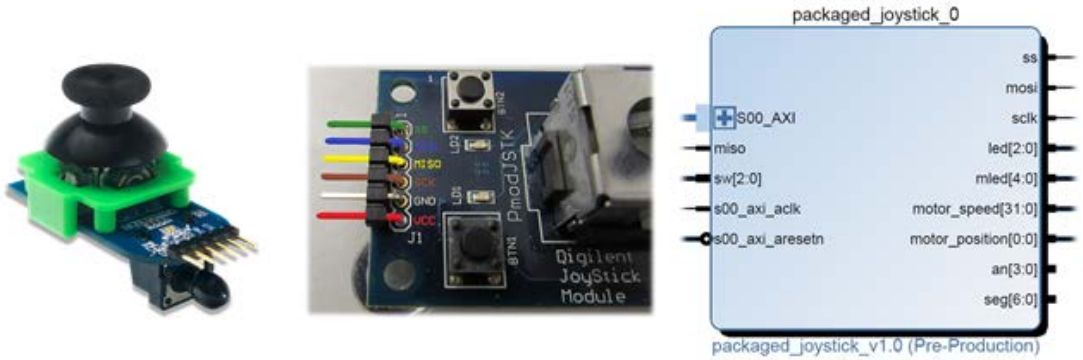

Figure 14: Physical joystick (left), PMOD pins (middle), finalized joystick IP (right)

Puck Detection & Trajectory Block (Custom)

The main purpose of the Puck Detection & trajectory IP core is to automate motor position and speed control if only one player wants to play.

The final Puck Detection & trajectory IP contains the following hdl source files:

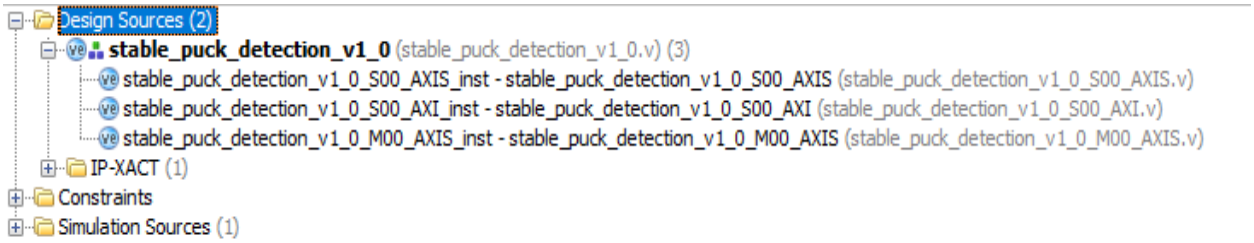

Figure 15: Puck Detection & Trajectory Design Source Hierarchy

Design Hierarchy

The *stable_puck_detection_v1_0.v* file instantiates the axi_stream slave, axi_stream master, and axi_Lite slave modules. This file also contains a 3.8 KB fifo for storing one line of video stream data (1280x24-bit RGB pixel data). The axi_stream slave, master, and lite hardware descriptions can be found in the *S00_AXIS.v, M00_AXIS.v,* and *S00_AXI.v* files respectively.

The *S00_AXIS.v* file contains an fsm for interfacing with the AXI4 video stream. This file also contains the hardware description for finding the x and y position of the puck in the video frame. The *M00_AXIS.v* file contains the fsm for outputting the modified video stream to the VDMA from the fifo. The *S00_AXI.v* file contains 15-slave registers that can be configured through software via the MicroBlaze to change the hardware's ability to track the puck in different lighting conditions, and change the output video stream.

I/O & Slave Register Descriptions

Note that when reading the table, 'Value written to SLV_REG X' indicates that the user writes the corresponding value to slv_reg x via MicroBlaze. 'Value read from SLV_REG X' indicates that this register can be read by the MicroBlaze only.

