Barkley Ball

A Fall 2009 CS150 Project

by

Anthony Glenn Demarco

Loring Scotty Hoag

Table of Contents

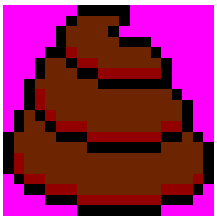

Figure 1: Poop

Abstract

Barkeley Ball is a fun-for-all-ages variation of the classic arcade game Pong using modern technology and the retro sights and sounds of old 8-bit era computer games. It was conceived as a student project for the class "Computer Science 150: Techniques for Digital Design" at the University of California at Berkeley in the Fall of 2009. The game was designed for the Xilinx Virtex-5 ML505 Evaluation Platform FPGA. Control input is obtained through a camera which tracks the position of brightly colored objects held by the players and allows the paddles to be mapped one-to-one with the players' real-world movements. A colored screen placed behind the players is replaced with a colorful backdrop onscreen to make the players feel like they are immersed in the game world. Animated characters interact with the players by littering the field with power-ups to tilt the balance of power. Users can also directly change in-game parameters via a terminal command line interface accessible via a serial connection. Several sophisticated compression techniques were employed to enable large amounts of graphic and sound content to fit within the harsh memory constraints of the platform. This document will detail all of the major components created for the game.

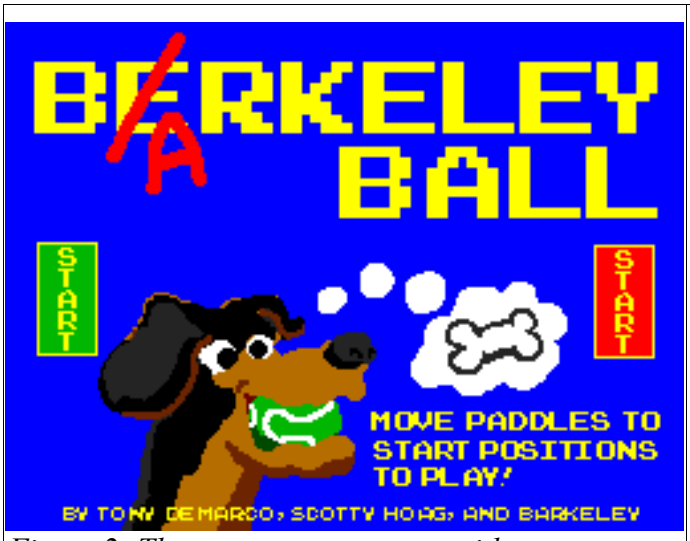

Figure 2: The most awesome game title screen, ever.

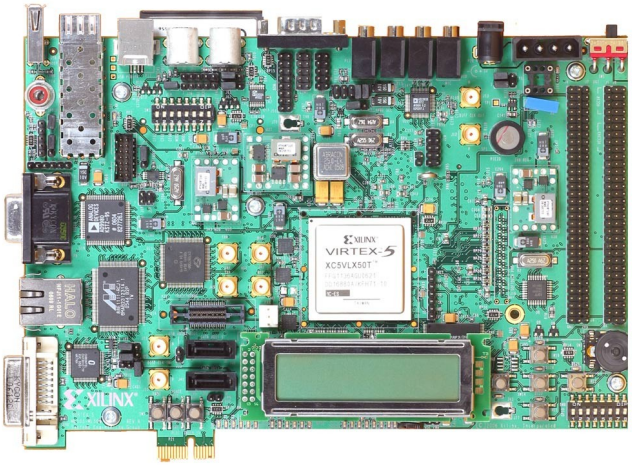

Figure 3: The Xilinx Virtex-5 ML505 FPGA. This is the hardware platform in use for Barkeley Ball.

