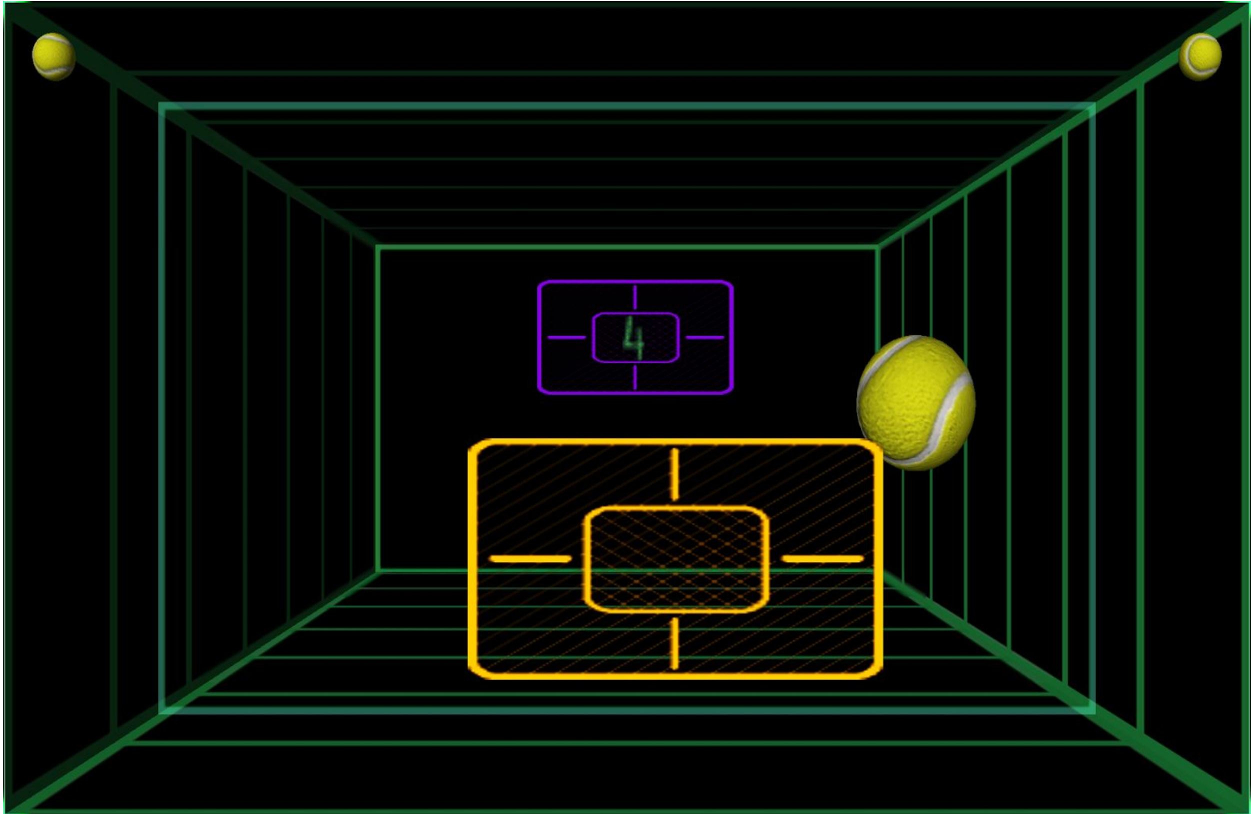
6.111 Final Project: 3D Multiplayer Pong



Louis Tao, Miguel Rodriguez, Paul Kalebu

Abstract

Our 3-D multiplayer pong project is inspired by our 2-D pong class assignment. Our goal was to create a 3-D 2-player pong game that allows players to control the positions of their paddles on the screen with colored objects. User input to the system was handled by an object recognition module with an NTSC camera. Physics-based considerations used the inputs of the system to determine the position of the ball in 3D space. The translation of the 3D coordinate positions of the balls and paddles to 2D display positions is implemented via a series of linear interpolation circuits. The system produces audio to increase the immersive experience for the players. For two players, we utilize two monitors, set up in such a way where each player can see the movement of his or her own paddle as well as his or her opponent's paddle.

Introduction - *Paul Kalebu*

Lab 3 tasked us with building a two-dimensional pong game, just like the very first sport arcade video game. This project takes that further by turning the game into a pseudo three-dimensional experience, by giving the playing field of view a z-axis with paddles on both ends. Similar to Lab 3, our project aims to add a multiplayer experience in addition to the single player game. Unlike Lab 3, however, this project uses a camera for user input instead of the labkit buttons. Initially, the project intended to track paddle movement by use of accelerometer and gyroscope, but issues brought about by drift in the accelerometer took too long to resolve, so the alternative solution was to use a camera instead. Overall, the project allowed all team members to explore their areas of interest. We all enjoy video games, and are very interested in integrating software development and hardware design. The game uses a physics module to translate data from the camera into coordinates later displayed by the graphics module via a series of linear interpolation circuits. The graphics module operates on two FPGAs to give the second player a display separate from player 1's display. The two displays allow each player to play from their own perspective. The project aims to achieve inter-FPGA communication with a parallel communication interface instead of serial. In addition to multiplayer gameplay and tracked user input, the project aims to enhance gameplay further with stereo audio, which outputs sound whenever certain events, like collisions occur during the game.

Below is a high-level overview of the system design for the game.

