ADVANCED DIGITAL DESIGN
18-545 / FALL 2015

Bradley Plaxen
Carl Nordgren
Rohan Saigal

This project and all associated materials including designs, code, reports, and presentations were made for academic purposes.

The members of Team Centipede give permission to the Carnegie Mellon University Department of Electrical and Computer Engineering to recreate and use these materials for future classes.

For more information, contact:
Bradley Plaxen- bplaxen@andrew.cmu.edu
Carl Nordgren- cnordgre@andrew.cmu.edu
Rohan Saigal- rsaigal@andrew.cmu.edu

TABLE OF CONTENTS

1. PROJECT OVERVIEW

1.1. BACKGROUND

Centipede was an arcade game released in 1981 produced by Atari. It was played as a top down vertical shooter in which the player spaceship must defend itself from waves of bug-like alien invaders. Played using a trackball to move the player around, the game was innovative and wildly successful. It spawned dozens of ported versions on game consoles after its release, and it has since become a cultural icon within the archives of video game history.

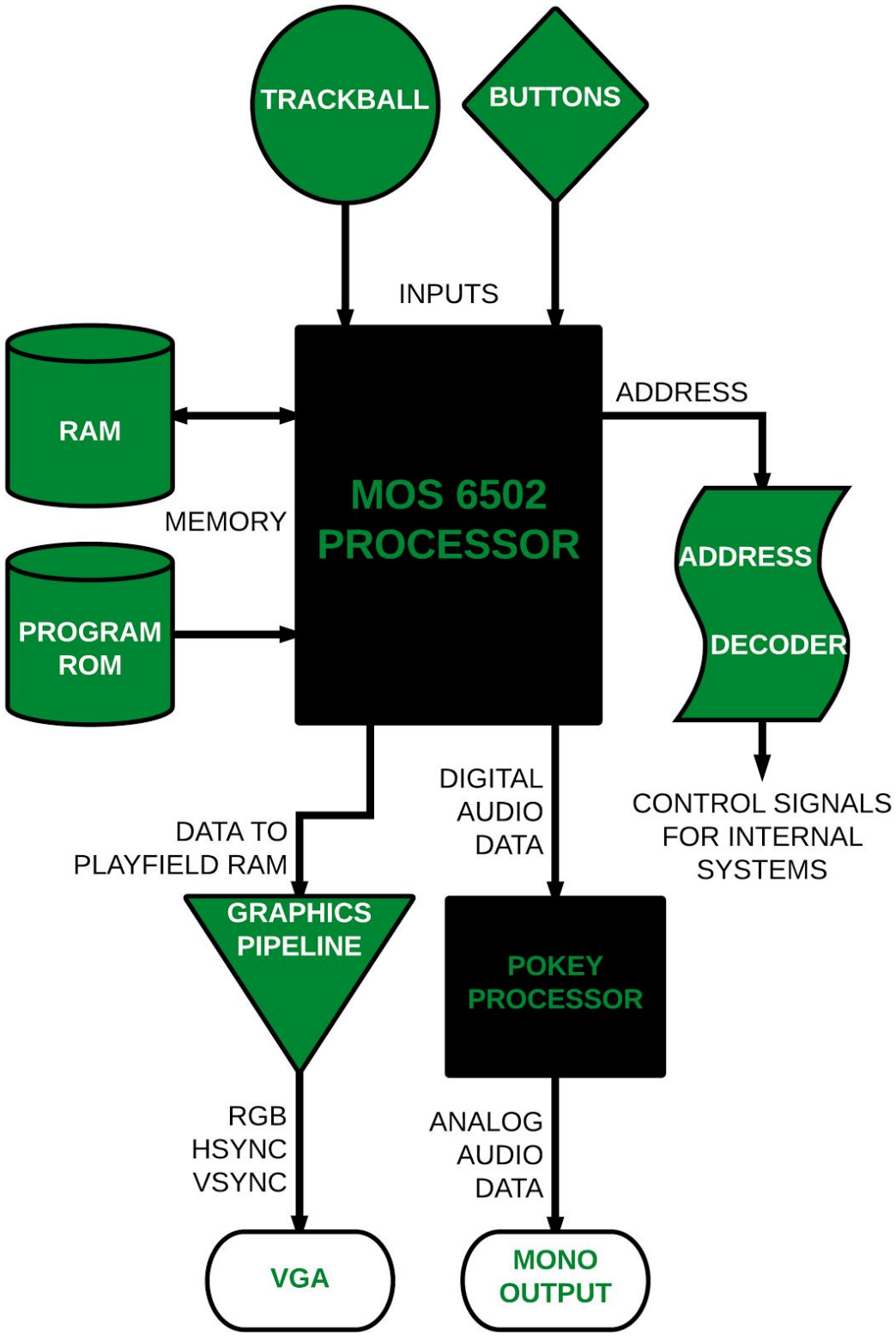
1.2. OBJECTIVE

Our goal was to recreate the hardware used to run the original upright arcade cabinet version of *Centipede*. This meant analyzing the schematics and specifications of the system, and implementing them on an FPGA system using a modern RTL description language.

1.3. HARDWARE DESCRIPTION

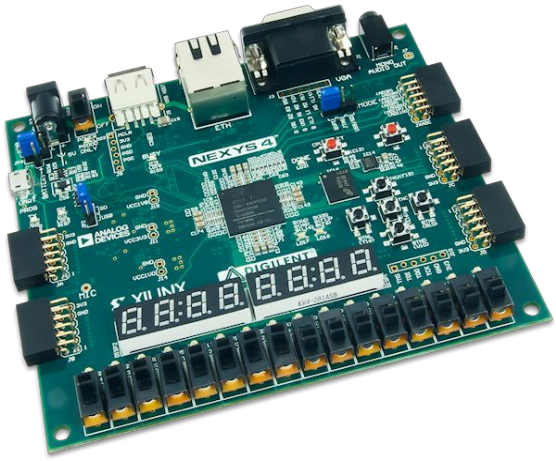
The original arcade cabinet motherboard can be broken down into five main subsystems: processing, audio, graphics, input, and memory. The core of the game's processing takes place on a MOS Technology 6502 processor which executes instructions and processes data from the game's core ROM. It periodically polls inputs from a series of buttons and a trackball. The movement of the trackball triggers a counting system which passes information up to the 6502. When the game needs to produce sound, it activates the Atari POKEY chip which sends square wave signals out through mono speakers. The graphics are processed using a frame buffered sprite pipeline that pulls icons from a memory bank and displays them at coordinates on the screen. All of these subsystems integrate using a network of control signals and data lines that ultimately allow the game to execute Assembly code successfully.