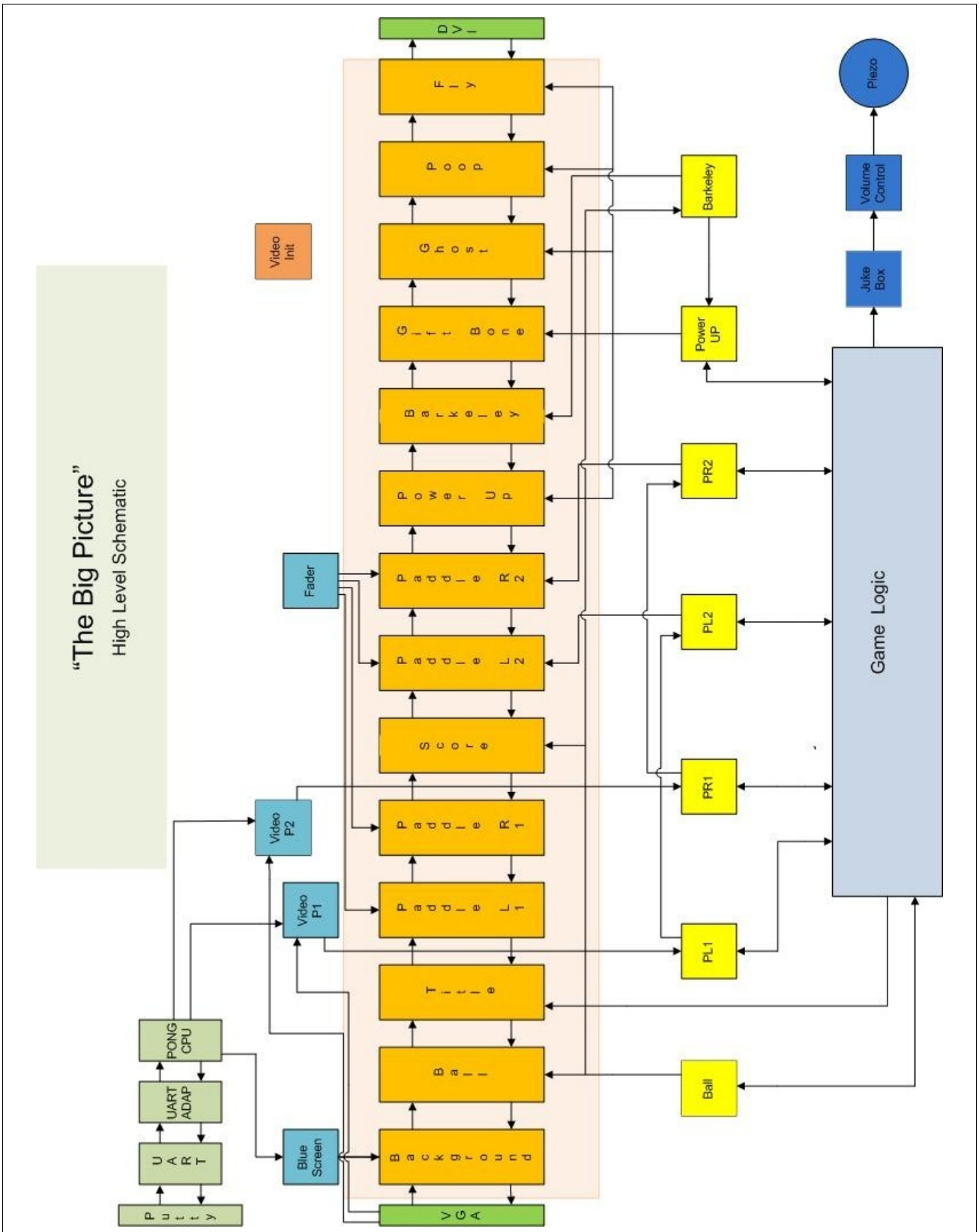

Figure 4: "The Big Picture." This is a very high-level data flow diagram of the entire circuit.

Graphics Pipeline

VGA Data:

There are two chips that must be properly initialized if any of the video processing is going to work. The AD9980 is initialized by the I2C bus that sets the registers to make it so the AD9980 can talk to the CH7301C. We initialize the chip for a 70Hz 1024x768 pixel signal. The I2C bus also initializes the AD9980 so that the pixels are being sent in a 75MHz stream. We set the clock to that frequency so that we can be sure that we are processing one pixel per clock tick. There is a handshake that occurs between the VGA and the DVI that must happen before the valid pixels will be read. We send VideoValid when we are ready to send pixels so that a handshake can occur. Upon receiving the VideoValid signal, the DVI sends the VideoReady signal to back to the AD9980 which tells it that the DVI chip is ready to receive valid pixels. The DVI only sends it once after a reset because it only needs the signal once.

Indexing the pixels is fairly straight forward, as we know how many pixel rows and columns that we have. We can count how many pixels and reset every 1024 pixels for the horizontal pixels and increases the vertical index by one. As long as the pixel is outside the Blanking region, it sets the predicate inWidth and inHeight which generates the VideoValid signal. The challenge is to deal with the Blanking regions. At the end of each frame, the old CRT screens would need time to reset but our LCD screens do not. We have to compensate for that by determining whether we are sending a valid pixel or a pixel that exists in the blanking region. There are both vertical and horizontal Blanking regions that the VESA standards should give us values that accurately compensate for the blanking region but they do not appear to be enough. We had to introduce additional offsets into the boundary limits to get it to work properly. There are several reasons why this would occur. One possibility is the differences in the screens could require more analog to digital conversion than we were had anticipated creating the need for an adjustment. The ChipScoped signal appears just as it should but a horizontal offset is the only thing that makes the screen appear correctly. If we do not include the horizontal offset, then we print the blanking region pixels.

We pass the X and Y coordinates that we generate in the VGA module to position every compositor in the design. The coordinates are how the compositors know where to draw the pictures on the screen and which pixels to access from memory so accuracy is important. The X and Y indexing methods are much more flexible than simple pixel counting. It also takes considerably less logic and is far more modular because the compositors already have meaningful data rather than having to breakdown an pixel number down to an XY coordinate; or in an even sloppier implementation, just hardcode all of the predicates.

Compositor:

The Compositor is a module that prints an image at a position on the screen. It does this by taking in a pixel and the index from the previous stage and the position data from the Game Logic to determine whether to switch the pixel from the previous stage for the pixel from memory or to pass it through. The predicate InCompBounds checks the dimensions of the image is the inbounds and when to write the picture to the screen. The dimensions are fed in as parameters and the position is indexed from the Block RAM that is instantiated inside the module. After the math and pixel switching occurs