Diagram Overview

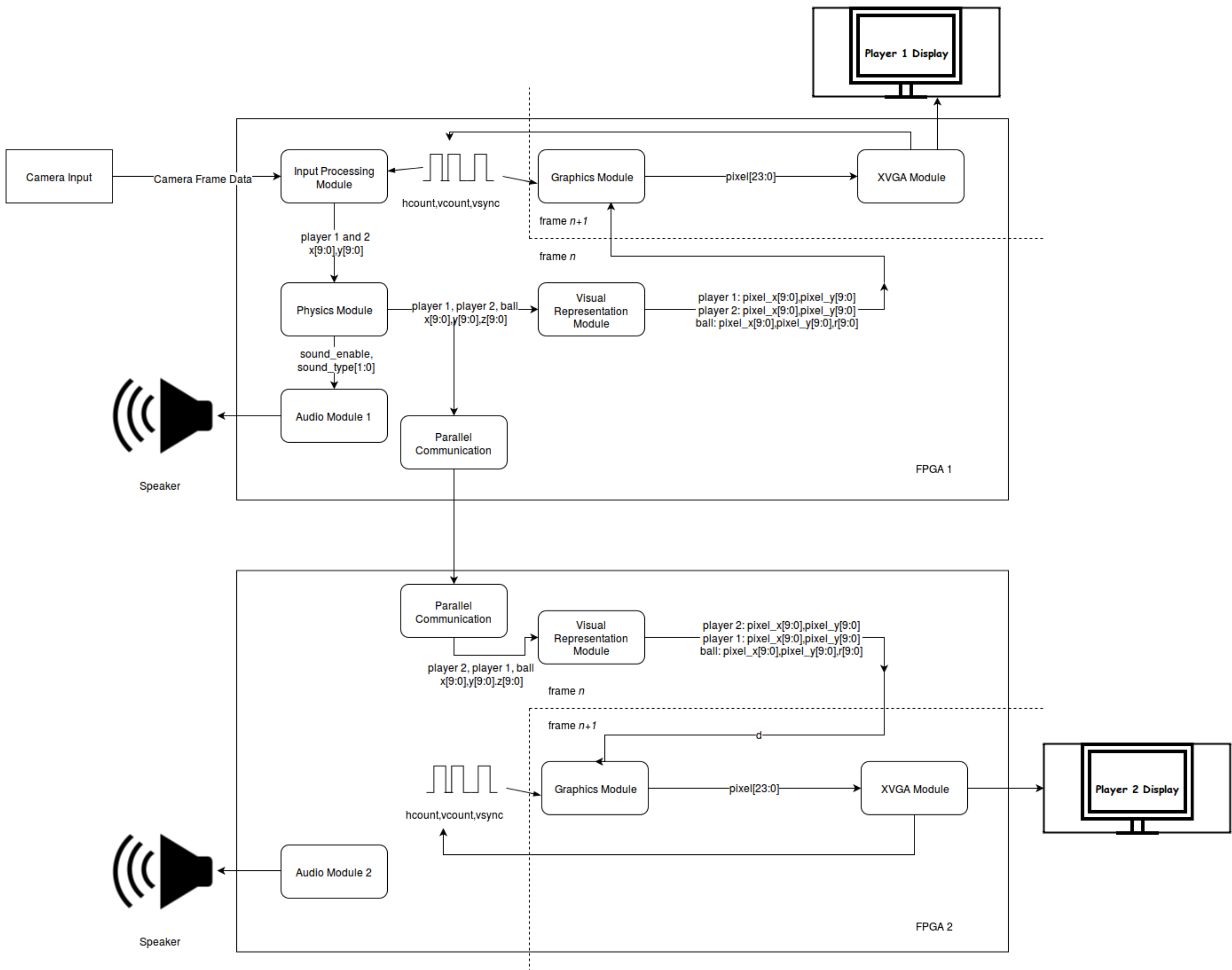


Figure 1: High-level overview of system design

NTSC Camera and Object Detection Modules - *Louis Tao*

One of the major functionalities of the 3-dimensional multiplayer pong game is computing a player's paddle coordinates by detecting colored objects with a

staff-provided NTSC camera. This component involved taking colored video from the NTSC camera and determining from the center of mass of a collection of pixels that corresponded to a colored object in front of the camera. Then, the center of mass of a recognized object would be transformed into coordinates that were usable in the 3-dimensional game space and which the physics performed computations on. In this section, we will present an in-depth discussion of the object recognition framework and discuss the workflow as well as any problems that I encountered along the way.

Colored Video Discussion

Fortunately, we were already presented with a staff sample implementation that allowed the FPGA to display grayscale images on an XVGA; however, object detection with the grayscale video proved to be an impossible task, as differentiating factors such as the Cr and Cb bits of the YCrCb color stream were discarded in the sample implementation.

Therefore, the first step involved modifying the vanilla NTSC labkit files to accommodate colored video. We altered the provided ntsc_to_zbt module to first separate the 30-bit YCrCb data vector into each of its respective 10-bit components. Then, we utilized the staff-provided YCrCb2RGB module to obtain each of the 8-bit values for RGB. Considering that we wished to write 2 18-bit values of RGB (we obtain the 6 most significant bits from each of the additive primary colors) to each location in ZBT memory, it was also necessary to modify this module to write two 18 bit values instead of four 8 bit values, as done for grayscale video. Since each memory location contains only two pixels, memory addressing changes twice as fast.

The next step involved modifying the vram_display module, which supplies pixels to the VGA display, to read the ZBT memory with respect to two pixel values from a single memory location, rather than four, as in the case with grayscale video.

Paddle Coordinate Tracking

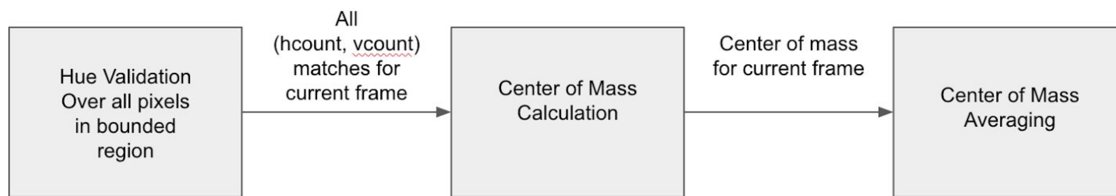


Figure 2: Overview of Paddle coordinate tracking

Computing the paddle coordinates involved several steps, given the colored video.

RGB is a simple linear transformation of the YCrCb data. However, color detection was found to work best with HSV data, which necessitated an explicit transformation of the YCrCb data to RGB. In this model, the hue dimension represents the color, saturation represents the dominance of that color and the value dimension represents the brightness. All of the values could be utilized in a color detection algorithm, but we found that utilizing only the hue was sufficient and greatly mitigated the need to develop potentially complex functions of the other values as a validator for pixels in the colored video, where a validator simply asserts a high signal for those pixels which meet a certain color threshold. Utilizing RGB values instead of HSV values would've required more research into specific values for the target RGB values, whereas we can envision the hue as a continuous color wheel. Such a model simply allowed us to check whether the hue of a certain pixel fell within a certain "epsilon" of a target hue. For a given frame, we sum up all of the vcounts and hcounts and utilize a CoreGen divider to divide the sums and get a center of mass. Additional techniques were implemented to ensure that the center of mass coordinates were robust, namely, eliminating pixels around the video boundaries (which were the source of unwanted noise) and a module which tracked the moving average of the center of mass from the most recent frames. Furthermore, to ensure a one-to-one movement of the paddle with the averaged center of mass, we implemented a module that scaled the center of mass coordinates from its position in a 700 x 500 pixel frame to a position in a 1024 x 768 frame. Because the video that is displayed is also a mirror image of what it is capturing, it was necessary to flip the center of mass across the middle of the screen. In the gameplay, the player should be able to move the center of mass to the right, if he or she moves a paddle to the right.

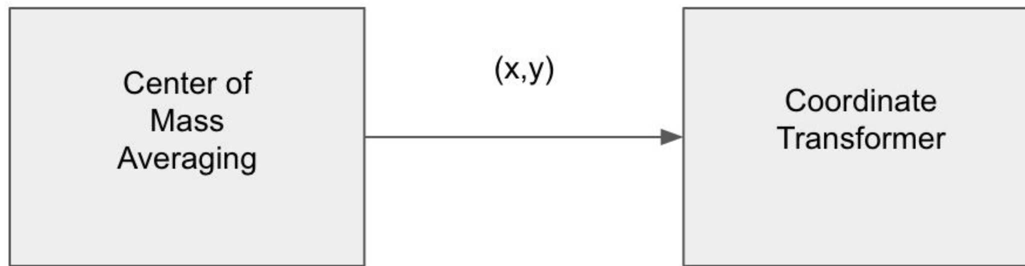


Figure 3: Coordinate generation

Physics Module - *Louis Tao*

In our physics model, we constrained the 3-dimensional space of our game to be 1024 x 768 x 2048 pixels, the width, height, and depth, respectively. At the most basic level, we implemented simple physics, where a ball travels in a straight line and has x, y, and z velocity components. For the purposes of ensuring that all three velocities were nonzero, initial x, y, and z velocities were seeded with nonzero values at the start of the game. When the ball hits a wall, one of its directions is negated, as per a simple model of collision. Unfortunately, we weren't able to implement more a more advanced model of game physics. However, potential improvements will be discussed in the conclusions.

Overall, this module kept track of the positions of both players' paddles and the ball. It was simplest to use blocking assignments and model the physics logic as a combinational circuit; at each positive edge of vsync, the physics module would update the positions of the ball based on the positions of the paddles. Each of the paddles had a width, height, and depth of 128 x 128 x 4, which we found was sufficient to hit the ball, which was a sphere with diameter 64. If the ball collided with the wall on a player's side and wasn't in the bounded region of that player's paddle, then the game would end, with all the movement paused, until the reset button is set. Otherwise, the ball collides and reverses direction in the z dimension.

Object Recognition and Physics Discussion

The object recognition and physics modules were the first modules that needed to be implemented because the later modules needed these to be successfully tested.

Overall, integration was pretty straightforward, with the transformed center of mass from the object recognition system being fed into the physics module.

Some of the issues we found along the way when building the object recognition and physics modules are noteworthy, and others who wish to implement the same behavior in the future can take away lessons from our experience.

Although there was a staff-provided NTSC implementation, we found that modifying the existing code to output colored video wasn't as straightforward as we expected, in part because of the lacking documentation. Some time was set aside early on in the project to figure out a way to modify the existing code. An example of such bugs we ran into when modifying the existing code was very jittery video; it turned out, we had to modify the addressing scheme in the ntsc_to_zbt module to write twice as fast, because we were storing two pixels' worth of data instead of four in each ZBT memory location. We knew that the modifications we had made were correct when we saw smooth color video on the XVGA display.

