An FPGA Vector-Graphics Game Emulator Based on *Tempest* (Atari© 1981)

Instructor:
Dr. Samuel Richie

Terms:
Spring 2011
Summer 2011

Group #3:
Robert Baker
Tony Camarano
Drew Hanson
Robert Higginbotham

Sponsor:
Mr. Don Harper
Director of Technology, EECS, UCF

Table of Contents

List of Figures

List of Tables

Section 1: Project Description

1.1 Executive Summary

Everyone has played or seen a video game in his/her lifetime. Most of society knows what a video game is and many have owned and played video games throughout his/her lifetime. Video games play a major role in society as a source of entertainment. According to Anita Frazier, video games industry analyst for The NPD Group, "Video games account for one-third of the average monthly consumer spending in the U.S. for core entertainment content, including music, video, games."[1] Throughout time, video games have evolved from the *Pac-Man* arcade game that most people love to the modern version of *Call of Duty: Black Ops*, which can be played on many platforms and in any home around the world. The evolution of video games in our society has been in large part due to advances in computer and electrical engineering. While current video games continue to push the technological limits, there is also an effort to preserve classic games among die-hard fans. Many of the classic games from the 1980s and 1990s have been preserved by video game enthusiasts who took the time and effort to create emulators for these games. These emulators provide the younger enthusiasts the opportunity to experience video games as they were originally intended.

Our group proposed the idea of creating an emulator for the arcade game *Tempest* by Atari©, Inc., which was released in October 1981.[2] The emulator was created using an FPGA and served as a way to preserve the arcade game using modern technology. We used the MOS Technology 6502 processor on the FPGA. The FPGA was connected to a laser that allowed the output of the game to be viewed on a large screen or wall. The user will have a simple set of controls interfaced with the FPGA and will be able to play the game as if they were playing the original coin-operated arcade version. The FPGA emulator provided much of the same functionalities found on the original *Tempest* game, such as multiple difficulty levels, multiple environments/playfields, and a dial/spin controller. However, the FPGA emulator didn't have the functionality of color output. This is due to the simplifying of the PCB that was created and also due to cost of the lasers that would have to be purchased.

Tempest is a tube-shooting game that utilizes a fixed environment shown in a 3D perspective.[3] *Tempest* is probably most notable for being one of the first arcade games with a selectable difficulty level and for its vector-generated graphics. The objective of the game is to destroy all of the enemies that are within the same segment of the playfield as the user-controlled. The enemies first appear at the far end of the playfield and then move toward the user's spaceship as time passes. This style of game was utilized by many other companies, as well as Atari©, when they created variations such as *Battlezone* and *Space Duel*. *Tempest* was initially meant to be a 3D remake of the Atari© smash-hit *Space Invaders*, but the design was never used due to problems with early versions.[3]

FPGAs are a type of modern technology that allow for a customer or designer to configure the integrated circuit after the FPGA has been manufactured.[4] Having the option to be configured based on the user's needs, allows the device to implement any logical function that can be programmed using ASIC or HDL. These functions are programmed using logic blocks that may consist of any number of arrangements of the simple logic gates such as AND-gates and NOR-gates. In addition to programming these simple logic gates, the logic blocks within the FPGA can be set up to execute intricate combinational logic functions. Along with the digital functions previously mentioned, FPGAs also have a variety of analog features that are available to the user. One of the most common analog features is the differential comparators on the input pins which are designed to be connected to differential signaling channels, which allows these "mixed signal FPGAs" to have the ability to implement Analog-to-Digital Converters (ADCs) and Digital-to-Analog Converters (ADCs).[4]

In our project, we added in many of the same features found in the original arcade version of *Tempest*, such as sound and an input controller that looks and feels almost identical. However, we decided to change the output from a screen to a laser. Implementing this type of output for our emulator gives it a vast amount of appeal to game enthusiasts by creating a vintage playing environment similar to the true environment experienced during the 80s. The laser also adds an extra feature that helps to make this emulator stand alone when compared to others that are made to run solely on a PC.

1.2 Motivation

For our Senior Design project, we formed our group to consist of four Engineering students: Tony Camarano (EE), Robert Baker (EE), Drew Hanson (CpE), and Robert Higginbotham (CpE). We first came to a unified decision to create a touch-screen version of the Rubik's Cube. We called this project The *Allspark*. This project was approved by Dr. Richie and would be self-funded by the group. After a few weeks of research, our group came to the conclusion that The *Allspark* would take more money than the group was willing to contribute as well as having also had some major complications in terms of physical design about which the group could not determine a plausible solution in our design.

After talking with Dr. Richie, the group decided to swap project ideas, while staying within the realm of video games. The new project, approved by Dr. Richie, was a sponsored project by Mr. Don Harper, Director of Technology, EECS at UCF. The project idea was to create an FPGA emulator for the *Tempest* game originally released by Atari© in October 1981. Our group decided to name this project *Vic Vector*.

The first motivation for this project was to find a way to re-create a vintage arcade game on a small piece of modern technology. Preserving this game will

allow video game enthusiasts to play the game as if they were back in the 80s. This game appeals to all video game fans whether they are young kids or middle-aged. This game has great meaning for enthusiasts due to the innovations that Atari© was able to accomplish in video games from the release of the *Tempest* arcade game. Creating this emulator became important to us because we, as a group, enjoy playing video games of all kinds of every age.

Another motivation for this project was that it was sponsored. With The *Allspark* project, each member in our group had intended to spend approximately 250USD. In addition to having received some outside funding, the sponsor of the group also had some background knowledge to help the group because of his attempts in past years at the same project. The amount of knowledge that Mr. Harper has from his trials on the project helped to give our group a running start in creating our design and in assisted our group in overcoming obstacles along the way.

We all wanted to work on a project that allowed us to have the opportunity to specialize with a certain technology that we have utilized during our studies at UCF. In any field, employers look for students that have gone above and beyond their normal studies in school and have worked to broaden their skill set on their own. FPGA design was an area that we each had experience and wished to improve our skills as venture into the workforce.

Another primary goal for the group was to learn new tools such as programs and even new coding languages. The main program we used was *Synopsys*. This program is used for Verilog coding and for the implementation of various schematic diagrams to an FPGA. We also hoped to learn more about Verilog, while also gaining the knowledge to create our own schematics and programs in VHDL.

Our final and most competitive motivation was the desire to complete a project where other UCF students had fallen short. In years past, a group worked on this very project and was also sponsored by Mr. Harper. After two semesters of work, that group failed to come close to implementing the emulator on the FPGA. As we began our project, we heard of the failures of the previous group from both our sponsor and our professor, Dr. Richie. Despite hearing of these obstacles, our group believed that we could improve upon the previous attempt and also believed that we could successfully complete the project.

1.3 Objectives

Our primary objective was to re-create the popular Atari© game *Tempest*. A successful re-creation includes reading the original ROMs, using an Atari© controller, and displaying the game using vector-generated graphics to control a laser. In order to accomplish these tasks, we needed to research the hardware used for the original game. This hardware is broken down into four major

components: the MOS Technology 6502 microprocessor, the Math Box, the Vector Generator and the Input/Output (I/O) management. A successful project will re-create every piece of these four modules and connect them properly. For example, we needed to understand how the input was controlled and how the vectors were derived to create the graphics. Our secondary objective is to learn more about the how older arcade games functioned and to learn about technology, both past and present.

To accomplish our task we implemented the original hardware using Verilog. This included the MOS Technology 6502 processor and any other chips originally used. This hardware reads in and processes the game's original ROM. When the hardware is implemented correctly, then we will be able to play *Tempest* using a controller that mimics an original Atari© controller. Also, since we fully re-created the game and used the original ROMs, any bugs which occurred in the original game will also occur in our version as well. The original game sounds and high score lists have also been implemented.

Since Robert Baker is interested in power systems, he was assigned to deciphering the power controls for *Tempest*. For his research, he needed to understand the controls that Atari© used for their arcade cabinets, and figure out what we need to change for our system. Since we powered a small printed circuit board and a full powered laser, Robert needed to figure out how to distribute the power. One of the team's goals was to power our system completely from a standard wall outlet, Robert had to make sure that we were able to draw enough power to control the light and still not blow out our PCB which was powered by a USB connection. Through this research, Robert has broadened his knowledge and understanding of power systems.

In a similar manner, Tony Camarano focused on FPGA design during his undergraduate studies, so he focused on re-creating the *Tempest* hardware and implementing it onto a FPGA. This included properly modeling the various pieces of the hardware, such as the MOS Technology 6502, and integrating this hardware with the modern pieces we used. One challenge was the use of a modern clock, which runs at 96 MHz and how to make it run on the much slower MOS Technology 6502 which allows operations on both the positive and negative cycles of the clock.

Finally, since Robert Higginbotham and Drew Hanson both studied Computer Engineering, they focused on the dataflow of the game and how to implement the existing ROMs. Most of this research dealt with understanding how to use certain file types and emulators and integrate them into our VHDL version of the hardware. Through their research, Robert and Drew gained a better understanding of classic gaming software and dataflow in complex systems.

1.4 Requirements and Specifications

The following is a list of specifications and requirements based on the general objectives of the project. This list takes the general concepts of the project and assigns specific, numerically attainable goals to the design process. The list is organized into sections pertaining to separate, integral design components:

Software Requirements:

- Accurate representation of the original arcade game *Tempest*
- Runs on original game timings at approximately 1.5 MHz clock speed

Hardware Requirements:

- PCB will be no larger than 5” x 5”
- 38 FPGA I/O Ports

User Interface and Gameplay Requirements:

- The game will save top 8 high scores
- Contains 16 different level shapes
- Contains 99 skill levels
- Three difficulty levels: Easy, Medium, Hard
- One player

Power Supply Requirements:

- Powered by 120 V power outlet

I/O Requirement:

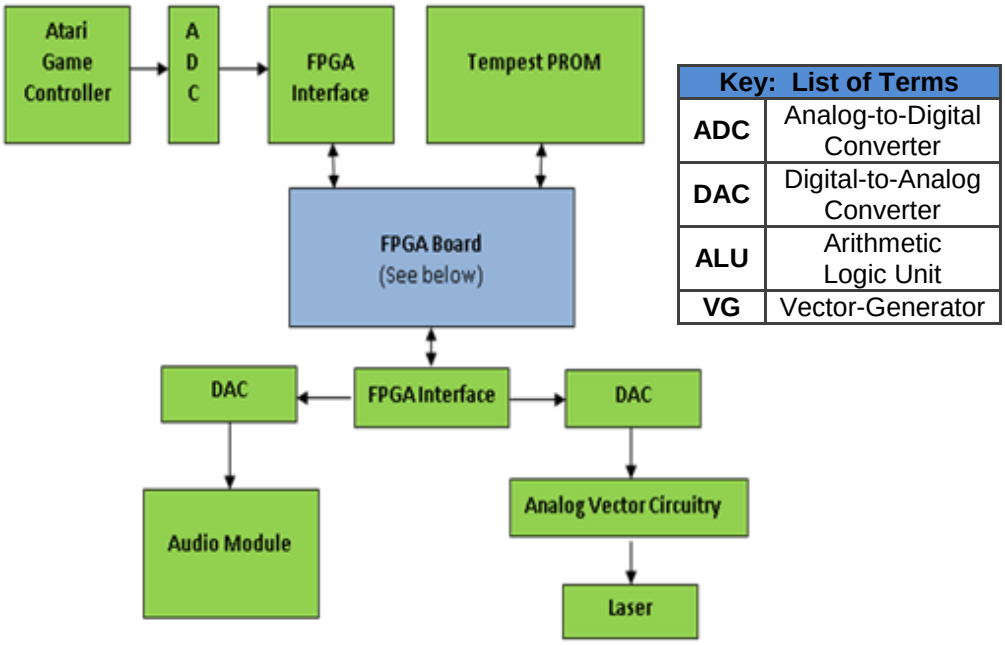
- Controlled by a dial capable of 360° of motion
- 4-button controller
- Controller deck 7” x 5”
- Output by laser galvanometer under $300.00
- Output refresh rates above 30 frames per second (fps)
- Greater than 20 kilo-packets per second (kpps)

Audio Requirements:

- Able to output audio frequencies from 130 - 524Hz

1.5 Block Diagrams

The FPGA receives input from the Atari© game controller originally used for *Tempest*. The *Tempest* ROM is loaded into ROM space on the FPGA board. Output from the FPGA includes digital values which correspond to X and Y coordinates, beam intensities, and vector scaling factors. The FPGA also outputs control information for the audio module. These values are converted to analog signals and some circuitry will be needed to interface them to the laser. Figure 1 illustrates the hardware components needed for this project



Key: List of Terms	
ADC	Analog-to-Digital Converter
DAC	Digital-to-Analog Converter
ALU	Arithmetic Logic Unit
VG	Vector-Generator

Figure 1: Hardware Block Diagram

Each of these blocks represents components of the original game hardware that are designed and implemented within the FPGA. This is meant to present a high-end diagram of the main components modeled in the FPGA. Game ROM may or may not be external. Some FPGA boards have PROM capability which allows for the storage of .bit files. Ideally, we will be able to load in any game ROM using this PROM. Figure 2 illustrates the components of the FPGA needed for this project.

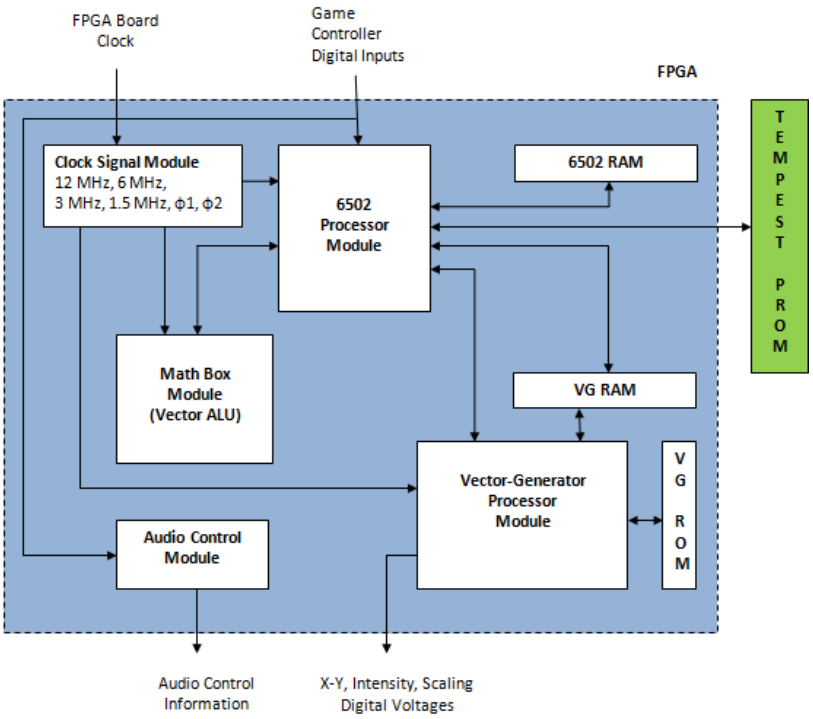


Figure 2: FPGA Block Diagram

1.6 Roles and Responsibilities

Table 1 details each member's responsibilities pertaining to the Senior Design I paper and the various sections the group has designated for the project.

Member	Description	Research	Design and Prototyping
Tony	Block Diagrams Roles, Division of Labor, Budget	Intro and Goals, 6502 Clock & ALU, Vector Generator FPGA Options	6502 Clock & ALU Verilog VG Verilog Memory Interfacing
Robert B.	Specs & Requirements	*Tempest* Background I/O Interface ADC/DAC Game Controller Audio/Laser Interface	Game Controller Interface Audio/Laser Interface ADC/DAC PCB to FPGA Interface
Drew	Goals & Objectives	Research Methods History & Background 6502 Instruction/µCode Math Box	6502 Instruction/µCode Verilog Math Box Verilog Memory Interfacing
Robert H.	Executive Sum. Motivation Milestones	6502 State Machine & Top Level Protocol Math Box *Synopsys* Simulator	6502 State Machine & Top Level Verilog Math Box Verilog Memory Interfacing

Table 1: Senior Design I Paper Roles and Responsibilities

The aspects mentioned in the table are the individual responsibilities for research and the design and prototyping stages for each component of the project. Each member of the group is responsible for, not only the design of their respective components, but also various research areas and administrative tasks. The communication and organization of our group was vital for successful and useful collaboration.

All of the group members were responsible for the rigorous simulation and testing of the project design once our prototype was created. Each member developed a series of test plans to ensure their respective project components were functioning properly, first on their own, and later integrated as a whole.

Tony was responsible for most of the group communication. This included information distribution of meeting times, setting up appointments with our sponsor, Mr. Harper, and the organization at group meetings. In addition, he divided most of the key components of the project to the other group members according to their strengths. He handled the research into the different options available for the selection of a FPGA for the project. He explored the various options available for FPGAs. There are several FPGA manufacturers with dozens of chips and variations from which to choose from. It was important to find a board that included all the necessary features of the project. This also included researching the different modeling and simulation tools available for FPGA design, such as Xilinx's free ISE software or *Synopsys*.

Robert Baker was essentially the group's lead analog designer. He handled all things from analog-to-digital conversion to power and laser control. He was also responsible for the game audio output. Since our PCB was comprised of primarily analog components and chips, Robert was the member working on the PCB design and its implementation with the FPGA board. This included research about the necessary configurations for power and components on the board. Administratively, he handled researching the original game, and was the one responsible for ensuring the project remained true to the original game.

Drew shared programming responsibilities with Robert Higginbotham. There are currently several emulator examples available which implement *Tempest*. These emulators are written in anything from C and Assembly to VHDL. Drew and Robert H. used these previous versions as a model for their design in Verilog. More importantly, Drew was responsible for making sure the code was properly translated and reflected the original hardware mechanics, rather than simply emulating them. Drew also introduced the group to Google Wave, which we used frequently to chat privately and collaborate on the project.

In addition to his programming responsibilities, Robert H. was essentially our group liaison. He acted as group correspondent to the different companies and organizations we need to contact. This included part providers such as Newark and Texas Instruments Inc., as well as organizations which hold copyrights on

information we wish to display for our research, such as Atari's© original game schematics. In addition, Robert H. records and posts group meeting documents on Google Documents so that we could access and edit them later.

1.7 Division of Labor

In order to be successful, the work load of this project was divided based on each member's field of expertise. Our group was made of up two Electrical and two Computer engineers. Out of the four of us, Tony, Drew, and Robert H. have had experience with FGPA design and Verilog HDL. Since roughly 75% of this project involved FPGA design, it was only natural that the three of us take on the responsibility of designing the HDL code. Robert B. had the least experience with Verilog, but since he was one of the Electrical Engineers in the group, he handled the 25% of the project that was related to analog circuitry. It was also important to divide the labor among group members so that if problems with design occurred while testing, the person working on that part of the project had the most knowledge in order to generate a solution.

Robert Baker handled most of the analog portion of the project. Since he was one of the Electrical Engineers in the group, he had significant exposure to analog circuit design. The Atari© game controller was the only input and needed to be converted to some digital input. In addition, the signals from this controller were used to determine audio output. For example, pressing one button may trigger the "firing" audio, while another might output the "zapping" audio. The FPGA delivered several digital output values, including change in X-Y coordinates, color, light intensity, and vector scaling factors. These values were converted from digital to analog signals and processed using some analog circuitry. Most of the schematics for the circuitry used in the original game hardware were openly available, but needed to be optimized and adapted by Robert for the laser interface.

Drew Hanson modeled the Atari© Math Box module. This section controls all of the resizing, movement, and manipulation of the data needed for vector generation. The difficulty in researching this module was that Atari© developed the Math Box as proprietary hardware and software. The game ROMs contain a wide array of elements, such as the micro-code on how different math functions are executed. Drew had to treat some components of the Math Box module as a black box and simply re-create the hardware. Included in this module are four AM2901 ALUs. Drew used open source code to emulate the ALUs. In order to simulate this code, some slight modifications were made. Drew also researched the POKEY chips. These chips are used to scan the game's controller and buttons for input. Similarly to the AM2901s, the POKEY code used for this project is open source VHDL.[5]

Tony Camarano will be designed the Vector Generator, another unique Atari© component. Originally, it was a very simple microprocessor made up of discrete

logic parts. When *Tempest* was designed, microprocessor speed and throughput were extremely limited. In order to work around this limitation, Atari© implemented the Vector Generator to handle the generation of vector changes. It outputs changes in X and Y coordinates while the 6502 processor handles game instructions. In addition, the Vector Generator has the ability to scale the output vectors. It operates using a simple program counter, stack, ROM for predetermined shapes, and RAM which is accessible via the 6502 processor. Tony was the other Electrical Engineer in the group and the Vector Generator interacts constantly with the output and analog circuitry, so his expertise allowed for a good link between the software and hardware requirements.

Robert Higginbotham worked on the design and modeling of the 6502 in VHDL. The VHDL code for this section was acquired through open source code.[6] Since this was the core component in the FPGA, it required all our efforts to ensure that the HDL code we used was true to the original functionality of the 6502 processor. The processor itself is over 30 years old; however, it embodies the core essentials that make up a modern CPU. It includes an instruction set and control logic to decode each process within the machine. An ALU performs all of the arithmetic and logical functions for each instruction. The data bus is 8-bits wide, and the address bus is divided up into two 8-bit busses: address high and low. A program counter and stack pointer control the instruction execution processes. The 6502 uses a two-phase clock system to operate. It receives a single clock input, and generates two non-overlapping clock signals which work internally to carry out instructions. These signals are also sent to the output to operate external components. Modeling the 6502 in Verilog properly was the biggest component in the project.

There are several areas within the 6502 design, including I/O to other modules within the FPGA, instruction execution and implementation, and control logic for said execution. There are currently implementations of this processor in C and VHDL which acted as cornerstones in our approach to designing the 6502 properly.

1.8 Milestones and Timelines

Our group decided to meet at least once a week in the Engineering 2 Atrium to discuss our individual research and to share any sections of the research paper that have been written. We also made it a point to regularly meet with our sponsor every couple of weeks to discuss what we had accomplished and also to talk about aspects of the project where we had reached a stand-still. Additional meetings were also been agreed as needed for portions of in-depth research. As a group, we created a series of shared folders on Google Docs to share work that was done along the way. Work was compiled by exchanging Word documents with the various sections that each group member had completed. Google Docs also allowed the group to back up the work in the event that one of the members should have lost his data. Google Wave and Skype were utilized

as ways to meet collectively whenever we were unable to actually meet in a face-to-face setting. Below is a detailed list of many of the main milestones that the group agreed on, shown in Figure 3.

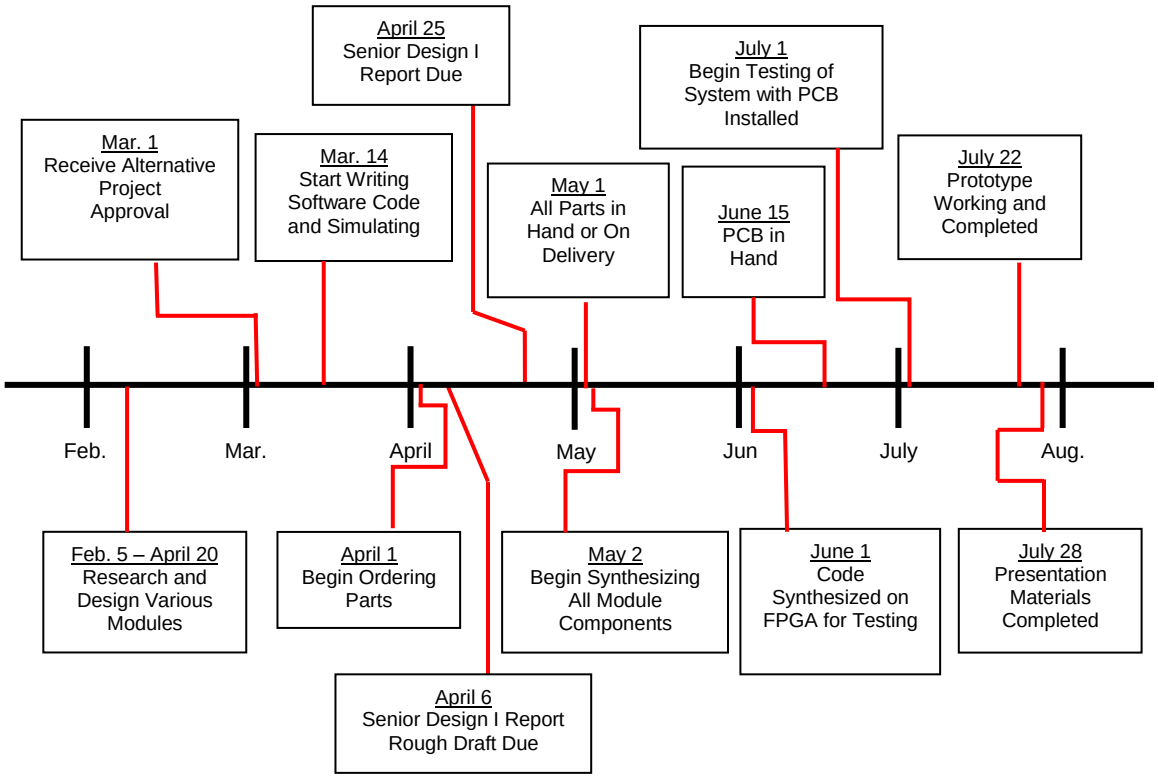


Figure 3: Timeline of Project Milestones

- **Project Approval - 3/1/2011:** This deadline was set to complete Introduction paper that described the project and gave an overview of what the group expected to encounter as we moved forward with the project. This document was turned into Dr. Richie. After which, he granted us permission to continue on with the project.

- **Preliminary Research Completion – 4/20/2011:** This was the first major milestone set for the group. It was set to allow each group member sufficient time to thoroughly research all aspects of the project and prepare to begin the actual design phase. By this deadline, all group members should have had all information needed to complete their individual sections. This amount of research allowed the group to decide on the specific components we felt necessary for the project. As well as locating these components at a competitive price within our budget.

- **Senior Design 1 Report Rough Draft Due – 4/6/2011:** This was the second major milestone for the group. Following this deadline, the group set up a meeting with Dr. Richie to discuss our progress up to this point

and to ensure that we were following the Senior Design 1 Report guidelines in the proper fashion.

- **Senior Design 1 Report Final Draft Due – 4/25/2011:** Our third major milestone was to complete our Senior Design 1 Report. The completion of this report demonstrated that the group had finished our Research and Design portion and had moved onto the Implementation portion of the project. This report provides a detailed explanation of each aspect that we researched and our intended design methods for each component of the project.

- **All Parts in Hand or On Delivery – 5/1/2011:** This deadline was set so that we would have sufficient time to begin building our project while allowing for ample opportunities to solve any errors that were discovered while in the implementation stage.

- **Begin Synthesizing on FPGA Development Board – 6/1/2011:** This milestone was set so that our group could begin testing certain components that contained all of the FPGA elements before we had the final PCB bard in hand. This stage also allowed us to work out any issues in preparing the board with all of the code that we had generated so that we could easily load the PCB with the files to obtain a working prototype.

- **Begin Testing System with PCB – 7/1/2011:** This was one of the final milestones for the group. This deadline allowed for at least two weeks of testing and troubleshooting any issues that were encountered.

- **Working Prototype – 7/22/2011:** This milestone marked the end of the testing phase. By this point, our group will have a fully functional prototype to preview to the judges for our presentation. At this point, our implementation was fully complete and we could modify any aspects of our final paper that needed to be handled so that all information in the final report is current.

- **Completion of Presentation Materials – 7/28/2011:** This is the final milestone for our group. At this point, we will have made sure that our project has met all of the requirements and specifications that we set forth for the project and that our prototype is fully functional. We will also have our presentation of the prototype worked out with our presentation materials so that we can flawlessly demonstrate our project to the panel of judges and answer all questions that they may have for us.

Section 2: Research Related to Project Definition

2.1 Research Methods

Our main source of information and guidance was Mr. Harper. Not only did he recommend this project, but he also has experience with this material. His previous research focused on reverse engineering and attempting to "hack" *Tempest* and use the graphics for a flight simulator. Through his research he gained knowledge about the inner workings of the system and how the individual modules work.

Mr. Harper also provided the team with access to the simulation tool *Synopsys*. *Synopsys* will be discussed more in depth in Section 2.9, but this tool allowed us to test our code for accuracy and functionality. This tool allowed us to check each module individually and allowed us to test the entire system before we implemented it onto an FPGA or PCB. As well, it allowed us to confirm that the vectors were functioning properly.

Using Google search queries, we located a website which posted all of the original schematics for the various versions of *Tempest*. With the help of these drawings we were able to trace specific control lines and dataflow from one module to another. Understanding these schematics was vital to the success of this project. If we hadn't understood how each module worked, then we wouldn't have been able to implement them onto the hardware and therefore not be able to re-create *Tempest*. Through our research, we also found similar projects using with different games. The authors of FPGAArcade.com have been able to re-create many older titles such as *Frogger* and *Asteroids Deluxe*. Since *Asteroids Deluxe* is a different game, the hardware configuration and ROMs was different. However, *Asteroids* is also a vector based game which runs on the MOS Technology 6502. The FPGA Arcade team used their FPGA to control a laser so we were able to learn from their methods.

2.2 History and Emulator Background

One major distinction that must be made is the difference between simulation and emulation. The goal of this project is "emulate" the hardware used to play *Tempest*, not to "simulate" the game play. Simulation simply mimics the game's output. A simulation of *Tempest* could use modern coding techniques to animate the game on a computer monitor. Emulation re-creates the game's hardware. For example, our project is controlled by a VHDL instantiation of a 6502 microprocessor. In a simulation, a "main" function would control the game's animation. Ian McGregor, a Simulation Business Development Manager for Brooks-PRI Automation, described simulation as "a model from which the user can test the outcomes of a system, whereas emulation parts of the original system to not only show outcomes, but also *how* the outcomes are reached.[7]"

Furthermore, Harrison Burris, a contributing author to the National Computer Conference and engineer for *TRW Defense and Space Systems Group,* adds that "[s]oftware development tools can be embedded in the emulation permitting software testing without destroying the integrity of the software under test."[8] To put it simply, "differences occur because simulators abstract a number of system attributes, and make several assumptions about [software behavior]."[9] This distinction is what defined our project goals.

Like many engineering disciplines, emulation has strong roots in military research. As early as 1970, the United States Army was studying emulation in hopes of keeping existing software in place, while utilizing the technological advantages in portability and reliability of hardware.[8] The outcome of this research was twofold. First, the US Army was able to continue using their older programs without being shackled to outdated hardware. Second, multiple programs were able to be run on one machine which was emulating several systems. Emulation became a mainstream topic around 1985. In the mid-1980's, there were few laws that dictated the legality of emulating another companies processor and the micro-code implemented therein. Much like the debate on file sharing, the proponents of emulation claimed that coding ideas could not be patented, while large processor manufactures claimed their products were intellectual property. One of the driving issues for CPU emulation was that large companies (such as IBM) would change their processors regularly and force users to upgrade to the latest chip.[6] Twenty-five years ago processors were not cheap, so switching computers was a major cost that crippled some small businesses. Luckily, some of the large CPU manufactures made their chips backwards compatible, so that old programs could run on new chips.

If we were to simply simulate the game *Tempest*, we could write some lines of source code and be finished in a matter of hours. What elevates the complexity of this project is that we run the game using the original ROMs from the arcade game. In order to do this we needed to re-create the original hardware used for the game. For some of the hardware components, we translated the original hardware into Verilog, which will be implemented onto an FPGA or a printed circuit board. For other sections of the hardware, we replicated the original components.