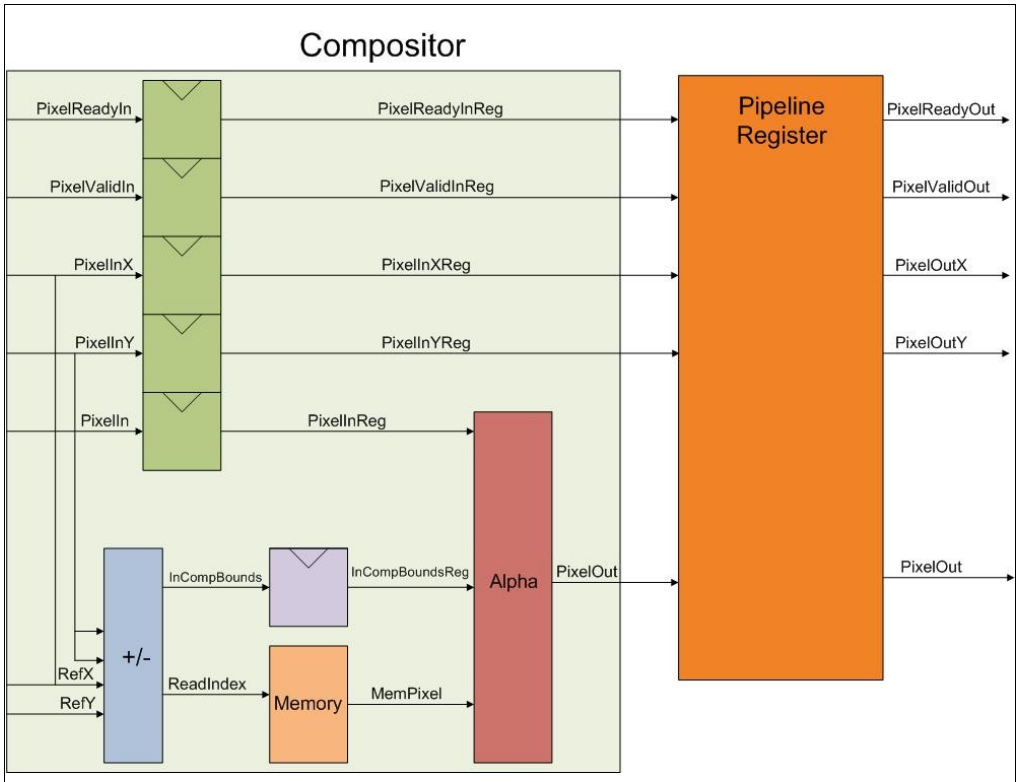

Figure 5: The general layout of the Compositor type modules.

the signals are then propagating the signals to the next pipelining stage.

The Block RAM contains the hexadecimal values of the picture that is to be drawn on the screen. The indexing that is fed into the compositor from the previous stage is also what is used to index the memory correctly. However, the data is synchronous. So to ensure that the correct pixel is being indexed from memory, some of the pixel data must be delayed by one clock cycle.

There are two ways that the invisible pixels are handled. One method is that we pick an extremely ugly color, like magenta, as the color that we choose not to draw. By not drawing those pixels, they appear to be transparent which allows us present a Ball that is actually a square but the non-ball pictures are just not drawn. The other method we use is Alpha Blending. This is a mathematical operation that allows us to fade images. Using pixel-by-pixel averaging, we perform an operation on each of the RGB parts of the pixel that gives the impression that the compositor has been faded. This can be observed by some of the power ups disappearing or the fading of the title screen when the paddles are at the starting position.

There are different types of compositors that are used for different purposes. We have used different compositors (32-bit, 4-bit, 3-bit, and 2-bit) so that the amount of RAM could be optimized. Then we use a color map that has been discussed under the Optimizations Section of this Document.

The Compositors also perform different functions. There are still compositors that are stationary such as the title screen or the background. Others take contain more than one block of memory in some Game Logic data to choose which block to index. The Score Board uses this compositor to print what the score is. Sprites are longer than normal images that have the animated pictures later in the image. As the frame clock cycles, so does the image. It gives the illusion of motion by shifting the boundaries of the memory that is being indexed. Some of the Sprites are used for the animations such as the Ball rotation or Barkeley walking across the screen.

Frame Clock:

All of the animations and moving compositors use the Frame Clock. It is simply an edge detector that detects when the Vertical Sync signal goes low and asserts the Frame Clock signal high for a cycle Pixel Clock. This ensures that all of the animation positions are updated when the frame is in the blanking region. This ensures the animations look smooth and prevents jitter from occurring.

Tracking Module:

The tracking module takes in the data from the camera and finds pixels that meet the threshold requirements. These requirements can be altered in real time by the PONG OS interface to help set color thresholds for different settings with varying lighting. There is a low and a high threshold for each of the 8bit color segments of the 24bit pixel that is being fed in from the previous stage. The color values of the pixel must be higher than the low value and higher than the low value that is being taken in. This determines whether the pixel that is being fed in is the pixel that we are looking for and if it should be counted or not.

If the pixel is the right color, then the pixel's X and Y coordinates are added to a register and the pixel is counted. At the positive edge of the Vertical Sync signal, the division module begins. This module takes 8 cycles from the positive edge of the Vertical Sync signal to calculate the position of the paddle. It updates the position based on the sum of all of the pixels that were within the set boundaries. This ensures the position will only be updated in the blanking region so that the animation appears smooth. The sum of all the X and Y positions are divided by the pixel count to give a pretty accurate estimate of the position that updates once per frame. On the Frame Clock, the position is updated and the counters and position sum registers are cleared, but the positions are being held by a different register. This register maintains the correct value and feeds the position data to the paddle compositor until the next frame.

Figure 6: The players standing in front of the blue screen. Pixels with similar RGB values are replaced with a background image.

Figure 7: The blue screen replaced.