We found that the implementation of the center of mass calculation from the colored video to be a little more straightforward than the modifying the the NTSC code to display color video. Because the video only appears in a subportion of the overall XVGA display, i.e. a 700 x 500 pixel region, it was important to only consider pixels within that range. Furthermore, the region was also in the middle of the XVGA display, so several iterations of our development efforts went towards empirically finding the range of that region. Determining which pixels went into the center of mass calculation was pretty simple, as discussed in the earlier section. Because we had already synthesized a CoreGen divider for use in generating colored video, we found that the same divider also did the job well for use in the dividing the sums in the center of mass calculation. To test the center of mass was moving along with the colored object we were holding in front of the camera, we implemented crosshairs that tracked the movement of that object. Further efforts included explicitly shading in the pixels that our code determined fell within a certain epsilon of a target hue.

The physics module wasn't able to be debugged until the graphics module was ready; with the graphics module that Miguel developed, we were finally able to iterate on the product until we were fully satisfied with the movements of the paddles and the ball.

Divider Module - *Juan Miguel Rodriguez*

Much of the graphics relied on linear interpolation. For this, we needed a way to divide numbers, since Verilog did not support this natively as they do multiplier circuits. Since we were pre-computing the graphics on the previous frame, we did not face any timing limitations, and did not have to resort to an efficient divider generated by the Xilinx design tool, like we did for the NTSC Camera Module. Instead, we sourced the divider module from the 6.111 online resources. The divider is easily controlled, following the protocol of being enabled by a start signal and producing a ready signal when the result has been computed.

The divider is based on a simple restoring divide algorithm. The number of iterations of the algorithm is proportional to the number of bits of the dividend and divider, which was upper bounded at 12 bits in our case. By means of some extra logic in the linear interpolator module (described in the next section) we were able to avoid needing to use signed numbers and the more complex signed number division circuitry. The unsigned logic is described as such: On each step, a trial subtraction occurs between the dividend and the divider. A temporary quotient/accumulator is maintained and multiplied by two on each step, whereas the temporary divider is divided by two. The trial subtraction becomes the new temporary dividend, and bits of the temporary quotient are set depending on the value of the trial subtraction. Once the algorithm carries out the 12 bits worth of iterations, the quotient contains our answer.
Here is an instance of the divider being controller and working:

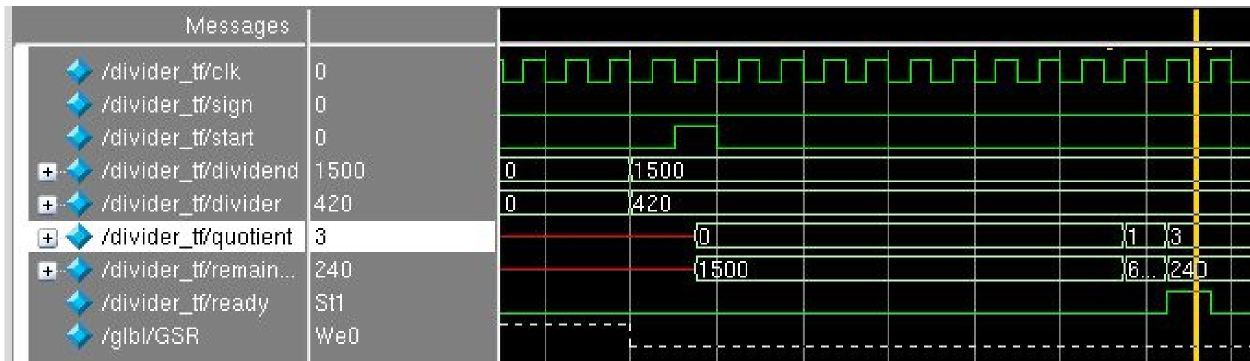


Figure 4: Modelsim Logic Analysis of Divider Module

Interpolator Module - *Juan Miguel Rodgriguez*

Having produced a divider module, the main idea behind the visual representation module (the module that takes in 3D space ball/player coordinates and outputs the pixel positions of where the sprites should go on the display) is linear interpolation. The Linear Interpolator Module does just this. Given two reference coordinates, X_1, Y_1 and X_2, Y_2, the module can calculate a corresponding y to a given x. The interpolator is easily controlled, following the protocol of being enabled by a

start signal and producing a ready signal when the result has been computed. With careful choice of reference coordinates and some extra logic described below, we were able to circumvent the need to use signed numbers, even with negative slopes.

The interpolator operates as a three stage state machine carrying out the steps for the following computation:

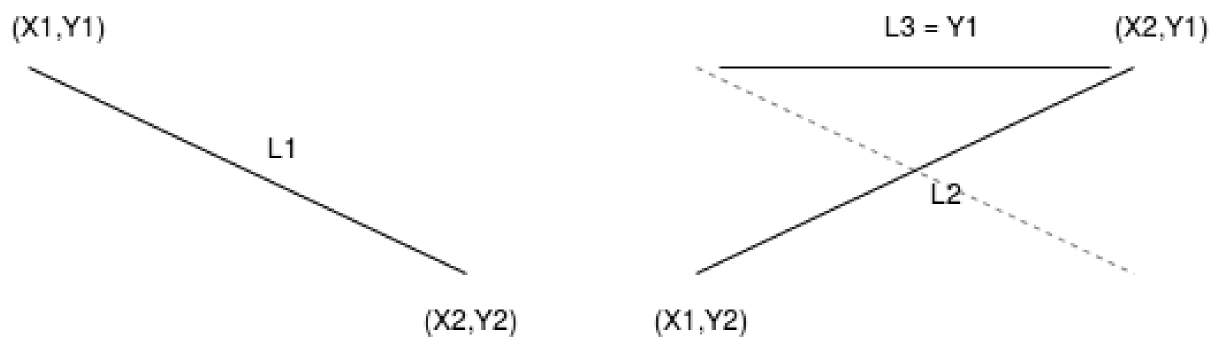
$$y=\frac{Y_2-Y_1}{X_2-X_1}(x-X_1)+Y_1$$

In the first state: the module calculates $(Y_2-Y_1)(x-X_1)$. This product is set as the numerator for the divider circuit, and (X_2-X_1) is set as the denominator. The divider is also set in motion and the state is transitioned to state 2. It is important to note that the divider has to operate on a bit width that is twice the bit width of the parameters, inputs, and outputs of the interpolator module, given that the product $(Y_2-Y_1)(x-X_1)$ can at most need twice the bit width of the multiplicand and multiplier. In the second state: the module stands by, waiting for the divider to finish computing the quotient, namely $(Y_2-Y_1)(x-X_1)/(X_2-X_1)$. Once the divider's result is ready, the interpolator's final result can be computed by adding Y_1 to the quotient from the divider. The interpolator's ready signal is also enabled at this point, and the state is transitioned. In the final state: the interpolator's ready signal is lowered so that it is only active for one cycle, and the state is transitioned to the done/idle state, awaiting another start signal for the next computation.

With careful choice of reference coordinates and some extra logic, we were able to circumvent the need to use signed numbers, even with negative slopes. One constraint/assumption that was made is that the x inputs and the y inputs will only be positive, along with positive reference coordinates as well. For the purposes of the Visual Representation module (described in the next section), this constraint provides no limitations. A second constraint is that (X_2-X_1) must be positive, or $X_2>X_1$, so that the divider module can also operate unsigned. This limitation is fine, because ordering of the reference points X_1,Y_1 and X_2,Y_2 is arbitrary, and swapping the points would yield the same line. A similar constraint arises with the numerator of the divider module, $(Y_2-Y_1)(x-X_1)$. The $(x-X_1)$ term must also be positive, or $x>X_1$. This constraint can easily be met if the operator of the module understands

the domain of $x \in [x_{min}, x_{max}]$ of interest, and chooses a reference coordinate (X_1, Y_1) on the line such that $x_{min} > X_1$.