1.4. SYSTEM DIAGRAM



2. DEVELOPMENT TOOLS

At the start of the semester, we were highly encouraged to pick an FPGA supported by Xilinx because of its excellent forum community. After reviewing the needs of our project, we determined that there was no need for a software core or expanded hardware network. Because the project would be entirely contained within the single hardware core, it made the most sense to pick the board that would make development the easiest. This lead us to picking the Nexys4 which was simple, yet had all the necessary components to run a 1980's arcade cabinet motherboard. The Nexys4 provided us with a VGA connector, mono Audio connector, and enough memory and logic units to hold our system.



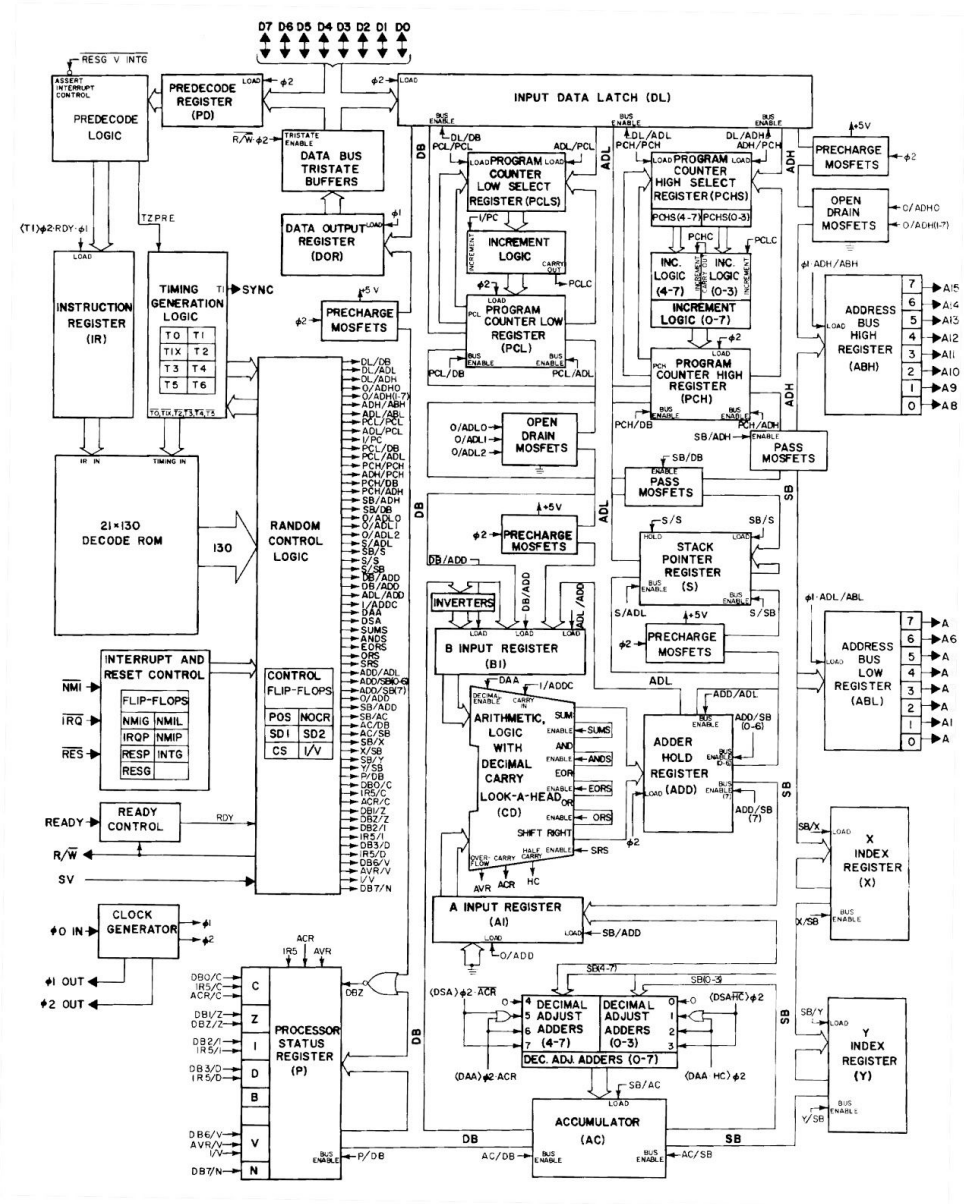
As we approached the end of the semester, we had implemented and synthesized all the individual submodules of the system successfully on the Nexys4. When we attempted to integrate every subsystem together, we ran into an error regarding the primary data bus of the hardware. We had implemented every databus interface as an "inout tri" type in SystemVerilog. After many hours of parsing through error messages in Vivado, we determined that the Nexys4 does not support internal tri-state buses the way we had implemented them. To get our top module functioning on the Nexys4 would have required recoding every tri-state interface as a multiplexer. Because of our limited time once this had been discovered, we decided to switch over to the Altera board.

The Altera board synthesized our data bus without issues, and Quartus, the software used to program it, compiled more quickly than Vivado did too. Although our project could have worked on the Nexys4 had we known about its limitations sooner, the decision to switch to the Altera board was a pragmatic one that ultimately sped up our debugging time at the end of the semester.