BlueScreen:

Blue screening is a relatively simple effect that adds a feeling of being "in the game" The Blue Screening occurs much the same way that that the paddle tracking does. It has a low and high threshold that is fed in from the PONG OS that is used to determine whether or not the background picture of Barkeley's house is printed or not. By hanging a blue sheet behind the players, the players are printed without printing the sheet as long as the sheet's color is in the acceptable threshold. This effect makes the players appear to be in the game.

Audio:

The audio is implemented by passing a signal at an appropriate frequency through the piezo speaker. This is done through the use of two modules; the Jukebox module generates all of the signals that are played through the speaker and the Music module reads the notes and tells the Jukebox what note to play and for how long. The Jukebox is a multiplexer that takes in a note and plays it by selecting the correct signal to send to the piezo speaker. The signals are generated by taking in the pixel clock using a counter to step down the frequency to a note that is in the either the 6th or 7th octaves. All together, there are 24 notes that can be played so it gives a realistic range that can play almost any song. The Music module has eight memory blocks that contain hexadecimal files that play a note for one eighth of a second for each note that is in the file. The module takes in a song as a selector and the module plays the song that has been requested. That module then moves into its singing state until either the song is complete or the mute signal is asserted. If the mute signal is asserted, then the music will stop. The music also has a stop note that is the silent note and tells the state machine to return to the silent state.

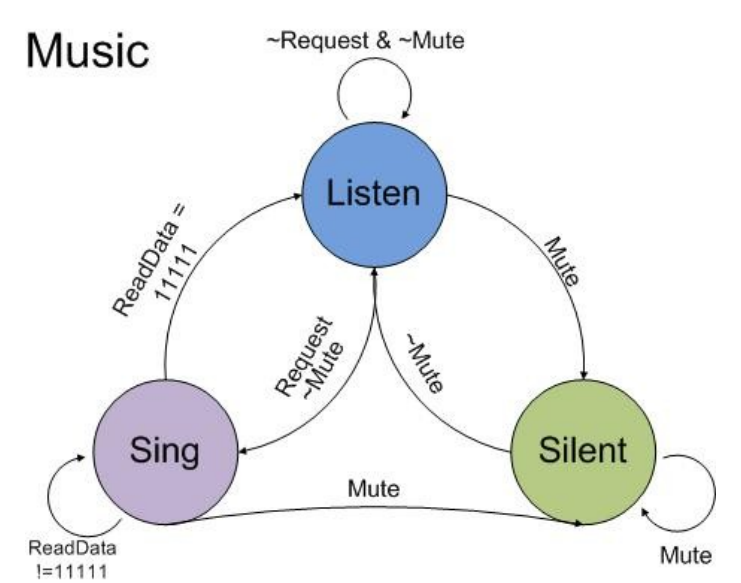

Figure 8: The FSM for the music controller.

Game Logic

The game is made up of three main components; The ball, the paddles, and the power ups. The GameLogic module ties them all together and also performs the bookkeeping involved in collision detection, activating power-ups, playing music, and starting and ending the game. The diagram below shows how each component is linked together via the GameLogic module.

Ball and Paddles

The Paddle modules takes in a set of coordinates from their corresponding Video module on inputs InX and InY as the suggested center point of the paddle, adjusts for values that are outside that player's field of reach (Outside the bounds of the playing field or past the center divider), and outputs fixed coordinates for the top left corner of the paddle on PaddleX and PaddleY at the rising edge of the frame clock. Not shown is the manualControl signal, which acts as an override to camera controls. When high, vertical movement occurs at a fixed speed in response to moveUp and moveDown input signals instead of the input from Video. This allows FPGA buttons to control the paddles for testing.