2.3 Atari© *Tempest* Background

The creator of *Tempest*, Dave Theuer, originally envisioned the game as a first-person view of the popular *Space Invaders* game. Theuer realized that this first-person game would not be nearly as popular as expected. As a result, Theuer decided to redesign the game after having a dream where monsters were crawling up from out of the ground. This idea led to the current design of *Tempest.*[10]

Game development like this imposed technical issues for Atari© during that time period. *Tempest* would use diagonal lines and fast refresh rates, so then-current hardware was not able to process the game. As a result, *Tempest* was the first game to use Color-Quadra scan or vector display. Vector display works similarly to an oscilloscope by drawing lines using X-Y coordinates. In *Tempest's* case, three colors are used: green, yellow and red, to draw the battlefield and the player. The result was *Tempest*, a stunning graphical display for the time period. However, a big disadvantage of the vector display was that only the outlines of shapes could be drawn; there is no shading in the primitive versions of vector display. Another issue with vector display is that only so many vectors can be drawn on the screen simultaneously due to the fact a laser has to draw the individual vectors.[12] As such, the project required a laser galvanometer similar to ones used in laser light shows. In order to imitate the *Tempest* by drawing the vectors with similar refresh rates, a galvanometer with around 20 Kilo packets per second was required.

Another technical issue that occurred with the development of *Tempest* was a phenomenon known as pincushion distortion. Pincushion distortion occurs when there is magnetic deflection between the two pairs of deflection plates inside the CRT. In addition to magnetic deflection, the shape of the CRT screen also can increase the pincushion distortion effect. The curved shape of the CRT screens increased the deflection. In order to fix this distortion, the *Tempest* game designers had to use onboard correction. This correction is evident in the *Tempest* "X-Y outputs schematic" as a sequence of four MC1495L chips that serve as multipliers, two multipliers per coordinate (X and Y). This type of correction can be done inside the CRT monitor, but games such as *Tempest*, *Space Duel* and *Black Widow* had to do the correction on the game board. The last few Atari© vector display games, *Star Wars* and *Major Havoc*, were able to correction the pincushion distortion through the monitor. The pincushion distortion effect is shown in Figure 4.

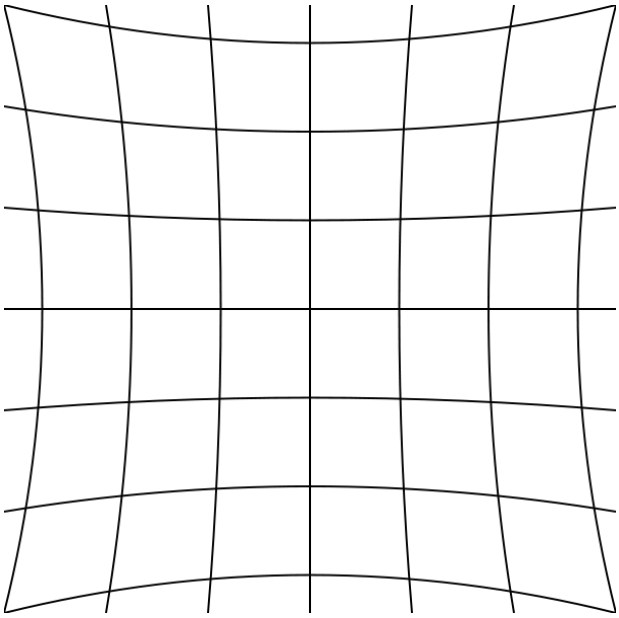


Figure 4: Pincushion Distortion[13]

In this project's design, things like pincushion detection and correction were an issue since the output was drawn with a laser instead of using a CRT monitor. Therefore, some of the additional circuitry that was used in the original *Tempest* arcade design was not necessary.

The original Atari© Arcade *Tempest* machine had many components that made up the whole, including the controller coin door systems, power supply schematic, the 6502 processor, RAM and ROM memory, Vector Generator, math box, vector timer control, high score memory and the output. The system works by first getting the user's input, running it to the external address bus and into the internal workings of the 6502. The user input is then interpreted by the Vector Generator. The Vector Generator uses the vector timing control in order to draw the line slowly enough for the user to be able to understand what is happening in the game. Additionally, the Vector Generator uses the information in the math box in order to allow the game to output simple geometric shapes to display as the output. The generator decides the motions of the user's spacecraft and outputs that to the output XY plot. The output decodes the digital data received from the Vector Generator output and converts that data into an analog output that the screen can display. The output of the *Tempest* system also has the built-in pincushion correction previously mentioned in this section. Additionally, the output has a charging circuit that makes the lines of the output display slower so they can be seen and reacted to by the player. The same charging circuit allows the output to be displayed in a straight line by using capacitors to charge the circuitry to allow it to output in a straight line. The math needed to make that correction is shown in Figure 5.

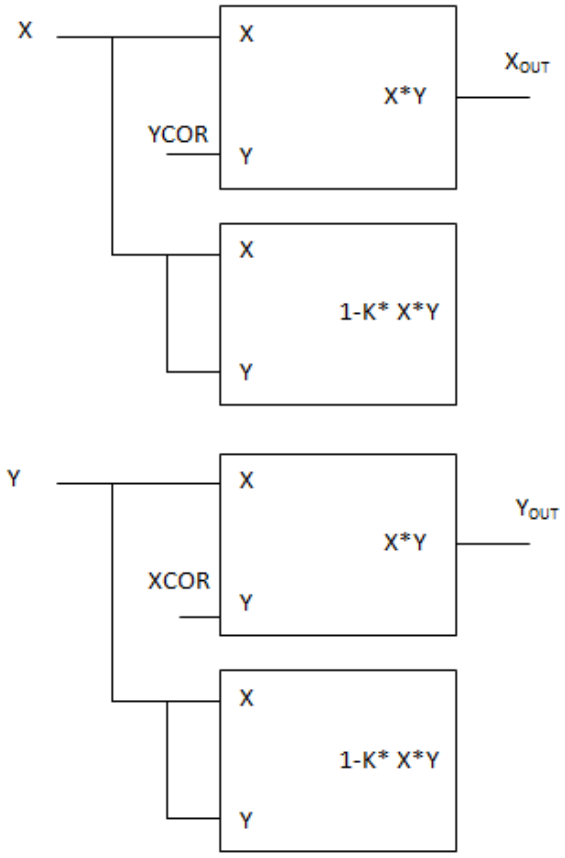


Figure 5: Capacitor Math Calculations

For the *Tempest* project, some components can be skipped or highly simplified. The coin door mechanism was skipped because we did not charge to play games on our project nor was the design project large enough to store change. The power supply mechanism, at least not in its relatively original form, was not needed as the power was supplied to the PCB and FPGA modules, not an entire arcade system. The separate high score memory block was not necessary as that memory was stored on the ROM on the game board. In the original arcade machine, a large CRT monitor was used with multiple lasers to display the output. In our design, one laser galvanometer was used to display a monochromatic game output. As a result some of the circuitry for outputting different colors for the different instructions was not necessary. A majority of the other features of the Atari© *Tempest* machine needed to be redesigned to meet the design criteria as well as to respect the intellectual rights of Atari©. Many of the actual circuit elements in this design project were converted to field-programmable gate array (FPGA) development. Most notably the 6502 processor, which is usually a hardwired component, will be converted into an FPGA using the Verilog programming language. The simplistic block diagram of the overview of the system is shown in Figure 6.

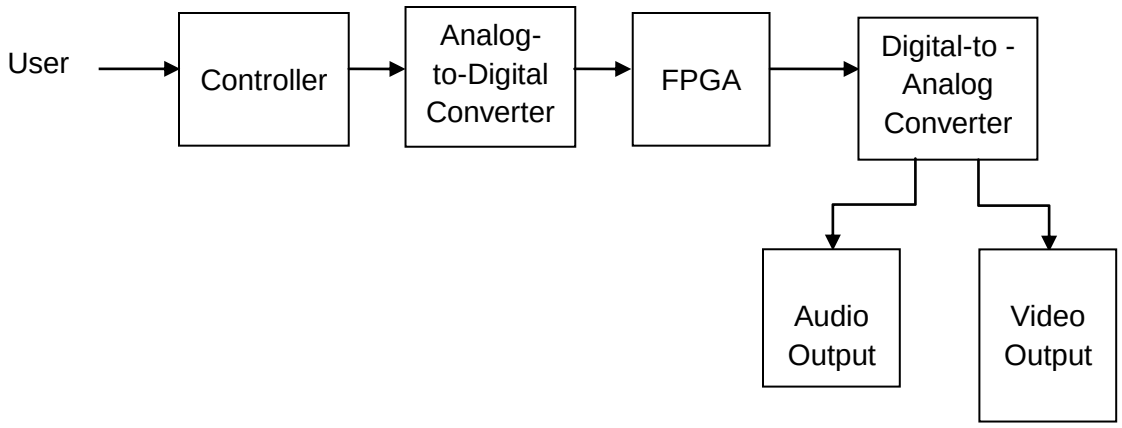


Figure 6: Project Block Diagram

As shown in Figure 6, the user inputs the data via the controller, the analog data that is input from that controller is converted from the analog signal into digital signal by way of the Atari© POKEY chip. The POKEY chip is described in further detail in subsequent sections. The digital output from the analog-to-digital converter is taken as an input into the FPGA board via the interface where the actual game is run. The new output digital data for the audio and video signals from the FPGA board is then sent through the digital-to-analog converter. The output from the digital-to-analog converter is spilt into audio and video signals, the output from both of the signals is then sent as analog data. The video output goes to the galvanometer laser, and the audio signal is sent to a speaker system.

2.4 Introduction and Goals

The FPGA is the most important component in our project. Through it, 90% of our design and implementation became reality. At a high level, an FPGA is basically a two-dimensional array of logical gates, connected via some routing hardware. Each logical block contains Look-Up Tables, or LUTs, which are mapped according to some desired output given an input. Most LUTs have a 4-input width, however, some FPGA devices use six or more. Flip-Flops are also available for each LUT, which allows for timing control of input and output signals. A series of routing channels and connection boxes relate each of the logical blocks correspondingly. These routing components have the ability to be remapped according to the logical design being implemented on the FPGA.

The basic idea of the above architecture is to allow the user to reconfigure the hardware according to a specific design. This is obviously a powerful tool, as it allows for the quick modification and prototyping of a system without the cost of completely remanufacturing a PCB or ASIC chip. The FPGA is a leading driver in the push for faster, more innovative technology. It can be applied to solutions in embedded systems, DSP, and hardware acceleration. For example, an FPGA can be configured to have extremely high through-put when applied to DSP projects. In general, they have higher performance compared to generic CPU or

GPU systems, but also have increased capability and configurability compared to ASIC solutions. FPGA devices are often benchmarked based on their capabilities for embedded processing, which is ironic, due to their inherent ability for parallel processing compared to a traditional serial CPU.

The FPGA has great potential as a promising computing platform. For example, the Xilinx Virtex 6 FPGA has the capacity for approximately 2.6 billion transistors. This density may be compared with the NVIDIA Fermi GPU architecture, which has around 3 billion transistors. It is clear that there are major tradeoffs between the two different technologies. However, it is important to note that the improvement of bandwidth in the Virtex 6 compared with the GPU is over 2000%. This is due to its ability for a large amount of parallel calculations, only available because it can be configured to optimize this process. These comparisons are illustrated in Table 2 and is referenced from UCF professor Dr. Mingjie Lin's FPGA Design course material.[14]

Device	Ports	Power	Capacity	Agg. Bandwidth
XC6V475T	1064	10s W	4.68 MB	5.22 TB/s
NVIDIA Fermi	~16	100s W	Many GBs	~0.23 TB/s

Table 2: Example Comparison of FPGA Features

These factors make the FPGA is an important, blossoming technology, gradually becoming a standard for efficient computations and solutions for a variety of electrical and computer fields.

We modeled each of the hardware components of the original Atari© *Tempest* within the FPGA. Ultimately, the functionality of our design is equivalent to its 30 year-old counterpart. As discussed, the FPGA is an extremely powerful computing tool which may be optimized to improve efficiency. Many of the hardware designs of the original game were limited due to the technology available at time. There are many designed tricks which are employed to efficiently utilize the discrete parts within the game PCBs. In our designed process, we not only modeled the functionality of the game hardware, but also address some of the unnecessary steps taken due to technological limitations. For example, much of the address decoding was accomplished using small decoders and clever wiring configurations in order to access the small capacity RAMs and ROMs. This moderately complex address decoding may be bypassed in an FPGA implementation, since the address bus and memory space may be appropriately sized to accommodate these needs. Thus, our design will reflect proper functionality while attempting to optimize processes when appropriate.

2.5 6502 Microprocessor

In 1975, the 6502 microprocessor was introduced into the market at the retail price of 25USD. This release occurred at the Wescon show. The processor was released under MOS Technology, Inc. (now known as Commodore

Semiconductor Group or CSG). The design of the Apple I computer was one of the first uses of the 6502 microprocessor in the public retail sector. Other notable uses of this processor came from its use in the designs of the Atari© 2600 game console, the Nintendo Famicom game console, and as a reference in the show *Futurama* for the robot character, “Bender,” having the 6502 microprocessor in his brain.

2.5.1 Arithmetic Logic Unit (ALU)

The ALU within the 6502 microprocessor is a fairly simple unit for executing arithmetic and logical operations. It has enable lines for AND, OR, XOR, and SUM operations. These functions are controlled by the random control logic signals, which enable a particular operation to be implemented. Therefore, if an instruction requires the addition of two numbers, the Random Control Logic loads the numbers into the A and B registers, and the SUM enable line will become active. Carry bits are sent to the ALU from the Flag register if the Carry In load signal is active. If an instruction needs to increment a value, such as the Program Counter (PC) for the next address, it can do so by loading 0 into A, the address into B, and setting the carry bit to 1. Clever use of the same hardware allows the ALU to consist of minimal components while still being able to implement a variety of arithmetic and logical functions.

The ALU also has the capability to shift right. There is a specific enable line dedicated to this operation, which shifts the entire A register one bit to the right. The MSB receives the carry bit value. The carry bit receives the LSB value. Other shifting operations, such as arithmetic shift left, are accomplished using the preset hardware. For example, A would simply need to be added to itself to perform the arithmetic shift to the left. This would shift a 0 into the LSB and all other bits left by one space. The MSB would become the carry bit.

The ALU sets bits within the Flag register after an operation is performed. Carry, Half-Carry, Zero, Overflow, and Sign bits are all updated depending on which operation was executed in the ALU. These bits are often used to carry out further instructions within the 6502.

One of the more complex features of the ALU is its ability to handle Binary-Coded Decimal (BCD) operations. In a separate module, the Decimal Adjustment Adder (DAA) receives ALU output and converts it to BCD format. The ALU itself receives a control signal, DAA, from the Random Control Logic which changes some of the logic within the unit to address the BCD operation. However, the only signals which need special attention for BCD functions are the Half-Carry and Carry bits. The DAA signal impacts these carry results accordingly when performing the BCD addition operation.

An important aspect of the ALU within the 6502 is that it is asynchronous, meaning the output is immediately available as soon as the input register values

change. The ALU itself is only controlled by the signals from the Random Control Logic block. The output register is synchronized with the Φ2 phase clock and will be loaded on its positive edge. The signal may then be fed back into the ALU if more computations are necessary, or can travel elsewhere on the data bus to execute logic in other 6502 components.

The input registers, A and B, have several control lines which are determined by the Random Control Logic block. Input register A is typically loaded with the accumulator value, address bytes or zero for certain instructions. Within the 6502, there is a specific control line for initializing the A register to zero. The other control line loads the address/accumulator bus. Input register B usually contains immediate data or values read from memory. It might also contain the low byte of the current address. Control lines for the different bus connections are available. More importantly, there is a control line tied to a data bus inverter. If this control is active, the compliment of the data bus will be loaded into B. This is useful for performing operations such as subtraction, where (A – B) can be rewritten (A + $\overline{B}$ + C), when C (carry) is equal to 1.

A simplified block diagram of the ALU, its registers, and all control lines is shown in Figure 7.

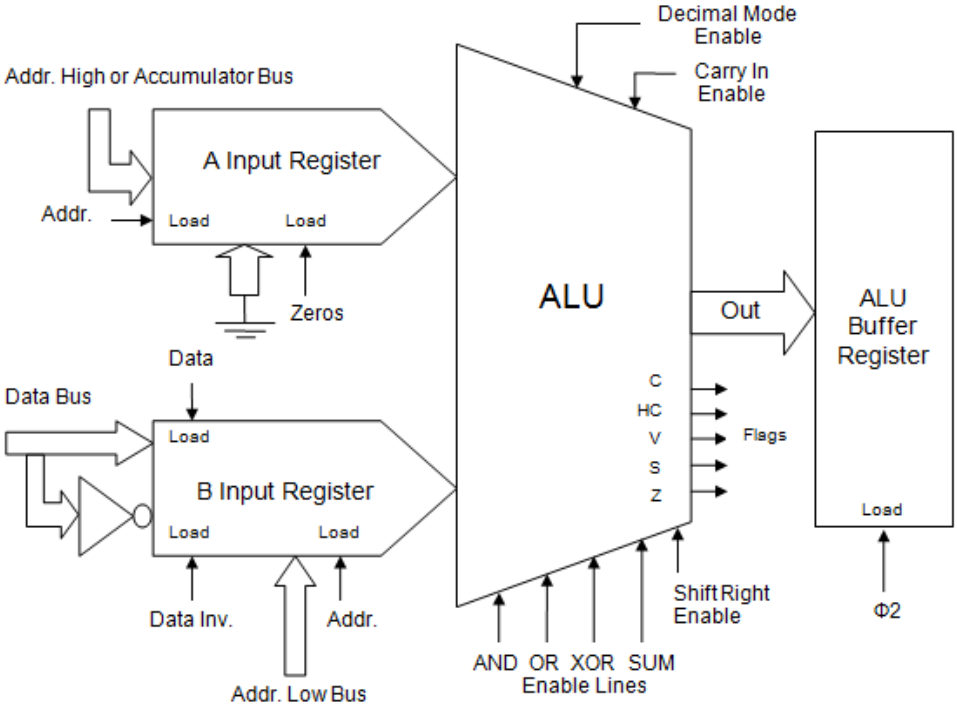


Figure 7: 6502 ALU

2.5.2 Address and Data Busses

There are three main bus lines within the 6502: data, address-high, and address-low. Each bus is 8 bits wide, allowing for 8-bit data words and 16-bit address

words. In general, the three busses are isolated; however, any one may be latched into another given the correct control signals from the Random Control Logic block. Given timing considerations, this allows for data to be transferred into an address space and vice versa.

The Φ1 phase clock controls the output registers. When Φ1 is high, the registers are latched, presenting any data on the address and data busses to the output busses, to be processed outside of the 6502. Φ2 controls the input data latch. It receives external data once Φ2 hits a positive edge. At this time, all processes within the 6502 are valid. These are operations such as instruction decoding, ALU processing, and stack interfacing. The two-phase system allows other external devices to operate while the processor is handling internal operations in the Φ2 phase. Through these means, a more efficient use of clock cycles is achieved.

Outside of the 6502, the data and address busses are connected in such a way as to properly interface the Vector Generator, Math Box, RAM, ROM, and any other components in the hardware. The data bus can be latched to the Math Box via some buffers and I/O select lines, determined by both the 6502 and the external hardware. It is it also connected to game ROM, 6502 processing RAM, and a Vector Generator data latch. Appropriate control signals are needed to access any of these components. These signals are decoded based on the address bus. In general, the address-high lines A8-A15 are used to determine which component is accessed. Since memory size was limited at the time this game was developed, small 2K blocks were connected together, with their chip-select lines acting as control signals to select which block was accessed. The chip-select (CS) lines became active through some intricate decoder routes tied to the address-high bus. In a similar fashion, the Vector Generator RAM can be written to or read from by decoding the top bits of the address bus.

All of these external components are asynchronous. Their input and output lines are updated immediately, given that their respective CS and enable signals are active. This allows all data to be accessed during the Φ2 clock phase, and new values to be calculated and latched during the Φ1 clock phase. For example, if the 6502 wishes to access vector information from the Math Box, process it, and send it to the Vector Generator to create the change in display vectors, it needs only a few cycles of these phase clocks. On the first Φ2 cycle, it may read data through the external latch connected to the Math Box. This data may be processed accordingly. When the correct address and data are available, in this case an instruction to draw a vector, it latches them to the output bus lines on a Φ1 phase. On the next Φ2 cycle, it can now latch the data bus to the Vector Generator, based on the address decoding and data latch signals. This data is then written to a specific address within the Vector Generator RAM, to later be processed and create a vector on the screen.

Atari© gives a detailed memory map, which summarizes how the address decoding works and what function is performed. It is important to note that this memory map does not contain a full address map, meaning from 0000h to FFFFh. This is due to the way the address decoding works. Since certain lines of the upper address correspond to specific functions, it is difficult, and sometimes unnecessary, to create a function for every combination of these lines. The Atari© Memory Map is shown in Table 3.

Atari© Memory Map										
Hexadecimal Address	**R/W**	**Data**								**Function**
		D7	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**	
0000-7FFF	R/W	D	D	D	D	D	D	D	D	Program RAM (2K)
0800-080F	W					D	D	D	D	Color RAM
0C00	R								D	Right Coin Switch
0C00	R							D		Center Coin Switch
0C00	R						D			Left Coin Switch
0C00	R					D				Slam Switch
0C00	R				D					Self-Test Switch
0C00	R			D						Diag. Step Switch
0C00	R		D							HALT
0C00	R	D								3KHz
0D00	R	D	D	D	D	D	D	D	D	Option Switch Inputs
0E00	R	D	D	D	D	D	D	D	D	Option Switch Inputs
2000-2FFF	R/W	D	D	D	D	D	D	D	D	Vector RAM (4K)
3000-3FFF	R	D	D	D	D	D	D	D	D	Vector ROM (4K)
4000	W								D	Right Coin Counter
4000	W							D		Center Coin Counter
4000	W					D				Video Invert X
4000	W				D					Video Invert Y
4000	W									VG GO
5000	W									WD CLEAR
5800	W									VG Reset
6000-603F	W	D	D	D	D	D	D	D	D	EAROM Write
6040	W	D	D	D	D	D	D	D	D	EAROM Control
6040	R	D								Math Box Status
6050	R	D	D	D	D	D	D	D	D	EAROM Read
6060	R	D	D	D	D	D	D	D	D	Math Box Read
6070	R	D	D	D	D	D	D	D	D	Math Box Read
6080-609F	W	D	D	D	D	D	D	D	D	Math Box Start
60C0-60CF	R/W	D	D	D	D	D	D	D	D	Custom Audio Chip 1
60D0-60DF	R/W	D	D	D	D	D	D	D	D	Custom Audio Chip 2
60E0	R								D	One Player Start
60E0	R							D		Two Player Start
60E0	R						D			FLIP
9000-DFFF	R	D	D	D	D	D	D	D	D	Program ROM (20K)
E000-FFFF	R	D	D	D	D	D	D	D	D	Reset/Interrupt Vectors

Table 3: Atari© *Tempest* Memory Map[15]

2.5.3 Instruction Set

The 6502 supports 56 total instructions such as Add With Carry (ADC) and Logical AND (AND). Many of these instructions can be expanded to contain multiple Op-Codes due to the addressing mode for the instruction. For example, the ADC instruction can take on the following address modes as shown in Table 4. The table also illustrates the Op-Code pertaining to that address mode, and the number of cycles need for execution.

Addressing Mode	Op-Code	Execution Cycles
Immediate	69h	2
Zero Page	65h	3
Zero Page, X	75h	4
Absolute	6Dh	4
Absolute, X	7Dh	4 (+1 if page crossed)
Absolute, Y	79h	4 (+1 if page crossed)
(Indirect, X)	61h	6
(Indirect, Y)	71h	5 (+1 if page crossed)

Table 4: 6502 Op-Codes for ADC Instruction[16]

These instructions are used in the 6502 to help pass data and control to the various components of the system. More specifically, these instructions will tell components such as the Math Box to perform specific instructions such as rotating the current image, then pass that to the Vector Generator for producing the appropriate output, and finally for storing any necessary data in memory. The instruction set for the 6502 is divided into four types of machine language instructions: R-Type, J-Type, I-Type, and Misc.

R-Type instructions are referenced as being register-type instructions. This type of instruction is the most complex in terms of converting the instruction into its binary components. A typical R-Type instruction in machine code can be seen as “add $rd, $rs, $rt.” Within this machine code, $rd, $rs, and $rt refer to registers. The semantics of this instruction example are R[d] = R[s] = R[t]. Based on the semantics, $rs is the destination register for the operation, whereas $rs and $rt are two source registers. The actual binary representation is formed as a 32-bit instruction that has the Op-Code stored first, followed by the two source registers, the destination register, the shift amount, and the function. This representation can be seen in Table 5. The Op-Code is specified based on the instruction and can be found in Table 5. The shift amount represents the number of bits to shift the result by and is used primarily in shift instructions. The function is used to help specify the operation, in addition to the Op-Code.

B_{31-26}	B_{25-21}	B_{20-16}	B_{15-11}	B_{10-6}	B_{5-0}
Op-Code	Register S	Register T	Register D	Shift Amount	Function

Table 5: Binary Representation for R-Type Instructions[17]

I-Type instructions are known as being immediate-type instructions. Typical I-Type instructions in machine code appear as “addi $rt, $rs, immed.” In this instruction type, $rt is the destination register, $rs is a source register, and immed is an immediate value. The semantics of this instruction are R[t] = R[s] + $(IR_{15})^{16}$ IR_{15-0}. IR represents the instruction register. $(IR_{15})^{18}$ means that bit B_{15} of the instruction register (sign bit for the immediate value) is repeated 16 times, and is then followed by IR_{15-0} (the 16 bit representation of the immediate value). The binary representation for this type of instruction is also formed as a 32-bit instruction, but differs in the fields represented. The instruction first stores the Op-Code followed by the destination register, the source register, and then the 16-bit representation of the immediate value. This format is shown in Table 6.

B_{31-26}	B_{25-21}	B_{20-16}	B_{15-0}
Op-Code	**Register S**	**Register T**	**Immediate**

Table 6: Binary Representation for I-Type Instructions

J-Type instructions refer to jump-type instructions. A typical J-Type instruction in machine code can be seen as “jmp target.” In the machine code, jmp represents the instruction and target represents the target address that the program will jump to. The semantics of this instruction are PC <- PC_{31-28} IR_{25-0} 00, where PC is the Program Counter storing the current address of the instruction being executed. The Program Counter is then updated by using the upper 4-bits of the Program Counter followed by the 26-bit representation of the target address, then followed by two 0’s. The binary representation of this address is formed as a 32-bit instruction that stores the Op-Code first, followed by the target address. This format is illustrated in Table 7.

B_{31-26}	B_{25-0}
Op-Code	**Target**

Table 7: Binary Representation for J-Type Instructions

Miscellaneous instructions are only used for No-Operation instructions (NOPs). NOPs are used to prevent hazards in programs and can also be viewed as inserting a stall. These instructions are 32-bits long but have no specific format, such that the upper 6-bits do not match any of the Op-Codes for the instruction set. A typical NOP instruction in machine code can be seen as NOP. The binary representation of this instruction is evident in Table 8.

B_{31-0}
1111 1111 1111 1111 1111 1111 1111 1111

Table 8: Binary Representation for Misc-Type Instructions

2.5.4 Timing Logic and Timing Decode

One unique feature of the 6502 is the two-phased clock processing. *Tempest* takes advantage of this feature by allowing the game’s modules (the 6502, the

Math Box, the Vector Generator, etc.) to compute during one clock phase (Φ1) and read new addresses from the 6502 during the second clock cycle (Φ2). Internally, the 6502 has a clock generator which produces Φ1 and Φ2 signals from a single Φ0 clock signal. During the Φ1 phase, all of the computations are performed. During the Φ2 cycle, the 6502 performs all loads needed for the next operation and loads the data which was computed during the previous Φ1 phase. The Address Bus, the Data Output Register and the Instruction Register are all clocked in during the Φ2 phase. After the new data has been clocked in, Φ1 will allow the computations to begin.

Specifically, the Φ2 signal also controls the load of the Input Data Latch, which sets the next Program Counter, Data Output Register, Address Registers and others. The Input Data Latch also sets the Accumulator, which in turn has some control over the Stack Pointer. In addition to clocking the Input Data Latch, Φ2 clocks the Predecode Register, which loads the next line of data to be processed by the Predecode Logic.

External to the 6502, the Φ2 clock signal that is generated controls the High Score Memory, Player Input and the Vector Generator. For the High Score Memory and Player Input, Φ2 is fed into the Auxiliary Board Address Decoder through the External Address Bus. This signal is applied the logical AND operation with a 3MHz clock and an ER/$\overline{W}$B signal whose resulting output is either passes as a clock signal to the Player Input module or the High Score module. The Φ2 signal in the Vector Generator module will be discussed in Section 2.7.6.

The Timing Generation Logic within the 6502 provides the timing scheme for the Decode ROM and also the Random Control Logic. This timing module is what determines when an instruction has had the required amount of clock cycles to finish execution. Once this determination has been made, the module allows the 6502 to decode the next instruction while also setting the associated control lines for the decoded instruction. The Timing Generation Logic uses control lines T0-T6 and T1X to signal the decode ROM and the Random Control Logic module that they are to proceed with their operations.

2.5.5 I/O

In addition to sending a clock signal to the rest of the game, the 6502 is used to handle the addresses and dataflow. The 6502 has an 8-bit bi-directional data bus and a 16-bit address bus, which is used to choose the microprocessor's instruction. The address lines and bi-directional data bus are connected to the Address Decoder, ROM and RAM modules. Depending on which address is written to the Address Bus, three different modules can be activated. The specific address map is described in Table 4. If the corresponding ROM address is written to, the address is read by two separate modules. First is the Address Decoder, which is a series of 2-to-4 decoders. Whichever ROM is specified by

the Address Bus is activated by the Address Decoder. The address is also sent to the ROM itself, which reads data from the desired address. This data is written to the External Data Bus. Further discussion about control and function of the ROM can be found in Section 2.13. Another module which can be activated is the RAM. The RAM behaves similar to the ROM, except the RAM can also be written to, so the External Data Bus can also write data to the specified location.

Another module which can be written to is the Auxiliary Board Address Decoder. This address is again decoded. Based on the desired function, the High Score Memory, the Player Input/Output module or the Math Box is activated. When one of these modules is called, they can then operate on the data provided by the External Data Bus. All of these modules can then write data back to the External Data Bus upon completion.

2.5.6 Program Counter

Within the 6502, the location of the current instruction is provided by the Program Counter. This Program Counter is split up into two 4-bit registers. Each has their own logic for incrementing the current address. The Program Counter for the lower 4-bits has a carry-out line that is connected to the counter for the upper 4-bits. Each of the registers get the next address to be executed as specified by the Input Data Latch as an input to the Program Counter Low/High Select registers (PCLS and PCHS). The function of the counter is specified by various control lines that are set in the Random Control Logic. These control lines are decoded during the Instruction Decode ROM.

2.5.7 Stack Pointer

In the 6502, the Stack Pointer is an 8-bit register which is used to store data from the Accumulator, Program Counter, or the ALU. The stack pointer is mainly used to store variables when a function call is made during the program's execution. The Stack Pointer operates in a "First In, Last Out" manner. In addition to holding information for the ALU, the Stack Pointer also is used by the Program Counter to store the return address of the program in the event of a jump or branch statement. The Stack Pointer is told to fetch data either from the ALU or the Program Counter based on the control lines coming from the Random Control Logic.

2.5.8 Decimal Adjust Adders

The Decimal Adjust Adders are used to convert numbers from binary format to decimal. It is split up into two separate adders; one is for the lower 4-bits and the other is for the upper 4-bits. After performing the necessary shifts within the conversion, the result is then forwarded to the Accumulator. The decision to make the conversion is determined by control lines from the Random Control Logic and the flags represented in the Processor Status Register.

2.5.9 Instruction Decode Logic

Data is passed into the 6502 microprocessor through eight 1-bit data lines labeled as D7-D0. These lines are passed into the Predecode Register and into the Input Data Latch. Once the data is clocked into the Predecode Register on a positive edge of the Φ2 clock signal, the data is fed to the Predecode Logic block. This Predecode Logic forwards the data to the instruction register while also sending a control line to the Timing Generation Logic. The control line fed to the Timing Generation Logic is a signal that determines how many cycles are needed for execution of the instruction that is next in line for execution. For example, if an Add with Carry (ADC) instruction with an immediate addressing mode is to be executed, then the Predecode Logic will signal the Time Generation Logic that the current instruction requires 2 cycles to execute.