3. MOS 6502 CPU

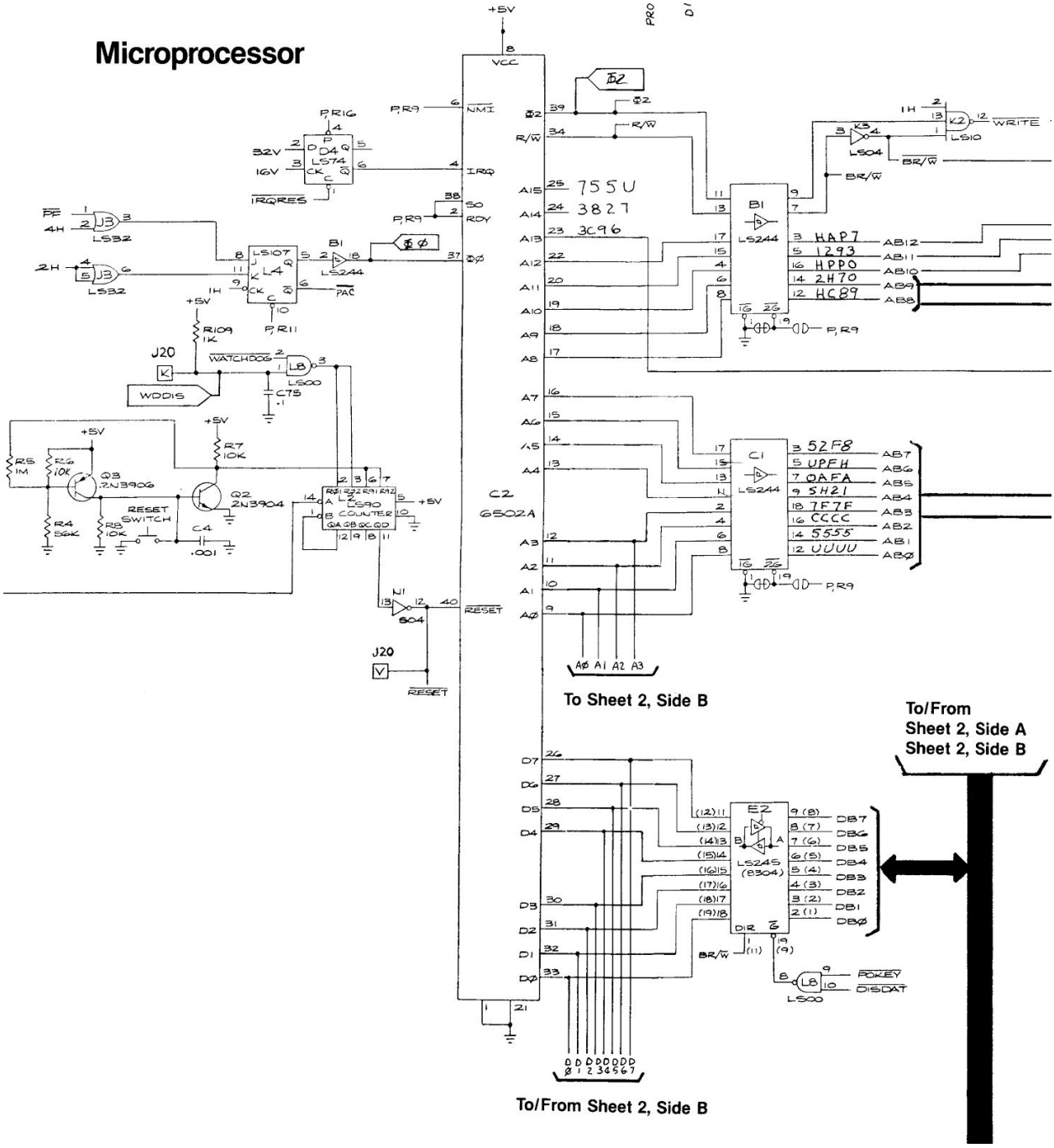
3.1. SPECIFICATIONS

The MOS 6502 is an 8-bit microprocessor that was originally produced in 1975. It executes Assembly instructions provided by the core ROM unit of the *Centipede* hardware. The MOS 6502 has a 16-bit address bus that it uses to activate the majority of the control signals throughout the system. The internal clock runs at 1.512MHz for this system. Internally, it has 6 registers: an 8-bit accumulator, two 8-bit index registers, a stack pointer, a 16-bit program counter, and a status register holding all the processor flags. It has two interrupt signals, the non-maskable interrupt and the IRQ as well as an overflow set input for external control.



3.2. IMPLEMENTATION

Because the MOS 6502 is such a complicated logic module to implement ourselves, we found an open-source version on Github made by Arlett Ottens. His code is broken into two files, cpu.sv and alu.sv, which together make up his model of the processor. His model of the MOS 6502 matches the original specifications exactly excluding some very minor changes. The primary one is that instead of a single input-output 8-bit data bus, Ottens implemented the MOS 6502 with an 8-bit input bus and an 8-bit output bus.



3.3. VERIFICATION

To ensure that Otten's MOS 6502 was functional and matched specification, we executed a test bench Assembly program on it in simulation. This test bench was provided by Ottens, and it was his internal testing tool for his model. The code was placed into a ROM and executed by the 6502 module. This code was examined by the team to confirm that it seemed to cover every primary use function of the MOS 6502. Had the processor made an error during simulation, the code would have exited abruptly, but because it ran through the entire gauntlet of instructions, we could be sure that it was up to specification.

Then, to ensure that it would work with the *Centipede* code on the Nexys4, the processor was given the *Centipede* ROM to execute. The program counter of this execution was outputted to the seven-segment displays of the FPGA, so we could see the way it parsed the code. The processor was clocked on the click of a button in order to slow down the program counter to a speed we could control. A characteristic of the 6502 is that the address of the first instruction is stored in the reset vector($FFFC/$FFFD). We knew our core processor was functional when it went to the reset vector and continued running through main instructions. Additionally, the way it processed the code was compared to a side-by-side run with the MAME simulator's execution of the same code to ensure verification even further.

3.4. CHALLENGES

Our main challenge with the CPU was integration. At the project deadline, the 6502 would execute code for ~200ms and then spin off into an undefined state owing to a bad branch instruction. We believe that this is due to a timing issue between the RAM/ROM modules and the CPU. It was designed with asynchronous read memory in mind and our block RAMs have synchronous outputs and I think this was causing us to violate timing on some specific portions of instructions. We unfortunately ran out of time to solve this issue by the end of the project.

3.5. ADDRESS DECODER

The address decoder is responsible for converting the output of the MOS 6502 into all of the necessary and appropriate control signals needed across the system. As such, the address decoder was a straightforward block of combinational logic to activate these bits. Depending on the address range outputted by the processor, the address decoder turned on particular flags and output bits that served as inputs for the POKEY, ROM, RAM, etc.