Signal name	**Signal type**	**Description**
threshold_pos1[31:0]	output to ILA	Indicates the minimum number of consecutive dark pixels that the core has received from the input video-stream. This minimum count is set by SLV-REG 5 called *pixel_count_target*
threshold_pos2[31:0]	output to ILA	Indicates the number of consecutive dark pixels on a video line that the core receives past the minimum count (threshold-pos1).
video_lines[31:0]	output to ILA	Indicates the video_line that is currently being processed by the core.
pixel_count_target[31:0]	output to ILA	Value written to SLV_REG 5: sets the minimum consecutive dark pixel count to issue a new puck location
threshold_level[31:0]	output to ILA	Value written to SLV_REG 4: between 0 and 255 indicating the minimum RGB value to qualify as a dark pixel.
new_color[31:0]	output to ILA	Value written to SLV_REG 6: indicating the output color of the puck after it has achieved the *pixel_count_target* (grey used in final demo)
h_start_value[31:0]	output to ILA	Value written to SLV_REG 0: indicating the pixel number on the video line to start finding the puck and modifying the video stream to output
h_stop_value[31:0]	output	Value written to SLV_REG 1: indicating the

	to ILA	pixel number on the video line to stop finding the puck and modifying the video stream to output.
v_start_value[31:0]	output to ILA	Value written to SLV_REG 2: indicating the video line in the frame to start finding the puck and modifying the video stream to output
v_stop_value[31:0]	output to ILA	Value written to SLV_REG 3: indicating the video line in the frame to stop finding the puck and modifying the video stream to output
motor_update_rate[31:0]	output to ILA	Value written to SLV_REG 7: serves no purpose in the design, since data is given to the motor-ip-core once every frame.
x_pos_per_frame[31:0]	output to ILA	Value read from SLV_REG 8: indicating the approximate x-position of the puck within the frame
y_pos_per_frame[31:0]	output to ILA & motor	Value read from SLV_REG 9: indicating the approximate y-position of the puck within the frame
valid_data	output to ILA	Trigger signal for ILA, going high upon passing the minimum pixel count target.
motor_offset_x[31:0]	output to ILA	Value written to SLV_REG 10: If the puck is not detected because it is out of bounds, this value is the default x-position the core outputs.
motor_offset_y[31:0]	output to ILA	Value written to SLV_REG 11: If the puck is not detected because it is out of bounds, this value is the default y-position the core outputs to the motor block
bottom_y_edge_case[31:0]	output to ILA	Value written to SLV_REG 13: If y-position of the puck is greater than this value then y_pos_per_frame gets assigned the highest position value (2100). This is to allow the motor to position itself at the bottom of the board.
top_y_edge_case[31:0]	output to ILA	Value written to SLV_REG 14: If y-position of the puck is less than this value then y_pos_per_frame gets assigned the lowest position value (0). This is to allow the motor to position itself at the top of the board.

use_default[31:0]	output to ILA	Value written to SLV_REG 12: If set to 1, this enables the use of values *top_y_edge_case*, and *bottom_y_edge_case*.