After going through the Predecode Logic, the data representing the instruction is passed into the Instruction Register. This register only serves as a way to hold the instruction that is to be executed next. When the time is deemed appropriate by the Timing Generation Logic, the Decode ROM is clocked so that it reads in the next instruction from the Instruction Register. This data that is fed into the Decode ROM represents where in the ROM to look up the associated control lines that need to be set high in the Random Control Logic. The 6502 Decode ROM is a 130 x 21 bit decode ROM (130 lines of 21 bits each) that takes in the inputs from the Instruction Register and Time Generation Logic.[18] All lines of the Decode ROM compare the instruction and the current clock cycle to determine if there is a match. If there is a match, then the ROM sends the resulting output from the instruction line. A simplified version of what each line would look like is shown in Table 9.

ON Bits								OFF Bits								Timing Bits					
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	T6	T5	T4	T3	T2	T1

Table 9: Example Line from 6502 Decode ROM[18]

In Table 9, "**ON Bits**" specifies which bits should be set high for this line to fire and "**OFF Bits**" specifies which bits should be clear for this line to fire. Examples of these fields can be seen in how the branch Op-Codes are encoded: aab10000. In the example for the branch Op-Codes, "aa" is the condition and "b" decides if the branch should be taken on a set or clear flag. The condition for this example specified by "aa" is given by the following flag statuses: N = 00, V = 01, C = 10, Z = 11. Table 10 shows a line that would fire on the first cycle of any branch. A more readable version is also located in Table 10 below the set of bits for the line.

ON Bits								OFF Bits								Timing Bits					
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	T6	T5	T4	T3	T2	T1
0	0	0	1	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	1
Mask								Cycle	Description												
X	X	X	1	0	0	0	0	T1	T1 of Bcc: Fetch Branch Offset												

Table 10: Example of Firing Line from Decode ROM[18]

When a line fires from the ROM, the output is a 1. This output is fed into the Random Control Logic, which AND/OR-combines the output from the Decode ROM to create a set of control lines that aid in passing the result into various parts of the 6502.

2.5.10 Program Status Register (Flags)

The Program Status Register is an 8-bit register that consists of seven flags. These flags are set by the instructions being executed. The flags are labeled as the Carry Flag (C), Zero Flag (Z), Interrupt Disable (I), Decimal Mode Flag (D), Break Command (B), Overflow Flag (V), and the Negative Flag (N). Therefore, bit 5 is always set to 1, since it is not used as a specific flag. Table 11 provides a description for what determines if a specific flag needs to be set.

Identifier	Flag Name	Set Condition
C	Carry Flag	When set to 1, indicates that an arithmetic operation has a carry out from the result
Z	Zero Flag	When set to 1, indicates that an arithmetic or logical operation produced a result of zero
I	Interrupt Disable Flag	When set to 1, indicates that an interrupt signal (RESET, NMI, IRQ, ABORT) has been detected and is disabled
D	Decimal Mode Flag	Indicates that the number is a BCD or a Binary number – Set to 1 for BCD and set to 0 for Binary
B	Break Command Flag	When set to 1, indicates that a software interrupt has been executed
V	Overflow Flag	When set to 1, indicates that an arithmetic operation produced an overflow
N	Negative Flag	When set to 1, indicates that an arithmetic operation produced a negative result

Table 11: Flag Labels and Set Conditions

2.5.11 Random Control Logic

The Random Control Logic module within the 6502 has the sole purpose of sending the appropriate control lines to the various parts of the microprocessor. This module takes inputs from the Decode ROM, Timing Generation Logic, and also the Interrupt and Reset Control. The Timing Generation Logic determines if the current instruction has used the required number of cycles needed for

execution. This timing signal keeps the control lines required for the given instruction set high for this length of time before allowing them to reset or change to the configuration needed for the next instruction. The input lines from the Interrupt and Reset Control are used to detect if a reset or interrupt to the microprocessor has been triggered. If the Random Control Logic is told that a given interrupt has been triggered, then it sets a given set of control lines that aid in setting the interrupt vector fetch address, the zero-page, and stack high address on the next Φ2 clock signal.

The inputs from the Decode ROM tell the Random Control Logic to do a multitude of processes. These processes include, but are not limited to loading data into the accumulator, allowing the Program Counter High and Low to increment, or even triggering the ALU to perform a given operation. For example, if the Decode ROM instructed the Random Control logic that it should load data from the SB Bus onto the X and Y Index Registers, the for control lines (two for the X Index Register and 2 for the Y Index Register) would be set as shown in Table 12.

Control Line	Status	Description
SB/Y	0	Loads data from the Y Index Register onto the SB Bus
Y/SB	1	Loads data from the SB Bus onto the Y Index Register
SB/X	0	Loads data from the X Index Register onto the SB Bus
X/SB	1	Loads data from the SB Bus onto the X Index Register

Table 12: Control Lines to Load X and Y Index Registers

The Random Control Logic has a total of 62 control lines that are passed throughout the 6502. Many of these control lines may be activated at the same time, as seen in the above example showing data being stored to the X and Y Index Registers, but that isn't always necessary. For instance, some instructions may require that only the X Index Register be set to load data from the SB Bus, while the Y Index Register may be instructed to put data onto the SB Bus for the X Index Register. At any given time, all 62 control lines will always have a value of 1 or 0, since they are 1-bit control lines.

2.6 Math Box

The concept of the Math Box was first designed for use in the Atari© game *Malibu Grand Prix*, but the game ended up being scrapped due to limited capacity of the hardware and limited graphics.[19] After modifying the idea of the Math Box for *Red Barron* and *Battlefield*, Atari© continued to improve its functionality and later released it as a part of *Tempest*. Understanding how the Math Box functions has proven to be difficult for many game enthusiasts throughout the years, but our understanding of its processes is described below.

The Math Box is treated as a memory mapped I/O device that is activated by writing to the external address bus.[19] The external address bus enables a microcode address from within the program counter that begins execution on a pre-specified routine. This address deals with three given components which include the Play Inputs and Audio Output, the High Score Memory, and the Math Box. The address bus and data bus are activated by a write of the 6502 microprocessor to the main ROM locations of 6060h-609Fh in the memory map shown in Table 3. The 8-bits stored in the address bus serve a multitude of purposes. The lower 5 bits control the A1 chip, which decodes the address within the Math Box ROM for the specified instruction to be executed. The A1 chip is a ROM chip (136002). This provides the beginning location for the pre-specified math function that is to be executed. Table 13 illustrates what components within the Math Box and its associated Address Bus Decoded are controlled by the respective bits.

Address Bus Bit Numbers	Components Controlled
EAB0-EAB4	A1 chip within Math Box
EAB4-EAB5	Bit-slice Processors with values of 10 and 11
EAB6-EAB7 ($\overline{I/O}$ Line as Select Line)	01 = Control Line for EAB4-EAB5 Selections 10 = MSTART Line to eventually control BI chip 11 = Control Line POKEY Selections

Table 13: Address Bus Line Controls

The Math Box uses the microcode to perform the desired operation on the data provided, which is viewed as rotating or drawing shapes on the screen. Once one operation has been performed, a series of logical units determine if the microcode should continue performing operations on the data or if the single operation is sufficient. Once all of the operations that needed to be performed have been executed, the resulting data is latched back to the External Data Bus. This bus is then read by other components such as the Vector Generator or the 6502 microprocessor.

One of the main controls for the Math Box is the $\overline{BEGIN}$ signal. When this signal is set to low, it enables the A1 chip which fetches the initial address for the microcode. Otherwise, the B1 chip is enabled so that the previous operation information is looped back for the next operation. The B1 chip is a D-type Latch (74LS374). This data flow is summarized in Figure 8. The data passed through the A1 and B1 chips then enter the Program Counter that provides the correct address to look up in the ROM. The control line $\overline{PCEN}$ for the Program Counter is enabled when $\overline{BEGIN}$ is high or there is data presently being operated on by the Math Box.

Once the Program Counter is enabled, it increments the current address in the ROM to provide the next address to be looked up. This address is then read in the ROM, which decided the operation to be performed and controls the data on the four 2901 bit-slice processors that actually carry out the operations. That address is then forwarded back to the B1 chip and awaits the next operation to be performed. Looping the data, continues until certain control bits LDAB, A12 (Accumulator 12), and S are set by accessing a specified location within the ROM. These bits correspond to the O4 register within the ROM. LDAB controls the B1 chip's clock by disabling the clock input. A12 controls the STOP line, which we believe terminates all operations within the Math Box. The S control bit aids in controlling the setting of the $\overline{PCEN}$ line for the Program Counter.

A general block diagram of a new operation occurring in the Math Box, through the A1 chip, is shown in Figure 8. Following that figure shows the routine of carrying out the operation after the instruction has been decoded by the A1 chip. The initial pass through the routine cycle is shown by the orange arrows, while the blue arrows illustrate the looping done to carry out the remaining operations to be executed for the initial instruction called by the External Address Bus.

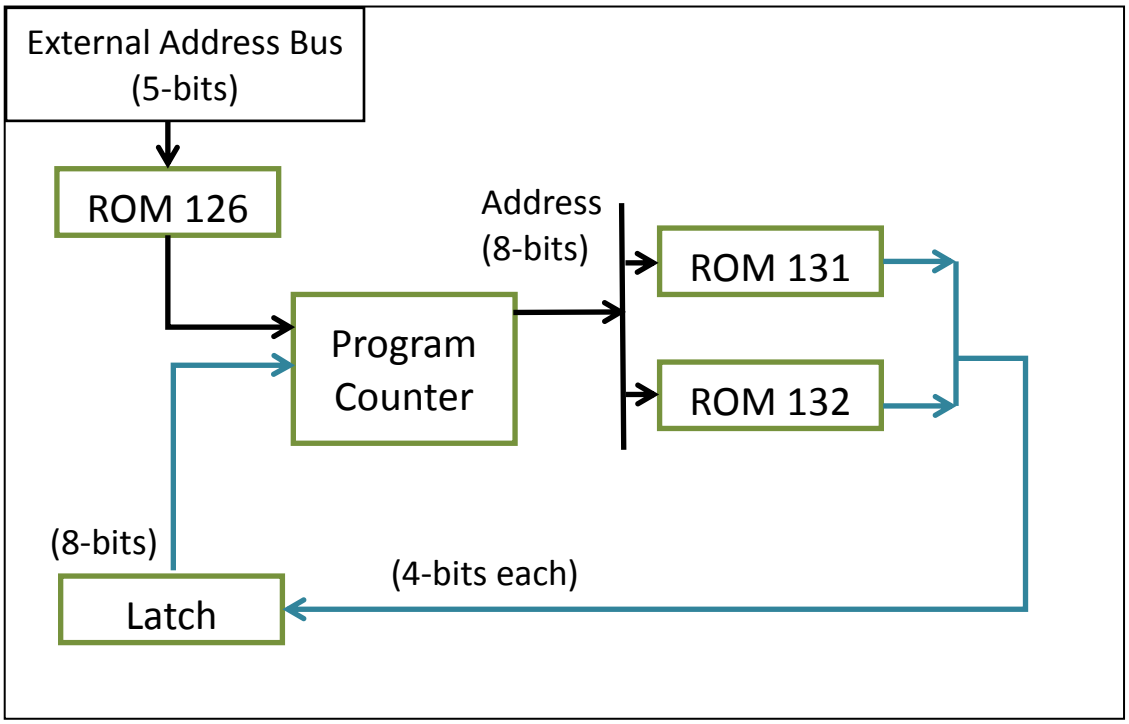


Figure 8: Math Box Block Diagram A

2.6.1 Trace-Through Example

This example will illustrate all of the controls and memory accesses within the ROMs for the Math box given that the 6502 microprocessor writes data to the address 6080 within the Memory Map. This places the hexadecimal value of 80 on the External Address Bus. This is treated as a binary value stored on bits

EAB7-EAB0 of the External Address Bus. Bits EAB4-EAB0 are sent to the address decoder A1 with the value of 00000b. This looks up location 0 that has the hexadecimal value of 20 stored at that location. In turn, the hexadecimal value of 20 is forwarded to the two Program Counters that are each 4-bits wide. Between these two counters, the ripple carry out of the lower 4-bits is connected to the carry in bit of the upper 4-bits. The next step is for the Program Counter to latch this current address to the Math Box ROMs because pin 7 (EP) and pin 10 (ET or $\overline{PCEN}$) of the Program Counter chips are set low initially. For other functions, the Program Counter will increment if these pins are both set to high on the same clock cycle.

The output of the Program Counter is the address that is read within all six of the Math Box ROMs (E1, F1, H1, J1, K1, L1), which all output a separate 4-bit value. Address 20 has the following hexadecimal values stored for the respective ROMs: L1: 0, K1: 1, J1: B, H1: 3, F1: 4, E1: 0. All of the values for L1 and K1 are passed to the four ALUs to control the function, while also being passed back to the B1 chip. Bits O1, O3 and O4 of J1 and bits O1- O3 of H1 and F1 are also passed to the ALU to control functions. All bits from E1, bit O1 of J1, and bit O4 of H1 and F1 are used as various control lines within the Math Box and the ALUs.

The ROMs outputs serving as control lines are forwarded to three main control blocks. The first control block is determined by the O4 bit of the H1 ROM. This O4 bit is first passed into a D Flip-Flop. In our example, the value of this O4 bit is a 1; therefore the output of the D Flip-Flop is a 1 on a positive-edge clock. This output is then passed to a logic gate that inverts the value along with the other input, a 3MHz clock signal, before performing an AND operation. The result of the logic gate is branched in the following manner: 1) the output is inverted and returned back as the clock input of the D Flip-Flop previously mentioned, 2) the output is passed to the next main control block, the clock input of the Program Counters, and the clock input of the ALUs. This situation is illustrated in Figure 9.

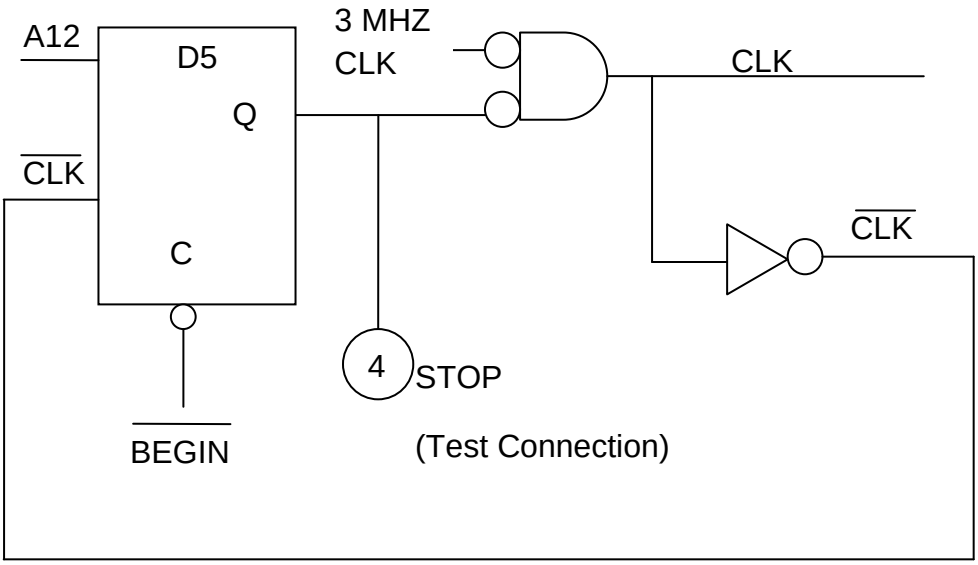


Figure 9: Math Box Control Block 1

The next main control block begins with another D Flip-Flop, which has the inputs of the clock signal passed from the previous control block and the Q0 bit. The determination of the Q0 bit will be discussed later in this section. The inverted output of this D Flip-Flop is then passed to an AND logic gate. The AND gate performs the AND operation of this signal with the O2 bit from the E1 ROM. The resulting output is passed to an XOR gate that takes in the other input as the O2 bit of the J1 ROM. In our example, the Q0 bit is undetermined and the output of the D Flip-Flop is sent to an AND gate. The other input, O2 bit of E1, is a 0, which will ultimately produce an output of a 0. Also in our example, this 0 valued output is passed to an XOR gate with the input of the O2 bit of J1 being 1. This output of the XOR gate, a 1, is sent to the ALUs as the A10* bit. A summary of this situation is illustrated in Figure 10.

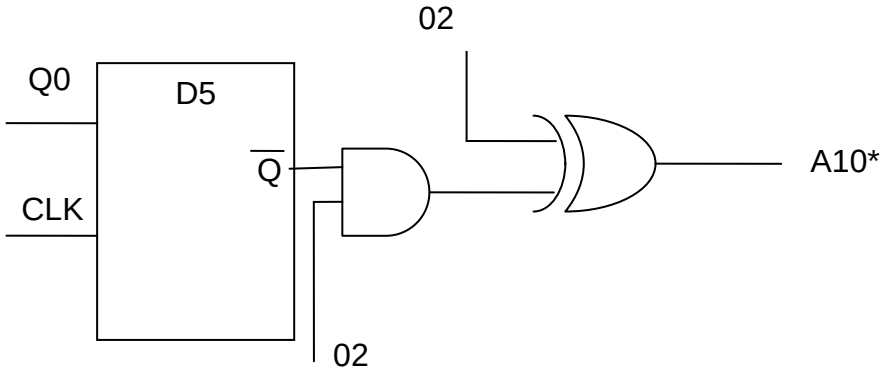


Figure 10: Math Box Control Block 2

The last of the main control blocks begins with the following inputs: the O2 bit of the F1 ROM, the O3 and O4 bits of the E1 ROM, MSB of the E2 ALU, the overflow (OVR) bit of the E2 ALU, and the $\overline{BEGIN}$ signal from the first of the main

control blocks. This control block performs a of series of logical operations that are described as follows. The MSB and OVR (S1 and S0) of the E2 ALU are passed as inputs to an XOR gate. The values of these two inputs are undetermined at this point. The result of this XOR gate is then passed to a NAND gate that takes an additional input, S (O4 bit of E1 ROM), which has the value of 0 in our example. The resulting value from the NAND operation is then passed as an input to two other NAND gates. This point will be referred to as P1. The remaining inputs for P1 are obtained in the following ways. The O2 bit of the F1 ROM is split into four signal lines that are each fed as the two inputs to two different NAND gates. Since the input signals are the same, each NAND gate performs an inversion of the O2 bit of the F1 ROM. Neither of the two NAND gates are connected in series, but are rather connected in parallel. The output of the first inversion is tied into the Q0 line of the K/L2 ALU, which is then passed back as an input to the D Flip-Flop mentioned in the second main control block. The remaining NAND gate that performs an inversion is passed as in input to one of the NAND gates in P1. The resulting output from one of the NAND gates in P1 is passed to R3 pin of the E2 ALU, which is the register stack of the ALU. The remaining NAND gate from P1 takes in the first input to P1 that was mentioned and the O3 bit of the E1 ROM. The output of this NAND gate is passed as in input to a NOR gate that has the other input as $\overline{BEGIN}$. Due to De Morgan's Law, this NOR gate acts as an AND gate of these two signals because the inputs are inverted just before they enter the NOR gate. The resulting signal turns the $\overline{PCEN}$ line that is passed back to the Program Counters. The values for each stage of this control block within our example are shown in Figure 11.

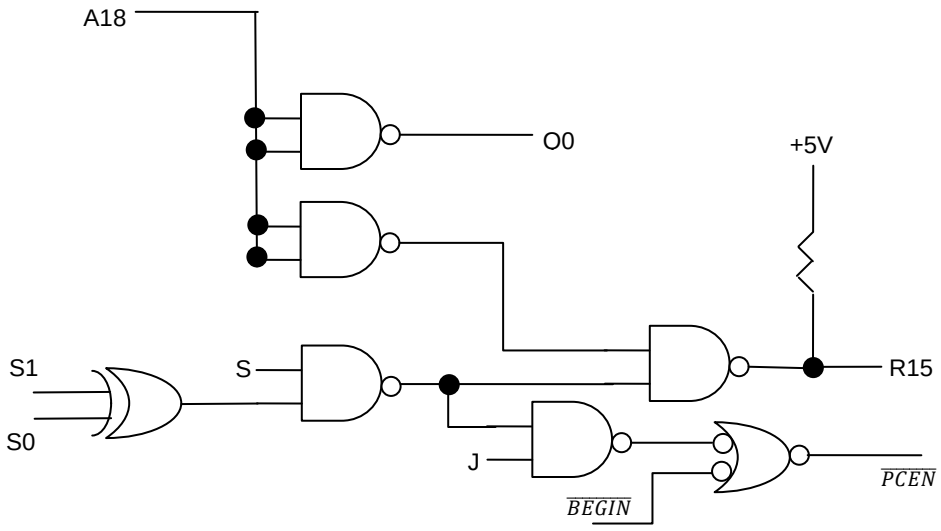


Figure 11: Math Box Control Block 3

After the initial pass through the Math Box with the address initially passed in by the External Address Bus, either the Program Counter is incremented or the O0-O4 bits for the J1 and K1 ROMs will be used to select the next address that is read within the Math Box ROMs. This action is determined based on the new

present values of the control bits previously stated. The cycle will continue until the control bits have determined that the desired function has been executed.

2.6.2 Arithmetic Logic Units (ALUs)

The ALU block within the Math Box is comprised of four Am2901 bit-slice processors. Each processor can operate on two two-bit signals, so they are tied together in series to act as a single 8-bit ALU. The four ALUs (E2, J2, F/H2, K/L2) are set up so that they may perform ripple carry operations. The ALUs are organized in such a way that the E2 ALU operates on the MSBs of the 8-bits and the K/L2 ALU operates upon the LSBs of the 8-bits. The address lines from the Math Box ROMs choose the functions that are to be executed on the bi-directional data lines. Since the External Data Bus is bi-directional but the ALUs have separate pins for data in and data out, the original game simply split the bus, and fed the same signals into the input and output pins. While each ALU can only perform simple shift and logical operations, the microcode from the Math Box ROMs will allow for more complex operations.

2.6.3 Math Box Auxiliary Board Address Decoder

The Auxiliary Board Address Decoder for the Math Box is organized as a series of 2-to-4 decoders, along with some other simple logic gates. Each of these decoders is controlled by EAB4-EAB7 bits of the External Address Bus. The varying combinations of these bits dictate what components are active. The components that are controlled include the Math Box, High Score Memory, and also the Player Input and Audio Output sections.

2.7 Vector Generator

Vector graphics were used by a number of different video games, most notably Atari© *Tempest* and Atari© *Asteroids*. Vector graphics use a multitude of basic geometric elements such as points, curves, and shapes. These basic elements are all determined from known mathematical equations. The complement to vector graphics is raster graphics, which takes arrays of pixels and arranges them in such a way that an image is produced. In Figure 12, the difference between vector graphics and raster graphics is illustrated when a portion of the image is magnified.

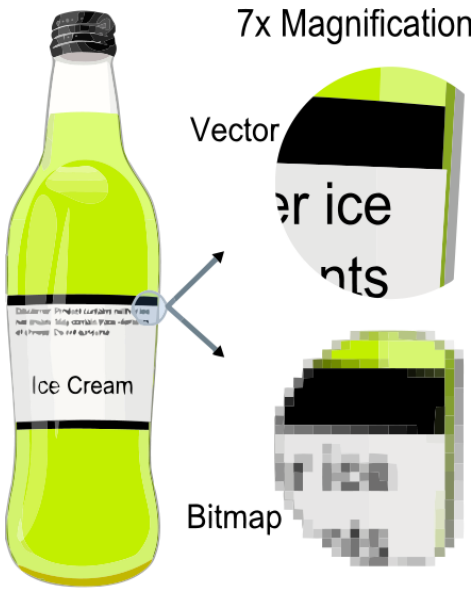


Figure 12: Vector Graphics vs. Raster Graphics[20]

The capabilities of microprocessors were extremely limited in the late 70's and early 80's. In order to meet the demands of games such as *Asteroid* and *Tempest*, Atari© invented the Vector Generator or VG. It is a very simple microprocessor, implemented entirely using discrete TTL ICs. While the 6502 microprocessor handles game logic, the VG is responsible for drawing the vector graphics to be displayed on the screen. Below is a list of the VG's core features.

- Outputs changes in X-Y vector coordinates, 10-bit 2's complement numbers
- 13-bit address bus
- 8-bit data bus
- 1024x1024 resolution
- Program counter and stack (4 words max)
- State machine with 8 micro-instructions
- VG ROM with instructions to draw predetermined shapes
- 16 levels of brightness intensity
- Linear and Binary scaling capabilities

As stated, the Vector Generator outputs the change in X-Y coordinates. This digital information is converted to an analog voltage which is then used to draw the change in the line. A vector is always drawn relative to its previous position. For example, to draw an "L", the coordinate output might appear as seen in Figure 13. The corresponding change in vectors is Step 1 and Step 2 in the figure below.

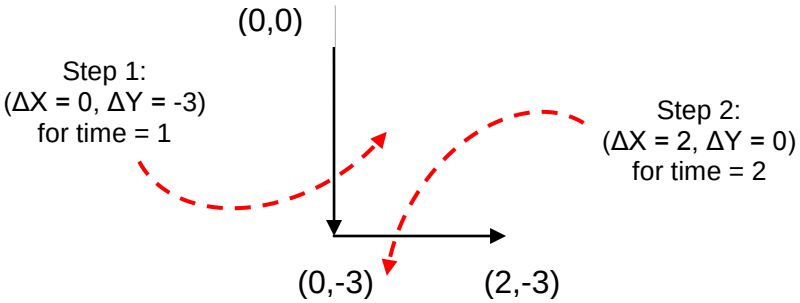


Figure 13: Example VG Output for "L"

The Vector Generator is also formatted as a little endian machine. This means that for any given two words, the Vector Generator will process the second word and then the first word. If the address space below is processed, the words will be read in the following order. Table 14 provides a set of example addresses and the data associated with those addresses, while also providing an example of the Read Order.

Address	Data
400h	47h
401h	6Ah
402h	F0h
403h	15h

Read Order: 6Ah, 47h, 15h, F0h

Table 14: Example VG Address and Data Table

2.7.1 State Machine and Instruction Set

The Vector Generator uses a state machine to implement micro-instructions through its hardware. The original state machine was enacted with 1K-bit TTL PROM (256 x 4). This memory receives, in order from MSB to LSB, a halt bit, a 3-bit operating code, and a 4-bit previous state value. Using the current Op-Code and previous state, the address for the next 4-bit state is generated. This means that a total of sixteen states are available, however only twelve are actually used. The Op-Code remains constant until the state to latch a new code is reached. If the halt bit (MSB) goes low at any time, the next state will be the "Idle" state. As the name implies, this state does nothing except wait for the next clock cycle to calculate the new state.

Shown in Figure 14 is the general state machine execution order. This state diagram has been derived through analysis of the *Tempest* Vector Generator hardware and state machine ROM. It is has been slightly modified from the state machine seen in the Analog Vector Generator document released by Atari©.

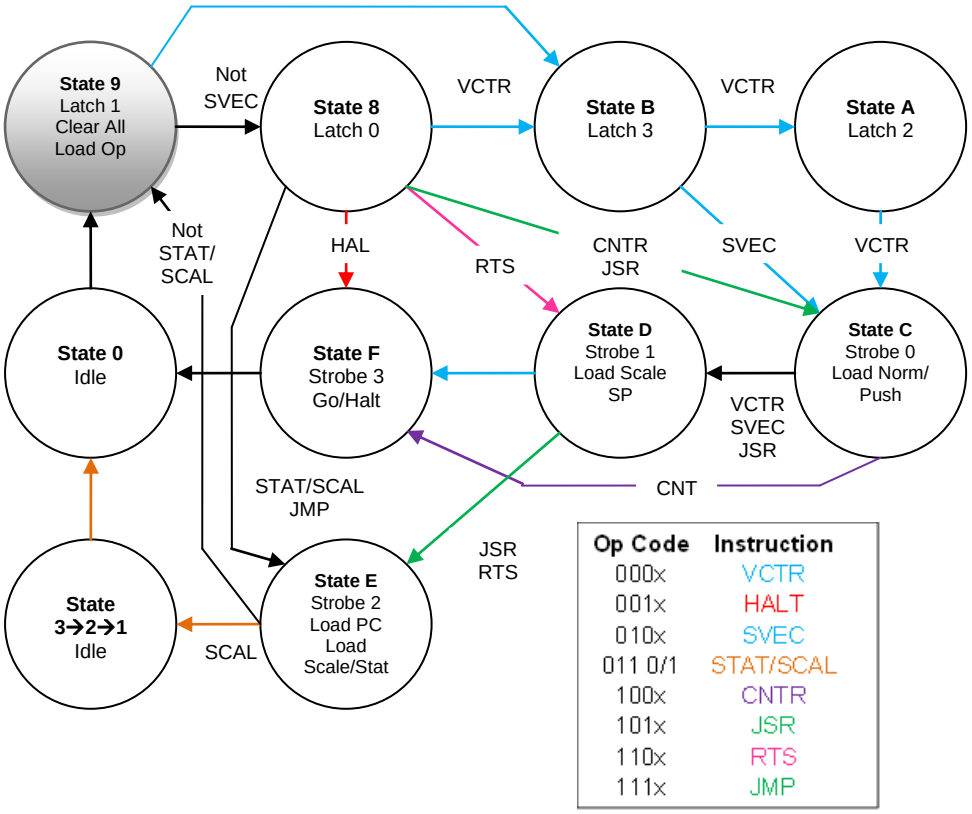


Figure 14: Vector Generator State Machine

All instructions begin in state 9, where the output data is cleared and a new Op-Code is loaded. Using this Op-Code and the current state value of 9 (1001), the next state is calculated. For example, if the Op-Code is 010, then SVEC is the current instruction, and the next state will be B (1011). The state value is used in a decoder to control the latch and strobe signals. As shown in the state diagram, for state B, latch 3 is set. Since most TTL logic in the early 80's was active low, latch 3 in the Atari© Vector Generator is set to 0. Similarly, any other state will set its strobe or latch control line to 0. Doing so implements a variety of functions, some of which are shown in the state diagram.

An explanation of each instruction's function and format is shown by Table 15.

Instruction	Bytes	Byte Information		Function
Vector (Long)	4	ΔY 8 LSBs	000, ΔY 5 MSBs	ΔX and ΔY are 13 bit 2's complement #'s. Z is 3 bit intensity information.
		ΔX 8 LSBs	ZZZ, ΔX MSBs	
Halt	2	xxxx xxxx	001, xxxxx	Halts the VG. Sets Halt flag.
Vector (Short)	2	ZZZ, ΔX 5 bits	010, ΔY 5 bits	ΔX and ΔY are 5 bit 2's complement #'s. Z is 3 bit intensity information.
Stats	2	ZZZZ, xxxx	011, 0, x, EN, H/L', I/O'	Z is a 4 bit intensity value used when 3 bit Z in VCTR or SVEC is 001. EN, H/L' and I/O' define a blanking window
Scale	2	8 bit linear	011, 1, x, 3 bit binary	Binary scale affects vector timer. Linear scale affects analog voltage output.
Center	2	01, xxxxxx	100, 00000	Center the beam to the middle of the screen
Jump SR	2	8 bit addr. LSBs	101, x, 4 bit addr. MSBs	Jumps to subroutine at 2 * 12 bit addr. Stores next address instack.
Return SR	2	xxxx xxxx	110, xxxxx	PC is now equal to the address at the top of the stack.
Jump	2	8 bit. addr. LSBs	111, x, 4 bit addr. LSBs	Jumps to address at 2 * 12 bit addr. to continue execution

Table 15: Vector Generator Instructions Format and Descriptions

In general, the latch control lines receive data from the data bus. Depending on the latch currently active, the data lines will be latched to different parts of the Vector Generator or to the output data bus. For example, when latch 3 is active low, the top three bits of the data bus are sent to the Z output register. This register is tied to hardware which implements brightness or intensity of the vectors being drawn. Bit 4 is the sign bit for the ΔX vector, which is tied to corresponding logic to this information. The remaining four lower nibble bits are sent to the DAC to be converted to analog voltages. Each latch performs other functions similar to those stated above.