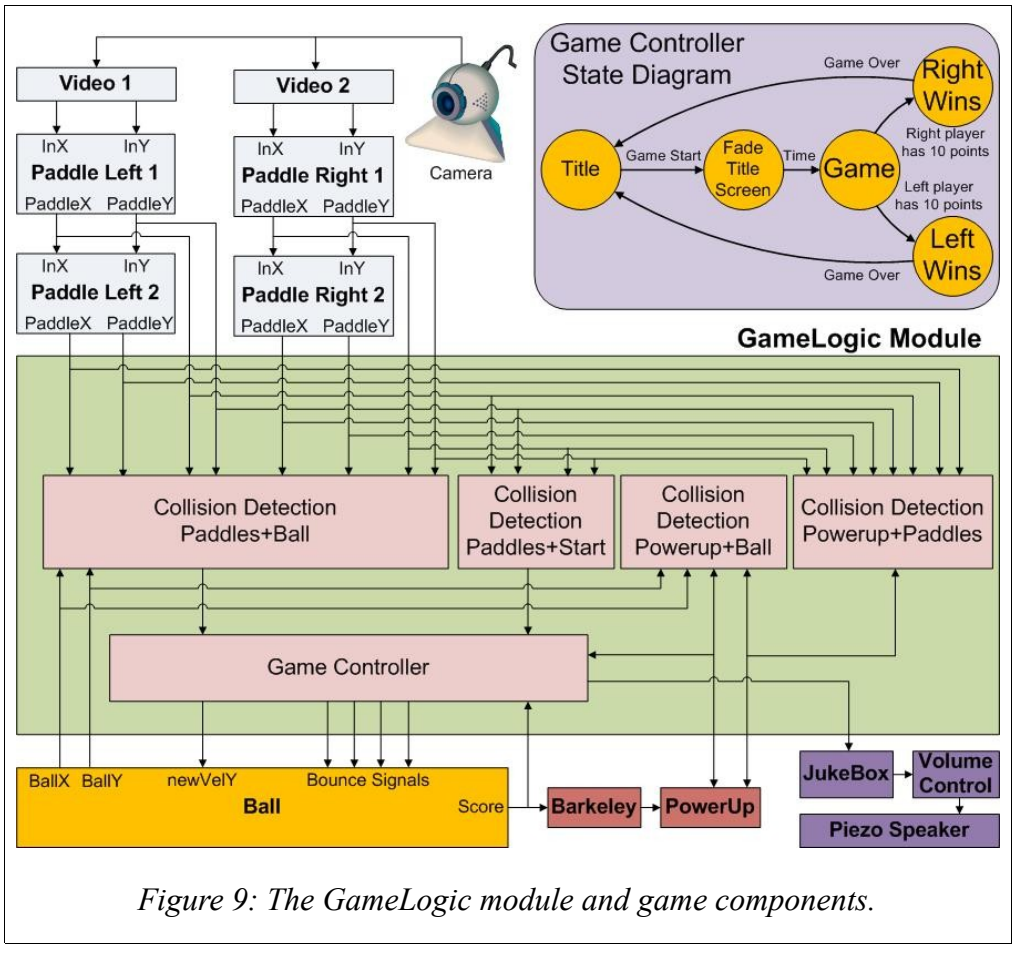

Figure 9: The GameLogic module and game components.

The Ball updates its horizontal and vertical coordinates on the positive edge of the frame clock. If none of its “Bounce” inputs are asserted, then it just adds its velocity values to its coordinate registers. If BounceLeft is high and the ball is traveling to the right or if BounceRight is high and the ball is traveling left, the ball will change its horizontal direction (Negating its horizontal velocity in the process). Likewise, the BounceUp and BounceDown signals cause the ball to change its vertical direction. Like the paddles, the ball also knows the dimensions of the field. If the ball would be past the top or bottom playing field boundaries given its current velocity, then the ball is moved to the respective field border and its vertical direction is changed. The ball's horizontal velocity is initially set to a small value each time the ball is served (This parameter is two pixels per frame by default) and increases with each BounceLeft or BounceRight signal until reaching a preset maximum speed. The ball's vertical velocity is set to the value of newVelY upon each high BounceLeft or BounceRight signal. This allows the angle of the ball to be calculated by an external source and more accurately respond to collisions (See figure 10). The fixed coordinates of the ball's top left corner are output on BallX and BallY.

GameLogic

The rest of the collision detection is done inside of the GameLogic module. There are nine cases that need to be checked to determine whether two objects have collided or not. The left side of figure 10 shows each case graphically using the ball and paddle as examples. Each corner of the source object (Ball) is checked to see if it currently exists between two horizontal and vertical corners of the target object (Paddle). The case in the center of the diagram where none of the source's corners are between the target's corners can be checked by swapping the source and target objects in the comparison. While the diagram only shows the ball and paddle, the same technique is used to test collisions between all objects that can interact with each other such as the paddles and power up items.

In the special case when the ball bounces off of a paddle, the GameLogic module also calculates a new vertical velocity for the ball. Similar to the original arcade Pong, the ball’s vertical velocity is based on the ball’s position relative to the paddle upon impact. As shown on the right hand side of figure 10, the paddle is split up into discrete segments. The green arrows approximate the direction that the ball will travel in after a collision based on which segment the ball’s center point (The red dot) lines up with most accurately.

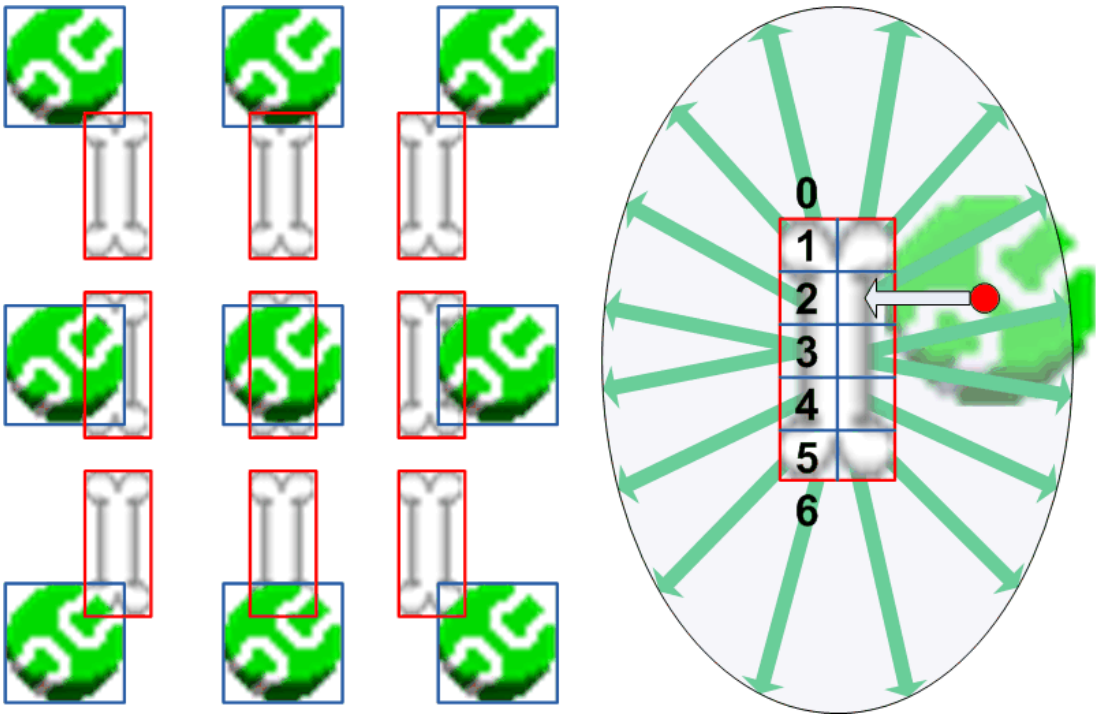

Figure 10: Collision detection and the bounce mechanic for the ball and paddle.