As for the $(Y_2 - Y_1)$ term, some extra logic is needed to circumvent the need to keep this positive. It is important to note that, under the assumption that $X_2 > X_1$, the line defined by the two reference points will have an ascending slope if $Y_2 > Y_1$ and a descending slope if $Y_2 < Y_1$. In the case of the descending slope, $(Y_2 - Y_1)$ would be negative, and our dividend for the divider module would then be negative. To get around this limitation to support negative slopes, the following must be taken advantage of:



From the diagram above, we can still use the Interpolator Module on a line with a descending slope (line L_1), even if we don't want a negative numerator for the Divider Module. To do this, we would instead interpolate over the line L_2 , the line defined by the points , X_1, Y_2 and X_2, Y_1 . We can then compute the interpolation over L_1 by noting that $L_1(x) = L_3(x) - L_2(x) = Y_1 - L_2(x)$. The Interpolator Module takes advantage of this by checking if $Y_2 < Y_1$ at the beginning to determine if the slope of the line will be negative.

Here is an example of the Interpolator Module iterating through the range $x \in [0, 2047]$ with $(X_1, Y_1) = (0, 500), (X_2, Y_2) = (2047, 1000)$ for the line on the bottom and $(X_1, Y_1) = (0, 1000), (X_2, Y_2) = (2047, 500)$ for the line above.

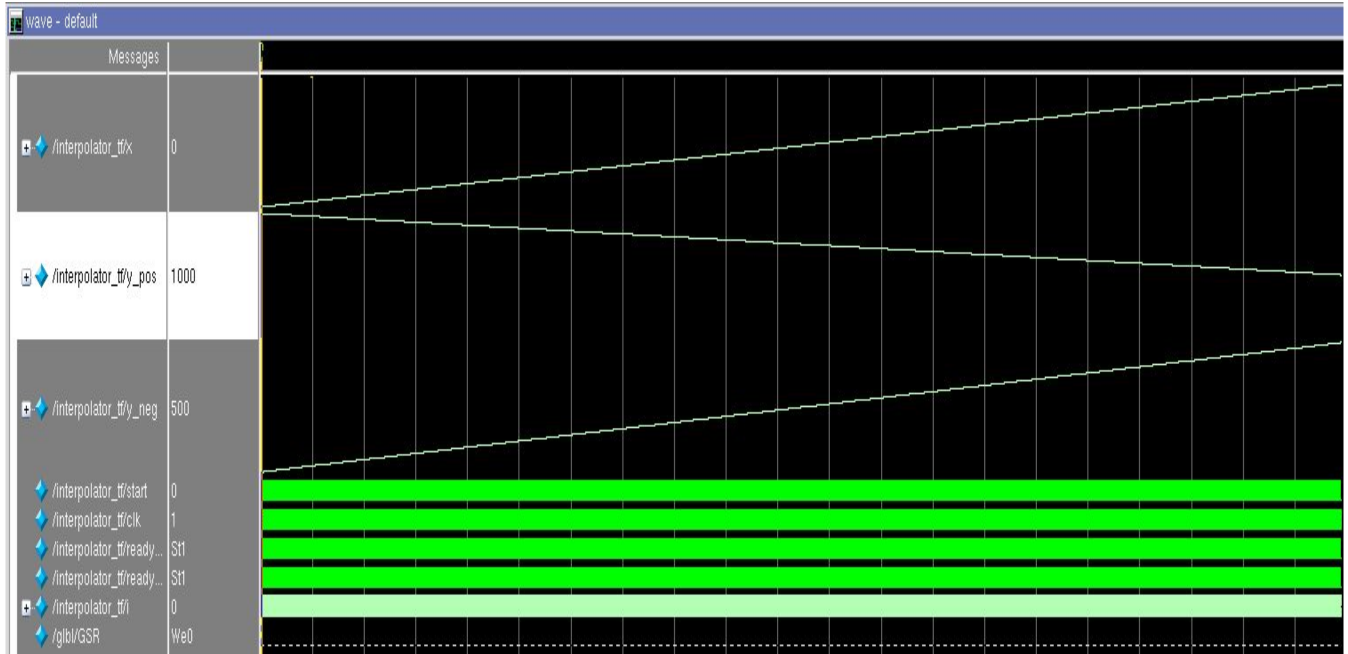


Figure 5: ModelSim visualization of Interpolation

Visual Representation Module - *Juan Miguel Rodriguez*

The Visual Representation Module serves as the core of the VGA display experience for the 3D pong game. At a glance, it takes the 3D spatial coordinate positions of the ball and paddles, and returns 2D pixel coordinate positions of the ball and paddles, along with the radius of the ball. A diagram summary is shown below:

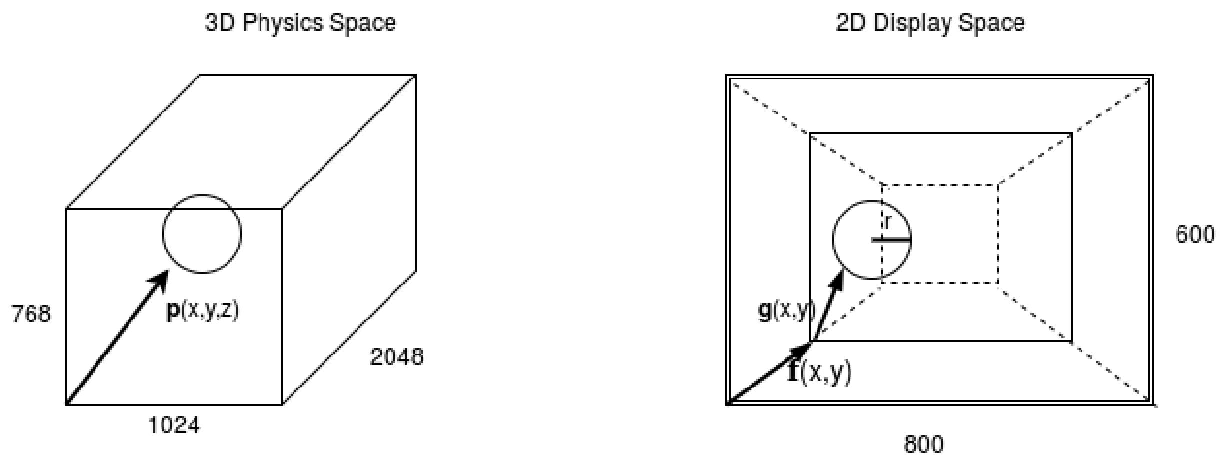


Figure 6: Visual representation of field of play

The are some key concepts here: the 2D display space simulates the 3D physics space by at all times having the ball exist in a frame on the screen. The dashed line wire

frames in the 2D Display Space figure above serve to show the corners of the frames as we go from the foreground to the background. The frame, and more specifically the position and size parameters of the frame, depend on the depth of the ball in the 3D space. The radius of the ball sprite also depends on the 3D depth of the ball, since we simulate the ball appearing farther away by becoming smaller. In short, the Visual Representation Module finds the pixel position of ball by adding the pixel position of the frame to the pixel position of the ball relative to the frame; and also computes the appropriate pixel radius for the ball.