Once all data is latched, the strobe states perform some micro-processing control depending on the instruction. For example, if the instruction is SCAL, during state E, strobe 2 will load the binary and linear scale information into the appropriate hardware in order to be used on the next vector draw. Most of the strobe functions are described elsewhere in this section.

2.7.2 State Machine Control

In order to control when the next state is updated, the Atari© Vector Generator uses four clock signals: 12 MHz, 6 MHz, 3 MHz, and 1.5 MHz. These signals control the register which stores the previous-state feedback for the state machine. The register will only be updated when all of these signals are zero. There are some additional constraints such as bits 2 and 3 of the current state, as well as a signal from the 6502 processor. These signals are set in place to provide the means to temporarily inhibit the state machine from updating the next state. The digital circuitry which implements this control is the basis for the cycle time of each instruction.

In addition to next state control, this logic also controls the little endian function in the Vector Generator. The LSB of the address is initialized to '1' at the start of any instruction. On the next state, this bit is toggled. This causes the second byte of a word to be read first. The LSB of the address toggles only during the latch states. Otherwise, the LSB is set to '1.' The rest of the address bus is tied either to the Vector Generator program counter or the address bus of the 6502 processor. Depending on the address sent from the 6502, the address decoding logic will selected between these two addresses in order to read and execute the next instruction. The program counter is controlled by the LSB of the address. Whenever it generates a positive edge, the program counter will increase. For example, if the PC and the LSB of the address are equal to both equal to '1', then the effective address is 3h. The LSB will toggle to '0', which will change the effective address to 2h. When the LSB toggles back to '1', the program counter will increase to 2h, and when combined with the LSB, creates an effective address of 5h. Thus, the functionality of little endian addressing is generated through the state machine timing and control logic simply by toggling the LSB of the address at appropriate times.

2.7.3 Vector Timer

This portion of the Atari© Vector Generator is basically a counter which determines the length of time it takes to draw a given vector. It is only active when executing VCTR, SVEC, or CNTR instructions during State F. The counter will increase until full, at which time it will cause the STOP signal to activate and the state to be changed.

The load line on the counters is tied to the scale and norm signals. Norm is a function of the sign bit and MSB of the X and Y vectors. Scale is active when binary scaling information is present. These signals determine whether or not the load line is active. If active, the counter output will be loaded into the counter input, shifted by one bit to the right. This causes the counting process to finish rapidly, which in turn stops the drawing of the vector. In this fashion, the vector is effectively scaled down by an amount relative to the actual value sent to the output.

2.7.4 Vector Timer Control

The control of the Vector Timer is based on the instruction being executed. This hardware consists of several logic functions to handle specific instruction micro-processes. As stated earlier, the Norm signal is set based on the sign bit and MSB of the X and Y vectors. The value is clocked during state C, when strobe 0 is active. When SCAL is being executed, the binary scaling value is loaded into a register. This register stores the value until the next vector or centering instruction is carried out. It also loads the linear scaling information into a DAC on the output circuitry. The binary value is loaded into a counter during these instructions and counts down on every cycle. Once the minimum value is reached, counting stops and the Scale signal activates in the Vector Timer circuit. Similarly, the STAT instruction loads its information into the output circuitry, which updates the default intensity value and creates parameters for generating a blank vector window. All of this happens when strobe 2 is active low. Figure 15 illustrates some of the schematic of the circuitry designed by Atari© to control the Vector Timer.

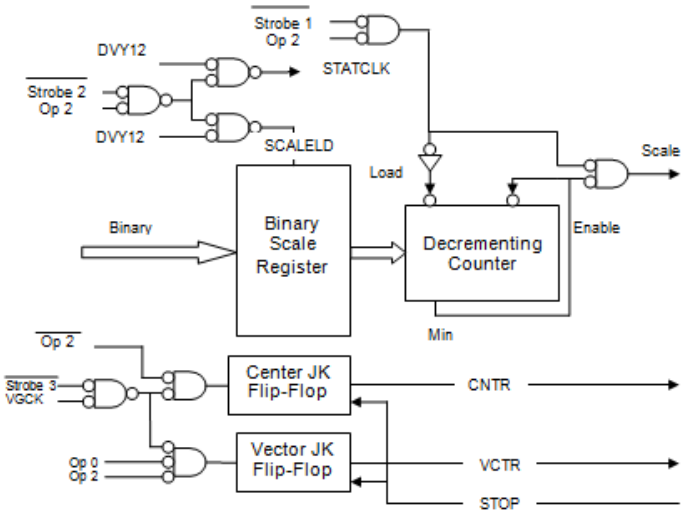


Figure 15: Vector Timer Schematic

If VCTR, SVEC, or CNTR is being executed, and strobe 3 is active low, the Vector Timer will be activated using a JK Flip-Flop. It will continue counting until all carry bits are '1', meaning the counter is full. At this time, the Stop signal is fed back to the K line on the JK Flip-Flop, which causes the next state, Idle, to be read.

2.7.5 Program Counter and Stack Pointer

The Vector Generator includes a simple program counter, which contains the next address to be read and the stack pointer, capable of holding four addresses.

It is only used for the JSR, RTS, and JMP instructions, since these deal with changing execution flow. If the instruction is JSR or JMP, the address value on the data lines will be clocked into registers tied to the program counter. For JSR, the next address, or the value currently on the address bus, is stored during state C, when strobe 0 is active. The next state will either increment or decrement the stack pointer, depending on if it is RTS or JSR. If it is RTS, the value that the stack pointer is now addressing will be read out to the program counter latch.

When strobe 2 is active, the next state, E, will cause whatever was on the register lines to be read into the PC lines. If it is a RTS instruction, these lines come from the stack pointer. If the instruction is JMP or JSR, these lines come from the register lines clocked during state C. Since the LSB address line comes from the state machine, the effective address will always be the 12 bits from these lines plus the LSB. The LSB is combined with these to form the new address where data is read.

2.7.6 Address and Data Busses

Since the 6502 microprocessor and the Vector Generator must communicate, some control logic for the address and data busses is needed. The Vector Generator receives a clock signal Φ2 from the 6502. This signal is approximately equivalent to the 1.5 MHz clock; however it is offset by a few nanoseconds on each edge. Exact timings are discussed in Section 2.8. If the 6502 wishes to read or write data, it must do so when Φ2 is high. This is to ensure that the Vector Generator has the proper amount of time to use the address and data busses to carry out its instructions. The 6502 may read or write data from the Vector Generator RAM. While the Vector Generator is carrying out an instruction, the 6502 can update the RAM with new instructions during the Φ2 high cycle.

The address bus is accessed via the VMEM signal. This signal selects whether the next address is from the 6502 or from the Vector Generator's program counter. If low, the next address will come from the 6502. By using this signal, R/W' and Φ2, the 6502 can properly access Vector Generator RAM and ROM to either read or write data. These signals also control the data bus, so that the 6502 will have access to data at the appropriate time.

2.7.7 Vector Generator ROM

Since most of the shapes drawn using the Vector Generator are very repetitive, these common shapes are predefined using Vector Generator ROM. In general, if a common shape is needed, an instruction in RAM will jump to the location in ROM which contains the correct data, execute these instructions, and then return to the main execution sequence in RAM. For example, if information for drawing a square was stored at location 400h in ROM, the data would appear as shown in Table 16.

Address	Data	Instruction	Function
400h	20h	SVEC	ΔX = 0 ΔY = +1
401h	41h		
402h	21h	SVEC	ΔX = +1 ΔY = 0
403h	40h		
404h	20h	SVEC	ΔX = 0 ΔY = -1
405h	5Fh		
406h	3Fh	SVEC	ΔX = -1 ΔY = 0
407h	40h		
408h	00h	RTS	Fetch next address
409h	C0h		

Table 16: Vector Generator ROM

As shown in Table 16, each word performs a different change in the vector beam, which creates a square. Scaling may be applied to use the same subroutine but instead creates a smaller or larger version of the same image. This makes for more efficient calculations of the vectors necessary to draw a particular image.

2.8 Clock Generation and Control

The Atari© *Tempest* hardware employs some clever timing techniques in order to save parts and maximize the clock signals established by the 6502 processor. There are a number of clock signals used within the hardware. The primary crystal oscillator has a 12 MHz frequency. All other signals are derived from this master clock.

The 6502 can operate on clock speeds up to 2 MHz. Since the other circuitry within the game hardware uses a 1.5 MHz frequency, this is the input frequency for the 6502. In order to generate this frequency with only one oscillator, Atari© used a simple 4-bit counter. The “Up” line is tied to the 12 MHz master clock, which allows the counter to increase on each positive edge. Based on these positive edge counts, the corresponding outputs are divisions of the input clock. In order from LSB to MSB, the 6 MHz, 3 MHz, 1.5 MHz, and 0.75 MHz clock signals are created. These signals may now be tied elsewhere in the hardware to implement the necessary timing control. All the clocks are perfectly synchronized with the 12 MHz clock, which makes for accurate control of all circuitry.

The 0.75 MHz clock and two additional counters are used to divide the signals even further. Through appropriate use of this clock signal and the counter output lines, a 24 kHz and 3 kHz clock are generated to be used within the game hardware.

The 1.5 MHz clock is tied to the 6502 input clock line. The processor generates two phases of this original signal, Φ1 and Φ2. The general timing diagram for these signals is shown in Figure 16.

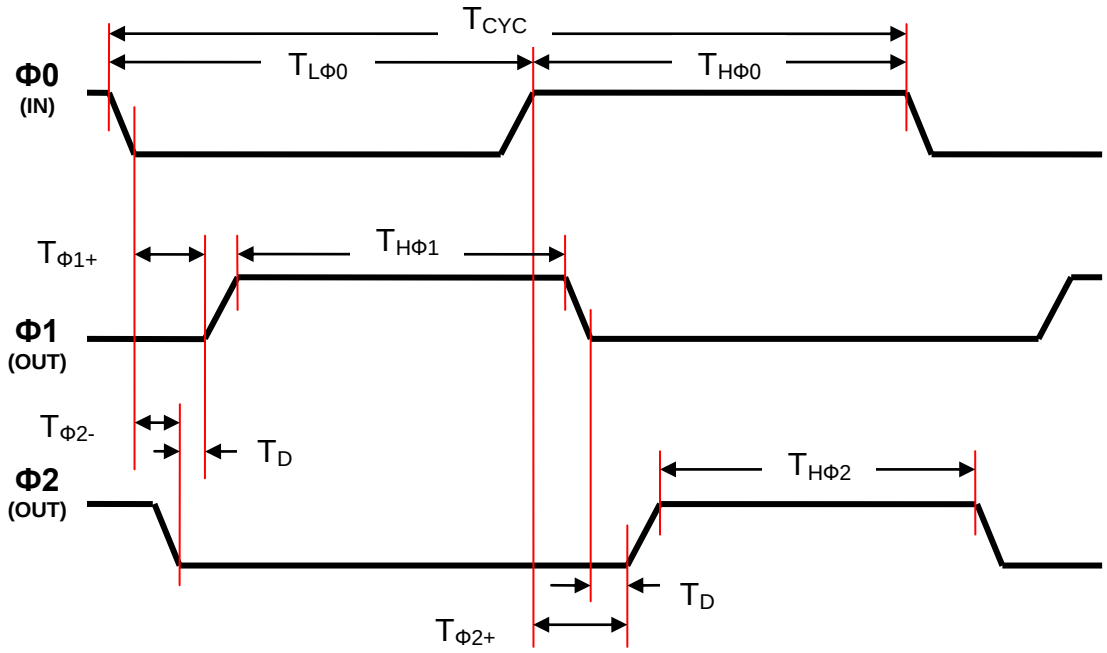


Figure 16: Processor Timing Diagram

Φ1 and Φ2 are non-overlapping clocks which control the timing of both operations within the 6502 and for external components. When Φ2 is low, the 6502 applies an address to the bus. When it is high, the processor completes the data transaction (read or write). Therefore, when Φ1 is high, logical operations which require external access to RAM or ROM are processed.

Using two phases allows peripherals, such as the Vector Generator, to be accessed by the 6502 at specific times. The processor shares the address bus with the Vector Generator. RAM is only modified when Φ2 is high. In general, the timing for the VG is active-low using the 1.5 MHz clock. As shown in the 6502 timing diagram, the 1.5 MHz clock will be the complimentary phase of the Φ2 clock. This allows the 6502 to communicate with the VG memory during the high Φ2 phase and the VG to execute its logic during Φ2's complimentary phase.

2.9 Software (*Synopsys*)

Our primary means of testing the design was the simulation software *Synopsys*. *Synopsys* allowed us to test each individual module that we instantiate, the system as a whole, our implementation onto an FPGA and eventually a printed circuit board. *Synopsys* possesses a number of tools which will help verify our design. One testing method is the built-in waveform tracking. Through this

tracking, we were able to monitor many different outputs and intermediary registers. For example, we were able to verify the addresses which are being accessed for the various ROMs. Also, we were able to track what values are being read from the ROM.

As an extra measure, the original game contained a testing mode. Testing mode allows the user to check to make sure the ROMs are still functional and the game is still operating properly by reading intermediate values of predefined testing wires. Testing mode runs through a battery of simple tests, such as displaying the alphabet or a simple grid.

If the testing output displays properly, we need to perform more rigorous tests. This is where *Synopsys* will help. Throughout the schematics, various testing points are labeled. For every test scenario, the values for the testing points are given. With *Synopsys*, we were able to check the values of these test points though the waveform tracking.

Another useful feature of *Synopsys* is the program's ability to translate Verilog code into a visual schematic interpretation. Since we converted schematics from picture to a description language, the ability to double check our work was very helpful. We utilized this feature during the initial implementation phase when we created each individual part, such as a 4-bit counter or a D Flip-Flop, and when we integrated the entire system together. We checked to make sure that each node was wired correctly and that each node behaves the way we wanted it to behave. Additionally, *Synopsys* recognizes module layering. This proved helpful since our project was very modular in nature. For example, the 6502 microprocessor contains an array of Flip-Flops, an ALU and various other components. Once all of the 6502's individual components are functioning separately, they can be grouped into a single header file. *Synopsys* then views this group as a single, functioning unit.

While there are many other simulation programs available, we felt that *Synopsys* was the best software to use. We considered Xilinx ISE 9.2i, which would be available to students. However, our personal experience with the software in various lab sessions from previous coursework had shown that it was prone to bugs. With such an important project on the line, we felt that this was not a risk worth taking. Additionally, *Synopsys* supports FPGA integration. This was obviously an important feature since we implemented our project onto an FPGA. Once our design was verified, *Synopsys* supported a printed circuit board mapping tool. This aided in creating our PCB design and checking for any undesirable side-effects which were not accounted for when simply coding the project.

While *Synopsys* was the best choice, it does have some drawbacks. One hurdle was learning the Linux operating system. Since this tool is used only on Linux, we had to learn how to obtain a user license and start the program in a Linux

command line. This hurdle was quickly overcome thanks to help from our project advisor, Mr. Harper. Another drawback to *Synopsys* is that the program does not have a built in Verilog compiler. This means that any edits that we needed to make to our source files will require us to edit them in a simple text editor and then re-compile the entire project before testing our changes. This also meant that any typographical error or syntax error may not be detected until after the project had been recompiled and tested. Despite these inconveniences, we felt that *Synopsys* was the best tool to fully test our project.

2.10 FPGA Board Options

Choosing the correct FPGA device to implement our design was a crucial part of our project. It was important to find a solution that was not only capable of handling our project, but also low-in-cost and included the correct I/O interface. Many boards currently in the market support numerous features including VGA, HDMI, Serial ports, Ethernet, and I/O devices including switches, buttons, and 7-segment LEDs. Our project had very little need for most of these features, making it unnecessary to purchase a prefabricated board for general applications. Customizing our board with only the necessary I/O interfaces allowed us to maximize the interactions with our FPGA device. The complexity of FPGA implementation onto a custom PCB needed to be considered. Since the FPGA needed to be programmed, some controller and I/O interfacing was required. Operating said controller was not really an element of the project we wished to oversee ourselves, so finding a solution that would handle the programming of our device without an in-depth design was crucial. This was a major factor in our choices, since we lacked experience with PCB design incorporating a FPGA.

There are several major FPGA providers such Xilinx and Altera. Most of our previous experience with FPGA programming was using Xilinx products. Our coursework in Digital Systems, Computer Architecture, and FPGA Design all utilized FPGA boards with Xilinx chips. More importantly was our software training for programming these boards. Xilinx's ISE Design software is an all-in-one package which allows users to write and debug code, simulate their design, and generate a bit-stream to program their respective devices. It includes a vast selection of libraries which accommodate all versions of their chips. It also has many key capabilities such as memory module generators used to instantiate RAM or ROM to be interfaced by an HDL file. These and other powerful features of the Xilinx ISE software were all necessary tools for our design and prototyping. Additionally, the ISE package was the only CAD software we were familiar with for programming FPGA devices. Due to this fact, our FPGA provider was Xilinx.

After some research into the different types of FPGA devices offered by Xilinx, we found that most low-cost solutions use Spartan 3 chips. The Spartan Family FPGAs are optimized for the lowest-cost logic, processing, and memory. However, none of the features of a higher-scale FPGA are sacrificed. Spartan 3

chips have 50,000 to 2 million logic gates. Some other features include 6-input Look-Up Tables (LUTs), 18KB Block RAM, Digital Clock Managers (DCMs), differential I/O pins, and dedicated multipliers for more efficient DSP solutions. Most of the FPGA boards we researched either have a Spartan 3E or 3A chip. These boards have prices range of approximately 50USD to 150USD, well within our group budget. Since our project was relatively small in scale; the number of logic gates on a Spartan 3 product was sufficient. The largest memory device used in the Atari© *Tempest* hardware is the 20 KB Game Rom. Therefore, the Block RAM on the Spartan 3 devices was more than satisfactory for our memory needs. All of the above specifications met the requirements for our project. Therefore, we chose to work with any of the devices in the Spartan 3 family. We chose a device with 200 K or more gates, based on the memory blocks needed.

2.10.1 Basys2 Digilent Board

Digilent is an FPGA board provider which distributes affordable and useful boards using Xilinx chips. They have several boards which use the Spartan 3 chipset. The Basys2 is one such board. It has 100 K or 250 K gate versions, the latter of which supports the desired range for our project. It is powered via a USB port or through battery terminals and runs using 5 V DC. The FPGA on the board is a Xilinx Spartan 3E which is interfaced using an Atmel AT90USB2 controller. It has platform Flash memory to be configured by the user's bit-stream. This allows the board to be disconnected from the PC and still run the code loaded into the Flash memory when powered. The board has a 50 MHz clock which can be switched to either 25 MHz or 100 MHz if desired. It also has a secondary clock port to incorporate a user-desired clock speed. Such capability will be necessary for the 12 MHz clock in order to achieve perfect division of this master frequency. The price of this board is 50USD with a student discount.

Some of its features include I/O devices: push-buttons, switches, LEDs and four 7-segment displays. These devices are extremely useful during coding, debugging and testing, as actual FPGA input can be controlled by switches and buttons, and the output displayed using either the eight LEDs or the 7-segment displays. Other features on this board include PS/2, VGA, and Pmod connectors. While the PS/2 and VGA are unnecessary features for our project, the Pmod connectors are extremely valuable. The board includes four connectors, each of which have four I/O lines that can be used either as digital or analog I/O lines. The Vector Generator outputs approximately 18 data lines to the analog circuitry for digital-to-analog conversion. This does not include the input lines coming from the game controller and coin slot inputs. Therefore, there were not enough I/O lines on this board to accommodate the needs of this project. The nice thing about this board was that it is already owned by one of the group members and was utilized in testing and debugging of code. The Pmod connectors were a good means to test small modules of our code to ensure they are functioning properly. However, this board wasn't used in our final design due to its I/O limitations.

2.10.2 Nexys2 Digilent Board

This board is the next product up from the Basys2 distributed by Digilent. Again, it includes a Xilinx Spartan 3E chip and an Atmel AT90USB2 controller to handle bit-stream transmission. There are two versions: a 500 K and 1200 K gate variation. It is powered via a USB port, power connector, or through battery terminals and runs using 5 V DC. Again, Flash memory can be configured with a bit-stream. In addition to this configurable Flash, it also has 16 MB of Flash ROM and 16 MB of SDRAM. This on-board memory would be extremely useful for our design, particularly for loading game ROM. By using on-board memory, other vector-graphics based games might also be loaded and run using our FPGA implementation. The on-board SDRAM would allow all the game RAM to be accessed in one location. Again, it has a 50 MHz clock built-in with capability for an optional secondary clock. This board costs approximately 100USD with a student discount.

The Nexys2 board includes all of the same I/O features mentioned for the Basys2, which again are LEDs, switches, push-buttons, and four 7-segment displays. It also includes four Pmod, VGA, and PS/2 connectors. The Pmod connectors on this board are 2x6, meaning there are 8 signals allowed per connector. In addition to these features, the Nexys2 adds an RS232 port and a high-speed Hirose FX2 connector with 43 signals available. While the serial port is not needed, the Hirose connector is a very attractive feature of this board. This connector, combined with the Pmod connectors, will allow for a total of 75 I/O signals. This is obviously the most important specification to meet, given that our project cannot work without enough I/O connectors.

2.10.3 Papilio One Board

The Papilio One board is a customizable FPGA board manufactured by Gadget Factory. It can accommodate the Xilinx Spartan 3E 100 K, 250 K, and 500 K chips. It is powered via a power connector or USB and includes independent power connections distributing 5V, 3.3V, 2.5V, and 1.2V. Xilinx JTAG cables are supported by this board for easy programming. The board includes a 32 MHz oscillator that can be modified using Xilinx's DCMs on the Spartan 3E to generate our desired clock speed of 12 MHz. 4 Mb of Flash memory is available to configure a bit-stream to the device. The price of this board ranges from 50USD to 75USD depending on the FPGA logic gate quantity.

The great feature of this board is its I/O wings. As shown in the figure below, the board can support up to 48 bidirectional I/O signals via its routing to the edge wings. This allows the user to customize the I/O features that interface with the FPGA. This type of solution is ideal for our project. We wanted to avoid unnecessary components (such as PS/2 and VGA included in the Digilent boards) so that we maintain only the bare necessities of the FPGA interface. This

board allows us to customize these features, while avoiding the complexity of configuring a controller to program the FPGA.

The other important feature to note about the Papilio One is that it is open source, meaning the schematic for the device and the Eagle PCB information is available to download. This would allow us to incorporate the layout and features of this board as well as customize some of the components we desire, such as a 12 MHz oscillator. Doing this would eliminate the need for exterior PCB hardware, instead allowing for all components to be streamlined onto one system. If, however, we had decide that using the Eagle layout for this board was too complex, there were also layouts available for customized wings to attach to the Papilio One board. These wings would include our input controller PCB as well as out output DAC and op-amp analog circuitry to interface with the laser. Either option is very attractive. Figure 17 shows many of the features on the Papilio One Board as previously mentioned.

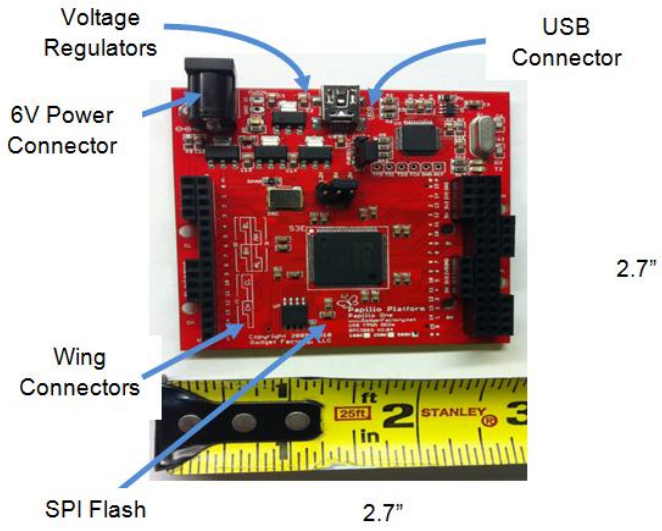


Figure 17: Papilio One Board and Features

At the time, this was our top choice for testing both our design and implementation onto a PCB.

2.10.4 XuLA Board

The XuLA FPGA board is another customizable device, manufactured by X Engineering Software Systems (XESS). It has a Xilinx Spartan 3A FPGA with a PIC microprocessor for programming the device. Memory includes an 8 MB SDRAM module and a 2 Mbit Flash for bit-stream configuration. 40 I/O pins are available and are routed to the edge of the board for external interface. It is powered via USB or power pins tied to their respective regulators, accepting either 5V or 3.3V. A 12 MHz oscillator is on board, making it a perfect fit for our frequency requirements. If needed, this clock signal can be multiplied or divided

using Xilinx DCMs. Standard JTAG cables are compatible with this device. The price of this board is $70.

Similar to the Papilio One board, the XuLA allows for customized I/O connections. As can be seen in the figure, the I/O pins are routed to the edge of the board to allow for easy prototyping on a breadboard or connection to a PCB. This feature will allow us to easily test our design without any of the excess components that come with other boards.

The XESS is an open source design. All schematics and PCB layouts of this board are available free of charge. This option would give us the freedom to incorporate the XESS design directly into our own customized PCB if desired. This eliminates the need for more than one board.

2.11 Input – ADC

The input is one of the more important aspects of this design project. Without input, the game is not playable. The input has to work well, because if the input does not work well, the game is compromised. The spinner wheel is the most important part of the controller for the Tempest design project. The spinner wheel turns 360 degrees clockwise and counterclockwise and allows the player to steer the ship around the border of the level. The project requires a reproduced version or accurate representation of the original Atari Tempest spinner wheel. The input from the user is the first and only external data that enters the system. The analog input from the spinner controller comes into the digital to analog converter by a pair of sine waves. These two sinusoidal waves or square waves are 90°s out of phase. The POKEY chip turns the sine wave into a square wave then selects the spinner wheel in operation if there are two spinners. The POKEY chip also converts the two (now square waves) into a clock line and a direction line. When the knob is turned in the other direction, the direction line changes sign and the player ship will move in the now correct direction. This data enters the system as analog input by way of four different buttons and a previously mentioned spinner wheel. The spinner works by using an encoder wheel to detect the velocity of the rotation of the wheel. The data received from the four buttons and the encoder wheel comes into the system as analog input. In order for the data to be used by the *Tempest* game system, the data must be converted into something the system can use; therefore, the data must be converted into a digital signal before it can be processed into the system. In order for this data to be converted, the input must be passed through a multiplexer on separate control lines. In the original Atari© system, after the data is passed through the multiplexer, the information is then passed into a POKEY chip. The data passed into the chip includes the start button, the fire button, the zap button, the encoded data from the spinner wheel, and the information from the external data bus. The fourth input, the coin in button does not pass through the POKEY chip, but is handled in the FPGA. The POKEY chip outputs some audio and the

digital signal used by the actual game system. A block diagram of the proposed design is shown in Figure 180

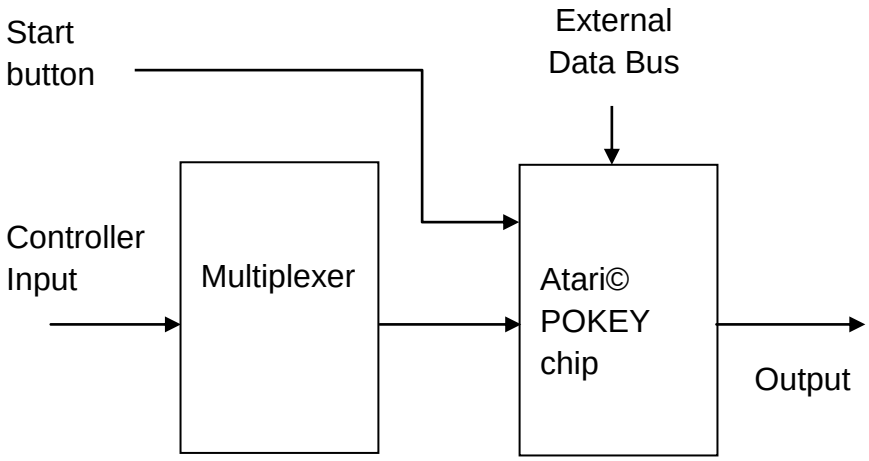


Figure 18: ADC Illustration

2.12 Input – Memory (ROM and RAM)

The *Tempest* system uses one large block of ROM that is divided up into a multitude of ways, as was evident in the Memory Map in Table 4. The following sections describe how the divisions of the main ROM are set up and the size of each block.

2.12.1 ROM

ROM is Read-Only Memory and is also referred to as a type of firmware. ROM chips are programmed with specific data when they are manufactured and can be found in most electronics, rather than only in computers as perceived by many. The Atari© *Tempest* Memory Map shows us that the ROM for the game is broken up into blocks that pertain to certain components of the game's circuitry. To begin with, the top 32K of the Memory Map belongs to ROM, however not all of the allocated space is used.[20] Of the 32K, the actual program resides in a block that is mapped from 9000h–DFFFh. A second image of the top 8K is made and stored at E000h-FFFFh for the purpose of the 6502 processor requiring reset vectors to be located at FFFAh–FFFFh.

Other sections of the main ROM that were split up include the following mappings: The Vector ROM is stored at 3000h–3FFFh. The two POKEY chips have their data stored in the ROM at the following locations: POKEY #1 at 60C0h–60CFh and POKEY #2 at 60D0h–60DFh. Of the other memory locations in the memory map, specific memory locations are not used as blocks of ROM, but rather are used to store control bits like the Vector State Machine RESET stored at memory location 5800h.

2.12.2 ROM – High Score Memory

The High Score Memory is another section of the ROM. The High Score Memory takes the address as decoded from the Auxiliary Board Address Decoder and then is passed to a 64 x8 bit ROM (ER2055 Chip) that determines whether to set a new high score or display previous high scores, based on the instruction stored at the desired memory location. Following this function, the data is rewritten to the External Data Bus that is then forwarded back to the 6502 microprocessor.

2.12.3 EEPROM

In the Atari© *Tempest* circuit boards, there is a single 64x8 EEPROM chip. Any Write instructions first pass through the write area, which is 6000h-603Fh in the memory map. This address in the main ROM is then latched to the address pins of the EEPROM. This means that a write instruction that does nothing needs to be completed before a specified location in the EEPROM may be read. A read instruction to the EEPROM latches the data pins onto the bus.

2.12.4 RAM

Memory from 0000h–00FFh is treated as the “zero page” by the 6502 and is used for addressing modes. 0100h–01FFh is used for the CPU stack, since the stack pointer is only one byte wide. This 2K mapping from 0000h–01FFh is placed at the bottom of the memory map and is used as RAM that is private to the CPU. The Vector RAM is stored at 2000h–2FFFh. The Color RAM for the outputs are stored near the bottom of the memory map at 0800h–080Fh. Due to the limitations of our laser not having options for color output, the Color RAM was not be utilized in this project.

Most FPGAs on the market today contain SRAM that can be setup as synchronous or asynchronous. This SRAM can also be programmed so that the processor treats a certain block of the SRAM is treated as ROM through programs such as CORE Generator from Xilinx.

2.12.5 Synchronous vs. Asynchronous

Synchronous memory refers to memory that has its communications synchronized with the clock that runs the processor. Having the memory synchronized with the processor clock creates simpler timing requirements, which allow the memory to operate at higher frequencies. In addition the higher memory bandwidths, synchronous memory is not prone to errors since the signals into and out of the memory are performed on either the positive or negative clock edges, depending on its configuration. In effect, this eliminates a write-enable circuit

Asynchronous memory does not have its communications synchronized with the processor's clock. It also requires a write-enable circuit to be created that produces a pulse signal whenever a write instruction is performed. With this type of memory, the write address has to be considered at setup and hold time, as well as considering the data at the same times on the rising and falling edge of the write-enable pulses.

2.13 Input – Game Controller

Since *Tempest* is driven by a user I/O, input is one of the more important aspects of this design project. If input does not work, the game isn't playable. Along with that, the input must work exceptionally well, otherwise the game is compromised. With that said, the design of the input system should be designed with no monetary limitation. The most important part of the controller for the *Tempest* design project is the spinner wheel. The spinner wheel is the part that allows the player to steer the ship around the border of the level. The spinner wheel must be the most accurate piece of the controller input; as such the project requires a reproduced version of the original Atari© *Tempest* spinner wheel. A product that is easily integrated, relatively inexpensive and accurate will be the best answer for the spinner wheel needs. In addition to the spinner wheel controller, the project required four buttons the user can press. The buttons will be used for inserting a coin, user start, fire and zap.