4. POKEY AUDIO

4.1. SPECIFICATIONS

The POKEY chip is the module responsible for producing the sound of the game. It converts digital control signals into usable analog outputs. The POKEY has eight potentiometer input lines. Each line contains its own dump transistor and an 8-bit latch. A binary counter to 228 initiates when the scan sequence starts and clears the dump transistors; counts once per line and causing the potentiometer lines to begin charging. When a line reaches a logic 1, the current counter value is latched into corresponding latch to be read by the CPU.

There are three polynomial counters (17 bit, 5 bit, and 4 bit) which are use to generate random noise. The outputs of these counters are sampled independently by four audio channels where each channel has polynomial counters clocked at its own frequency. The frequency dividers act as low-pass filter clocks where each audio register is controlled by bits 5, 6, and 7. Bits 0 to 3 of this register controls volume. 4'b1111 represents maximum volume, 4'b1000 represents middle strength, and 4'b0000 sets the volume off. The 4th bit turns on the "Volume Control only" mode.

4.2. IMPLEMENTATION

Our first attempt utilized an open-source code solution from Github, but in simulation, we found that it did not match our specifications. Luckily, we were able to collaborate with another 18-545 group, Team Battlezone, on an implementation of the POKEY which followed the specifications of the official Atari data sheet while leaving out unwanted I/O.

In our system, the POKEY's data in and out ports were connected to the databus, and it takes in both 1.5MHZ and 50 MHZ clocks to run. The address space for the POKEY connect to the first 4 bits of 16-bit address bus, and it gets activated by the main CPU write line.

4.3. VERIFICATION

To verify synthesis on the Nexys4 board, we overlaid two different signals in order to guarantee outputting a non-pure tone from the POKEY chip. Because the signal would contain more than one sinusoidal component, this particular audio could be measured and heard by the average ear easily. Ultimately, we knew it was functional when we listened to and quantified the audio coming from the speakers.

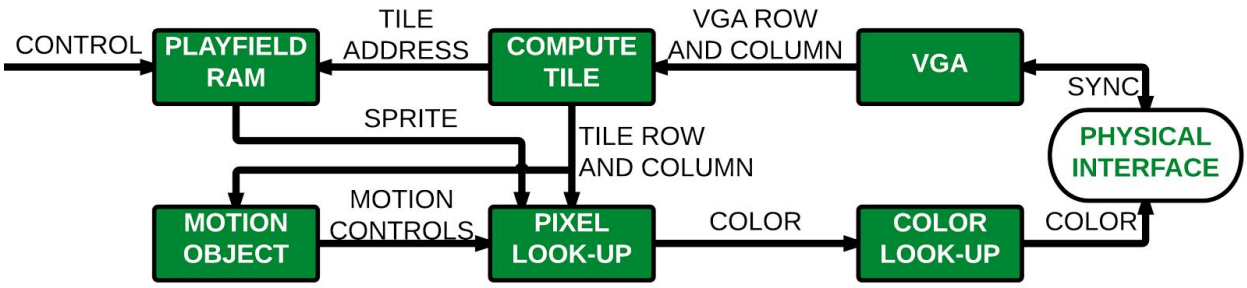
5. GRAPHICS PIPELINE

5.1. SPECIFICATIONS

The graphics pipeline is responsible for taking playfield information (written to the playfield ram by the 6502 CPU) and turning that into a displayed image. The screen is arranged into a 32x30 grid of tiles for the background with each tile holding a sprite. The playfield ram also holds data for up to 16 motion objects (centipede segments, spiders, etc) these sprites can be located anywhere on the screen and are drawn over the top of the background tiles. Since we are outputting over a VGA interface, the game needs to render in a rasterized fashion rather than updating a frame buffer. This drives the implementation of the display logic.

5.2. IMPLEMENTATION

Our rendering hardware is implemented using a series of lookup blocks. First, the compute tile module takes in row and column for the current VGA pixel and figures out what background tile we are in. This tile address is then used to pull the sprite ID from playfield ram. Simultaneously, the motion object hardware checks if the current pixel (row, col) intersects one of the motion objects. It then outputs the correct sprite ID for that motion object along with which pixel in the object to draw. The pixel lookup block takes in both of these and based on a hit signal from the motion object circuitry, selects the correct sprite ID for the pixel and then uses the sprite row, col signals to figure out which pixel in the sprite to draw. This pixel lookup table is a large block ROM that holds pixel color data for every sprite. This color code is then sent to the color palette lookup which writes the RGB values to the vga interface. This whole system is clocked at 50 MHz using the 18-240 VGA implementation as a base.



5.3. VERIFICATION

The graphics pipeline was tested in simulation in pieces, verifying that each module matches its spec through each change and addition. After all of the pieces matched their individual specifications, the entire subsystem was simulated with some basic test vectors to render specific sprites that could be easily checked in simulation. Then, the whole system was synthesized with the playfield ram loaded with a frame dump from MAME. Once the system was displaying that frame correctly, we declared the module functional and moved on to other priorities.

5.4. CHALLENGES

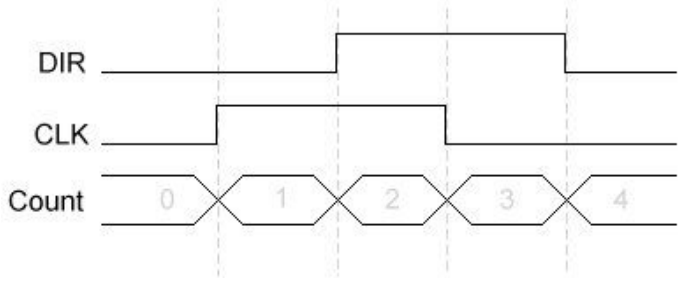
The biggest challenge with this subsystem was deriving the specification by reverse engineering the playfield ram memory arrangement to determine what data was where and in what format. This required a great deal of time working with MAME in debug mode and the game in test mode, manually moving sprites around and watching for changes in the memory contents. Actual implementation of the system was pretty smooth, taking about 100 hours total. The decision to throw out Atari's highly compressed pixel data ROM in favor of recreating it using raw images was invaluable to this component's completion.