The Visual Representation Module is a state machine that manages and integrates multiple Interpolator Module and Divider Module calculations. Although many such computations take place, a lot of them can be carried out in parallel. By carrying them out in parallel, we can think of the program flow in the following way:

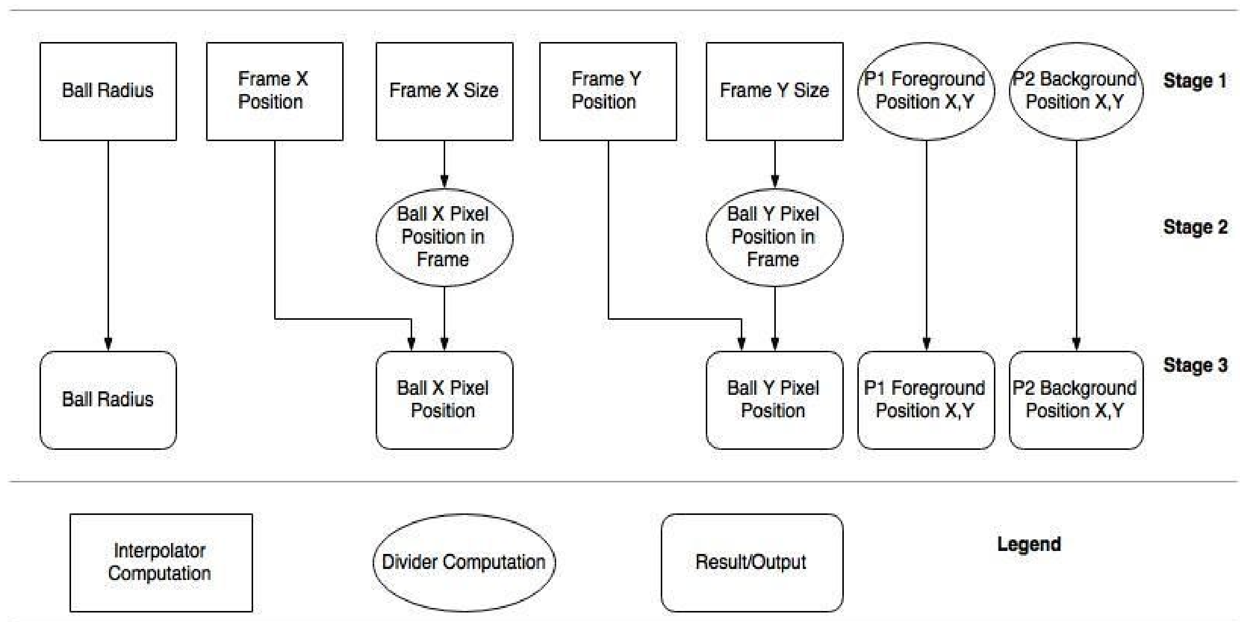


Figure 7: Graphic representation of graphics module

In the first stage, five Interpolator Modules and 4 Divider Modules are loaded with the correct inputs and enabled. Five of these computations (ball radius, player 1 paddle's pixel coordinates in the foreground, player two's pixel coordinates in the background) directly become outputs of the system. Four of these computations (the frame's pixel coordinates and the frame's size parameters) are used by later stages to compute the pixel coordinates of the ball. In the second stage, two dividers are loaded with the correct inputs and enabled to compute the pixel coordinates of the ball relative to the frame. The inputs to the dividers depend on the values of frame size parameters computed in the first stage. In the third and final stage, the Visual Representation

Module computes the pixel position of ball by adding the pixel position of the frame to the pixel position of the ball relative to the frame. The other results (computed in the first stage) are now outputted by the system as well. This description of the system hopefully emphasizes the importance of the start signal/ ready signal control scheme of the Divider Module and the Linear Interpolator Module. Otherwise, such asynchronous, multi-stage computation would not be possible.

Here is a summary of how each module is individually operated:

TABLE OF MODULE PARAMETERS, INPUTS, OUTPUTS

MODULE	INPUTS	OUTPUTS
Ball Radius	(X1,Y1) = (0,FOREGROUND_RADIUS) (X2, Y2) = (2048, BACKGROUND_RADIUS) X = Z depth of ball	Y = BALL_RADIUS
Frame X Position	(X1,Y1) = (0,0) (X2, Y2) = (2048, BACKGROUND_FRAME_POSITION_X) X = Z depth of ball	Y = FRAME_X_POSITION
Frame X Size	(X1,Y1) = (0,0) (X2, Y2) = (2048, BACKGROUND_FRAME_SIZE_X) X = Z depth of ball	Y = FRAME_X_SIZE
Frame Y Position	(X1,Y1) = (0,0) (X2, Y2) = (2048, BACKGROUND_FRAME_POSITION_Y) X = Z depth of ball	Y = FRAME_Y_POSITION
Frame Y Size	(X1,Y1) = (0,0) (X2, Y2) = (2048, BACKGROUND_FRAME_SIZE_Y) X = Z depth of ball	Y = FRAME_Y_SIZE
P1 Foreground Position X,Y	DIVIDEND: P1_X*800, P1_Y*600 DIVIDER: 1024, 768	QUOTIENT=P1_PIXEL_COORDS
P2 Background Position X,Y	DIVIDEND: P2_X*BG_SIZE_X,P2_Y*BG_SIZE_Y DIVIDER: 1024,768	QUOTIENT=P2_PIXEL_COORDS
Ball X Pixel Position in Frame	DIVIDEND: BALL_X*FRAME_X_SIZE DIVIDER: 1024	QUOTIENT= ball pixel x relative to frame
Ball Y Pixel Position in Frame	DIVIDEND: BALL_Y*FRAME_Y_SIZE DIVIDER: 768	QUOTIENT= ball pixel y relative to frame

Figure 8: Legend for graphics parameters

Audio - *Paul Kalebu*

Overview

The project aimed to enhance the player experience from the what the two- dimensional pong game built in Lab 3 provided. One of the avenues for this enhanced experience was the audio module. The audio module adds sound to the game, corresponding to various events during the game, to make the game as realistic as possible. Such events include collisions (for example a paddle striking the ball, or the ball bouncing off one of the walls), points being scored, the game beginning, and the game ending.

While it may seem quite auxiliary to the core project, the audio module was well-distributed across the tiers of goals for the project. The most basic commitment set out to be achieved was sound effects for whenever a player struck the ball with a paddle, and for whenever the ball bounced off one of the four walls in the game environment, as this was essential to achieving a realistic gaming experience at the most elementary level. This version of the audio module was completed during the third week of working on the final project –after other essential modules handling user input and physics, but before moving on to more complicated levels of the project like advanced gameplay and general system integration.

The general approach to designing the audio module was similar to that taken in Lab 5, where the task was to record and playback sounds on the labkit through the AC97 (audio codec) interface. The module takes, as input, a flag for each event requiring a sound effect, and selects the appropriate sound to play as output (*Figure 1*). For instance, if a collision happens, the collision flag's positive edge triggers playback through the AC97, by mapping the output to the collision sound clip. The necessary sound clips for each event were stored in block memory (BRAM) to later be played on the labkit whenever appropriate.