There are numerous options available for making or purchasing an arcade spinner wheel. One such option is through GroovyGameGear.com. They make a product called the TurboTwist 2™ arcade spinner control for 69.95USD.[22] This spinner is portrayed in Figure 19. An interesting feature of this spinner controller is the ability to purchase different style knobs for the controller. There are thirteen different colors and styles to choose from when creating an individualized arcade machine. This particular spinner wheel would possibly be one of the most accurate devices that could be used for this project. Another perk of this product is its small footprint in the arcade machine control panel; the spinner can fit into a slot that normal push buttons will fit into. Additional spinner controllers can be purchased for a lower price to allow for multiple players. The TurboTwist 2™ spinner controller is a professional product that was a top candidate for this design project. Price must be balanced with means, and in this case the cost was too great for our budget.

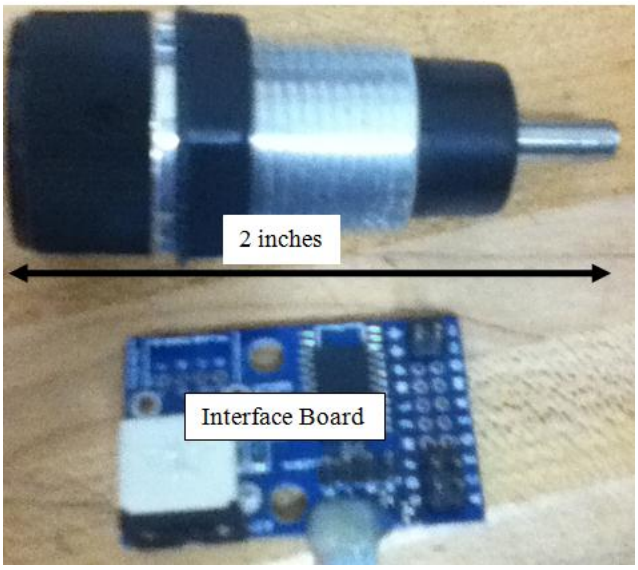


Figure 19: TurboTwist 2™ Arcade Spinner

The next product under consideration was the SpinTrak precision rotary control. The price of the SpinTrak itself is 69.00USD, but the knob is 12USD and the flyweight for counter balancing is 10USD making the total price 91USD.[23] The price evaluation between the two options makes the TurboTwist a more attractive option. However, the SpinTrak boasts to be the first spinner with USB 2.0 support. The USB 2.0 support is a useful feature, but was not necessary for this project since *Tempest* runs at slower speeds, and the PS/2 interface will most likely be chosen. More features include the Tornado-style spinner tops which are advertised as the best style of knobs, at a 10.00USD price tag. The SpinTrak features two modes of operation: a very accurate controller at higher rates of spinning or a slow spinning wheel for games like *Araknoid*. There are two spinner flywheels that can be connected to end of the shaft for longer spinning times. The large flywheel is capable of spinning for 20 seconds. The SpinTrak also features Opti-PAC or Mini-Pac interfaces as well as the USB support. The UTILIMARC Opti-PAC interface is a four pin connector available for easy interfacing.

A third option is also available. Instead of purchasing a retail spinner knob; a similar type of knob could be designed independently. There are various "build-your-own-controller" guides available: one such guide is "The Cheep Spinner." This spinner build requires that an old ball mouse be taken apart, and the encoder wheel be salvaged and used in the final spinner. Additionally, this project requires various hardware parts and tools from the hardware store. The total part cost of the spinner wheel project is 38.47USD, which is the cost of the hardware and parts minus the tools. Based on the three choices for the spinner controller, the TurboTwist 2™ seemed like the best choice. It is superior to the SpinTrak in the price category while all the components come standard with the TurboTwist 2™. While the custom controller would save money, the time and energy spent coming up with some of the older components, disassembling them

and acquiring the necessary tools is probably wasn't worth the effort and that design component was not in the same scope as the rest of the project.

The other piece of the controller for the *Tempest* arcade machine is the buttons. This controller design features four buttons, one for fire, zap, coin in and player start. A button is constructed by implementing a micro-switch. In the case of SparkFun's concave arcade-style button, the button uses a three terminal micro-switch. In three terminal micro-switches, there are three terminals. The first terminal is the NO terminal, or normally open, which is the terminal that when open, means that the button is normally not being pressed down. When the button is pressed, the switch becomes closed and the electrical signal is sent from the device. The second terminal is the NC terminal, or normally closed, which in its normal state it is closed. As a result when the push button is pressed, the NC terminal becomes open and will send an electrical signal from the device. The third terminal is a common contact. A three terminal micro-switch that is connected to an arcade style button is shown in Figure 20. The SparkFun's black concave button was a perfect fit for our game controller needs. The button was inexpensive costing only 1.95USD and is made from nylon, a durable material suitable for mashing. Additionally, the micro-switch is rated at a reliability of up to 10 million clicks. The micro-switch can run three amps at 120 volts AC. The product is lightweight and was advertised to have a "clicky" feel which is certainly a plus for arcade enthusiasts.

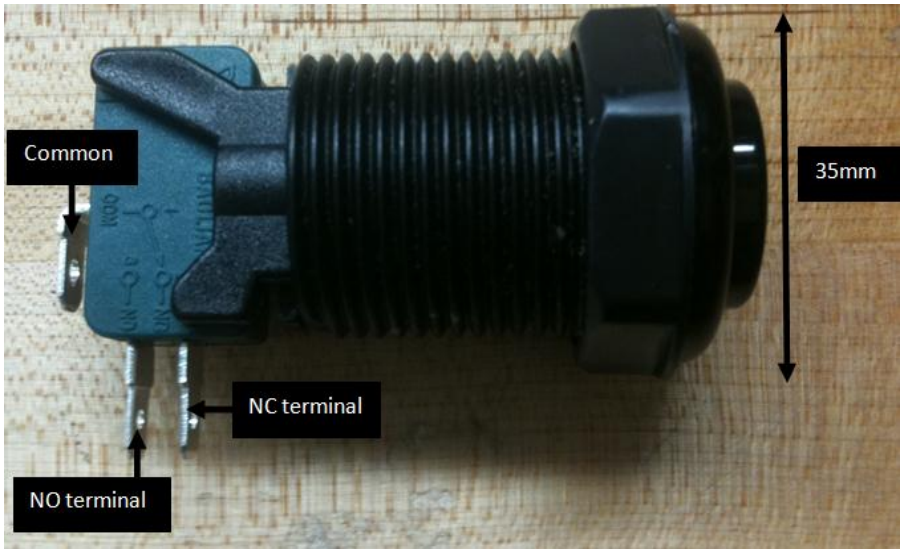


Figure 20: Concave Button

The controller for the Atari© *Tempest* machine was then comprised of both the TurboTwist arcade spinner control and the four black, concave buttons.

2.14 Output - Lasers

Laser galvanometer scanners, or galvanometer-scanners for short, are devices used in laser light shows to display lasers on walls or in the air for an audience's entertainment. The galvanometer itself has metal pin attached to a coil, and when electricity is run through the coil, the magnetic field deflects the pin depending on the amount of current applied. This meter movement alone is not enough to create an image as shown in laser light shows. In order to make complicated drawings on walls or in the air, a mirror was attached to the device to allow the smaller movements to be picked up. Galvanometer-scanners are rotational electric motors that can turn only so far based on the amount of current applied.

According to LaserFX's article on the laser show "Scanning System," laser scanners work by using, "two permanent magnets (to) create a strong flux in the gaps of the central pole pieces of the armature. The rotor moves – moving the shaft with attached mirror – in response to variations in this magnetic flux caused by current applied to the drive coils."[25]

There are two different types of galvanometer scanners. One is the open loop scanner and the other is the closed loop scanner. In the open loop scanner, the scanner is deflected by the current as usual and the beam will draw. However, the scanner is open loop so there is no way for the system to know where the shaft is currently drawing. As a result, the open loop scanner is not the suitable tool for laser light shows, due to its lack of precision. In a closed loop scanner, the shaft has a position detector that knows where the shaft is located during the entire scanning processes. A system designed this way allows precise, accurate control of the mirror that is ideal for projecting laser light shows. To draw both the X and the Y coordinates two scanners must be used for the laser system. The scanners are placed perpendicular to each other in order to display both coordinate systems. Modern galvanometer scanners use a process called "blanking", in which the laser turns off briefly in order to prevent the connection of letters. Another important issue with laser galvanometer-scanners is the speed (measured in kilo-packets per second) at which they can draw to a wall or other surface.

The scanning system, SM204, manufactured by Sonima will most likely be chosen for our design project. This scanner is used in professional laser light shows. The galvanometer-scanner operates with an input voltage of plus or minus 5 volts and can control gain, size and position via the driver. The laser operates at 20 kilo-packets per second and runs off a 100 volt AC power supply. The interface is supplied by a +/- 15 volt DC signal. This voltage is supplied by the JMD20 which is also manufacture by Sonima. Data must enter the galvanometer-scanner interface with a plus and minus X and Y coordinates as AC voltage. The voltage signal is sent through the galvanometer-controller. This particular scanning system does not include a laser; instead an external, high

powered laser provided by Mr. Harper is used in the design. A picture of the SM2804 is shown in Figure 21. The power pins on the interface board require the +/- 15 volt external DC signal, this input power will be provided by the FPGA board. The input signals come from the output from the FPGA and after they are converted to an analog signal they can be sent to the input signal of the laser interface. The VR1 through VR7 pins are shown on the diagram, these pins represent potentiometers. VR1 controls the size of the vectors to be drawn, VR2 controls the servo gain of the laser galvanometer-scanner, VR3 and VR4 control the low and high frequency damping respectively. VR5 controls the linearity of the drawing. VR6 controls the input scale of the vectors draw, and finally the VR7 controls the position of the input scaling.[26]

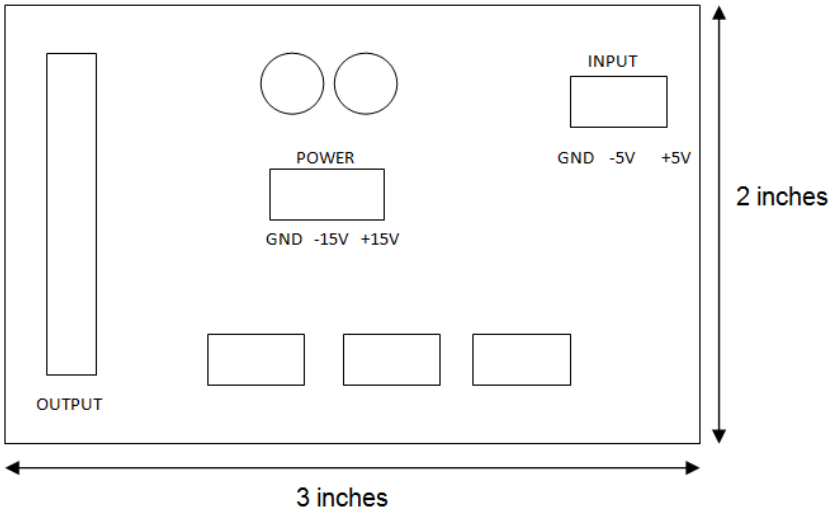


Figure 21: CW20 Block Diagram

Once the proper connections are made, and input voltages are to the laser controller the output goes from the controller to the X and Y scanner of the laser and is ready to draw to a surface. The laser (not shown) in Figure 25 also requires a voltage from its 100 volt AC power supply. The laser is powered through the supply and now takes the input from both the X and Y coordinate values from the controller.

There are a number of other closed loop laser galvanometer scanners that are available. All these products have faster laser speeds at 20 kilo-packets per second and more impressive features, however none of them fall within the projects budget range.

2.15 Output - DAC

The internal data coming from the 6502 processor, Vector Generator, and Math Box is digital. This digital data must be converted to analog data by a digital-to-analog converter (DAC). The DA conversion was done in order to process data

from the Vector Generator. In this case, the project will be using a laser galvanometer as the output. The galvanometer needs analog data to properly display the lines to be drawn to the screen. In order to convert the digital data to analog data, a DAC was used. There were three additional blocked components needed to properly display the desired output.

The actual part of the circuit that determines the charge time is the integrator circuit. The integrator circuit is used to correct for the natural charge-up of the capacitor. Integrators allow for a linear charge-up as opposed to the exponential curve that occurs over a regular circuit with only a capacitor. The input into this integrator comes from the DAC which outputs a current. That current is added from the voltage scaling block. A block diagram for the system described above is shown in the Figure 22.

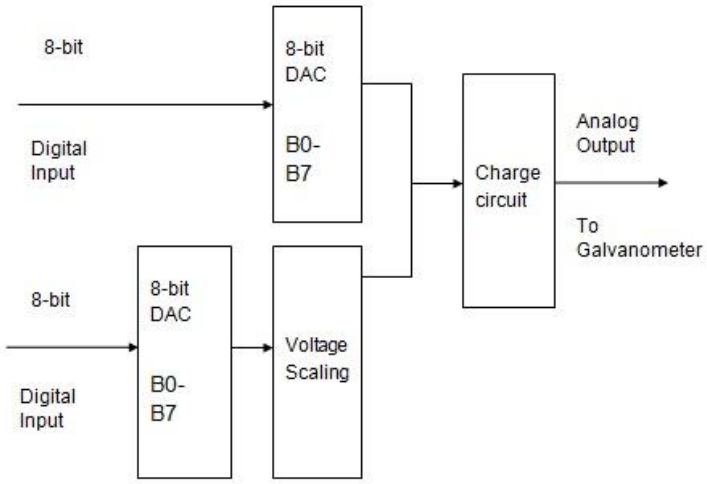


Figure 22: DAC Illustration

The charging circuit is the most important component of video output design. The digital data sent out from the FPGA is sent through two multiplying digital-to-analog converters. The first DAC handles the vector scaling portion and the second DAC handles the physical XY positioning of the vector to be drawn. The first DAC handles the vector scaling and sends two voltage references that affect the length of the vectors to be drawn. As shown in Figure 23, the two currents sent into the second DAC are I_{REF} and bi-polar output (BIP). Since DACs typically output currents, resistors R1 and R2 had to be added to the circuit. The I_{REF} current is governed by the resistance R1. The BIP current is controlled by the resistor labeled R2 which allows the vector to be drawn to the negative side or the positive side of the screen. When there is no vector to be drawn, the V_{REF} signal is switched off and there will be no signal to draw. As long as there is a V_{REF} signal, the second DAC will know to draw a vector and, depending on the I_{REF} and bipolar output current, how long the vector should be. The output current of the second DAC determines the slope of the vector that is to be drawn to the screen. One of these blocked components will be a voltage scaling circuit. The circuit can take the higher DC voltage data from the Vector Generator and

scale that voltage down so it can be used by the display. Additionally this block serves as a way to scale the size of the vector that is to be drawn to the screen and whether this is a vector that needs to be drawn on the screen. The second block component will be a charging circuit, this charging circuit will serve to slow the analog output of the circuit so the vectors will be drawn slower so the user will actually be able to see the output. The charging circuit uses an integrator in order to have the charge circuit draw the vector to as a straight light as opposed to the curved line that a simple circuit with a capacitor would. The integrator acts a perfect capacitor to output the straight line. Since capacitors store charge of all the old values used in the circuit, a switch is used on the integrator to completely discharge the voltage values on the capacitor. Without a discharge, vectors would drift into incorrect positions on the screen. This discharge will happen periodically to ensure proper drawing of vector lines. A special consideration had to be made with this capacitor. Capacitors are made with different types of materials, such that the proper one had to be selected for this application. Due to the fact that *Tempest* draws vectors at a relatively quick rate, a capacitor that can charge and discharge quickly was necessary for this project design.[13]

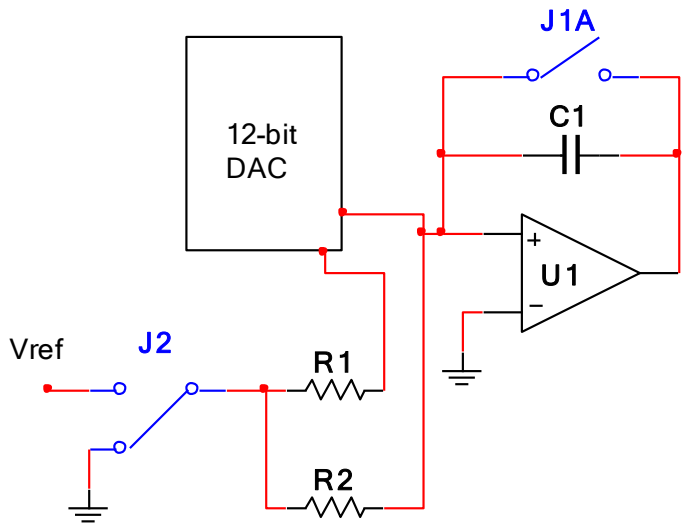


Figure 23: Charging Circuit

2.16 Output - Audio

In the original design of the *Tempest* Arcade video game, the audio for the game was produced by a digital I/O chip made by Atari© known as a Pot Keyboard Integrated Circuit otherwise referred to as POKEY. The POKEY chip was most commonly found in the home gaming consoles and arcade games. The chip is capable of using data from game spinner wheels found in arcade machines as well as inputs from computer keyboards. The POKEY chip is also able to output game audio. In modern times, the audio of the POKEY chip was converted into software by way of emulation.

The POKEY chip uses 16 registers; the read registers are shown in Table 17.

Register	Name	Description
D200h	AUDF1	Channel 1 Frequency
D201h	AUDC1	Channel 1 Generator
D202h	AUDF2	Channel 2 Frequency
D203h	AUDC2	Channel 2 Generator
D204h	AUDF3	Channel 3 Frequency
D205h	AUDC3	Channel 3 Generator
D206h	AUDF4	Channel 4 Frequency
D207h	AUDC4	Channel 4 Generator
D208h	AUDCTL	Control over audio channels
D209h	STIMER	Timer start
D20Ah	SKRES	Serial port status reset
D20Bh	POTGO	Start port scan sequence
D20Ch	N/A	Unused
D20Dh	SEROUT	Serial port output register
D20Eh	IRQEN	IRQ interrupts activation
D20Fh	SKCTL	Control over serial port

Table 17: POKEY Read Register Table

Although the POKEY can be used for audio, its usefulness is not limited therein. The POKEY chip is also used in Atari© systems in order to take input from various devices. In the arcade machines, the chip can take data in from simple buttons or switches, but can also take data from more complicated inputs such as spinner wheels found on arcade machine games such as *Tempest*. It can also take input from a keyboard of up to 64 keys. The input for such spinner wheels is measured by potentiometers. The POKEY chip handles these inputs through eight ports, which have 8-bit resolution. Additionally the chip has three timers that will trigger interrupts when the clocks cross zero as well as a random number generator. The POKEY chip also handles interrupts and has serial input and output ports.

The AUDC1 through AUDC4 are registers capable of being written to. These registers have control over the volume of the audio output via the bit 0-3, and they control the frequency divider to on or off (1 or 0) via bit 5. The bits 5 through 7 on this register shift another register that makes the sound or distortion produced by the POKEY chip. A 000b instruction on bit 5 through 7 indicates the 5-bit and 17-bit register to shift. The instruction 001b shifts the 5-bit register; the instruction 010b shifts the 5-bit and 4-bit register. A 100b instruction on bit 5 through 7 indicates the 17-bit register should be shifted. The 101b instruction indicates the register does not shift and therefore produced an undistorted tone. A 110b instruction shifts the 4-bit register. A 111b instruction does not shift the register and doubles the frequency.

The AUDIOCTL register is able to be written to, and controls all of the AUDIOF registers. Bit 0 controls the choice of frequency divider; if 0 the divider rate is 64 kHz. If 1, divider rate is chosen as 15 kHz. Bit 1 controls the high-pass filter for channel two. Bit 2 controls the high-pass filter for channel one. Bit 3, if set to 1,

controls the connection of dividers 4 and 3 in order to receive 16-bit accuracy. Bit 4, if set to 1, controls the connection of dividers 2 and 1 to receive 13-bit accuracy. Bit 5 sets channel 3's frequency to 1.77 MHz for PAL (Phase Alternating Line more commonly used in European countries) if the bit is set to 1, and if the bit is set to 0 thus the channel 1 frequency is set to 1.79 MHz for NTSC (National Television System Committee more commonly used in North and South America). Bit 6 sets channel 1 frequency to 1.77 MHZ for PAL if the bit is set to 1, and if the bit is set to 0 the channel 1 frequency is set to 1.79 MHz for NTSC. Bit 7 is responsible for switching the shift register to 17-bit if set to 0, and switching the shift register to 9-bit if set to 1.

The register IRQEN can be written to, and handles the interrupts requests for the POKEY chip. The BREAK instruction, if enabled, is a general interrupt for any input device to the chip. The K instruction, if enabled, is a keyboard interrupts for taking any user input from a keyboard. The SIR instruction is an interrupt that will pause the program from reading the serial input when enabled. The ODN instruction, if enabled, causes the program to interrupt, allowing output data to be sent. The XD instruction, if enabled, ends the serial transmission interrupt causing the program to continue running normally. The T1, T2, and T4 instructions, if enabled, cause the program to interrupt to use timer 1, timer two, or timer 4 respectively. When any of the timers are being used by the chip, the audio channels are all reset. The register IRQSTAT contains the interrupt status. The POKEY chip uses 16 registers; the write registers are shown in Table 18

Register	Name	Description
D200h	POT0	Potentiometer 0
D201h	POT1	Potentiometer 1
D202h	POT2	Potentiometer 2
D203h	POT3	Potentiometer 3
D204h	POT4	Potentiometer 4
D205h	POT5	Potentiometer 5
D206h	POT6	Potentiometer 6
D207h	POT7	Potentiometer 7
D208h	POTSTAT	Read 8 POT port lines
D209h	KBCODE	Last pressed key code
D20Ah	RANDOM	Random number gen
D20Bh	N/A	Unused
D20Ch	N/A	Unused
D20Dh	SERIN	Serial port input reg.
D20Eh	IRQSTAT	IRQ interrupts status
D20Fh	SKSTAT	Serial port status

Table 18: POKEY Write Register Table

The write registers $D200 to $D207 control the potentiometers POT0 through POT7 and measure the increasing input. A built in counter will count to value 228 and will be reset by a strobe of the POTGO command. Once the counter value gets to 228, the values saved in the latches are released and the potentiometers will change. When a spinner knob is turned clockwise, the pots decrease in value. The $D20D register handles the serial input data, this register is used as a

result of the serial data interrupt being received. The $D20A register handles the random number generator in the POKEY chip.[24]

In order to play the audio generated by the Atari© POKEY chip, the project must physically have a speaker. For the purpose of this design project, a loud speaker wasn't necessary, simply having a speaker that played the original game sounds of the *Tempest* Atari© game is proof of concept. As such, a small speaker found in any small electronic devices will suffice, for instance a two and a quarter inch 8Ω speaker.

2.17 Output - Video

Originally the idea of this design project was to be able to provide output video via the galvanometer laser as well as potentially to a CRT computer monitor. On a VGA display pixels are first drawn from left to right to create lines, and then there are lines that draw vertical to create a frame. The VGA driver works by splitting the horizontal rows into the horizontal synchronization (HSYNC) signal and the vertical columns into the vertical synchronization (VSYNC). In one second all of the horizontal columns in one vertical row are refreshed in one vertical refresh cycle. The issue with designing a FPGA was making the display timings match up perfectly. Additionally, in order to display the VGA output to the computer monitor, another circuit element had to be designed to assign RGB color to the screen. In Figure 24, the timing diagram for the HSYNC is shown, as the signal for the horizontal is high, the RBG input on the first signal is shown on the screen.

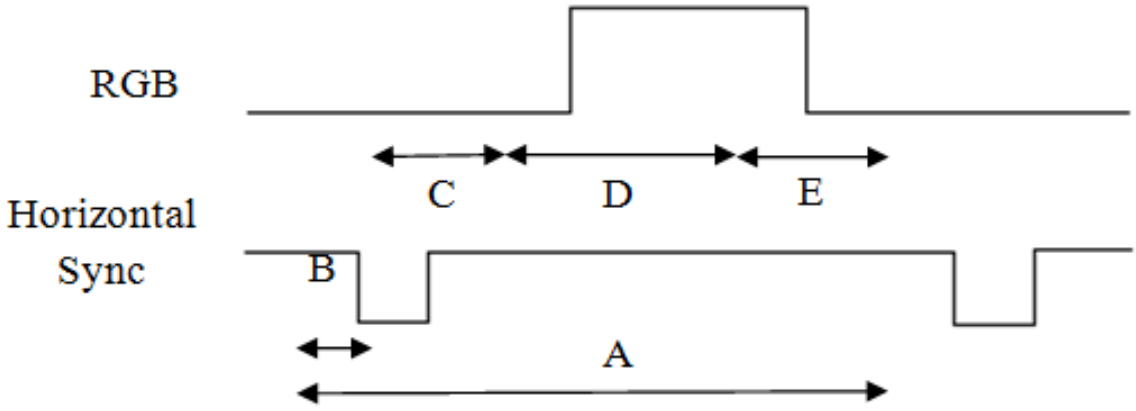


Figure 24: Timing Diagram for the HSYNC

The VSYNC signal is shown in Figure 25. When the VSYNC input is high, the RGB input from each of the horizontal lines are displayed to the screen creating a refresh rate of 60 frames per one second.

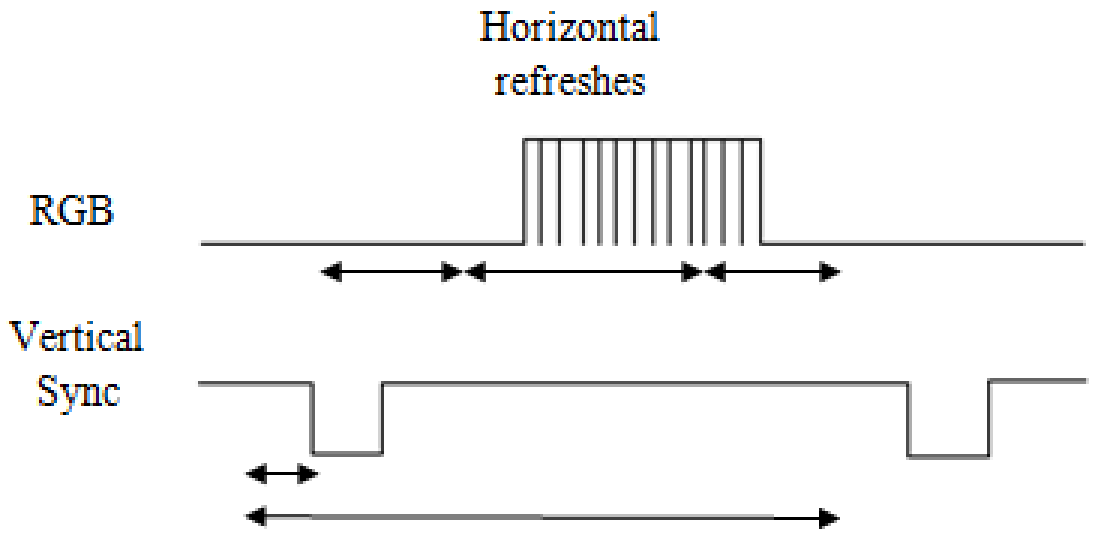


Figure 25: Timing Diagram for the VSYNC

In order to design a VGA driver for the system, a complex timing circuit had to be added in the software to take into account the timing for the vectors to be drawn diagonally across the screen, lighting up each pixel as the line is to be drawn via the output signal from the digital-to-analog circuitry. A VGA driver was originally considered; however, after some research into how VGA drivers work, the VGA drivers were determined to be out of the scope of this design project. Instead, the testing to ensure the proper operations of the digital-to-analog circuitry can be confirmed on a regular oscilloscope. Oscilloscopes operate similarly to how the laser galvanometers work in that they take input X and Y coordinates and show them on the screen.

Section 3: Project Design

3.1 Design Summary

The top-level design for our project is illustrated by Figure 26. The player sends inputs to the system through the user control, which contains a set of push buttons along with a spinner wheel. These inputs are passed through a small set of circuitry to convert the analog signals to digital signals for analysis. The analysis of the signals is done by passing the inputs read in through the POKEY chips to the 6502 microprocessor, via the data bus. Once the data has reached the 6502, it is then processed internally and forwards a master address to the Address Decoder. The Address Decoder then takes the data from the 6502 and activates the required components, such as the Math Box, Vector Generator, and System Memory (RAM and ROM). Once these components have performed their respective operations, the data is passed as output for the system. This output goes through another small set of circuitry that converts the digital signals back to analog signals before passing the data onto the laser for output to the screen.

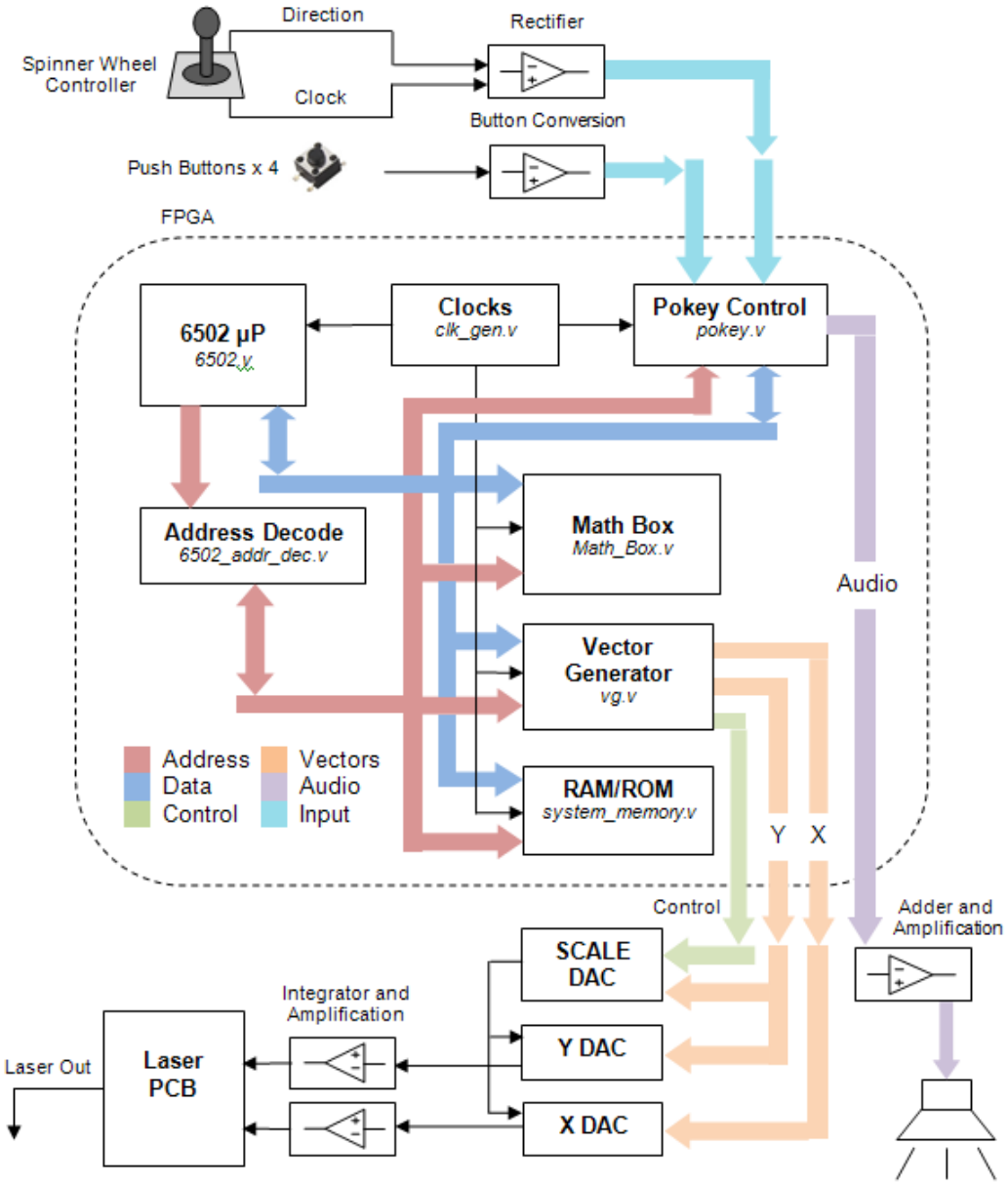


Figure 26: High-Level Design Block Diagram

The input of the controller was designed in reference to the original Atari© *Tempest* arcade machine. In the original machine, the controller input was designed by using a spinner wheel controller with two input signals. The input signals were created by having two sinusoidal waves that are 90°s out of phase with each other. The purpose for having these signals out of phase with each other is because it makes it easier to determine when the spinner wheel reverses direction. When the wheel reverses direction, there is some distortion between the waves, but the POKEY chip runs slower than the speed of the signal; therefore, the distortion between the waves is neglected as if it were noise coming into the system. This controller also sends to inputs to the system using

two signals, a clock and a direction. By using different pieces of circuitry, an approximation of the original signal could be maintained.