Barkeley and the Power Ups

Power ups are triggered by Barkeley the Dog who walks by whenever his Appear input is asserted. This occurs whenever either players' score is equal to three or six. Barkeley is a simple FSM that uses the value of the players' scores as a trigger. As Barkeley passes the center of the screen, he outputs a high signal to the PowerUp module on its Appear line. The signal stays high until the happy dachshund walks off screen.

The PowerUp module itself is a bit more complex. It slowly floats to the top of the screen by incrementing a counter. It checks for the top of the playing field the same way that the ball and paddles do and, upon hitting a boundary, changes directions. When a collision with the ball is sensed on the input HitPresent, the power up stops drawing the present picture and instead draws a randomly selected item with the same x and y values. When a collision with the item and a paddle is sensed on the input HitItem, the PowerUp begins to decrement a counter that is output as an alpha value on the output FadeCounter. When the counter has reached zero, the PowerUp sends low draw signals to all item and present compositors and waits for the next Appear signal. The GameLogic module keeps track of which paddle actually collided with the item and who the power up item affects.

PongCPU

It became apparent that a quick way of changing game parameters would greatly speed up testing by allowing variables to be changed and tweaked without having to reprogram the board. The PongCPU module began as response to this concern, but quickly grew into a powerful real-time interface between the user and the FPGA. The keyboard information is sent to the FPGA through a serial connection and is received using a UART interface. The UART then passes the information to the PongCPU via an adapter using a Ready-Valid FIFO handshake protocol. Several ports on the adapter are mapped to dedicated input/output pin memory addresses which the PongCPU can read from or write to. In particular, the PongCPU asks the adapter for permission to read information when in the "Try to Read" state and for permission to write in the "Try to Write" state. The adapter uses these requests as signals for the FIFO handshake and passes information back and forth accordingly. This exchange is detailed in the diagram below.

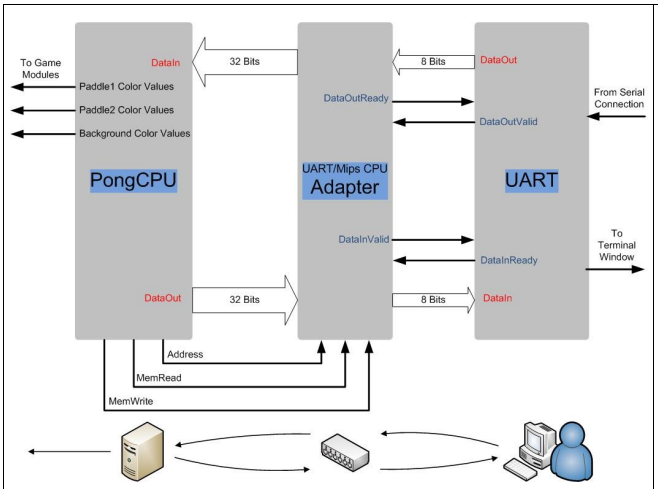

Figure 11: The High-Level FIFO Handshake.

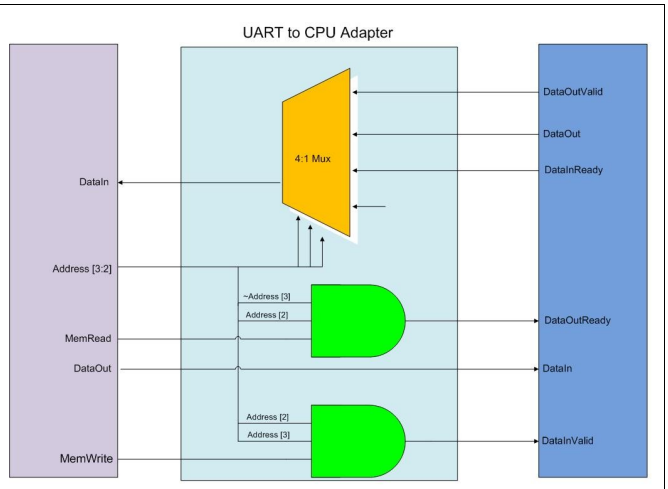

Figure 12: The Low-Level FIFO Handshake.

The input character is placed onto a buffer for parsing. If a command is not currently being processed and the last key pressed was the "Enter" or "Return" key, the contents of the buffer are checked to see if they match any of the known command words. If there is no match, the command prompt is printed on the next line. This is done by filling an output buffer with a string consisting of a carriage return, line feed, and the characters that make up the prompt "PONG-OS>", and then setting a write-cycle counter to the number of characters matching that string. This counter causes the PongCPU to loop through its "Try to Write" and "Write" states while sending out characters one at a time from the output buffer to the terminal. If a command is matched, the output from the command is printed the same way, but future input is converted from ASCII to the format expected by the command. After the

expected number of characters has been typed, the specified register values are updated and the PongCPU reverts to printing the command prompt and parsing commands. The dual state machine logic is explained below.