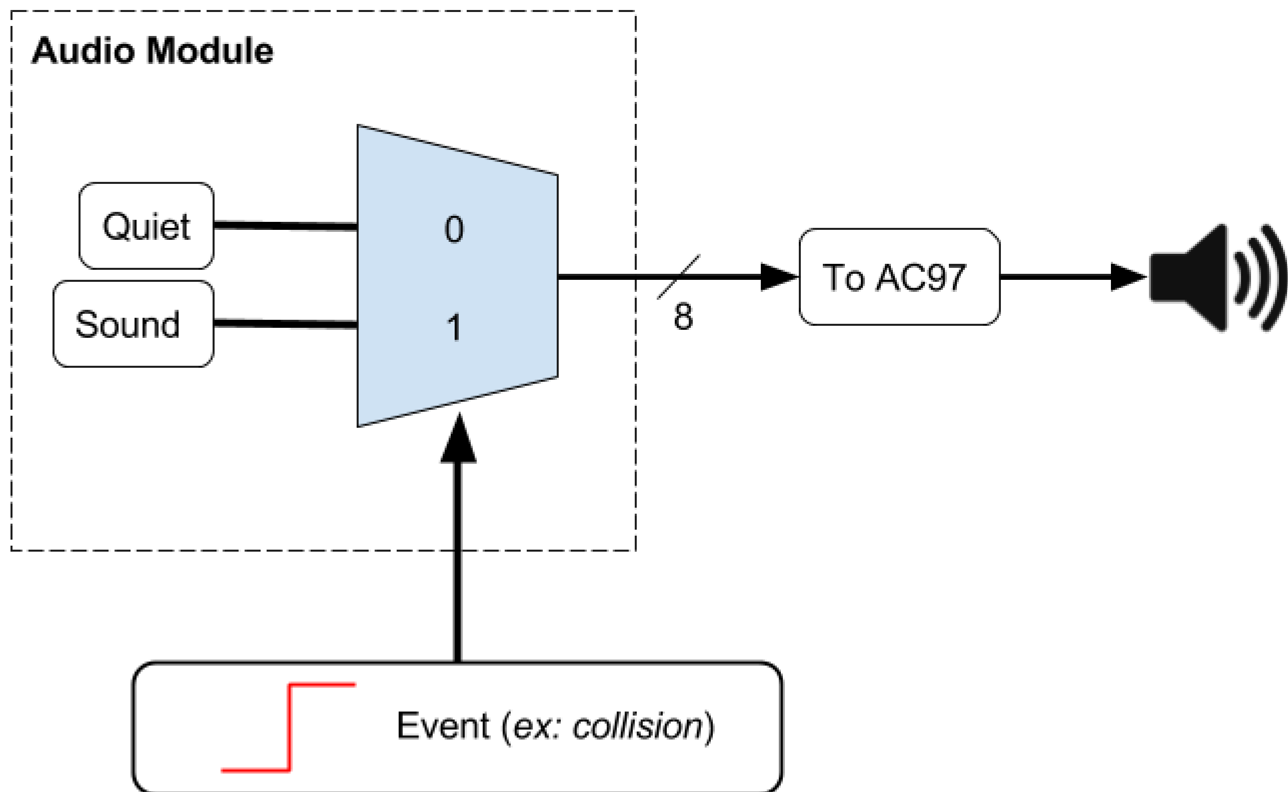


Figure 9: Basic graphical overview of the audio module

The basic core design for the audio module set the stage to later add on to the the module for further improvements to the project. The modular structure for inputs and outputs allowed for more sound effects for other events, and even stereo sound when communicating with a second FPGA for a second player.
As for sound effects for other events, each event was assigned a flag, which was equal to a logic 1 when the event happened and 0 otherwise. The flags were then used to select the appropriate sound within a case statement which mapped the output data to the appropriately selected sound clip stored in block memory.
Stereo sound in this project is theoretically defined along the z-axis, where the two different players are on opposite sides. Each FPGA has its own set of speakers and plays the appropriate sound when triggered to do so. For instance, if player 1 strikes the ball, the speakers on FPGA 1 output the sound effect for collision between the ball and paddle while those on FPGA 2 stay quiet. On the other hand, if player two strikes the ball, the speakers on FPGA 2 output the sound effect for the paddle hitting the ball while those on FPGA 1 stay quiet. Similarly, points scored by a certain player will trigger the sound effect for scoring on the scoring player's corresponding FPGA, while the other player's stays quiet. However, certain sounds are shared by both FPGAs, for example both FPGAs will output the appropriate sound clip for when the game begins or ends. Stereo is achieved by connecting the audio modules for both FPGAs across the FPGA communication interface and using the aforementioned trigger flags to select which FPGA plays the selected sound. There are several alternatives for how to implement the

audio module, and each comes with benefits and disadvantages, which will be assessed in the following subsections.

Design Alternatives

There were many options for implementing sound storage and audio playback, and with each one came tradeoffs on design simplicity, design robustness, storage size for sound, and overall sound quality. Below is a description of those considered for the project:

1. *Simple tones generated by sine wave*
   The primary approach, as outlined in the initial project presentation, was to generate a distinct sine wave tone unique to each event during gameplay. This approach was explored in Lab 4 to generate the car alarm, and offered simplicity and good quality of sound being played back. However, the approach was very limiting in the sounds to be played back. In particular, the sounds would come off as more of a gimmick in addition to the gaming experience instead of achieving realistic gameplay. For instance, there was no simple way of replicating the realistic sound effect of when a ping pong is struck by a paddle. The main goal of the audio module was to enhance how realistic it felt to play the game. In spite of the afforded simplicity, a different approach to implementing the audio module had to be considered because generating sine wave tones failed to achieve adequately realistic sounds effects.

2. *Record/Playback*
   Lab 5 was focused on recording, filtering and playing back sounds through and analog-to-digital converter (ADC) and across the AC97 interface. A similar approach could be used to record the desired sound effects, store them in block memory, and play them back when appropriate. Although it took a whole lab to implement the recording, filtering, and playback correctly, this approach also offered the added benefit of simplicity and minimal effort since the modules for Lab 5 had already been implemented. However, similar to the previously mentioned approach, this approach also lacks sound quality. It would require better signal processing beyond a low pass filter to cleanly reproduce the recorded sound. Additionally, aside from the necessary signal processing, producing the correct sound effects to be recorded was also challenging. For instance it would be difficult to correctly and most accurately replicate the in-game sound effect of a paddle striking a ping pong ball. Poor replication and inadequate signal processing would risk producing impure, unrealistic sound effects, causing the module to fall just short of achieving its goal of a realistic gaming experience.

3. *Convert .wav file to .coe for playback*
   Unlike the previous two approaches both fail to achieve the desired sound quality, this approach takes advantage of the abundance of pre-made sound effects online. Although most free sound effects online are in *.mp3* file format, they can be easily converted to *.wav* file format with the help of online resources. The audio module uses a 48kHz sampling rate for the pulse code modulation (PCM) data sent to across AC97 to the speaker. This sampling rate offered adequately high resolution to achieve the desired

sound quality. For each downloaded *.wav* sound effect file, a Matlab script is used to check the sampling rate for 48kHz. In the event that the sample is not at 48kHz, the script converts the sample to 48kHz and saves the output file after confirming the previous sample rate and the new sample rate of 48kHz. A second Matlab script is then used to convert the new *.wav* file with a sample rate of 48kHz is to a *.coe* file, which is readable by the AC97 codec. Both Matlab scripts were found in the tools section on the 6.111 class webpage. Each *.coe* file is stored in its own instance of 64Kx8 BRAM, and played back when appropriate. Out of all three approaches, this approach was chosen for the following reasons:

a. Although this approach involves searching the internet for specific sound effects that may not exist, and two levels of cumbersome file conversion, it ultimately helps achieve the main purpose of the audio module by **offering the desired sound quality** as evidenced by the clarity in playback.
b. Additionally, the approach **offers flexibility** and abundance when it comes to choosing sound effects. The internet is rich with sound effects, both realistic and electronically created, that can be fit to the various events that take place during gameplay. In addition to the guaranteed sound quality after conversion, downloading and converting these sound effects is a lot easier than manually recreating them.
c. The different events that require sound effects during the game only last a few seconds, and thus require short sound clips that do not take up much memory. This conveniently allows for the use of BRAM for storage, **eliminating the need for external memory hardware**.
d. The modular nature of the design, and the fact that it is all contained within the labkit (Verilog and internal memory) allows for the **abstraction** of the audio module, which enables **simple integration** with the rest of the system.
e. The use of internal tools like memory and file conversion eliminates the need for external hardware like memory devices and recording equipment, which makes the design a lot **easier and cheaper to implement**. This leaves little room for error in implementation, as a result.

In summary, choosing the third of the three design options allowed for flexibility in choosing sound effects, eliminated the need for external recording and storage hardware, made for cheap, simple implementation, and above all achieved good sound quality.