The final elements of the *Tempest* Arcade controller are the start, fire and zap buttons. Additionally, the coin in slot, diagnostic step, slam and self-test signals are required. Each of the controller buttons are operated by a micro-switch. When the button is depressed, an impulse of current is sent to the system. The impulse of current must be converted into a digital signal by turning it into a rectified square wave. The original *Tempest* circuitry implemented this by adding a resistance to reduce the voltage and a low-pass filter to square off the sides of the sinusoidal wave to create a square wave, digital signal. In our design, we chose a black, concave button with a three terminal micro-switch to serve as our four buttons on the controller for zap, fire, start and the coin in button. When the button is depressed the normally open terminal is closed and the normally closed terminal is open and the current impulse is sent to the FPGA input.

The 6502 microprocessor functions as the brain of the system. Within the 6502, the input data from the controller is analyzed. The data tells the 6502 whether it needs to execute instructions internally or forward data to the other components such as the Math Box, Vector Generator, Memory, or if data needs to be sent as output to the laser.

The ALU module in the 6502 handles all arithmetic and logical operations. The implementation will receive an Op-Code which is used in a case statement to decode the operation desired. This Op-Code is generated from the instruction decoding module. The ALU module receives registers A and B as inputs and operates asynchronously on these registers. In addition to standard arithmetic and logical operations, it also has the capability to perform logical shifts and BCD addition. If BCD mode is enabled, the appropriate carry values are generated to accommodate for the change is number formatting. The module outputs zero, sign, overflow, carry, and half-carry flags. The output value is clocked to an output register while Φ2 is high. From this register, the ALU output may either be passed back within the 6502 or tied to the external data bus.

The 6502 outputs its address and data busses to the address decoder. This decoder generates the appropriate interconnects based on the Atari© Memory Map in Table 4. The decoding logic allows the 6502 access to system memory, Vector Generator memory, and Math Box output data. Since all memory in this system is asynchronous, the address decoder provides the chip select lines which tie the address and data busses of the 6502 to these different components. The original Atari© design used smaller memory modules, accessed individually using these select lines from the decoder. We used the chip select lines to index one memory module of the appropriate size. The chip select lines are then converted to an index to determine which part of the memory is accessed. It also interprets specific address locations to system operations, such as Vector Generator start and reset. The 6502 may monitor

status locations such as one or two player start by writing the respective address to the decoder.

The Instruction Decoder is a compilation of the Predecode Register, Predecode Logic, Instruction Register, Timing Generation and the Random Control Logic. On every load cycle of the 6502, the next instruction is passed through the control logic. If the instruction will take multiple processor cycles to complete, the Timing Generation will control when the next instruction can be loaded. Each instruction also controls which of the 62 signals will be set from the Random Control Logic.

The Stack Pointer Register tracks where in the stack memory the top of the Stack is located. The Stack is a top down allocation in memory, so adding a value to the Stack decrements the address. The Stack is used when there is a subroutine call or a return from a previous call.

The Program Counter controls which instruction will be read from memory next. Typically the Program Counter will simply be incremented by one. In the event of a subroutine call, return from subroutine or a branch, the new address written to the Program Counter is provided on the address bus. The original processor uses two counters, a high and low, to control the Program Counter. Despite the fact that better technology was available, we replicated this two-register implementation to avoid the need to alter the micro-code of the processor.

The Decimal Adjust Adder is contained within the 6502 ALU and converts a binary number to the decimal equivalent. This is done by two separate decimal adjusters, one for the high and one for the low. This converted value is passed onto the Accumulator.

Our implementation of the 6502 contains three main modules. One module handles all of the ALU functions and passes the resulting outputs to the top module for forwarding to other parts of the 6502 or external components. The next module consists of decoding the Op-Code passed into the 6502 and handles all execution of the Op-Code, including the timing, data passage, and updating of the system registers. The final module passes the data fed into the 6502 to the other modules. This top module also assists in updating the system registers and will forward the results of the Op-Code executions to the Address Decoder.

Data will be passed to and from the various components through the use of the Address Decoder. This Address Decoder uses the address produced by the 6502 and the various clock signals to create a set of enable lines that will activate data busses to and from the 6502, Vector Generator, Program ROM, Program RAM, and Math Box. The Address Decoder effectively implements the memory mapped I/O attribute of the Atari© *Tempest* game.

The Math Box handles all of the mathematical computations for the gameplay such as shifting and rotating the figures that are drawn as output. Data is passed in from the 6502, in accordance to the input signals from the controller. The data passed in activates a set of mathematical computations that are stored in the Math Box ROMs, but are passed to a series of bit slice ALUs. The Math Box has its own control blocks that determine if an instruction has been fully executed or if the Math Box needs to continue execution on the data. The operations executed by the Math Box may vary from sines and cosines to more difficult calculations. The following section provides further detail into the internal components of the Math Box.

The Math Box was designed in such a way that there was a header file for the entire module and sub-modules were programmed as separate Verilog files. Inputs to the Math Box come from the External Data Bus, External Address Bus, and various clock signals. The input signals from the External Address Bus are decoded through the Auxiliary Address Decoder. Once the addresses are decoded and sent to the indexing ROMs for the Math Box, the operations required such as sines and cosines are instructed through the lookup within the six 2K ROMs and then passed to the four bit-slice processors for the actual execution of the operation. Following these operations and any necessary looping within the Math Box to execute the operations, the final data is passed back to the other components of the system through the External Data Bus. This module is in direct communication with the 6502 since the 6502 will determine which operations are desired after reading in the input from the user controller.

The next component is the Vector Generator. This component creates the output images as digital signals, which are later forwarded to the output circuitry. This section will detail the inner-workings of the Vector Generator.

The Vector Generator module (vg.v) receives inputs from the 6502 module. These input signals include start, reset, and clocks (Φ2, 1.5 MHz, 3 MHz, 6 MHz, and 12 MHz). It also receives VMEM and R/W lines, which control the address bus used within the Vector Generator. If VMEM is valid, then the VG will select the 6502 output address bus and, depending on the value of the R/W line, either read the VG RAM or write the contents of the 6502 output data bus to it. Other than these signals, the remaining operations within the VG are completely internal.

When the Vector Generator receives the start signal from the 6502, it will initialize the first address of the either the 6502 address bus or VG Program Counter (PC). If the 6502 is accessing the VG RAM, it will use the VG data bus to either read or write data. VG ROM might also be accessed during instruction execution. This memory contains predetermined instructions for drawing common shapes. The first state of the machine will read a new Op-Code from the VG data bus and find the next state of the instruction. The calculation of the next state depends on the halt bit, Op-Code, and current state. The next state is

generated based on certain timing constraints. These control signals are derived from the clock signals and current state bits in the State Machine Control logic. This circuitry controls the clocking lines for the next state and address. Once the Finite State Machine (FSM) generates a new state, its value is decoded to activate one of the latch or strobe signals.

The latch and strobe lines actuate most of the functions within the Vector Generator. There are four latch signals active for different states of the FSM. Each latch will tie the VG data bus to a different portion of the module or output. For example, Latch 1 ties the data bus to the output Y bus and the stack input. Depending of the instruction being executed, this information will be used either to draw a vector or modify the stack. The four strobe signals are used to process data that has been latched to various VG components. For example, if the instruction is JSR, the stack will need to receive the next address, available in the PC. This address is pushed onto the 4-word stack when Strobe 0 is activated. Then the stack pointer is increased by activating Strobe 1. Each strobe signal happens on a different state of the FSM.

The stack is four words deep and updated using strobe signals. It is only accessed for instructions such as JMP, JSR, and RTS. The program counter calculates the next address during the latching states. It can also write or read from the stack. Vector Timer Control (VTC) logic releases control signals for drawing vectors. It receives data from the latch states and uses the instruction Op-Codes to implement the appropriate function, such as loading scaling information. All of this logic is processed and affects the Vector Timer logic. This is essentially a counter which allows the machine to draw until the counter is full. The control signals from the VTC can speed up or slow down this counting process.

The Vector Generator outputs values to the X and Y output data busses. This data may be shifted depending on the Norm control signal, generated from the VTC logic. The VG also outputs linear scaling values via the Y output data busses. This data is clocked using a SCALELD signal, which comes from the VTC logic. VCTR and CENTER signals are sent to the output in order to control drawing time. Lastly, the VG releases color intensity information which can change based on the vector's length and time-to-draw.

The ROM and RAM for the system are comprised of numerous blocks ranging from 2K to 20K. Many of the components had multiple 2K ROM chips that were connected in series. For example, the Math Box had six 2K ROMs connected together. The .bin files for all of the ROMs throughout the system were obtained through open source code websites/databases. These .bin files were converted to a list of hex numbers by the use of a script that was written by the group in the C Programming language. After decoding these .bin files, the ROMs will be programmed using a Coregen and later loaded into a Verilog file. Due to simplicity of programming the ROM, all of the ROM was programmed as one

large block rather than splitting it up into individual blocks. This is slightly different from how the memory was set up in the original *Tempest* game. This change required a slight modification to the system; the addition of an indexing system. This indexing system takes in all of the inputs to each ROM as was specified on the original schematics, but then deciphers which index to look up for the newly arranged indexes. Within this master ROM block, each component still has its ROM blocked together.

In the original *Tempest* design schematic, two digital-to-analog converters (DAC) were used to create the output necessary for the CRT screen in the upright machine. In our design, we used Laserpic's CW20 Laser Galvanometer Scanner to physically display the vectors created by the DACs. One DAC was responsible for telling the laser how long to make the vector. This DAC was labeled the scaling DAC. The second DAC was responsible for the slope of the vector that is drawn. This DAC was labeled the slope DAC. These currents combined to create the final vector product that is shown by the galvanometer-scanner.

The audio output was handled by the Atari© POKEY chip, which can take an input of 5 volts DC from the FPGA output port. The data lines selected on the POKEY chip determine the frequency at which the square wave output is fed into the audio-in to the PCB. The PCB takes the frequencies from both POKEY chip and combines them using a summing amplifier. The output of the summing amplifier goes directly into the Audio+ input for the speaker. The output of the summing amplifier is fed into an inverting amplifier to invert the signal and send that newly inverted signal to the Audio- of the speaker.

3.2 FPGA Design

3.2.1 6502 Microprocessor – T65_Pack.vhd

The following section details the design process for each sub-component of the 6502 microprocessor. The design description will distinguish whether it was instantiated in VHSIC hardware description language (VHDL). The T65 package is obtained through the use of open-source software from OpenCores.org. This code has been used in other projects similar to this project, such as in an FPGA emulation project for the Atari© *Asteroids* game.

3.2.1.1 6502 Top Module – T65.vhd

The top module for the T65 VHDL Package, T65.vhd, is used to handle the initialization and updating of all of the system registers such as the Program Status Register (Flag Register), Program Counter, Index Register, and Stack Pointer Register. In addition to these registers, the T65 module also handles the passing of information to and from the T65_ALU and T65_MCode modules.

Interrupts are also handed within this module. If an interrupt is received, the module will update the Program Counter and Flags as required by the interrupt received and will access the appropriate memory location within the E000-FFFF hexadecimal range to execute the subroutine associated for the triggered interrupt.

In relation to the T65_MCode module, the T65 module must decode the data input for the 6502 into the appropriate Op-Code. The top module also determines a set of conditions that dictate whether or not an Op-Code is able to be executed by the 6502. The first condition is for the *really_rdy* signal. This signal is the gated result of the *Rdy* input signal for the 6502 and the $R/\overline{W}$ signal to make sure that the 6502 is safe to stop, when the *really_rdy* signal is set low, meaning that the *Rdy* signal has been disabled or set low. The next condition is to check for both the *Enable* and *clk* signals to be high at the same time. This effectively achieves the timing effect of the use of the Φ_1 and Φ_2 clocking signals from the original 6502 chip.

3.2.1.2 Arithmetic Logic Unit – T65_ALU.vhd

Since the Arithmetic Logic Unit (ALU) is a relatively independent component in the 6502 microprocessor, it receives its own VHDL module. It is asynchronous with input and output registers to control the flow of its data. Inputs include the A and B registers, the carry-in bit, decimal mode enable, shift right, and an Op-Code. The A and B registers contain values latched from either the data or address busses, depending on the current instruction and state being executed. This latching does not take place within the ALU Verilog module, as it will receive only the current value of whatever was most recently latched to these registers. The instruction and state also determine some special cases for these registers, such as latching zeros into register A, or inverting the data bus input latched to register B.

The Op-Code is generated in the 6502 top module. This code is specific to the ALU design and does not need to be generalized to a particular function. The Op-Code is selected based on the current instruction and cycle state. It is used in a case statement within the ALU module to determine what operation to perform.

Recall that the original 6502 ALU had control lines AND, OR, XOR, SUM, Carry-In Enable, and Decimal Mode Enable. Essentially, the Op-Code input to the ALU Verilog module will reflect these control lines, but in a more concise structure. Given these control signals, the ALU is able to implement addition, subtraction, OR-ing, AND-ing, and XOR-ing of registers A and B. It can also perform an arithmetic shift left on register A by adding A to itself. Finally, it can simply latch register A to the output register.

Decimal Mode Enable determines the Half Carry and Carry output flags. If this signal in enabled, the ALU module recognizes that the numbers being added are

in BCD format and calculate the carry flags accordingly. The Shift Right Enable will perform a right shift with carry. The carry bit is shifted into the MSB of register A and all other bits move one position to the right.

The function of the Decimal Adjust Adder is to convert a binary number to a binary coded decimal. This functionality is handled internally to the T65_ALU module.

The output data lines are forwarded to the T65 and T65_MCode modules for further use or for forwarding to external components. Zero, Sign, Overflow, Carry, and Half-Carry flags are also generated from this module and update the 6502 status register immediately.

3.2.1.3 Microcode (Instruction Execution) – T65_MCode.vhd

The microcode module (T65_MCode.vhd) of the T65 package serves as the complete Instruction Decoder of the original 6502 microprocessor chip. This module is passed an instruction Op-Code and then proceeds to decode the Op-Code. This Op-Code will be compared through a case statement to determine the exact instruction that is to be executed. Once the correct instruction has been selected, the Op-Code will perform the execution and update the appropriate registers as determined within the case selected from the case statement. Following the execution of the Op-Code, the resulting data is then either forwarded to the T65_ALU module or the T65 module as seen fit for the selected Op-Code. The execution of an Op-Code within the T65_MCode module is only performed if a specified set of conditions are met within the T65 module.

3.2.1.4 Address Decoder – T65_Address_Decoder.v

The 6502 accesses most of the external components, such as the Math Box and Vector Generator, via a single address bus and single data bus. This module implements the logic to determine what components are accessed by the 6502 by latching the respective data and address busses. It uses the 16-bit address bus of the 6502 and the Memory Map released by Atari© to decode the various functions to be performed. In general, the upper nibble of the address bus is decoded to generate a variety of control signals.

Memory access lines are decoded to enable the system RAM and ROM. The original *Tempest* hardware used 4KB memory blocks and chip-enable lines to index which block was accessed. For example, the game ROM was made up of five 4KB memory blocks, for a total of 20KB. The Atari© Memory Map dictates that hexadecimal addresses 9000h through DFFFh are used to access these 4KB blocks. This means that the decoding logic will select the appropriate chip-enable line to read from when one of these addresses is generated from the 6502. Similarly, there is a range of addresses which are valid for RAM access.

Given the memory being accessed, RAM or ROM, the data bus for the 6502 will be latched to the appropriate memory bus.

In addition to system memory, the 6502 can also control and access parts of the external modules such as the Vector Generator. For most of these control lines, the upper 3 bits of the 6502 address bus are used. For example, when the 6502 wants to write to Vector Generator RAM, it must output an address between 2000h and 2FFFh. The upper 3 bits, 001b, are decoded to enable the VMEM signal. VMEM chooses the 6502 address bus within the Vector Generator Verilog module for address decoding. Other functions available through this decoding process include the Vector Generator start and reset, Math Box read and start, One or Two Player start, and Audio read or write. These functions are modeled using the Atari© Memory Map in order to ensure that when the 6502 releases an address, it performs the desired function.

3.2.2 Math Box, POKEY and External Address Decoder

The Math Box and POKEY are all activated by the External Address Decoder. The module which will control and link all of these individual modules is named "Schematic_3B.v." The External Address Bus, various control signals and the External Data Bus will all be fed from the 6502 to "Schematic_3B.v." This module act as a top module be decoding and forwarding the data from the 6502 to the separate modules. The Verilog module hierarchy is summarized in Figure 27.

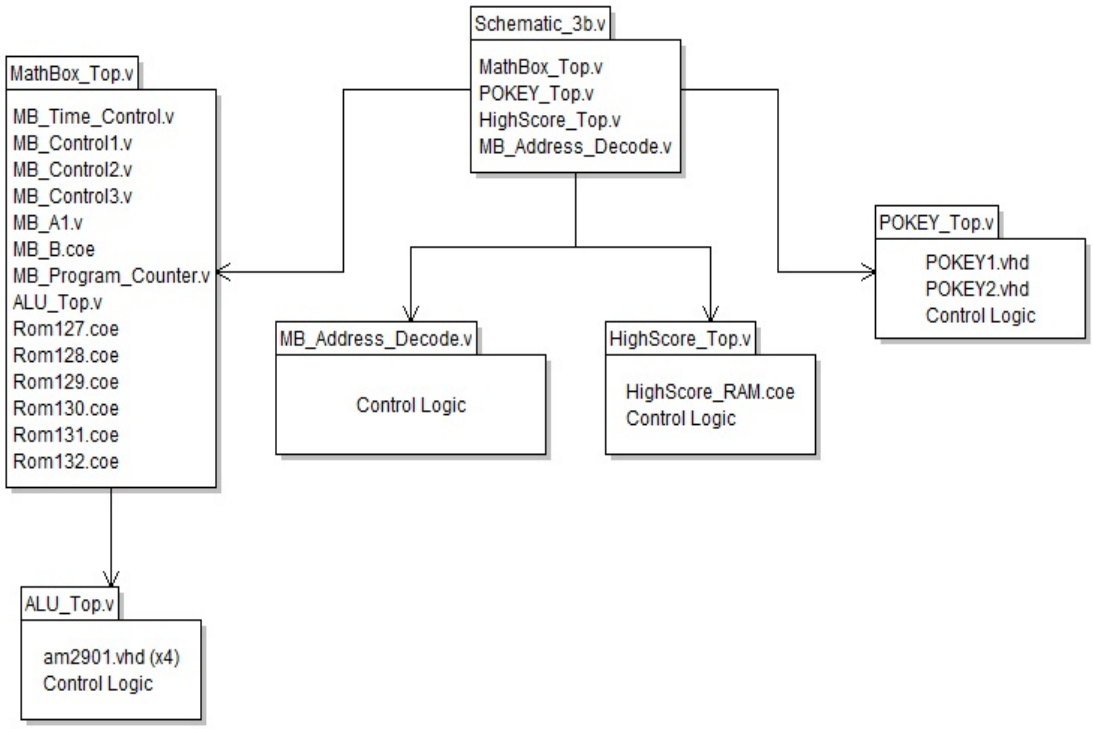


Figure 27: Math Box Verilog Software Overview

3.2.2.1 External Address Decoder – MB_Address_Decode.v

The External Address Decoder activates the Math Box, the POKEYs or the High Score Ram, based on the address and control logic set by the 6502. Only bits 5-8 from the EAB are needed to determine the chosen function. The other bits are used in the individual modules to choose specific functions.

3.2.2.2 Math Box – MathBox_Top.v

The Math Box contains all of the microcode to run the game's four AM2901 ALUs. When the 6502 needs the Math Box to perform a new calculation, it writes an address corresponding to the Math Box's range to the External Address Bus. The top four bits are used by the External Address Decoder.

3.2.2.2.1 Initial Address Decoder – MB_A1.v

The first module we created was the A1 module. This module takes in the bottom 5 data lines from the EAB. This module then produces the initial address that will pass to the Program Counter module as 8-bit data lines. This module had two options for its implementation. The first way was to use a small code script that was written in C, to convert the .bin file from the original ROM to a list of hexadecimal values. These values were then used in a Core Generator to program an actual ROM chip. The other option was to create a Verilog file that uses 32 case-statements to check the input of the bits from the External Address Bus and produce the corresponding 8-bit output. The 8-bit output was hardwired into the Verilog file by using the hexadecimal values that are gathered from running the C script on the .bin file for this ROM. In either situation, the 8-bit output is passed as inputs to the Program Counter module. Since we had a limited amount ROM space available and still had room for extra registers, we opted for the 32-case statement implementation.

3.2.2.2.2 Data Loop Latch – MB_B1.v

The next module is for the B1 chip, which takes its inputs from the four data lines from the L1 and K1 ROMs, along with the corresponding clock and enable lines. This module simply acts as a latch to forward the data on the input lines from the L1 and K1 ROMs to the Program Counter module. The latch is determined by either a case-statement or an if-statement that checks the specified conditions for latching data as specified in Section 2.6.1. The control logic and ROM code will reside in the MathBox_Top.v file.

3.2.2.2.3 Program Counter – MB_Program_Counter.v

Following the B1 chip, the next module is the Program Counter module. This module has the filename MB_Program_Counter.v. In the original schematic from Atari©, this program counter was created by using two 4-bit counters, which was most likely due to the technology available at the time. This same counter can be created by using Verilog to make a single 8-bit Program Counter, which is the method that we selected. This module takes either the data lines from the A1 or B1 modules as input lines, along with corresponding clock and enable lines. Based on the specifications provided in Section 2.6.1, the Program Counter simply increments the input by one or latches the current address, if necessary. Then the value is passed to the module controlling the Math Box ROMs.

3.2.2.2.4 ROMs – L1.coe, K1.coe, J1.coe, H1.coe, F1.coe, E1.coe

The Math Box ROMs are instantiated as separate ROM files. These files were loaded with a corresponding .coe file that is created through a Core Generator. The .coe files were produced by using the Core Generator to program the hex values from running the C script on the .bin file for each ROM. Each of the Verilog modules for these ROMs takes the 8-bit output from the Program Counter module and use these lines as address lines. These address lines look up the same index in each ROM and produce a 4-bit output for each ROM as specified in Section 2.6.1.

3.2.2.2.5 ALU – MB_ALU_Top.v and AM2901.vhd

The MB_ALU_Top.v module will control and pass data between the top Math Box module and the four ALUs. The four ALUs we used are based on open source VHDL code that we acquired. This code was modified slightly to make simulation in Xilinx and Synopsys easier. The main changes were to change some of the input and output signals from “buffers” to “input” or “output” signals. Also, the original AM2901 designated some signals, such as the carry in or stack counters, as a bi-directional input/output port. Since some of these signals are only used as either an input or an output, and simulators have difficulty interpreting the correct behavior, the signals were changed to either be strictly “input” or “output” signals. This created a new problem, as some signals were strictly output on the ALU dealing with operations on the MSB, but input on the ALU dealing with the LSBs. To fix this problem, separate copies of the VHDL code were made, with these slight modifications to each copy.

3.2.2.2.6 Control Block 1 – MB_Control1.v

Control Block 1 was created as its own module and titled MB_Control1.v. This module is comprised of a D Flip-Flop and a couple basic logic gates. The inputs to this module are the LSB from the H1 ROM and corresponding clock and enable signals. The input must first traverse through the D Flip-Flop and then

through the logic gates as described in Section 2.6.1. The output produced by this module is then forwarded to the B1, ALU, and Control Block 2 modules. The output to each these modules could first undergo inversion as specified by the schematic and described in Section 2.6.1.

3.2.2.2.7 Control Block 2 – MB_Control2.v

Control Block 2 was created in a similar manner as Control Block 1. The module was titled MB_Control2.v. This module consists of a D Flip-Flop and basic logic gates. The inputs of the module are the output of Control Block 1, the Q0 bit from Control Block 3, the O2 bit of the E1 ROM, the O2 bit of the J1 ROM, and the corresponding clock and enable lines. As with Control Block 1, some of the inputs are passed through the D Flip-Flop, which then is passed to the series of logic gates. This results in a single output line is passed A10* bit of the ALU module.

3.2.2.2.8 Control Block 3 – MB_Control3.v

Control Block 3 consists of a series of logic gates that are connected in series, while some are also connected in parallel. This module is titled MB_Control3.v. The inputs to this module are the MSB and OVR lines from the ALU module, the O3 and O4 bits from the E1 ROM, the O2 bit from the F1 ROM, and the $\overline{\text{BEGIN}}$ line from Control Block 1. These inputs go through a set of logical operations as described in Section 2.6.1. The corresponding outputs from these logical operations are passed to the following modules: Control Block 2 (Q0 bit), ALU (R3 bit for the Register Stack), and the Program Counter ($\overline{\text{PCEN}}$ line).

3.2.2.3 POKEYs – POKEY_Top.v

The POKEYs interfaces the game's controller and buttons to the 6502. The POKEY_Top.v module will forward the address from the 6502 to the two POKEY chips. Some of the functions that the POKEY supports are generating random numbers, creating audio signals and tracking the position of the game controller. The open source code was originally designed to emulate Atari's© *Asteroids Deluxe.* Since the POKEY was used in a similar configuration in *Asteroids Deluxe* and *Tempest*, no modifications to the original code were needed. Figure 28 illustrates the data flow with the POKEY and its simulation.

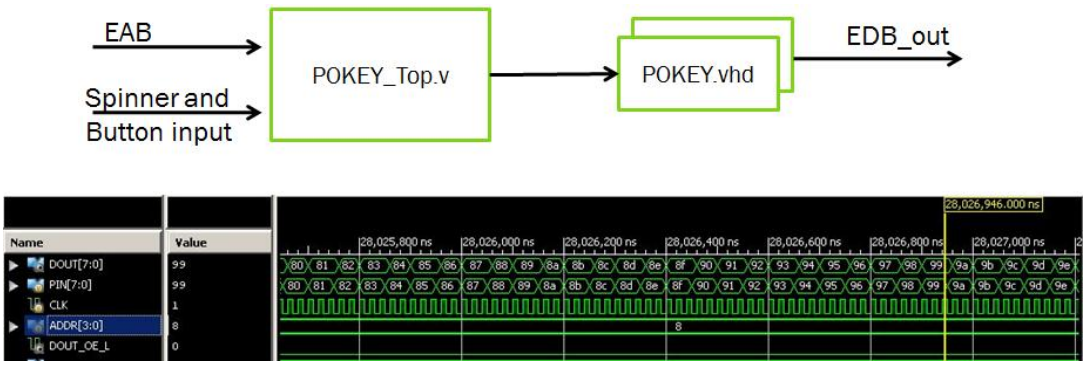


Figure 28: POKEY Data Flow and Simulation

3.2.3 Vector Generator

The Vector Generator consisted of a combination of Verilog (.v) and schematic (.sch) files. Ultimately, the Vector Generator design reflects the original Atari© Vector Generator. The goal was to properly model each of the major components within the Vector Generator using Verilog and schematics. The choice between schematics and Verilog code was simple. When it is more efficient and effective to simply reproduce the combinational and sequential logic of a particular Vector Generator component, a schematic will be used. If a part is very simple in logic structure, rather than trying to design the component in Verilog, which may often be a tedious process, it will be easier and more accurate to generate the circuit model for said part. Many of the components in the Vector Generator are easier to design using Verilog code. If the functionality of a component, such as a state machine decoder, is easy to design using coding logic, it is more efficient to generate this part in Verilog. While the original circuitry was changed, the specific function of the component remained intact and was properly implemented in a more concise way of reflecting a particular logical path.

Each Verilog and schematic file was interconnected with other modules within the overall Vector Generator Verilog design. In general, the signal path begins with some control signals received from the 6502, ordering the Vector Generator to begin operation. This initializes all address and control signals in the Vector Generator. The address bus is then decoded and read from either RAM or ROM. In most cases, RAM is read at the start of Vector Generator operations. Once the address has been decoded and memory is accessed, the data bus will contain a valid instruction. The Finite State Machine (FSM) then latchs this data to receive the current instruction's Op-Code. Each next state is calculated once the FSM Control logic generates a valid clocking signal. Meanwhile, the current state is decoded and a variety of operations may take place elsewhere in the Vector Generator. Depending on the signal generated from the FSM decoder, the Vector Timer Control logic, Stack Pointer, or Program Counter may be updated. Additionally, data may be latched to the output FPGA interface. The Stack Pointer and Program Counter receive control signals from the FSM decoder and output latches. Based on these signals, a new address is calculated and sent to the Address Decoder. The Vector Timer Control logic determines the control

signals for the Vector Timer, which times the drawing of a vector on the output. Once all components have been updated based all of these various control signals, the next address is decoded for the execution of the next instruction. The overall flow of data between each individual module is shown in Figure 29.

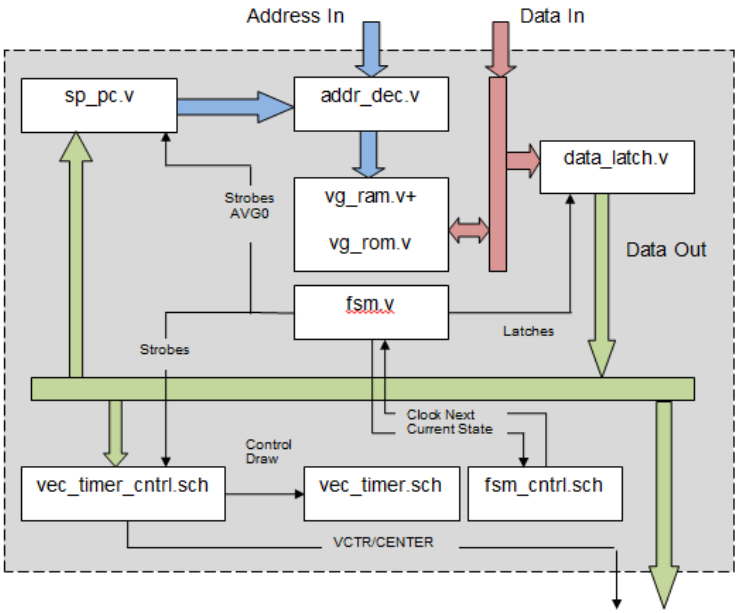


Figure 29: Dataflow Between Vector Generator Modules

3.2.3.1 Address Decoder – addr_dec.v

The Address Decoder receives two 13-bit address busses. One bus is from the 6502 and the other is from the Vector Generator program counter. A control signal, VMEM, which selected between the two address busses. If the 6502 wants to read or write to Vector Generator RAM, it simply writes the corresponding address in its own decoder logic which activates VMEM and selects the 6502 address bus. It may only access VG RAM when Φ2 is low, meaning that a valid address is latched on the 6502's address bus. If VMEM is not active, then the address decoder will default to the program counter bus. This means the Vector Generator is in standard operation mode, executing instructions within the VG RAM. The top bits of the 13-bit address selected by the latch will be decoded in order to select the appropriate 2KB RAM or ROM location. In order to avoid complications with game operations, this address decoding method remained intact. Some indexing was used to maintain the original games decoding process while allowing the RAM and ROM to be instantiated in one module.