6. INPUT CONTROLS

6.1. TRACKBALL

6.1.1. SPECIFICATIONS

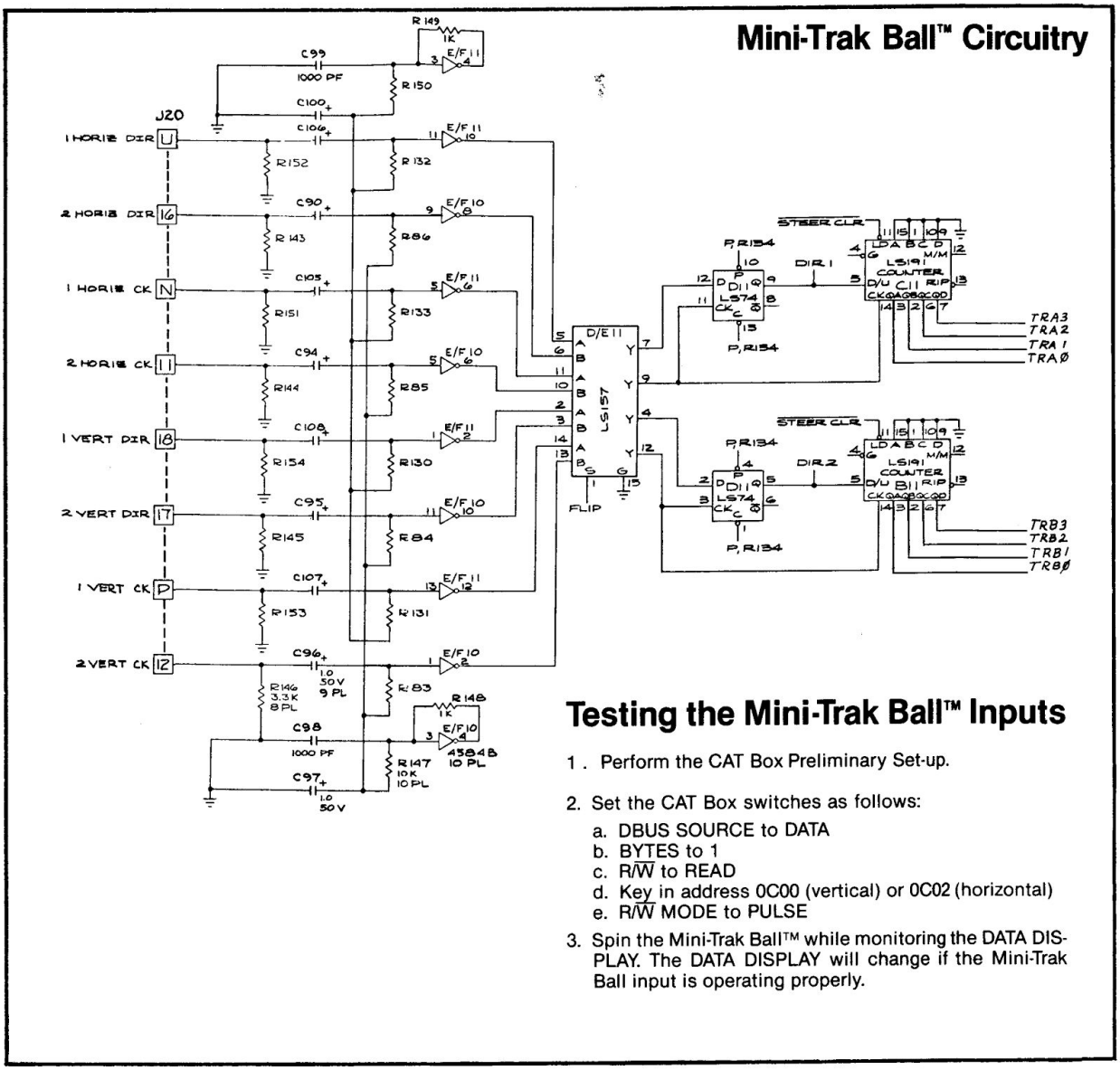
The trackball is a spherical controller that players can roll in order to move around the player character on the screen. It has two mechanical rollers, one for each axis, that monitor how much the ball has rolled in either direction. These rollers convert the mechanical motion into two oscillating signals for each axis: direction and clocking. These two signals will be phase shifted to show whether the ball is rolling left or right (up or down) as displayed in the figure below:



As these signals oscillate, a counting chip tracks how many edges have been seen and either counts up or down based on if the ball is moving in one direction or the other. In the diagram above, because the direction line is phase shifted ninety degrees delayed from the clocking line, the counting chip knows to count up. For each axis, a separate counting chip tracks these oscillations. When the MOS 6502 polls the counting chips for how much the trackball has moved, it resets them. The counting chip acts as a delta from the last time the processor read from them.

6.1.2. IMPLEMENTATION

The original trackball used for *Centipede* was made by a company called Suzo-Happ. Fortunately, they still manufacture trackballs with the original specifications. The trackball we purchased contained all of the necessary mechanical components and outputted the direction line and clocking line for each axis. These were inputted into the Nexys4 through the PMOD input connectors.



The hardware was implemented according to the original schematic above. Both trackballs were fed into a multiplexer which selects the trackball to monitor. The vertical and horizontal pairs of signals were put into single bit flip flops to hold the direction across a roll. Then, these were fed into incrementing/decrementing counting chips. The 4 bit outputs from these were then sent to the data bus back to the MOS 6502 processor.

6.1.3. VERIFICATION

After writing a simulated test bench which triggered a complete “spin” or the ball in each direction, the counting chip was determined to be functioning correctly. Once this was complete, the entire trackball circuitry was verified in synthesis. We hooked the trackball into the Neyxs4 using the PMOD port, and the four bit output of each axis was displayed via the seven segment displays. By shutting off certain control signals, we determined that the spinning of the trackball was being correctly displayed.