Detailed module and submodule description

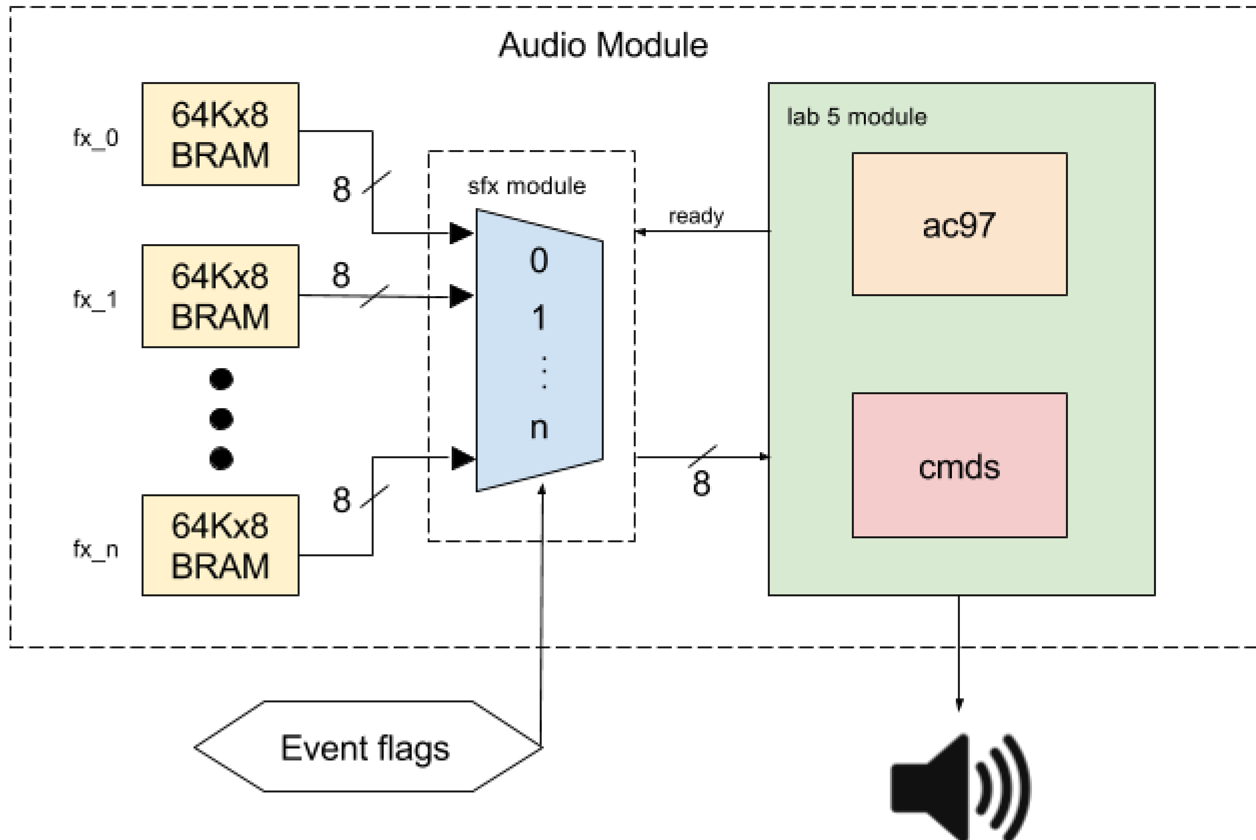


Figure 10: Detailed graphical representation of audio module with submodules

SFX module: This is the core submodule to the audio module. It instantiates memory and selects what sound clips to output as playback data based on the ready signal from the ac97 and the input value of a given event flag. This module also instantiates the block RAM memory in which *.coe* files for the sound clips of interest have been stored. The module takes in and outputs 8 bits of the selected file at a time, as the memory blocks are 8 bits wide and about 64000 addresses deep. Whenever a flag is asserted, this module reads the first address in the memory block corresponding to the asserted flag, and outputs the 8-bit data in the address, before moving on to the next address. Once the event flag is deasserted, the module resets the value of the memory address being read to 0 such that the next time the flag is asserted, playback does not begin in the middle of the sound clip's *.coe* file in memory, but rather at the beginning. Additionally, at times the blocks of memory are much larger than the data for the short sound clips being played. One glitch that resulted from this is a five second delay in repeated playback (instances when the module was required to loop through the data being read to play a sound effect multiple times). For instance, the sound effect for scoring was a siren-like tone that lasted about half a second, and was being looped through by the module. During the loop, there was an apparent quietness before the sound played again, which was not the desired effect. As a result, the maximum address to be read was limited to 10000 for all sound clips as they all lasted about a second. This helped fix the glitch and allowed repeated playback to sound as intended.

Lab 5 module: This module is of similar structure to the Lab 5 labkit setup. It instantiates two modules that help interface with the AC97 codec:

1. **AC97 module:** This module defines data going to and being received from the AC97 chip and how it is generally transmitted across the interface.
2. **CMDs module:** This module initializes the commands for talking to the AC97 chip in hex radix.

The above defined submodules maintain synchrony with the AC97 chip as audio data is being transmitted across the codec interface. Altogether, the Lab 5 module acts as the middle-man between the SFX module which selects the appropriate sound data to be transmitted to the AC97 before sound is played back.

Implementation Insights

1. **Limitations and Pitfalls**
    a. The approach taken towards designing the audio module is limited to playing back only short sounds. Even though there are 144 blocks of BRAM on the labkit used for this project, there is not enough memory in a single block of RAM to store large sound clips like songs. While this was not an issue for this project in particular, it may be an issue for a project where the designer intends to play long clips of sound for several minutes, for example background music. This design approach rules out the possibility of a soundtrack for the game as a result. An alternative would be to use external flash storage for larger sound clips. This way, the design could exclusively use flash storage for both short sound effects and long sound clips, or mix it up and use both —BRAM for smaller sound clips and flash storage for songs.
    b. When implementing this design on a different FPGA, this approach lacks flexibility. The designer would need to reload the sound data onto new blocks of memory on the new FPGA. Additionally, this different FPGA could be of a different brand (ex: Nexys instead of Xilinx) and come with different specifications for its block memory, which needs to be kept in consideration. When using external storage, however, the designer is afforded the versatility to very easily transfer their design to a different FPGA with the only setback being the cost of the external storage.
    c. A benign limitation is that the playback frequency is limited by the longer the *.coe* file for a given sound clip is. In this project's particular case, there is no disadvantageous effect; however, in a situation where, for example, a sound effect was supposed to play for a certain period of time in synchrony with a given repetitive action, playback would have to be cut short stay in synchrony, or the playback would become asynchronous. As a result, a shorter clip would be needed or the SFX module would have to be developed to accommodate for playback of multiple sounds simultaneously. A clearer example is if the sound effect for the ball bouncing off a wall was an echoing "boing," but the ball bounced faster than it took for one cycle of the "boing" to stop playing.

d. Along the same vein, another benign limitation is simultaneous playback. The implementation of the SFX module is based on a case statement which can only select one sound to be played. Luckily, the nature of our project does not require sounds to play simultaneously. For instance a paddle cannot hit the ball at the same time the ball is bouncing off the wall, or at the same time a point is being scored. A possible solution to this would be interleaving the playback, shuffling between the different blocks of memory whose data is sampled for playback.

2. **Lessons Learned**
   Ultimately, the audio module was never successfully integrated into the rest of the project as it was not considered as big a priority as achieving two-player, and testing and debugging two-player took too long to allow for the eventual integration of the audio module. Nevertheless, there were some positive lessons that arose from designing the audio module:
   a. **Modular design and abstraction allows for easy testing.** The audio module only required flags as inputs for when specific events happened, instead of large chunks of data to be processed, for example the coordinates of the ball and paddle to check for a collision. This made the audio module very easy to test. Debugging was done by using debounced labkit button inputs as dummy triggers for specific events, and inspecting playback for correctness. This allowed for the quick and easy identification of bugs, for example sound clip playback beginning in the middle of the memory block and not at address 0.
   b. The audio module should have been done last. Quick and easy testing allowed for the audio module to be polished for high quality function, but took time away from modules that actually mattered and were more central to the gameplay, like the inter-FPGA communication interface. The audio module does not fully define game play, and while it does make gameplay more realistic, it admittedly is not high on the priority list. Prioritization is important for one not to get distracted by all the interesting, yet not-as-important facets of this project.

Overall, implementing the audio module offered many lessons about how to design a simple system with high quality output, but took the focus away from the bigger goal that was integrating the entire system.