3.2.3.2 Memory – vg_ram.v and vg_rom.v

These memory modules were instantiated within the FPGA's Block RAMs. They are asynchronous memory devices, which needed to be considered in order to properly access the device. Block RAMs within an FPGA are typically

synchronous modules. To solve this problem, a high speed clock, perhaps around 100 MHz, was utilized. This ensured that the memory was read quickly enough to virtually become asynchronous to the Vector Generator, as it had only a minor impact in the timing of signal propagation after memory is accessed.

3.2.3.3 Finite State Machine – fsm.v

The Finite State Machine receives a halt bit, current Op-Code, and previous state. If the halt bit is '0', then the next state is always Idle. The next state will be calculated using a case statement. This statement will read the Op-Code and determine the next state based on the previous state. It is asynchronous and will update immediately given a change in previous state or Op-Code. Once a new state is found, it is decoded to activate one of the latch or strobe signals, discussed in the Atari© Vector Generator research section. The new state also may update the LSB of the Vector Generators next address, AVG0. This bit will swap according to the current instruction and the state of that instructions execution.

3.2.3.4 Finite State Machine Control – fsm_cntrl.sch

This schematic will determine the timing for FSM updates. Since the FSM is asynchronous, it requires a latch to control when the previous state is fed back to the FSM. This module also controls the clock for AVG0 and acts as clear for the FSM decoder. When the AVG0 logic is clocked, the decoder is cleared in order to prevent any unwanted signal propagation from the latch and strobe signals. The FSM Control logic was fairly simple and was based solely on timing signals and state bits. In the event that the 6502 processor is accessing Vector Generator RAM, the FSM Control will be reset using the VMEM signal. The schematic for the FSM Control is showed in Figure 30.

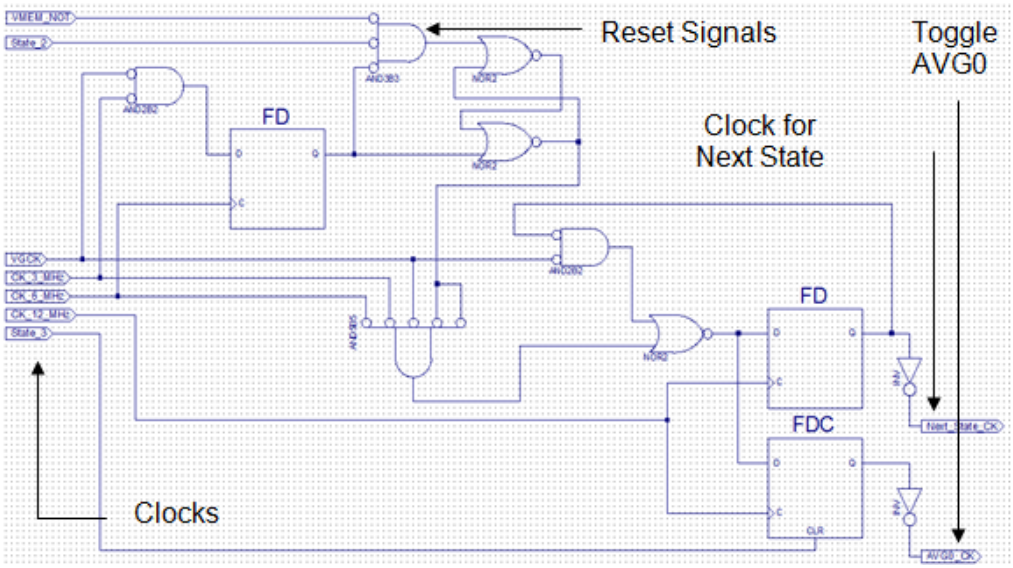


Figure 30: Schematic for FSM Control

3.2.3.5 Data Latch and Shifter – data_latch.v

This module receives control signals from both the FSM and Vector Timer Control. Based on the current latch signal, the data bus will latch into different output or feedback lines. This includes the output X and Y data busses, Op-Code lines, and color intensity output lines. The latches can also act as shifters, based on the control signal NORM, received from the Vector Timer Control logic. This signal, when active, shifts the output data busses to the left. This function is performed based on some control information for the vector being drawn.

3.2.3.6 Stack Pointer and Program Counter – sp_pc.v

This logic receives several control lines, including strobe signals, the Op-Code, and AVG0. AVG0 is used to increase the program counter when appropriate. Depending on the Op-Code, when strobe 2 is active, the program counter may load a new value. This data is latched from the output Y data bus via the Data Latch. This will only occur when using JSR or JMP instructions. The Stack Pointer receives Op-Code and strobe information as well. For instance, the stack pointer will read or write data based on the Op-Code value. It will also increase or decrease after storing or read a value based on Op-Code information. The Op-Code bits control a stack pointer counter which is increased or decreased appropriately given the instruction being executed. Given a JSR or RTS instruction, the stack pointer will write the program counter into its current location when strobe 0 is active. When the stack pointer is read, these data lines are later clocked into the program counter, to be sent to the Vector Generator address bus.

3.2.3.7 Vector Timer and Control – vec_timer.sch and vec_timer_cntrl.sch

The Vector Timer Control schematic receives an array of signals which influence its circuitry. The basic function of this circuitry is to implement various tasks to be executed given a specific instruction. For example, when a SCAL instruction is executed, the Vector Timer Control latches the binary scaling information, which will be used to influence the timing of a vector draw. The output signals of the Vector Timer Control are fed directly to the Vector Timer. In Figure 31, the capitalized signals represent signals that are not only used in the Vector Timer Control but also are sent to the output pins. The Vector Timer Control performs different functions depending on the current state, represented by the strobe lines, and the current instruction, defined by the Op-Code signals. The STOP signal is a feedback line from the Vector Timer circuitry which controls the JK Flip-Flops for VTCR and CENTER. These signals dictate the valid timing for a vector draw or centering function.

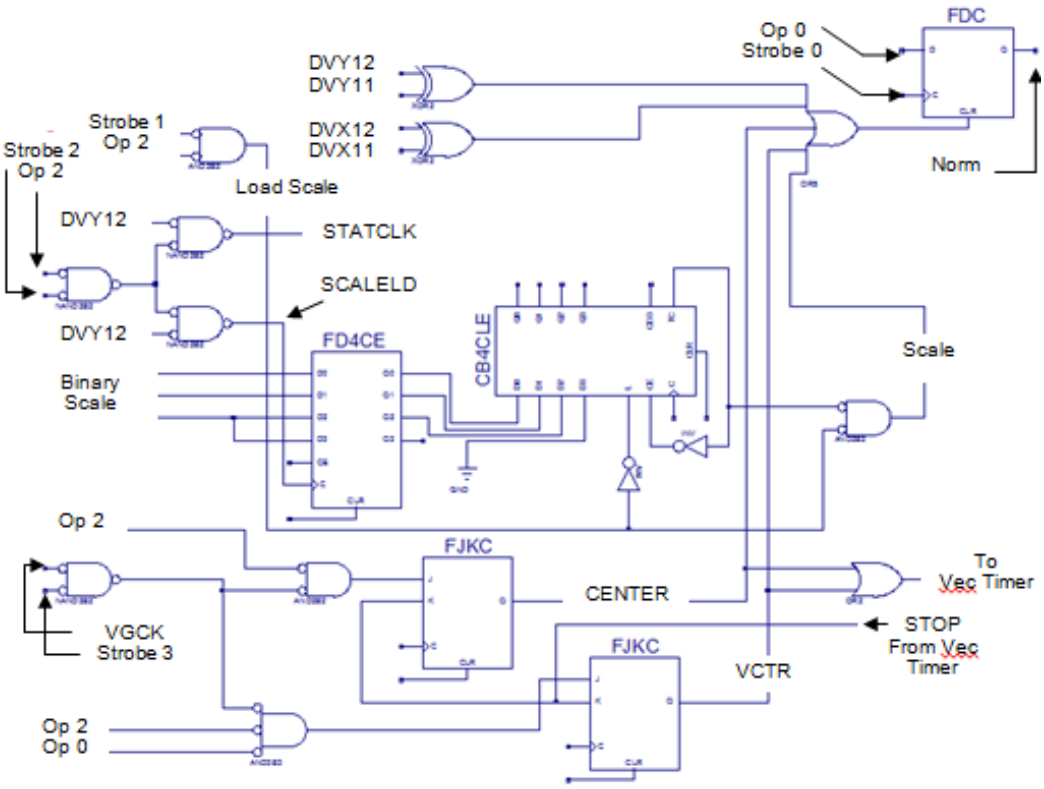


Figure 31: Vector Timer Schematic

The Vector Timer is similar to a counter which increases until full. While counting, a vector may be drawn on the output. When the counter is full, the drawing stops. This is implemented by sending the STOP signal back to the JK Flip-Flops within the Vector Timer Control. Based on the timer control signals Scale and Norm, the time to count to full can be reduced, using a shifting feedback connection for each 4-bit counter. The Norm and Scale signals allow the input to the counter to be loaded. This input is the bit to the left, so when the load lines for each counter are active, the circuit performs a shift to the right. Two of the counters are not pictured in Figure 32 due to their redundancies with the components shown. The STOP signal is activated once all counters are full, meaning the carry-out bit for every counter is high. The carry-out bits are ANDed together to produce the STOP bit.

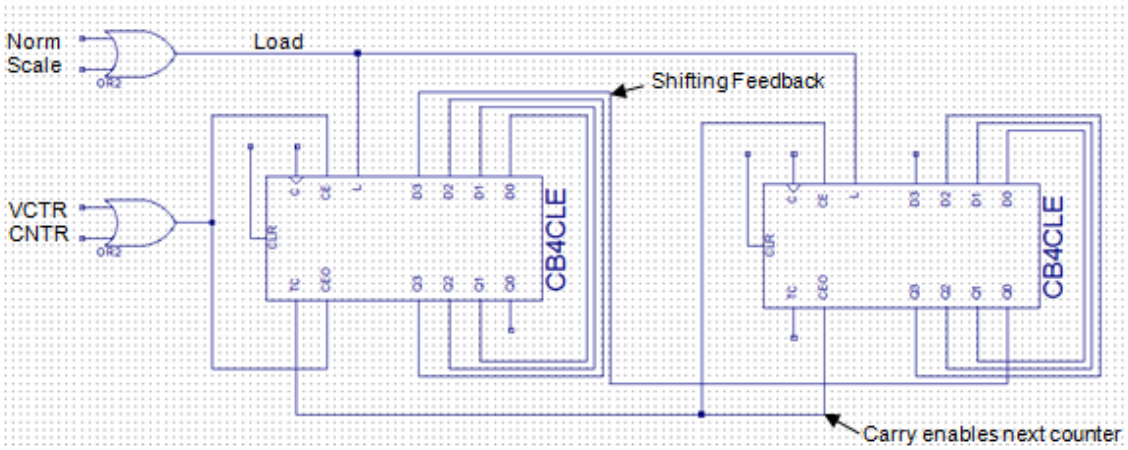


Figure 32: Vector Timer Control Schematic

3.2.4 Clock Generation – clk_gen_top.v

The Verilog and schematic modules within our FPGA required the following clock signals: 12 MHz, 6 MHz, 3 MHz, 1.5 MHz, 24 kHz, 3 kHz, Φ1, and Φ2. The input to the FPGA is a 96 MHz clock. This clock is divided four times to produce the 12 MHz clock. This clock will be an input to a 4-bit counter which counts up on each positive edge of the 12 MHz signal. By doing this, the four output lines represent divisions by 2 of the input signal. Thus, a 6 MHz, 3 MHz, and 1.5 MHz clock may be created. This process continues further with two additional 4-bit counters. The lines are tied together such that a 24 kHz and 3 kHz clock can be generated from a 1.5 MHz counting signal.

Φ1 and Φ2 are generated using the 96 MHz clock. The 96 MHz allows for a slight offset of these signals from the 1.5 MHz clock. Φ1 and Φ2 are non-overlapping, so this will be taken into consideration based on the 6502 clock signal timing diagrams. Once these clock signals are created, they are used in the 6502 to operate all modules. All other clock signals are distributed across the FPGA to be used in the necessary modules.

3.3 I/O Design

In this section, all design plans for any of the I/O components will be discussed. These components include the Memory, Game Controller, Laser, and the Analog Circuitry needed. Each section details the individual test cases that will be used for testing the component, while also expressing the desired result that would signify a successful test case.

3.3.1 Memory (ROM and RAM)

The ROM that will be used for the emulator will be handled as a part of the standard amount of SRAM that is contained within the FPGA. The FPGA can

program parts of the SRAM to be treated as the various types of ROM and RAM that are needed, while also setting them to be synchronous or asynchronous based on the given specifications. No additional memory was needed since the FPGA has enough memory to far exceed the amount we needed.

To begin, we needed to get the data from the various ROM chips that are found on the actual *Tempest* circuit boards. This information is available online through various open source websites as multiple binary files (.bin). The next step for certain components was to load these .bin files into an actual ROM chip that will be placed on the PCB. Another option was to convert these .bin files into Verilog code that is loaded into the FPGA along with the remaining source code. This second option allows the FPGA to treat this ROM as virtual memory.

One downfall to loading the ROM data into the FPGA's SRAM is that the data would have to be loaded into the FPGA every time after the power to the board was disconnected, as is done with the main source code. This process could become lengthy in time since the ROM Verilog files would not be the only thing to program within the FPGA, but additionally, the group would still have had to program major components such as the Math Box, Vector Generator, and more importantly the 6502 microprocessor. These individual components can consist of two or more Verilog files in their own right, which were large as they covered a multitude of instructions within themselves. On the other hand, designing ROM that is not programmed within the FPGA but rather is comprised of actual ROM chips on the PCB has its detriment. Using actual ROM chips can pose the problem of taking up a great deal of space on the PCB depending on the type of chips that the group selects for the project. These chips also have to be interfaced with each of the major components of the project. Due to the complexity of interfacing the individual ROM chips with the 6502 processor and the high performance degradation experienced by the FPGA for the ROM location look-ups, the group decided to use an FPGA that has enough memory on the board to handle the 32K ROM that was needed in the various sections. This ROM was programmed in the manner mentioned above. In an attempt to simplify the ROM within the system, the ROM wasn't programmed to the FPGA as separate modules, so that it emulates the separate physical ROM chips. They were actually instantiated as a single block of ROM, but while being indexed within the single block. This modification caused the group to add some small circuitry to decode the proper index to look up based on the original ROM that was intended to be looked up. This change was structurally be different from the original *Tempest* console boards, but was believed to provide more ease in programming for the system.

The RAM in the system was designed as a specific allocation within the SRAM of the FPGA. This allocation served as the RAM specified previously in the Atari© Memory Map, which had two allocations: CPU and Vector Generator RAM. Programmable ROM or RAM within the FPGA is performed through a program called CORE Generator with Xilinx. Since we used a program called *Synopsys*

to write the Verilog code for the components of the *Tempest* game, we also utilized their program that mimics CORE Generator. The process for programming the ROM described below was based on using the CORE Generator program, due to the group's familiarity with that program.

To begin programming the ROM/RAM, the user opens up his/her respective Coregen and starts a new project by going to "File -> New Project." In this case, it is Xilinx CORE Generator. This will pop open a small window that will ask the user for the title of the project he/she wishes to create and where to store the directory that will contain the files associated with this newly created project. Upon completion of this step, the user now must then select the appropriate package of information that pertains to the FPGA being used. This package selection will specify the Family, Device, Package, and Speed Grade. An example of this selection may be the following selections: Family: Virtex2P, Device: xc2vp30, Package: ff896, Speed Grade: -7. These selections can vary due to the FPGA board being used by the user.

At this point, the ROM/RAM is initialized and available to be edited. The user now clicks on "Tools -> Memory Editors." This opens up two windows that are shown side-by-side. In the window on the left, the user now clicks on the "Add Block" button to name the block. The remaining parameters need to then be initialized as illustrated below in Table 19. The parameters with a setting of XX indicate that the information can vary based on the number of lines for the ROM/RAM and/or the bit width of each line.

Parameter	Setting
Block Depth	XX
Data Width	XX
Address Radix	XX
Data Radix	XX
Supply Memory Content	(00000000~11111111)

Table 19: ROM/RAM Initial Parameter Settings

Now that all of the parameters have been set, the user needs to generate the .coe file that will later be loaded into a Verilog module. To generate the file, the user chooses "File->Generate." A window will pop up as seen in Figure 33.

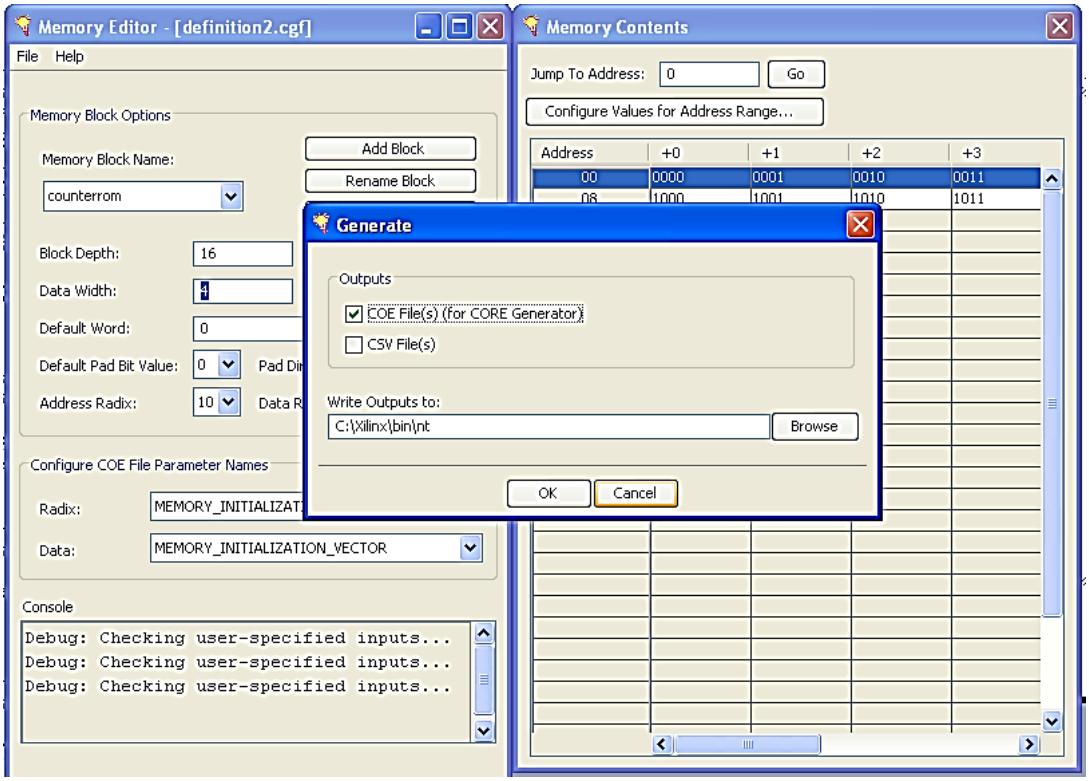


Figure 33: ROM/RAM Initial Parameter Settings Window

In the newly opened window, click “Output->coe file(s) for core generators” and then click “ok” to save it in an appropriate file location. A notification that the file was generated successfully will appear as shown in Figure 34.

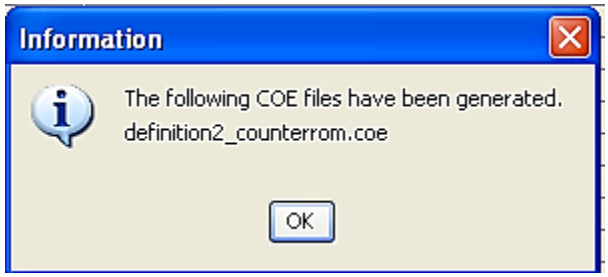


Figure 34: Notification of Successfully Generated .coe File

The final step in creating the ROM or RAM is to load the .coe file into a Verilog project. Begin this by opening up the desired ISE. The example will be illustrating the steps through Xilinx ISE 9.2i. The user must first select “Project ->New Source->IP (corgen & architecture wizard) -> Next” to open up the window shown in Figure 35.

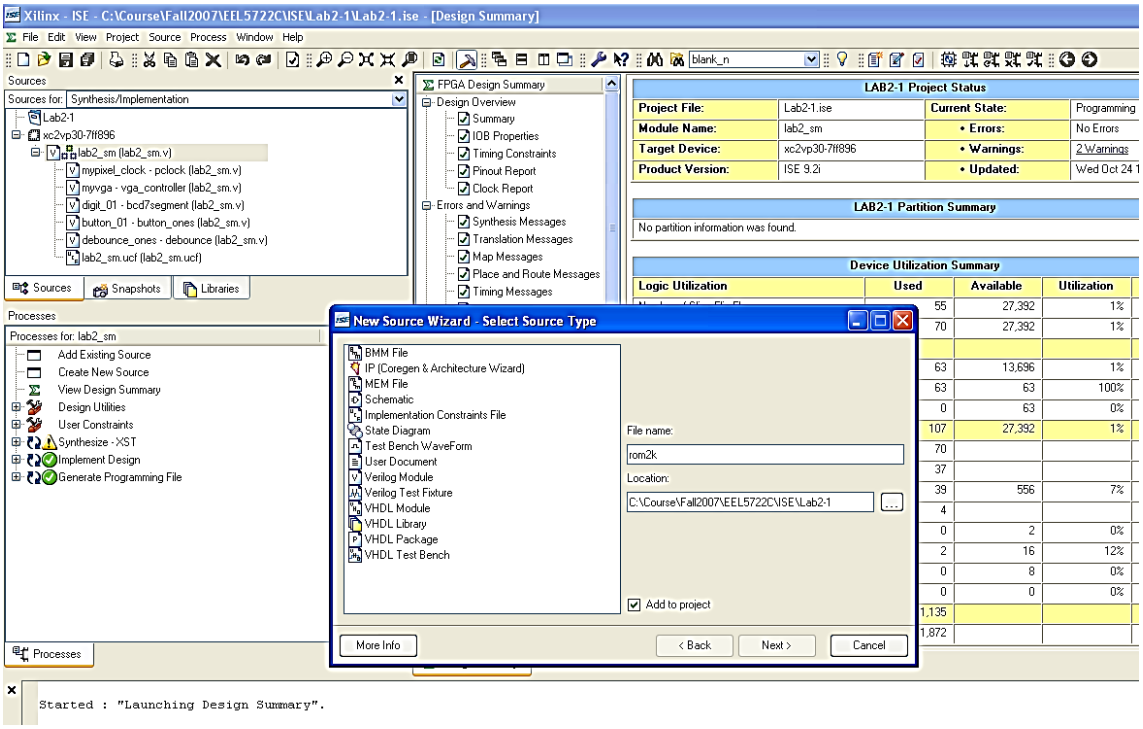


Figure 35: Create New Project in Xilinx ISE 9.2i

Once this window opens up, select “Memories & Storage Elements -> RAM & ROM -> Block Memory Generator v2.4 -> Next -> Finish.” This will open a new window shown in Figure 36. As illustrated by Figure 36, the user then selects the memory type according to his/her needs (i.e. ROM or RAM) and set the field called Algorithm to the setting of Minimum Area, then Select “Next.” On the next screen, select the memory size to have the Read Width and Depth to match the needs of the ROM/RAM, while setting the Enable field to be Enable Always and the Output Reset to be 0. Proceed by selecting the “Next” button. On the next screen, the user must set the Memory Size, Read Width, Read Depth, Enable, and Output Reset fields as in accordance for the ROM being created. Once setting these fields, select “Load Init File” by using the .coe file that was previously generated, then select “Next -> Finish.”

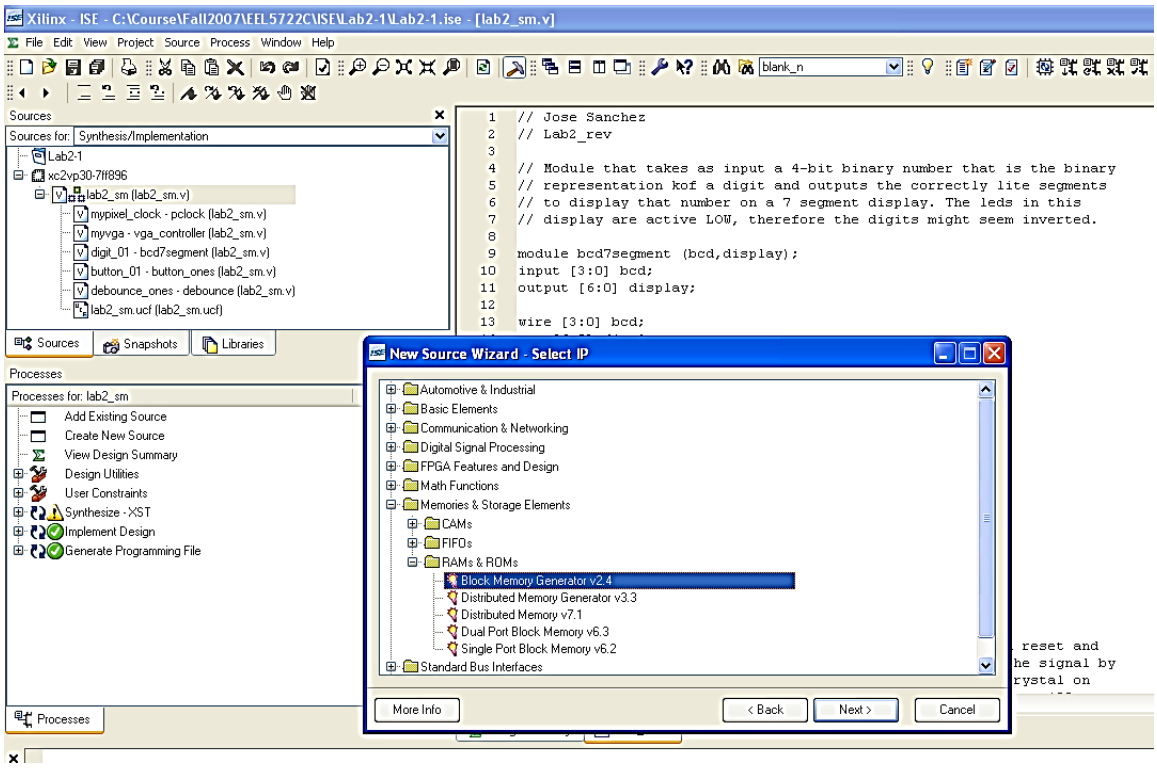


Figure 36: Memory Type Selection Window

At this point, the file has been loaded as a Verilog file within the project that was created in the respective ISE. This can be verified by selecting "Edit -> Languages Template -> Coregen -> Verilog Component Instantiation" to see the definition of the newly generated core interface. This core is now treated as a module and used in the design of your project.

3.3.2 Game Controller

The game controller needed to be designed in order to complete this design project. Some mechanical skills was necessary to design a metal or wood rectangular box that encompassed the TurboTwist arcade spinner wheel and the black concave buttons mentioned in Section 3.7.3, Game Controller Input. The box needed to be at least 7" x 5" x 4" to completely cover the area needed by the four buttons and the spinner wheel; the bottom of the box was left open for display purposes. Each terminal of the micro-switches needed to be hardwired via solder to connect the outputs to the printed circuit board inputs. The spinner wheel had a PS/2 interface and needed to be connected to the PCB input. The physical controller layout is shown in Figure 37.

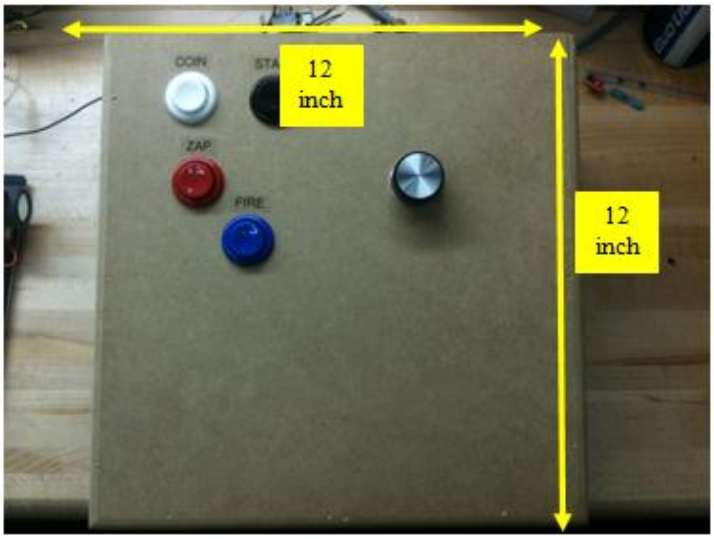


Figure 37: Controller Layout

3.3.3 Laser

The laser chosen for the design project was the CW20 scanning galvanometer. This particular galvanometer required a 5 volt input signal to provide its output. The onboard driver controls gain, size and position of the vector that is drawn on the screen. In Figure 38, the picture illustrates the interface for only one coordinate system. Another duplicate interface was required for full operation of the laser galvanometer. The 15V +/- pins on the upper right of the board are the digital power pins; this power came from the PCB. The unmarked pins on the lower right of the board take the analog input that comes from the digital-to-analog converter. The galvanometer scanner interface takes these signals and outputs the data to the laser to be drawn to the "screen."

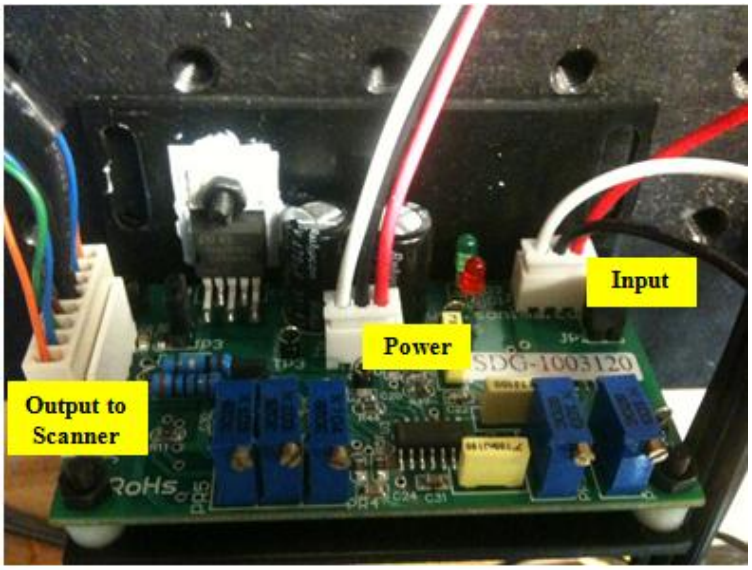


Figure 38: Interface for Single Coordinate System

3.3.4 Analog Design

The following section will detail the design process for each sub-component for the analog circuitry needed for the input and output interfaces. The design description will distinguish what circuits are needed for the design of the project.

3.3.4.1 Input Design

The design of this project required analog input from the user via the controller to enter the FPGA for digital processing. The controller was comprised of an arcade wheel spinner known as the Turbo Twist 2™ by GroovyGameGear and four push buttons. The wheel spinner creates two inputs, one for the clock and one for the direction that the wheel is currently spinning. The wheel direction and clock lines are input as two sinusoidal wave inputs that are 90 °s out of phase. The circuitry allows the two signals to first be converted into a square wave to get the circuit into more of a digital form. The resistor and the capacitor connected in parallel make the current of the capacitor 90° out of phase with the resistor. This relationship is illustrated by the equation $I_C = jwCV_{IN}$ and by the current and voltage graph in Figure 39.