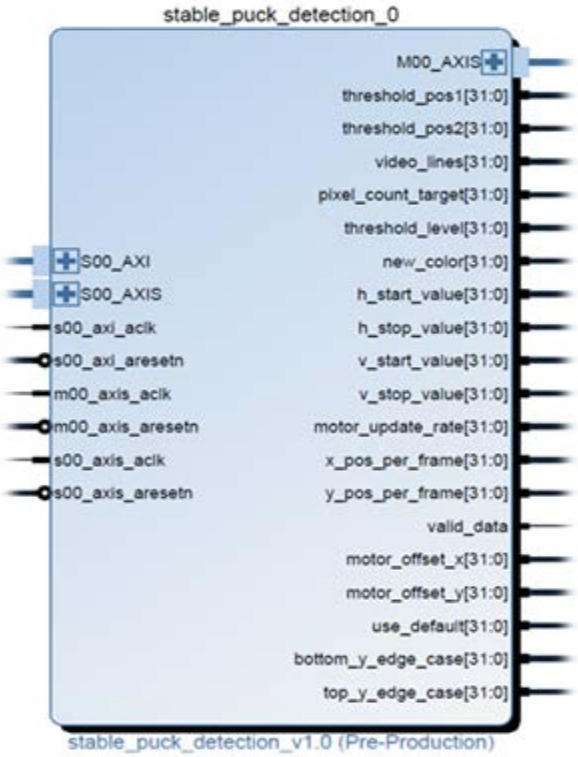

Figure 16: Puck Detection & Trajectory block

Description of functionality

AXI-stream data enters the IP core from the Video-In-to-AXI-stream IP block via the axi_stream slave interface. Upon entering the core, each 24-bit pixel in a video line is counted. Since the camera outputs 1280x720, there are 1280 pixels per video line, and 720 video lines per frame. Once the pixel count passes *h_start_value* and *v_start_value*, the core starts checking for pixels with RGB data less than *threshold_level.* If the value of the pixel data is less than *threshold_level,* then '0' is written into the data fifo to indicate black, otherwise 0xffffff is written to indicate white. In the special case where the number of consecutive dark pixels passes *pixel_count_target* for each video line*, new_color* is written into the data fifo until there are no more consecutive dark pixels in the line. Video output modification stops when the video line value or the horizontal pixel count value exceeds *v_stop_value* and *h_stop_value*. *x_pos_per_frame* and *y_pos_per_frame* are updated once per frame. They hold the x and y position of the puck averaged over the video lines where the *pixel_count_target* was met. Since the motor position range is 0 to 2100, and the actual video lines only range from 0 to 720, a multiplier factor of 3 was used before sending

y_pos_per_frame to the ILA and motor-ip-core. *bottom_y_edge_case* and *top_y_edge_case* are register values used in the special case we need to adjust the *y_pos_per_frame* along the corner cases. Please see the I/O description table for more information about these registers.

AXI-stream data is outputted via the axi-stream master interface. The master interface reads pixel data from the fifo and sends this data to the VDMA to be stored in a frame buffer and outputted to the HDMI display. Figure 17 depicts a typical example of how an HDMI output display would look like. The grey area on the puck is because new_c*olor* (grey) writes to the data fifo once the *pixel_count_target* is met.

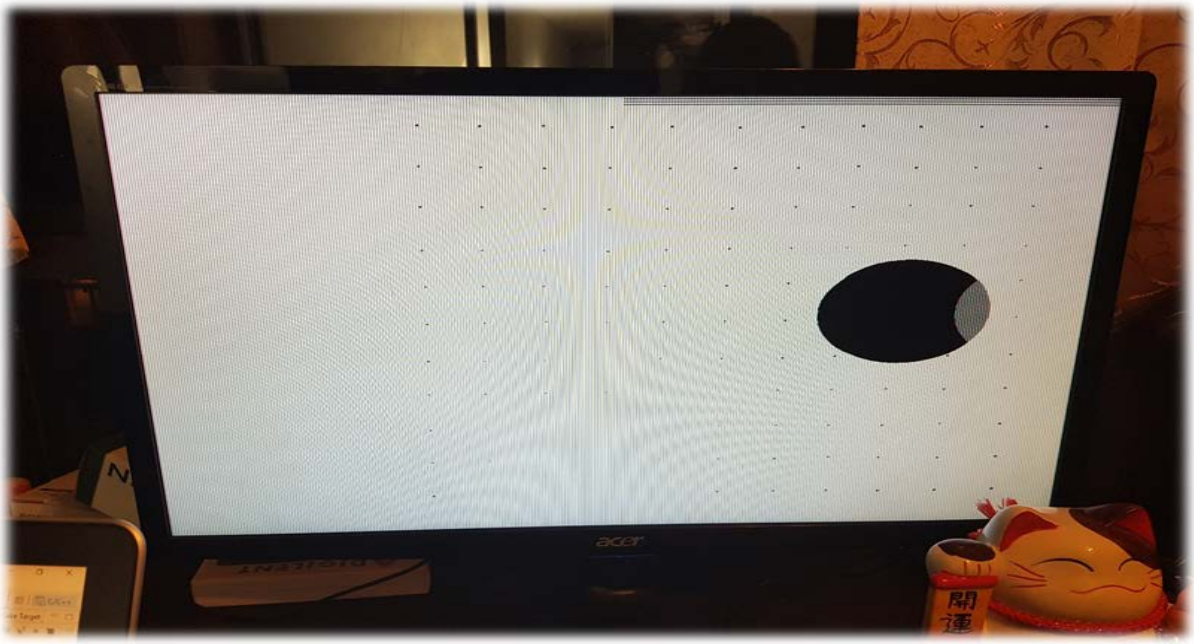

Figure 17: Output video stream (grey pixels indicate the approximate puck location)