Inter-FPGA Communication Interface - *Paul Kalebu*

Overview

As shown in the higher-level overview of the entire system, the second FPGA only runs a graphics module for the second display, and an audio module to allow for stereo audio, while the first FPGA processes player input by camera and runs a physics module to determine object positioning within the game. We eventually switched to using button inputs on the second FPGA for player 2's gameplay as we were unable to acquire a second camera for player 2. While the game can very easily be played by one player, the gaming experience is much more entertaining with two players. To enable multiplayer mode, the game had to run on two FPGAs

to allow for display on a second monitor. The communication interface was designed to transmit four sets of data: coordinates for paddle 1, coordinates for paddle 2, coordinates for the ball, and audio for stereo. Within the final project timeline, the inter-FPGA communication interface was scheduled to be implemented in the last week after the one player version of the game had been finalized. Perhaps we underestimated how long it would take to integrate the module with the rest of the system. The general approach to inter-FPGA communication was wiring the necessary number of user I/O pins between the two FPGAs all in parallel. This approach had been tested on the audio module for stereo and had worked out fine, but a possible cause for failure upon integration is that the signal integrity was lost due to the high speed communication. Technical approach and implementation insights are all described further below.

Design Alternatives

There were two obvious choices for implementing the communication interface: parallel and serial.

1. *Parallel*

   This approach was the simplest in terms of digital design for synchrony, as there was seemingly none required. However, it was the most cumbersome in terms of hardware setup as it required a great deal of wiring. In terms of Verilog code, all there was to do was write assign statements to assign values to individual user I/O pins on the labkit. Additionally, because the setup was in the last week, this approach seemed to be the most straightforward in terms of debugging since all there was to do was check for signal propagation across the FPGA interface. This approach was chosen to save time as it was easy to implement and integrate after wiring. Under testing, the approach seemed robust and reliable. However, under integration with the game graphics, the approach failed to work. While the main reason for failure was not discovered, it is believed that the approach breaks down with faster communication, as this was the only difference between the testing environment and the integration environment.

2. *Serial*

   This approach was the least cumbersome in terms of hardware setup as it required very little wiring, but required an entire UART protocol to be designed for communication. The approach was not chosen as it would have required a lot of time to set up and debug the UART protocol with very little visibility of the signal propagation; however, in hindsight, perhaps this approach should have been chosen as the primary method of implementation, and the parallel approach chosen as the fall-back incase serial communication had not been achieved. Additionally we are uncertain how well the serial communication would have held up under high-speed communication for the graphics display.

3. *UART Cable*

   An approach we did not consider until it was too late is using a UART cable to communicate between both FPGAs. The cable is designed for this exact application and can is guaranteed to be reliable under high speed communication. If given the

opportunity to redo the project, this would be the first consideration for implementing the communication interface across two FPGAs

The next section goes into detail of how the parallel communication protocol was designed.

Design description

When wiring up the user I/O pins across both FPGAs, one FPGA was rotated 180 degrees such that the top sides of both FPGAs were facing each other, so as to make the wiring as short as possible to avoid any signal loss. This brought up the issue of reversed connection, i.e pin 31 on FPGA 1 was wired to pin 0 on FPGA 2. This was unavoidable if we wanted direct wiring. As a result, we had to design a handshake module to facilitate a reordering of bits abstracted away from the other modules involved with the inter-FPGA communication. The tables below depicts how the user I/O pins were connected.

1. **Paddle 1**

Data Set	**Tx (FPGA 1)**	**Rx (FPGA 2)**
x1,y1	user2[31:11]	user1[20:0]

2. **Paddle 2**

Data Set	**Tx (FPGA 2)**	**Rx (FPGA 1)**
x2,y2	user2[0:20]	user1[31:11]

3. **Ball**

Data Set	**Tx (FPGA 1)**	**Rx (FPGA 2)**
xb,yb,zb	{user4[31:11], user3[31:20]}	{user3[20:0], user4[11:0]}

4. **Audio**

Data Set	**Tx (FPGA 1)**	**Rx (FPGA 2)**
audio	user3[19:18]	user4[13:12]

For each data set, the bits were reordered by creating bitfix modules based on FPGA and whether it was transmitting or receiving. For example, bits transmitted from FPGA 1 were reordered in the module defined below:

```
module bitfix_tx1(
                input [20:0] paddle1,
                input [32:0] ball,
                output [31:0] user2, user3, user4
                );
```

User I/O pins were assigned to inputs from the physics module that had been ordered in more intuitive order. This allowed for modularity in integration with the rest of the system. Testing was then done by transmitting data across the interface to light up the same LED on both FPGAs. While this was successful, similar success was not seen upon integrating with the rest of the system.

Implementation insights

This module proved more important to the success of the project that had been imagined. Unfortunately, not enough time was spent on ensuring that it functioned correctly under conditions similar to those put in place by the game itself, i.e the transmission speed for the graphics module. Even with failure, there are several insights that made implementation a lot easier:

1. Shorter wiring distance was used to minimize noise across the communication interface.
2. The wiring was grouped by coordinates and colored differently according to which group of user I/O pins it was connected to to facilitate for easier debugging.
3. While cumbersome, testing was iterated through all the wires to test for signal integrity. Even with the module's failure, this allowed for elimination of other causes of failure besides breakdown at high-speed communication.

Lessons learned

Given the opportunity to redo the project, I would reevaluate which modules have the most impact on the project's success. This would allow for more time to be dedicated towards them being implemented well, as they hold most of the project together. Furthermore, better time management and distribution would have enabled us to foresee being held back by long compile times and dedicating an earlier slot to work on the inter-FPGA communication interface.

In addition to reevaluating prioritization and better time management, better testing would have enabled us to identify the issues with the parallel approach much earlier. Had the approach been tested with communication signals at a frequency similar to that of the the graphics frame rate, perhaps there would have been time to pivot towards a different solution for a multiplayer gaming experience.

Takeaways

One of the biggest challenges we stumbled upon in developing the game was in incorporating all of our separate modules together towards the end of the project cycle. At the start, we partitioned the work in such a way so that each extra person's

contribution would build on top of what was already built. In particular, Louis modified the NTSC camera code to display colored video, and developed the paddle tracking. The paddle tracking outputted in-game coordinates, which would be fed directly into the physics module, which Louis also developed. The next step, the graphics generation was done by Miguel. After the completion of the graphics module, we got together and incorporated the two parts into a usable game; the graphics module allowed the gameplay to be visible, making concrete the algorithms in the physics module. The final steps involved extending a single player version to a two-player version, as well as incorporating an audio module, which allowed for in-game sounds.

Unfortunately, we were met with unforeseen bottlenecks in the development process. Unlike projects with a small number of moderately sized modules, the game we developed had many, non-trivial modules which required lengthy compilation times. With the compilation times hovering around 15-20 minutes, we found that there was much idle time. Whenever small bugs were made, the entire project had to be recompiled from scratch. We were attempting to follow an incremental build model, where we implement and test incrementally, but we underestimated the amount of time we needed to set aside to accommodate the aforementioned build times. Furthermore, interfacing between two FPGAs proved to be harder in reality than on paper; we didn't write code that was sophisticated enough to resolve timing issues that would arise during the transfer of information between the two FPGAs. In particular, such a protocol would serve as reasonable grounds for future work.

In terms of teamwork, everyone contributed a fairly equal amount of work. However, we made the mistake of not finishing a minimum viable product by a personally defined checkpoint, and instead, having to do everything at once, during the span of the last two weeks. Essentially, already having a minimum viable product would allow time to reiterate on things that went wrong the first time.

Conclusion

We successfully implemented a playable 3-dimensional pong game. The first player could control an on-screen paddle with a colored object, which in this case, was blue. The choice of color can be easily programmed in our code. A clean, 3-dimensional graphics representation was successfully developed, rendering the gameplay in a realistic manner.

However, we were also met with challenges, more so in the area of effectively managing the timeline of our project. We implemented a number of modules that were ready for integration with the final product, such as an audio module that would enable

sounds for paddle and ball collisions as well as a suite of FPGA-to-FPGA communication protocols.

Despite the challenges we faced, we found this experience to be highly instructive. We found the process of implementing certain modules, such as the object recognition and graphics modules, to be very interesting, and we came away with a better understanding of how to utilize the FPGA as a bedrock upon which external devices could be used with.