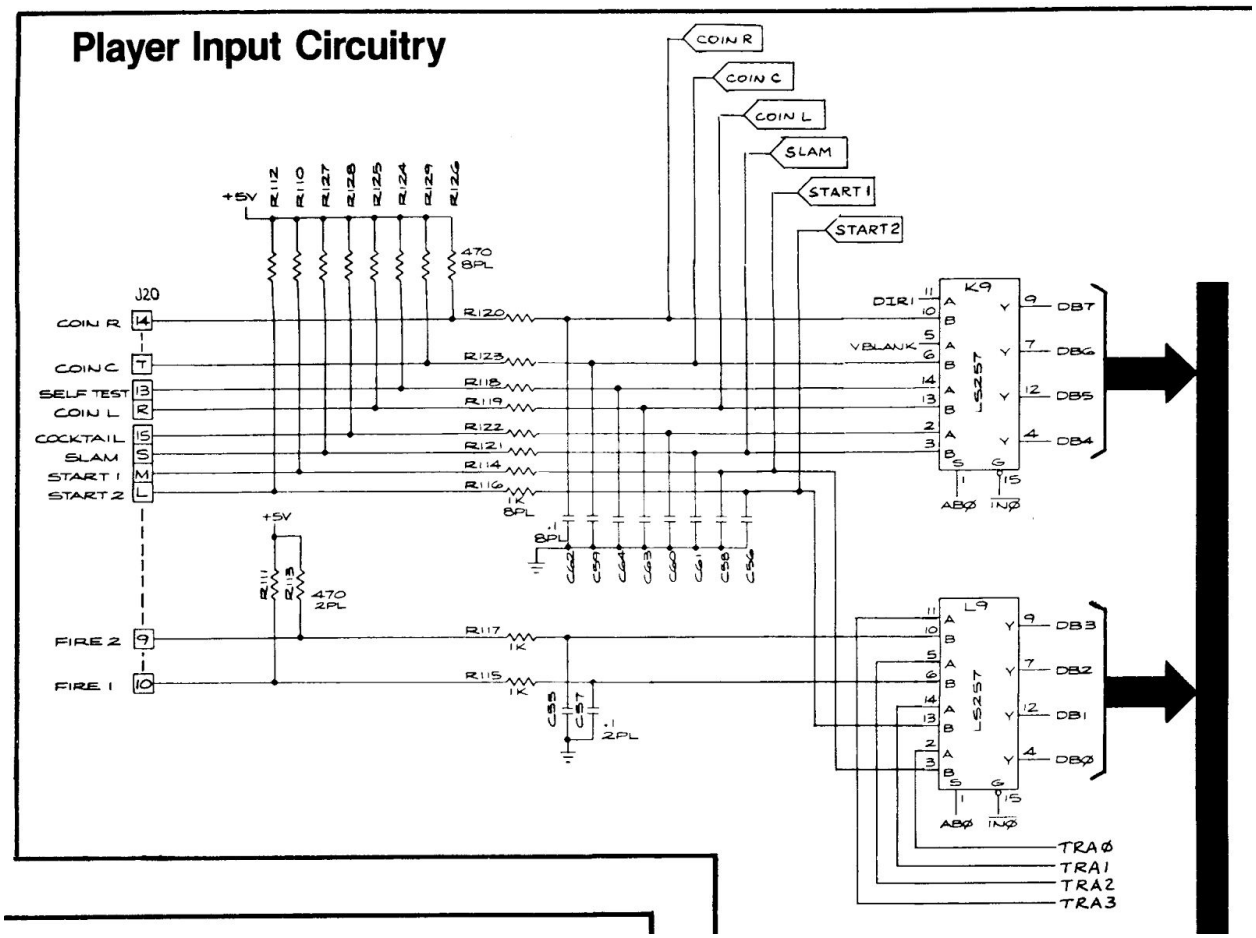
6.1.4. CHALLENGES

The trackball specifications outputted the direction and clocking lines at 12 volts which was far too high for the Nexys4 FPGA to take. In order to connect these through the PMOD ports, they had to be brought down to 3.3 volts. A problem arose when it was discovered that the trackball output voltages were very inconsistent ranging from around 10.5 volts all the way to 11.5 volts. In order to fix this problem, the trackball had to be powered at closer to 11 volts which made the outputs somewhat more predictable. Then, they were passed through a voltage divider and fed into the Nexys4 boards.

Another challenge was determining the resolution at which to count the input cycles. The original trackball had a poorer resolution for tracking the spinning than modern purchased one. This meant that spinning the trackball quickly would result in overflowing the counter chip. This problem was solved by examining specifications on the LETA chip. The LETA was a common trackball monitoring counter of many early 80’s arcade games. It had two resolution modes, one which counted every edge, and one which only counted complete cycles of both the direction and clocking lines. This led to the conclusion that our modern trackball had a resolution four times greater than the original trackball, and we compensated for this in the design accordingly.

6.2. BUTTONS AND OPTION SWITCHES

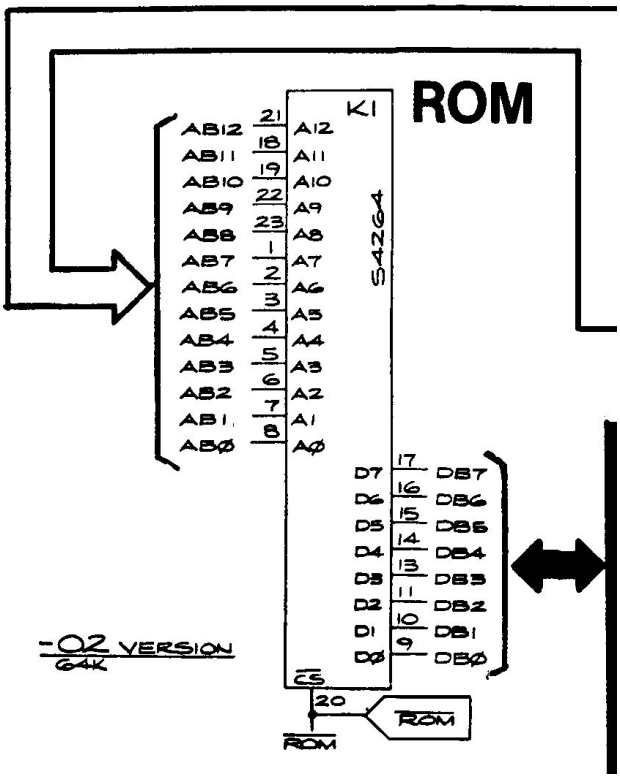
Centipede uses buttons to fire the player's lasers and switches to control option signals for the CPU. Because of the time constraints of debugging the system, these were simply connected to the buttons and switches built into the FPGA. All of these signals are passed through tri-state buffered multiplexers which feed directly into the central data bus. They are controlled by the MOS 6502 central processor.