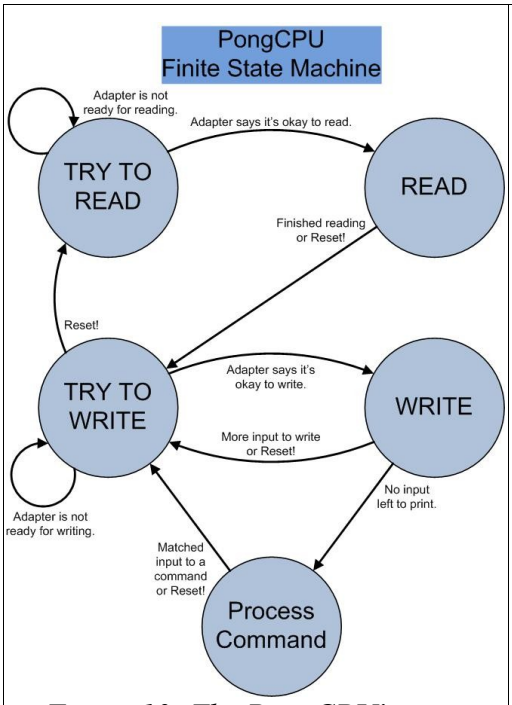

Figure 13: The PongCPU's state transition diagram.

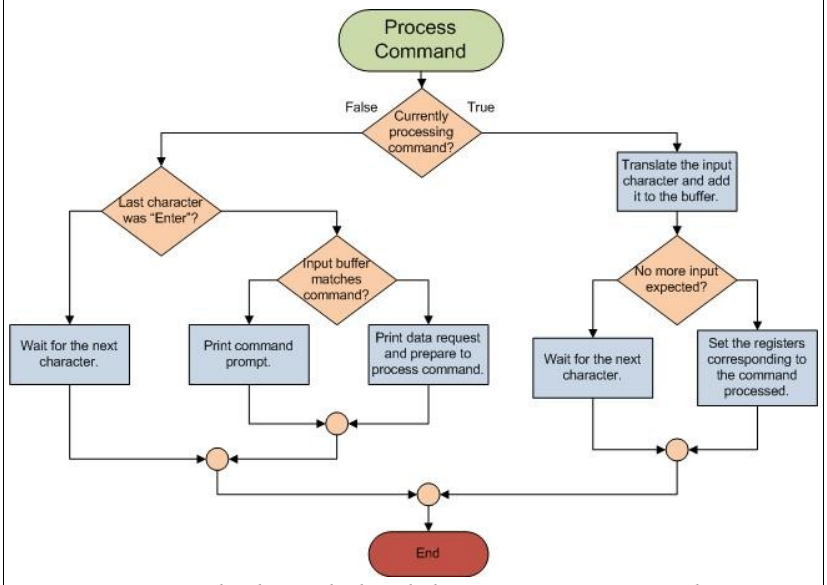

Figure 14: The logic behind the ProcessCommand state.

Design Metrics

LUT Usage

The main two resources on the board are LUTs (Look Up Tables), which implement all of the transistor level logic, and BlockRAMs which can be used as memory to store game assets like pictures and music. The amount of these resources used increased dramatically during the last few days of coding, which is due to the large amount of new content added during that time. The quickly approaching deadline meant that optimizing these new features for space held a much lower priority than just getting them to work in the first place. The LUT usage of the entire project weighs in at 11,303 Slice LUTs. As for memory usage, the game uses 30 18 kilobyte BlockRAM units – 540 kilobytes of the 5,328 kilobytes available – which is roughly 10% of the BlockRAM resources.

The GameLogic module makes up a large portion of the LUT usage with 3,684 Slice LUTs, which in itself is about 5% of the Virtex-5's total LUT resources. This is due to all of the comparisons being made to perform accurate collision detection. As shown in figure 9, each pair of objects that can collide currently has its own dedicated collision detection unit. This could be optimized by sharing collision detection units amongst several objects during the same video frame.

The two Video modules which calculate the position of the paddles take 857 Slice LUTs each for a total of 1,714 Slice LUTs. Each one performs complex division operations in order to approximate the center of its target color on screen, which is expensive in terms of hardware usage. One major optimization that was implemented is in how the dividers are set up to perform the division over eight pixel clock cycles rather than one. This saved around 4,000 LUTs and had no noticeable effect on performance.

The third major source of resource usage comes from the compositors. For example, a single compositor module takes up somewhere between 130-230 Slice LUTs and about 1 BlockRAM unit (18 kilobytes). Also, each PipelineReg module contributes another 90 Slice LUTs. While this may not seem like much in when compared to the modules mentioned above, there are 14 compositor-pipeline reg pairs contributing about 300 LUTs each. This translates to roughly 4,200 LUTs.

Graphics Compression

In order to fit the amount of content we wanted onto the board, we had to heavily compress our images. We did this by borrowing two techniques from old video game consoles. First: we decreased the resolution of the image space. While we were working with a 1024x768 pixel resolution screen, we

treated it as if it were only capable of 256x192 pixels. This effectively made our compositor images appear four times larger than their actual size while still taking up the same amount of memory, Each pixel in a compositor picture with the lower resolution settings takes up a 4x4 pixel block on the screen, so a large image could be drawn with only 1/16 of the bits actually stored in memory. This effect was accomplished with simple bit shifting tricks and is therefore extremely cheap in hardware.