Testing functionality

A preliminary version of this IP core was presented in the test bench demo during the course. A test pattern generator (TPG) HLS core was used to input axi-stream video data into the puck detection block. Another TPG core was used to accept the filtered output of the block. Behavioural simulation on this test bench setup was used to verify the core's functionality. Please see the test bench setup in figure 18 below.

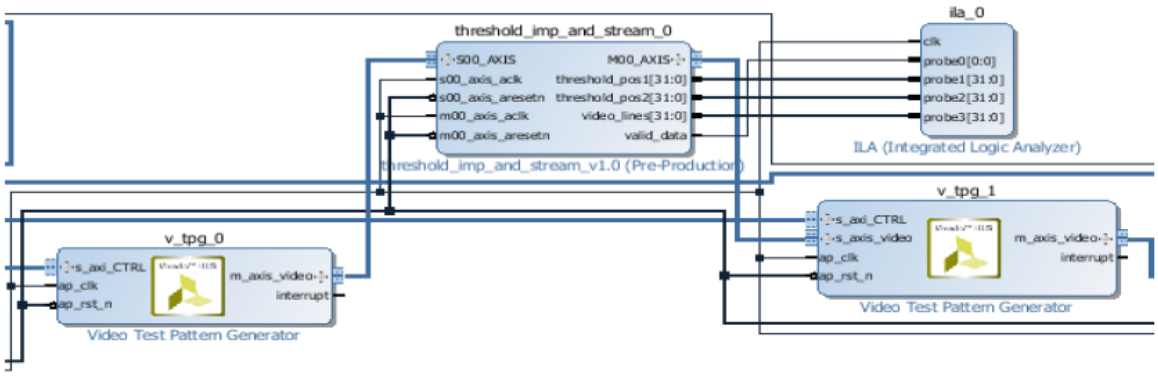

Figure 18: Testbench setup for preliminary puck detection block

DMA Audio Demo Project (Digilent)

The main purpose of the DMA Audio Demo Project from Digilent is to store a 5 second input stream of audio data and play it back when a button is pressed. This project was chosen over other versions of audio projects provided by Xilinx, due to its lower resource usage and relatively less modification required and easier integration.

Audio demo is responsible for taking input from one of Line-in or Mic connectors on the Nexys Video board and play it back on Line-out or Headphone connectors. Audio input and sampling is done by I2S. Since this is a IIC device, IIC IP block is used to put this data on the AXI interface. DMA block then allows to store this data in the memory - DDR in the original demo provided by Digilent. GPIO IP block controlled on-board buttons that were used to initiate record or playback. Original audio demo project included a MicroBlaze soft-processor. Finally, this demo is run through SDK - this made hardware integration of the IP blocks into the top level of our project easier. SDK project controls the duration of the recording which can be adjusted by the changing the sampling number of the audio codec. Pressed buttons are mapped to certain actions inside the SDK project as well.

IP blocks used by the audio demo, their versions and brief descriptions can be found in the Figure 19 and the table below. All IP blocks are sourced from Xilinx.