7. MEMORY

7.1. ROM

The main program memory is stored in ROM that is accessed by the main CPU. It is addressed using the bottom thirteen bits of the address line out of the MOS 6502, and it outputs a byte of data back to the databus. The original memory chip used in *Centipede* was combinational, but our implementation used a clocked ROM block instead.



The main program memory for the game was pulled from a ROM file found on a website that provides them for simulation using MAME. The uncompressed hex code was then extracted from MAME using a debug setting, and it was processed into correct formatting using a Python script. The code was then injected into the SystemVerilog using the memreadh command.

7.2. RAM

For the RAM of the system, we decided to implement our own version somewhat differently than how it was done in the original game. Both versions are 10-bit addressed, 8-bit wide RAM blocks with an enable and read/write line. The primary difference was that ours was synchronous, whereas theirs was asynchronous.

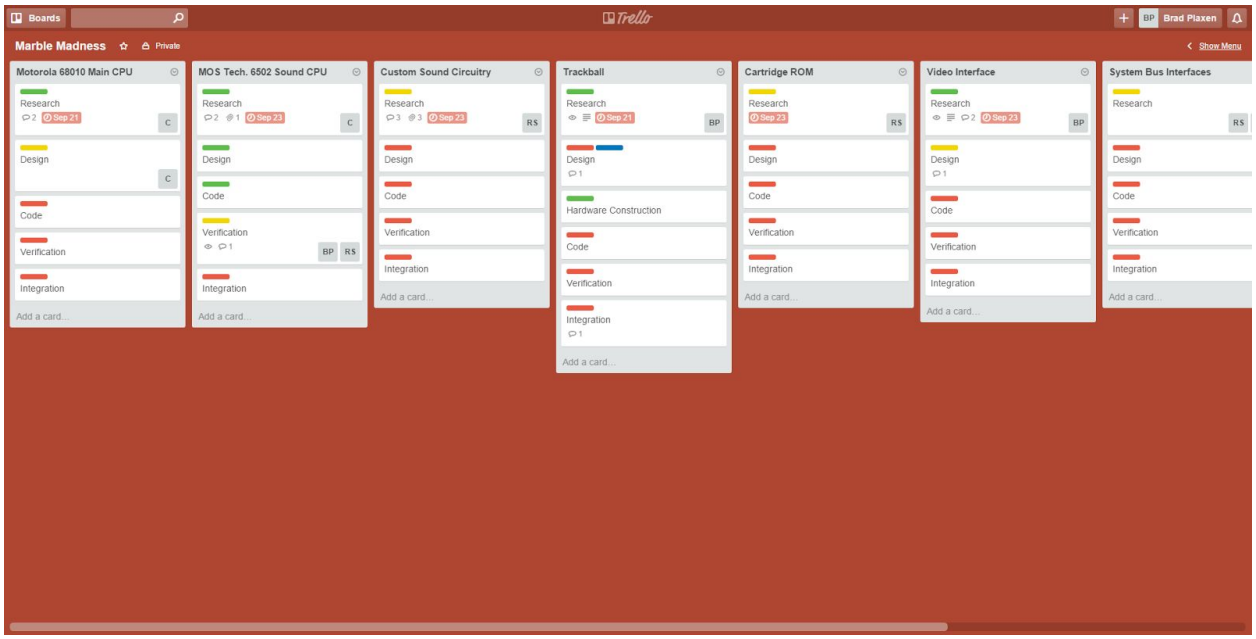
8. PLANNING AND MANAGEMENT

8.1. MARBLE MADNESS

Initially, our group's goal was to recreate the hardware for the arcade game *Marble Madness*. It game out several years after *Centipede*, and instead of one core processor, it had two: the MOS 6502 and the Motorola 68000. Eleven weeks into the semester, the group determined that integrating both of the processors together was going to be too lofty a goal to complete in the time remaining. After analyzing the specifications of other Atari arcade games from the early 80's, we discovered that many of them used similar standard hardware components. This meant we would be able to recycle most of the code we had already constructed for *Marble Madness*. *Centipede* had the most overlap with the material we had already engineered, so we made the transition at that point in time.

8.2. TIMELINE MANAGEMENT STRATEGY

Our first attempt for group management was utilization of a board on Trello. As shown in the figure below, the project was broken down into the various systems we first deemed crucial. Then, each system was broken down into steps for completion. These systems were subsequently assigned to the various members of the team.



8.3. GITHUB

All of our code was stored on a single Github account that was available to every team member. We separated verification and test code out from the final project folder therein.

Here is the link to our open repository:
https://github.com/cnordgren/545_Project

9. LESSONS LEARNED

9.1. IDENTIFYING HUMAN LIMITATIONS

The biggest mistake made on this project was failing to recognize the complexities of the project early on, and then not dedicating the proper amount of work towards that goal. Members of the team had a wide range of SystemVerilog skills and experiences coming into the project. Had we been able to identify that sooner, we could have managed the team responsibilities significantly better. Instead, tasks were assigned poorly, and deadlines were not made as a result.

Along the same lines, having better communication across these tasks could have solved some of these issues. The majority of the work was being done independently instead of together, so while some components moved forward rapidly, others got delayed because of bottlenecks of information. The lesson learned is that it is crucial to collaborate, even if specific tasks are assigned to specific individuals. The skill gaps within our group would have been dramatically reduced, and we would have been much more successful in our project.

These mistakes were ones learned around the time we switched projects from *Marble Madness* to *Centipede*. At that point in the semester, we began scheduling specific hours during which all of us would be present in the lab together. This is one of the reasons our development of *Centipede* went so much smoother than our initial attempt. Additionally, we realized the scope of the project more clearly, and thus dedicated more hours per week into the project.

9.2. KNOWING HARDWARE LIMITATIONS

During integration, one of the biggest roadblocks was determining the error with our data bus. Ultimately, it was discovered that our implementation using tri-state buffers was not supported by the FPGA we had first picked. Because we focused more heavily on researching our specific project in the early phases of the semester, we failed to learn as much about the tools we were using. Had we done more research on the Nexys4 in the beginning, we may have avoided the pitfall with the tri-state buffers.