The other trick employed was limiting the number of colors in use. The images were going to use only a fraction of the full range of 24-bit RGB color any way, so bounding the number of colors representable made sense. Each compositor is created using a specific number of bits to represent only the color values it needs. An n-bit compositor can represent 2^n colors. Each compositor is created with a custom palette that map each n-bits to a color. For example, the tennis ball uses a 3-bit color map which allows it to represent 7 colors (One color is designated as the transparent color). Among these are several shades of green, white and gray. To use more bits to represent this image is unnecessary.

A python script was even written that took in an image and color map and output the compressed bit file. This allowed us to quickly convert images into whatever format that was needed. The script is included at the end of this document.

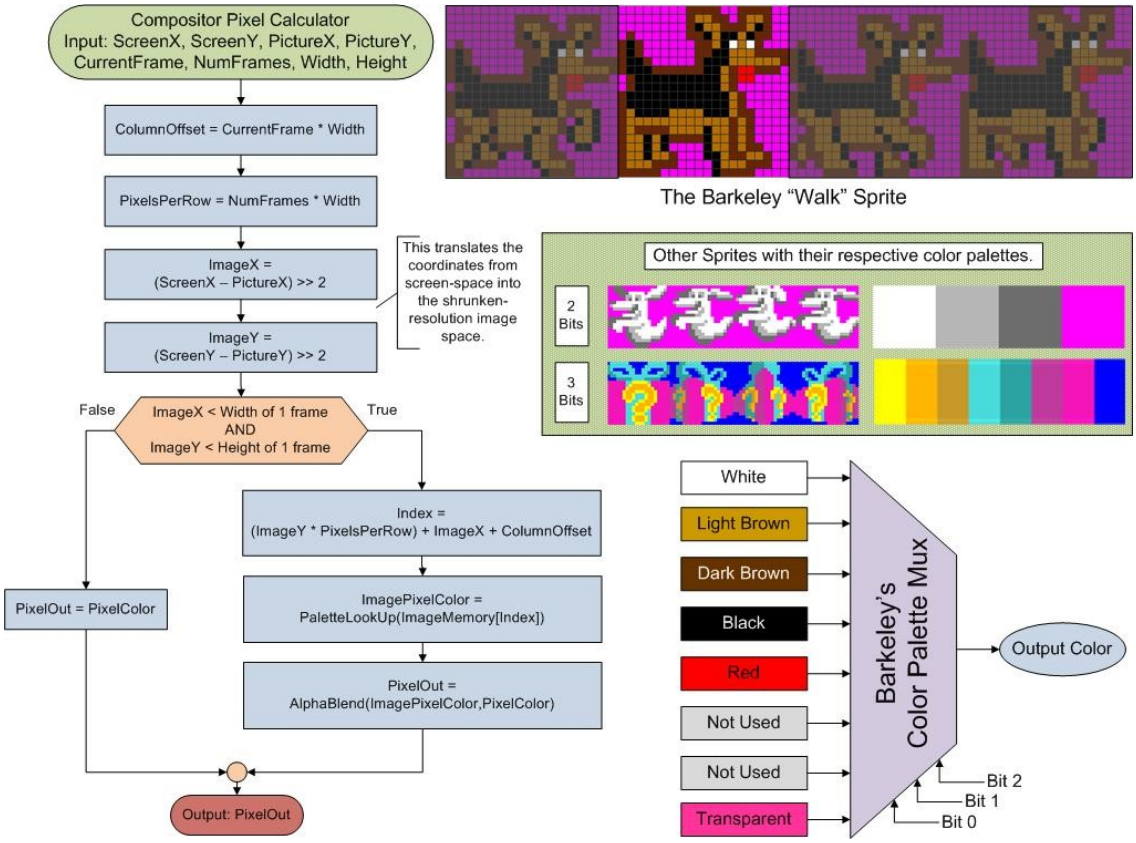

Figure 15: The compositor zoom and color mapping logic.

Conclusion

Suggestions for the Next Semester

We would like to start by saying that we had a great time in this class and we learned a lot. The TAs did an excellent job putting together the checkpoints and making sure that the expected progression between checkpoints was realistic. Switching to Pong over the Robots was probably a pretty good idea. We were initially disappointed, but after working with the FPGA we understand that it would have become a monstrous undertaking. However, we felt like sometimes the material for the class was just too much to cover in a realistic amount of time. We understand the CS150 has a reputation to maintain but with the project, homework and quizzes and the fact that we have other classes, it may have just been better to drop some of the material and go a little slower and emphasize more on the fundamentals. We also felt that the class material and the stuff that we did in the lab was more often than not disjointed. In the begging of the semester, there was a very close correlation but right around digital logic, the two diverged and never seemed to interrelate after that.

The biggest problem we had was with the cameras. We had lots of problems with auto adjust that made it extremely difficult to set correct thresholds. If you are planning to do anything with those cameras in the future we would suggest some type of work around or not to use them as input because of how difficult it is to get a stable input from the video. Blue Screening actually helped in this regard because we had a stable background which made it the camera far more predictable because of the constant blue background. We eventually figured out that if one of us wore light colors and the other wore dark colors then the camera would usually adjust correctly. The problem was that we spent far too much time fighting with the camera rather than implementing our project. The lab lectures were generally pointless. It really served no purpose and we don't understand why they were necessary. We saw a lab lecture from Spring 2004 that seemed like it would be the best blueprint for what a lab lecture should be like.

Figure 16: Sprite sheet.

The Last Word:

The project was fun,
But the best part was all the
Animated poop.

Figure 17: The "Dog House" background image used in Barkeley Ball. Who let the dogs out?