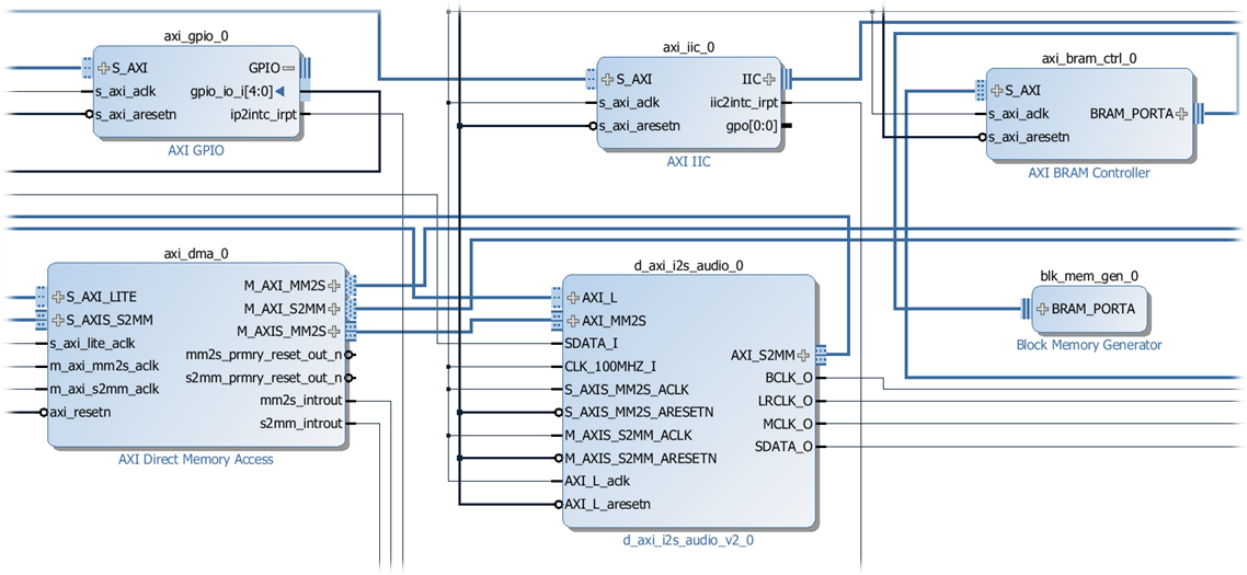

Figure 19: Audio demo project implemented using BRAM

IP Block	Version	Description
GPIO	2.0	Provides a general purpose input/output interface to AXI interface https://www.xilinx.com/support/documentation/ip_documentation/axi_gpio/v2_0/pg144-axi-gpio.pdf
IIC	2.0	Provides connection between IIC bus devices and AXI interface https://japan.xilinx.com/support/documentation/ip_documentation/axi_iic/v2_0/pg090-axi-iic.pdf
I2S	1.2	Enables connection to external audio devices https://www.xilinx.com/products/intellectual-property/1-f31nkv.html
DMA	7.1	Provides high-bandwidth direct memory access for external peripherals on AXI interface https://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf
BRAM Controller	4.0	Provides access to local block memory using AXI interface https://www.xilinx.com/support/documentation/ip_documentation/axi_bram_ctrl/v4_0/pg078-axi-bram-ctrl.pdf
BRAM Generator	8.2	Used to generate local memory using block RAM resources on Xilinx FPGAs https://www.xilinx.com/support/documentation/ip_documentation/blk_mem_gen/v8_2/pg058-blk-mem-gen.pdf

The main challenge of audio demo project came up during integration stage. Since it is not packages as an IP block (like custom joystick and motor blocks), we had to include each IP used by the audio demo into the top level of our main project. The project had to be separated from the original MicroBlaze it came with and made sure that all correct settings were maintained.

During this stage, it was also discovered that there can be only one MIG IP in the design. First option suggested was to design a data handler to allow two masters utilize DDR memory, as both HDMI and Audio require a continuous read/write access to the data in memory. Xilinx does not provide such capability for MIG out-of-the-box and a custom design could lead to bigger issues that would affect successful completion of the project. Second option was to take out DDR memory from the audio demo completely and replace it with local block memory. Total BRAM size on the Nexys Video board allows for a storage of a 5 second audio sample, but as mentioned earlier the duration could be easily changed in the SDK. This way, DDR memory was left untouched to image processing and the issue was resolved.

During the integration stage, SDK project for the audio demo had to be modified to run together with HDMI project. All initialization and record/playback procedures were contained in demo.c file, with function definitions in the header files. Digilent did not split initialization from the rest of the code and everything was contained inside the main function in a while(1) loop. This code had to be split up, to follow a better style presented in the HDMI demo project, where initialization was done separately and then the main function was called. Audio recording was moved to the initialization stage, since the sample was only to be recording once at the beginning, while the game is being set up.