9.3. RESPECTING INTEGRATION

A huge mistake was underestimating the amount of time integration was going to take our team. The assumption was made that because every subsystem had been vetted thoroughly, the integration would be relatively smooth. The fabric connecting every subsystem becomes just as important as the subsystems themselves, and it was much more difficult than anticipated. The professors warned that this would be the case, and it should have been heeded. If we were to do it again, we would certainly aim to complete the initial module building much earlier.

10. INDIVIDUAL RESPONSES

10.1. BRADLEY PLAXEN

At the beginning of the semester, my initial task was to research the trackball input system. I read through documentation of the original system, determined how it worked, and ultimately made the decision as to which trackball to purchase. I tested the purchased trackball using the lab oscilloscope and determined how the output signals functioned. Later on in the semester, I wrote the hardware description for the input network including all of the buttons and the trackballs. This included writing and debugging the counting registers for each of the axes of the trackball and ensuring that they met specification expectations of the MOS 6502.

Throughout the middle of the semester, I took on the role of finding and verifying the MOS 6502 processor. Once I had selected the open-source version from Artlett Ottens, I compiled and verified it using the Assembly code test bench provided. I was responsible for converting the code to the proper formatting to work with SystemVerilog's ROM blocks. This process included about a week of debugging the code to get it to verify correctly. Once I had proven its correctness, I handed it over to Rohan to integrate with the *Centipede* ROM and assisted him with that debugging process.

On the technical front, much of my semester was spent debugging Rohan's code and assisting him with his modules while Carl was working independently. For instance, while Carl was working on the graphics pipeline, I was debugging Rohan's address decoder. This was a common pattern throughout the latter half of the project, particularly once we switched to *Centipede*. I also wrote the initial code for the majority of our top modules. This included the top module for our sound demo part-way through the semester and the initial draft of our final top module. During the first half of the semester, I spent about 8 hours a week on the project which I realized was too few, and once we switched to *Centipede,* I spent an average of 12 hours a week towards completion. This excludes the final week during which I spent around 40 hours working on debugging systems and the creation of our final submission materials.

Beyond the technical scope, I attempted to layout a schedule for the group to use for team management. I created the entire Trello board as an attempt to keep everyone in communication. This meant I created all of the tasks and assigned them through that platform. Unfortunately, group members failed to take advantage of the platform I established, and this ended about halfway through the semester.

Additionally, on the non-technical side, I was responsible for the content of the major document submissions we made. I crafted every system diagram we made ourselves for use in our documentation. When the end of the semester came, I made our entire final presentation, poster, and drafted the overwhelming majority of this final report.

10.2. CARL NORDGREN

What I worked on:
Throughout the semester I was generally in charge of the high level design of our project while the other group members worked on pieces of that design. I broke the systems we worked on down into components and we then split these up among the team members. I think the reason this strategy was unsuccessful was that our group underestimated how much time each piece would take and was unable to get back on schedule. My main contribution in terms of system components was the graphics output logic. This logic takes data from the playfield ram (which is where the processor writes display information) and renders it to the screen. This was definitely the most complicated piece of team designed logic in the project and amounted to a series of lookups based on the pixel being rendered at any given time. For a full description see the relevant section of the report. As for time spent, I usually averaged 10-12 hours of time spent on the project per week up until the project switch happened. After that it was around 20 hours per week. In the last week I spent nearly 90 hours in the lab working on the project in an effort to make it work to the exclusion of all else.

My impressions, comments, etc:
Honestly, I didn't enjoy this class nearly as much as I had expected and hoped to due to group issues. Group members continually didn't meet their specified goals for each week resulting in constant schedule slip. There was nothing keeping group members accountable for their intermediate work other than pressure from within the group (me). I believe a requirement to submit meeting minutes for each group meeting during a week and using that as a group status report would be a better model than the individual ones. It's very easy to make your individual status report look like things are fine when they are in fact not. More structure to required group meetings and status updates will definitely be an aid in keeping marginal groups on track. Generally pressure from professors and TAs regarding progress is much more effective that pressure from other group members.

10.3. ROHAN SAIGAL

The first few weeks of the course I spent learning the system architecture of marble madness and how all the sub-components are involved. Pretty early on we split up the project and my responsibility was to focus on the audio subsystem. It was this portion of the semester where I spent time learning how the sound microprocessor, the 6502, communicated with the custom sound effects chip(POKEY) and address decoder to properly connect to either ROM or RAM. I played a role in proving our Pokey RTL code synthesizes on the FPGA board for demo day. I was able to find a working module on Github and proved it met our system architecture specifications and verified in simulation.

After the mid-semester demo my focus was to get the 6502 working. Other groups in the class had already found a 6502 core from open source so I decided to go with it. I spent at least two weeks to make sure I was confident in verifying the microprocessor. My next task was to be able to show that the 6502 is executing the instructions from main. My plan of attack was to output the program counter for the 6502 and have it display on the seven segment display of the nexys4. This took me much longer than I had intended. My lack of experience in FPGA programming set me back a little but nonetheless was able to get through it and make it work while learning a lot from my teammates and the teaching assistants.

After our decision of switching from Marble Madness to Centipede the lesson of better time management had triggered. As a group we started meeting four days during the week and made a point to work together by assisting with design and debugging. This caused me to me more productive and efficient in less time. Also at this point I had become much more comfortable in synthesizing in Vivado. In the final weeks of the class I successfully combined the 6502(main cpu) with the address decoder, Block RAM, and Block ROM in both simulation and synthesis.

With one week remaining all the components of Centipede's system had been independently verified and our top module code written leading to the final debugging stage. Unfortunately, we ran into the issue where the xilinx we were using does not support internal tri state buses which is a SystemVerilog tool we assumed we would have when designing our RTL code. The forced us to make a decision; we decided to switch to the Altera FPGA board for debugging the integration.

I had not taken 18-341(Logic Design and Verification) prior to the class so there was definitely a learning curve on my end. Evenso, looking at my progress from the beginning of the class to demo day, technically speaking, I had greatly improved in my system verilog code style writing, logic design, debugging, FPGA programming, and most of all my ability to operate Xilinx's Vivado.