Playback functionality had to be included into a loop inside the HDMI project to ensure a continuous music in background. Since HDMI project main loop occupies the system with an indefinite loop that is waiting for an input on UART (which would signify a user input in attempt to make a change and perform a certain action), audio playback had to be called in multiple locations. During testing it was also discovered that audio stream was not let to empty and it would be attempted to load it again, output sound would be corrupted. This required inclusion of sleep timers before “playback done” signal was asserted. Another challenge that arose during integration was due to the interrupts from GPIO. All buttons were taken out of playback control and this issue was resolved.

Description of Design Tree

The github repository for this project is available at the following link: https://github.com/emilkarimov/G3_airhockey. The following information is also available in the README.md file at the top level of the repository. The main files in the design tree have been identified below:

docs - Project documentation folder containing the final presentation, report, block diagram and video.

- ➔ **ECE532 Final Presentation.pptx** - This powerpoint presentation was used during the Milestone 7 final demo.
- ➔ **ECE532 Final Report.pdf** - This is the final report that explain the 532 project (this document).
- ➔ **hdmi.pdf** - This file is the final block diagram for the project.
- ➔ **ECE532.mp4** - This is the video recording of the Milestone 7 final demo. This video shows the project working with camera input, joystick input, motor output, display output, and audio playback for background music.

src/image_proc - Source files for the project (Vivado 2016.2)

- ➔ **NexysVideo-master_v3** - Main project built from the HDMI demo
  - ➔ **Projects/hdmi/proj/hdmi.xpr** - This is the main Vivado project file required to launch the final project.
- ➔ **Projects/hdmi/proj/hdmi.sdk** - This directory contains the hardware platforms, BSP, hardware description files, and the SDK components.
  - ➔ **Projects/hdmi/src/constraints/NexysVideo_Master.xdc** - This file provides the top level Xilinx Design Constraint for the project. This includes switches, LEDs, HDMI in/out, audio codecs, and PMODs JA and JB.

- ➔ **motor_ip_mar27_1.0** - Custom IP repository for Motor Control
  - ➔ **hdl** and **src** directories provide the verilog code required for the IP block.
  - ➔ **drivers/motor_ip_mar27_v1_0** directory provide the hardware drivers required by the SDK.

- ➔ **packaged-joystick/ip_repo/packaged_joystick_1.0** - Custom IP repository for Joystick Input
  - ➔ The **hdl** directory contains all of the code required for the IP block, written in verilog.
  - ➔ **drivers/packaged_joystick_v1_0** directory contains the hardware drivers required by the SDK.
  - ➔ **edit_packaged_joystick_v1_0.xpr** allows the user to edit the IP block.

- ➔ **repo/repo/local** - IF and IP repositories required for the HDMI demo
  - ➔ **if/tmds_v1_0** directory contains the XML interface specification required for the HDMI project (input and output).
  - ➔ **ip/axi_dynclk_v1_0** - This IP block is used to generate a dynamic pixel clock which changes based on the input and output video resolutions.
  - ➔ **ip/dvi2rgb_v1_5** - This IP block is used in the input HDMI stream to convert from TMDS DVI to RGB data.
  - ➔ **ip/rgb2dvi_v1_2** - This IP block is used in the output HDMI stream to convert from RGB to TMDS DVI video format.

- ➔ **stable_puck_detection/ip_repo** - Custom IP repository for Puck Detection
  - ➔ **stable_puck_detection_1.0/hdl** directory contains the verilog code used in the IP block.
  - ➔ **drivers/stable_puck_detection_v1_0** directory contains the hardware drivers required by the SDK.
  - ➔ **edit_stable_puck_detection_v1_0.xpr** allows the user to edit the IP block.

Tips and Tricks

- When adding new pins to the XDC, the HDL wrapper will not regenerate during synthesis or implementation. The wrapper must be re-generated manually, even though Vivado has a pop up that lets Vivado auto-manage the wrapper.
- Synthesis and implementation can take hours with a big project, so either offload projects to a powerful computer (Vivado doesn't benefit from more cores, but will benefit from more RAM), or plan synthesis runs carefully.
- Try to understand as much as possible, even though a feature works now, it may break or become unstable when integrating other designs.