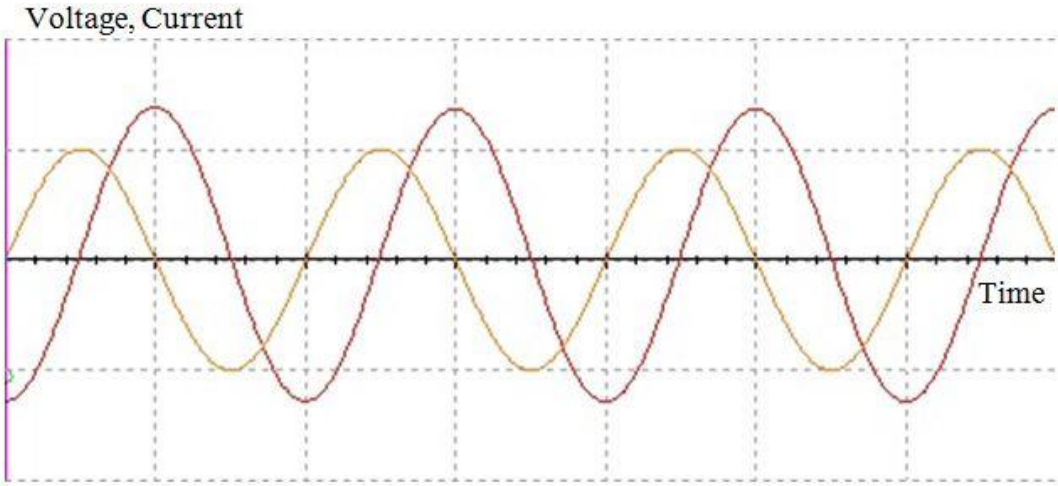


Figure 39: Current vs. Voltage Graph

The next piece of the circuitry for the input encoder wheel is the low-pass filter followed by an inverter gate. The result of these components creates a rectified square wave. The result of the parallel RC, low pass filter, and inverter gate is shown in Figure 40. Figure 40 shows the input signal as the sinusoidal wave and the output signal, which is converted into a 5V digital signal so it can be used by the FPGA I/O.

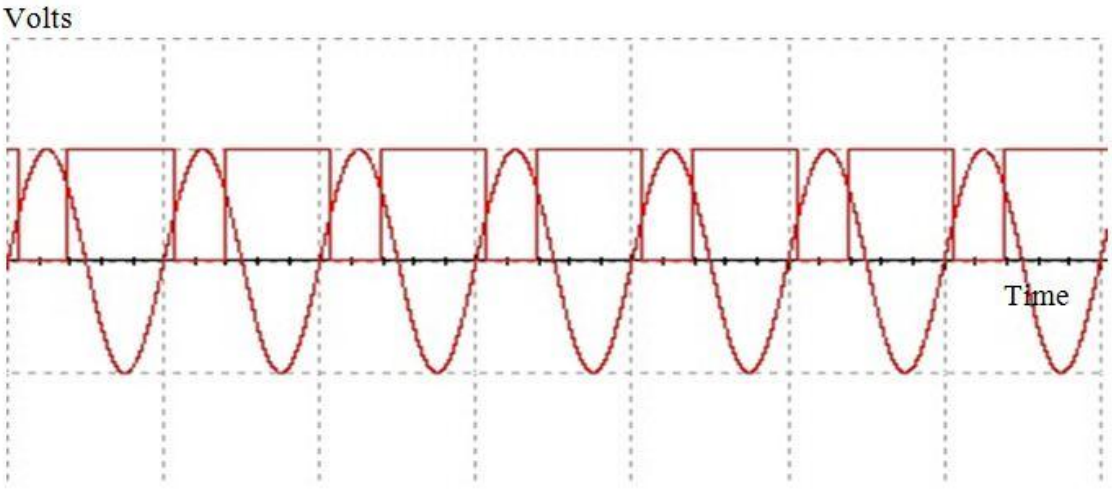


Figure 40: Parallel RC Circuit, Low Pass Filter, and Inverter Gate Result

The circuitry and results discussed and shown in Figure 40 are identical for both the direction and clock signals for the encoder spinner wheel. The actual circuitry will be shown in Section 4.1 PCB Design with resistor and capacitor values. If the spinner wheel is turned in the clockwise direction, the input is negative and it decreases the counter in the FPGA. If the spinner wheel is turned in the counter-clockwise direction the input is a zero. The clock input handles the case where the spinner wheel does not move. When the wheel is kept still the clock is a zero input voltage and does not increment the counter in either direction. Otherwise the clock is on and the player's ship will move in the appropriate direction.

Additionally, the controller input has four push buttons. The first push button is the coin button, in the original *Tempest* arcade machine the player had to insert coins in order to play the game. In our design, the player will simply be able to press a button to give a credit to play the game. The next push button is the start button; the player presses this button when they are ready to begin play. The third push button is the fire button, the player presses this button when they intend to fire on an enemy ship. The final push button is the zap button; the player presses this button in order to eliminate all enemies on the screen at once. The original *Tempest* design had ways to test the video settings by way of the diagnostic step, slam, and self-test, but our design will not need those diagnostic settings, and therefore will be tied to ground. Each of these buttons operates by sending current impulses when the micro-switches are depressed. As such each button will be connected by some resistors and capacitors in order to convert the current impulses into a sinusoidal wave signal to be converted into a square wave for digital processing. The conversion can be seen in Figure 41, the detail circuit with resistor and capacitor values will be shown in Section 4.1 PCB Design.

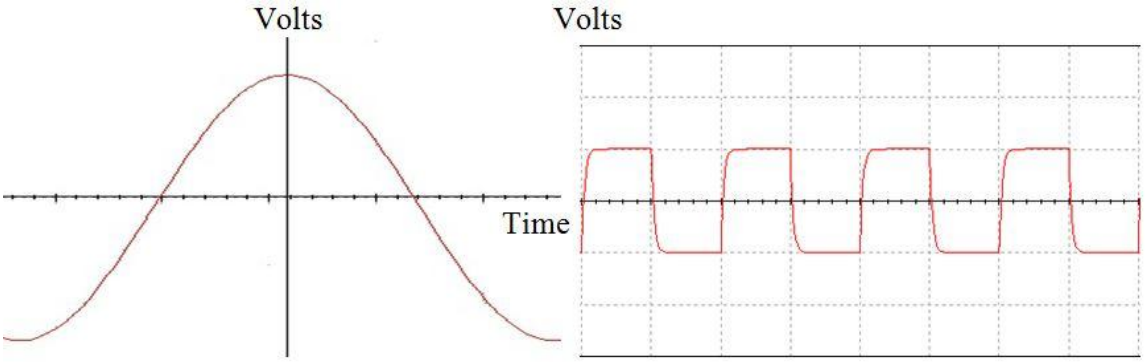


Figure 41: Conversion from Sinusoidal Wave to Square Wave

3.3.4.2 Output Design

The output schematics deal with the output audio and the video (laser) output. The laser output was controlled through two digital-to-analog converters whose input is received from the output for the FPGA. The DACs output current determine where the vector is drawn, and how long the vector is. The first DACs purpose is to handle the vector scaling; this scaling is controlled by its output current. The larger the current, the longer the vector line is to be drawn. If the first DAC does not send a reference voltage to the second DAC, no vector will be drawn to the screen. The second DACs output current determines the slope of the vector that will be drawn. As the output current becomes larger, the slope of the vector line also increases. In order to be able to draw a straight line, a capacitor must be used to charge up the voltage; otherwise the output vector would be a straight, horizontal line. An issue with using a capacitor to charge the circuit is the natural response of a capacitor gives a curved line as shown in Figure 42. As a result we needed to use an integrator circuit create a “perfect” capacitor. The response of the “perfect” capacitor generated by the integrator circuit is also shown in Figure 42.

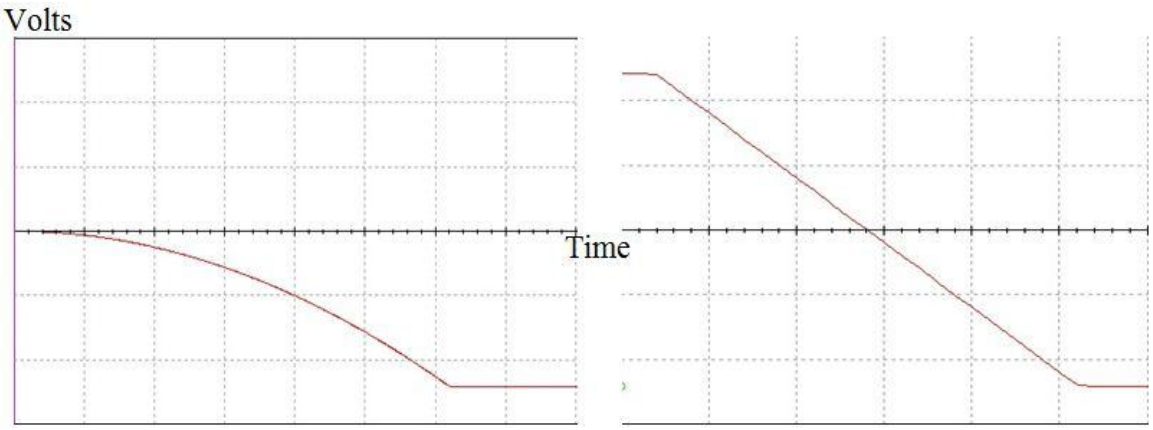


Figure 42: Effect of Integrator Circuit on Capacitor

A detailed schematic of this circuit described will be found in Section 4.1, PCB Design.

3.3.4.3 Audio Design

The audio out signals from the two POKEY chips are sent to the input of the Audio-Out Schematic located in Section 4.1, PCB Design. Their outputs are loaded into a summing operation amplifier in order to combine their frequencies. To be sent to the speakers. The output from the summing amplifier is send directly to the Audio+ line of the two and a quarter inch, 8Ω speaker. The output sent to the speaker is show in Figure 43.

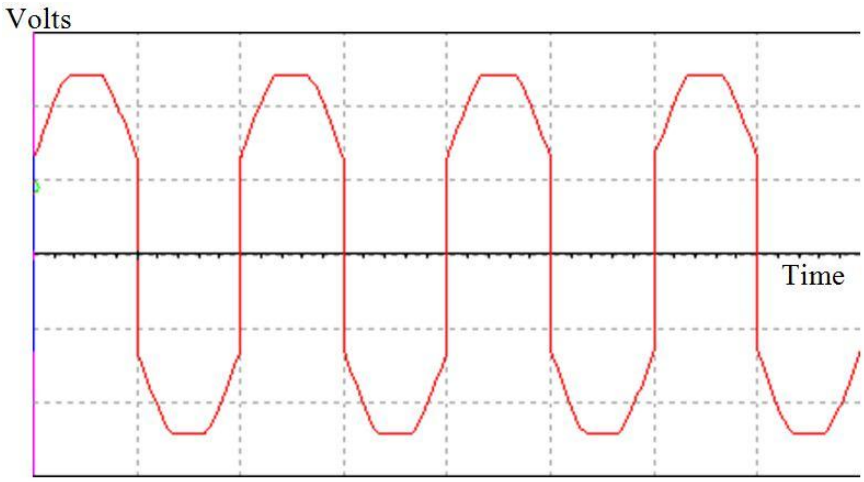


Figure 43: Output from Summing Amplifier

The signal for the Audio output is still needed to play sound frequencies from the speaker. An inverting op-amp was designed with no gain to invert the positive signal shown in Figure 43 into the inverted signal in Figure 44

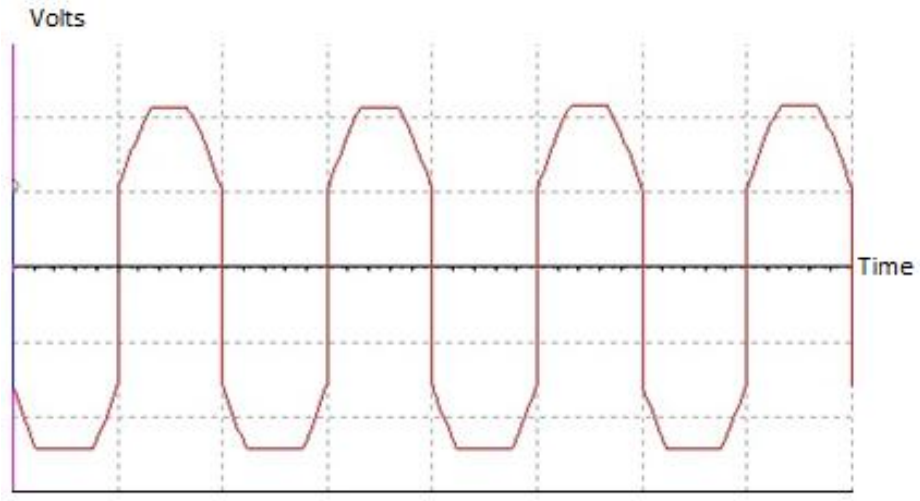


Figure 44: Inverted Output from Summing Amplifier

3.4 Budget

The projected budget for this project was at or less than 500.00USD. The budget was financed using the group's personal funds. Robert Higginbotham was responsible for making purchases which required a credit card. Once the project was completed, the total costs were divided evenly amongst the four members. Each member was expected to compensate Robert for his fair share.

A preliminary estimated budget was created after thorough research on the necessary components for the project. Table 20 lists all of the prospective components, as well as the quantity required, estimated cost of each item, and the total cost. Additionally, some costs may have been reduced if similar or equivalent components were found for cheaper prices than listed in Table 20 below.

Component	Quantity	Cost Per Unit	Estimated Total Cost	Actual Total Cost
PCM1741E – IC DAC	3	$4.19	$12.57	$28.36
LM324AM/NOPB – IC Op Amp	4	$1.22	$4.88	$10.52
Concave Button	4	$1.95	$7.80	$16.96
NPN-BJT Transistor	2	$0.419	$0.838	$0
CMOS Inverter	2	$0.279	$0.558	$0
Resistors, Capacitors, etc.	NA	NA	≈$5.00	$10.00
Arcade Spinner Wheel	1	$69.95	$69.95	$79.60
Papilio One 500K FPGA Board	1	$74.99	$74.99	$81.41
Galvo-Scanner & Laser	1	$106	$106	$0
PCB Milling Costs	NA	NA	≈$75.00	$39.31
Unforeseen Costs	NA	NA	≈$50.00	$84.08
Total			**$413.59**	**$350.24**

Table 20: Expenses

Section 4: Printed Circuit Board (PCB)

4.1 Overview

Earlier we discussed different FPGA board vendors that we had considered for out project. After researching all of our required specifications, we found that Papilio One board manufactured by Gadget Factory met all of our requirements. The most important consideration when choosing this board was the number of I/O pins available for us to interface with the FPGA. Our total number of I/O signals came out to 38, and the Papilio One board has 48 available connections with the FPGA. While the XuLA board was our first choice, it could not meet our I/O needs and instead we went with the Papilio One board.

The Papilio One board can accommodate the Xilinx Spartan 3E 100K, 250K, and 500K chip sizes. This is simply a difference in the quantity of gates and LUTs within the FPGA. It also determines the number of Block RAM modules available for internal memory devices. Since the Papilio One board does not have any external memory devices on the board, and most of our I/O pins are being used, it is necessary that all of our memory requirements for the project be satisfied within the given FPGA. The Spartan 3E 500K FPGA is the only one of the chips offered for the Papilio board which meets the memory specifications. It contains 20 Block RAMs, each of which is 2KB. Thus, the total addressable memory within the device is 40KB. The original *Tempest* hardware utilized approximately 30KB of memory. This included system RAM, game ROM, and Vector Generator RAM and ROM. All of these memory modules were instantiated within the FPGA in order to avoid the use of external memory devices. Having an external memory module on our board would have increased the number of I/O lines necessary to satisfy our design, and would have meant finding a board other than the Papilio One board.

Our external PCBs consisedt of two major parts. The first was the input PCB. This board included an interface with the arcade spinner wheel and push-buttons for all inputs. These components had to be mounted so that a user can access both the spinning wheel and buttons for game-play. The spinner wheel had its own PCB which controls the signals it releases. The output interface on this board was PS/2 and included a data and clock signal to be tied to some circuitry on out input PCB. The push-buttons were mounted with the spinner wheel and connected to circuitry on our custom board to properly manage their signals for FPGA interfacing.

The second board was the output PCB, which included digital-to-analog conversion circuitry that generates the analog X and Y voltages. These voltages can be scaled using some linear scaling digital values, which are converted to analog and relayed to the X and Y voltages. These voltages are then sent to the laser interface. The laser was powered separately and had its own PCBs which handled the control and generation of the lasers. Our output board provided the

laser PCBs with the changes in X and Y voltages, which were then used to generate a laser output.

Our PCB was manufactured by Sunstone Circuits, more commonly referred to as pcb123.com. This company offers free PCB design software, which allowed us to layout all of our specifications and needs with ease. We were able to custom-design our boards to easily attach to the Papilio One board's wings. Our boards received all necessary power from the Papilio One, which is capable of providing 5V, 3.3V and 2.5V DC voltage to external wings. Sunstone Circuits offers a variety of PCB services for small-scale projects such as ours. 2-Layer, 4-Layer, and 6-Layer board designs are all available for affordable prices. Since our design was relatively low in complexity, a 2-Layer board was sufficient. The shapes can be made custom, which will be important in order for our PCBs to connect to the Papilio One board. The Quickturn PCB service allowed our boards to be shipped very quickly, with no more than a one-week turnaround. For the given price, the many features of this service, such as dual-sided surface mounting and solder mask, unlimited number of holes, 24 drill sizes, and up to 168 in^2 sizes were all very appealing for our choice of this company.

4.2 Input PCB

As described in Section 3.2.4, the spinner controller input had two lines, a direction and clock line. The 3.3kΩ resistor in parallel with the 1µF resistor creates the 90° phase shift between the current and the voltage to allow the controller to function correctly. Next, the 1µF and 10kΩ resistor create a low-pass filter, and when that is connected to the inverter allows the phase shifted waveform to turn into a rectified square wave as a digital signal. These inputs go directly to the FPGA after some conversions, shown in the schematic in Figure 45.

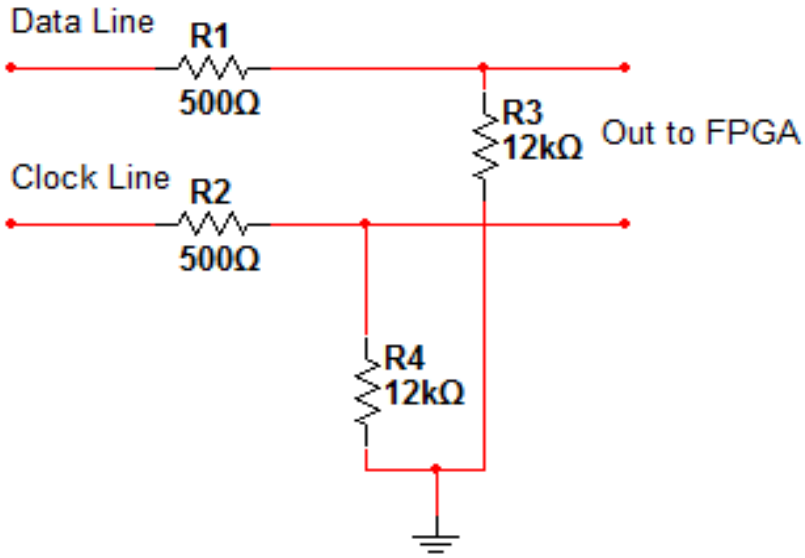


Figure 45: Spinner Controller Input Schematics

As previously discussed in Section 3.2.4, the input schematic for the controller buttons are connected to some resistors and capacitors in order to convert the current impulses into digital signals. The buttons are Start, Zap, Fire, Coin L/R/C (left, right and center). The line for the diagnostic settings within the original *Tempest* are shown and grounded as they were not used in the scope of our design project. When a button is pressed, the current impulse is sent out and goes through the 500Ω resistors and the 1kΩ resistor which converts that impulse current into 5V so it can be used by the FPGA I/O. Additionally, the capacitor and resistor create a low-pass filter to turn the analog signal into a digital signal. The coin slots left, right and center are tied together as shown in the schematic in Figure 46.

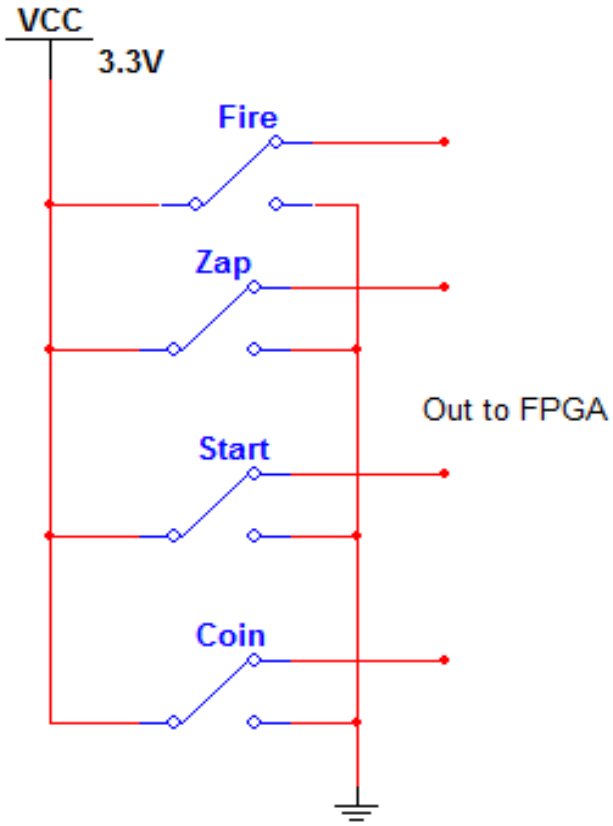


Figure 46: Input Schematic for the User Controller

4.3 Output PCB

As previously described in Section 3.3.3, the data going to the galvanometer scanner is created by using two digital-to-analog converters. The Scaling DACs I_{OUT} is connected to the VCTR control line which determines if there is a vector that needs to be drawn. As long as the VCTR line is not zero, there will be a vector drawn. The I_{OUT} of the scaling DAC also adds to the I_{OUT} of the Slope DAC and is known as the bipolar output current or bipolar output current. The bipolar output current determines where on the screen the vector should be

drawn. The output current of the Slope DAC determines the slope of the vector to be drawn. The I_{OUT} of the Slope DAC goes directly into the integrator circuit, as the capacitor charges up, a straight line vector is drawn to the screen. As the capacitor keeps some residual current as it continuously draws vectors, there is a switch controlled by the CENTER line, if the CENTER control line turns off, the latch is opened and the capacitor is completely drained of voltage. This must happen periodically for the vectors to be drawn accurate and is controlled by the FPGA. The schematic for the output data going to the laser is shown in Figure 47.

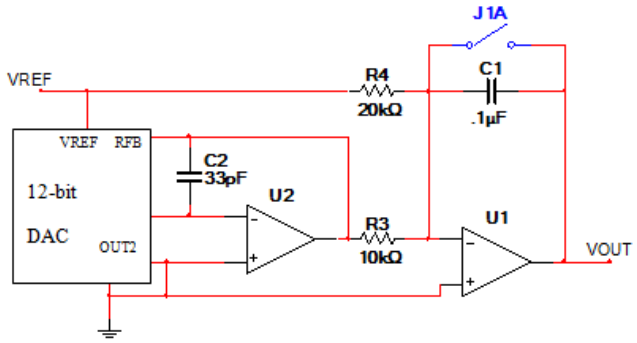


Figure 47: Data Output/Laser Input Schematic

4.3.1 Audio Output Schematic

The audio output schematic shown in Figure 48 uses the data coming from the two POKEY chips and adds them using a summing operational amplifier. In that way, the circuit can play the two frequencies generated by the POKEY chips simultaneously. The signal coming directly out of the summing amplifier is sent out to the speakers as the Audio+. The output signal Audio+ is then passed through an inverting amplifier, the inverting amplifier simply inverts the signal so it can be sent out of the system as the Audio- signal.

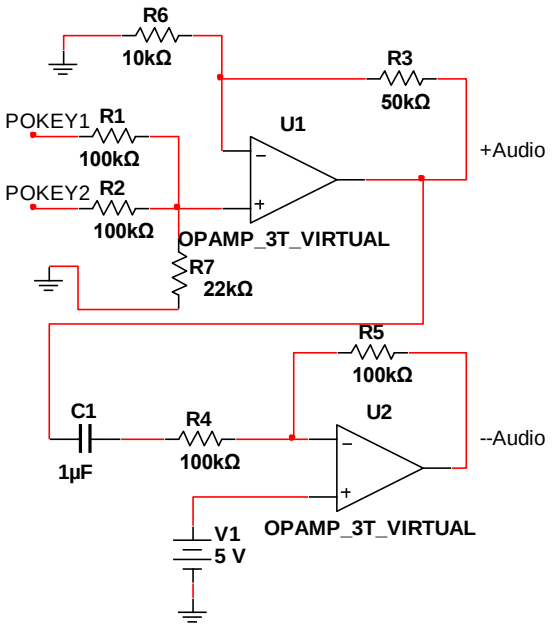


Figure 48: Audio Output Schematic

4.3.2 Overall Output Schematic

The schematic shown in Figure 49 displays the final implementation of the output circuit used in the design. The only piece missing from this design is the voltage regulator supplying the power connections for all components and the digital input going to the DACs.

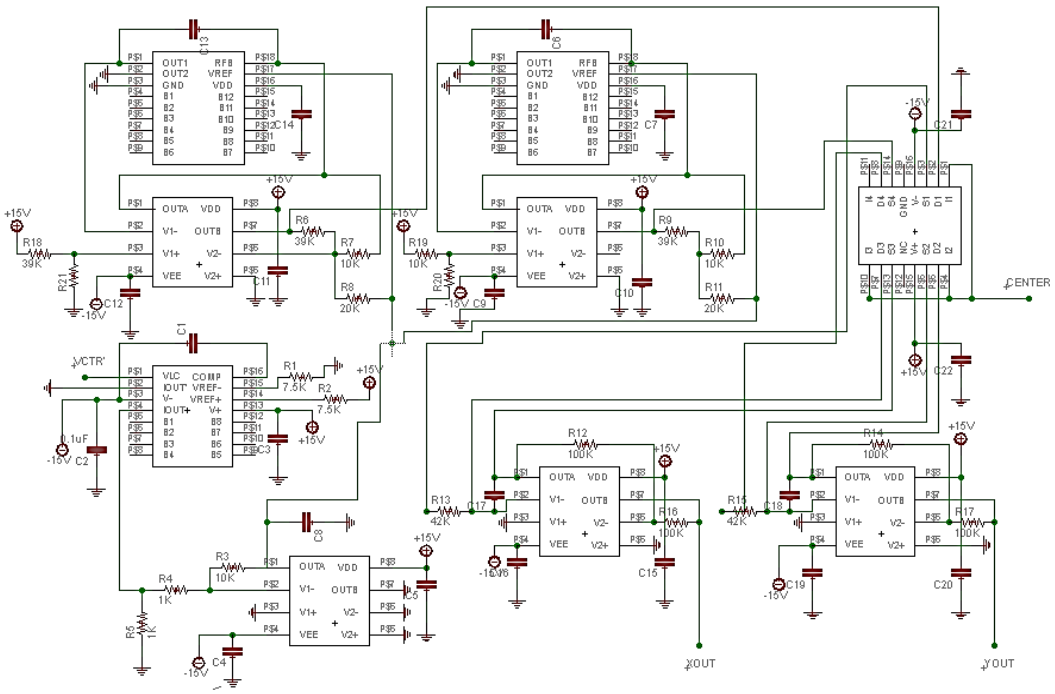


Figure 49: Overall Output Schematic

4.3.3 Final PCB Design

The final PCB was designed using PCB Artist and manufactured through 4PCB.com. The board's size is within the original specification of it being at most a 5"x5" board. For simplicity of assembly all circuit components used in the design are DIP chipsets. Due to the increased space of using all through-hole components and the amount of digital lines tied to the DACs, a four-layered PCB was needed. The final PCB design is shown in Figure 50.

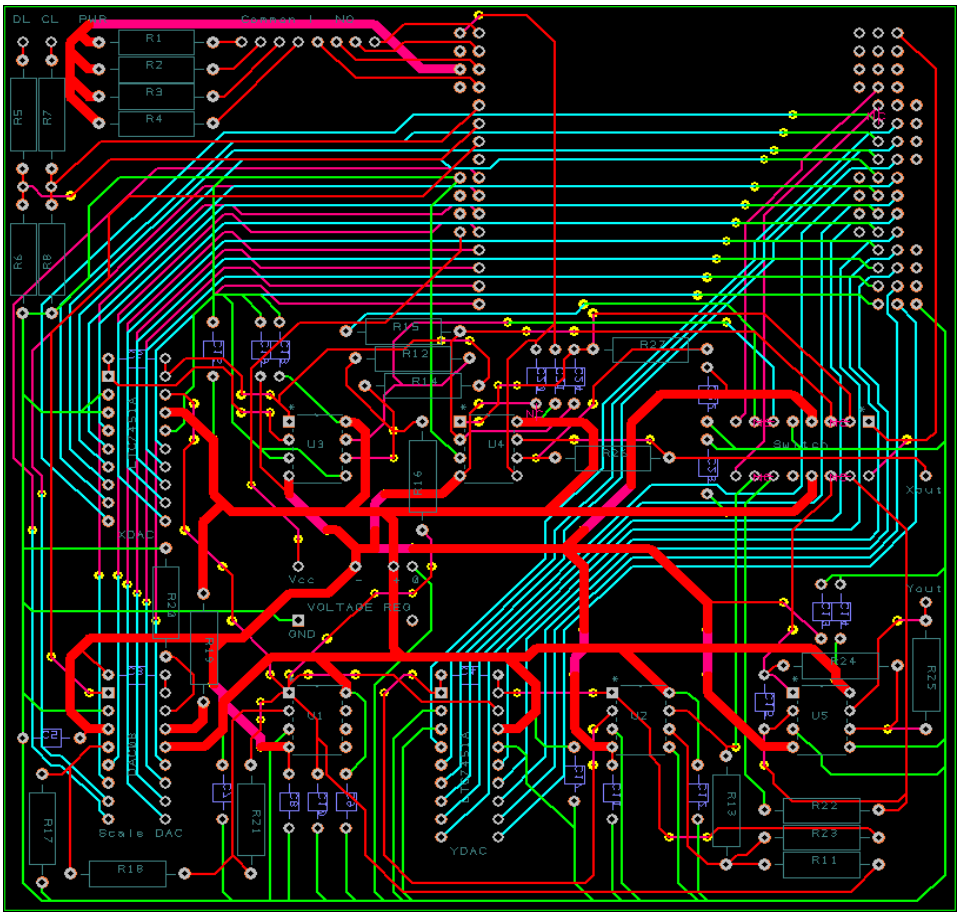


Figure 52: Final PCB Design

4.4 FPGA Interface

In order to integrate the FPGA with exterior components, some interface protocol had to be implemented. Drivers written in Verilog will latch input and output data. These lines are tied to specific pins on the Papilio board, which is then connected to the external PCB. The Papilio board was selected due to its ability to accommodate both a greater amount of I/O pins and memory. It has up to 48 I/O pins, which satisfies our requirement for 38 I/O lines. This is due to the FPGA chip set on the board, Spartan 3A, compared with the Spartan 3E on the XuLA

which only has 32 available I/O ports. Additionally, the Spartan 3A 500K gate FPGA has up to 40KB of addressable memory given 20 Block RAMs, each of which is 2KB.

The input consists of two lines from the arcade spinner wheel and three push buttons tied to ‘start’, ‘zap’ and ‘fire’ signals. The FPGA will interface these lines via the POKEY Verilog modules. The arcade spinner wheel input releases a data and clock signal. These inputs will be tied to positions C0 and C1 on the Papilio board, which corresponds to FPGA pins P91 and P92. The signals are latched on every positive edge of the spinner wheel’s clock signal. The data line controls a counter which is either increased or decreased based the direction of the wheel’s rotation. There are a total of 16 positions generated via this counter, which controls where on the game level map the user is able to shoot enemies. The start, zap, and fire signals were connected via some push-buttons circuitry. They will be mapped to the positions C2, C3, and C4 on the Papilio board, which corresponds to FPGA pins P94, P95, and P98, respectively. In addition to these inputs, there was also a coin slot input. This was a push-button input, tied to position C5 on the Papilio board, and pin P2 for the FPGA.

There were 32 output lines to be mapped to the FPGA. The Vector Generator Verilog module output the majority of these signals. It releases both an X and Y data bus. Each bus is 13-bits wide; however, the X bus only uses the 10 most significant bits of the bus. The X-Y coordinate DACs receive bits 3 through 12 of the X and Y data busses. No additional latching is needed for these busses. The lower nibble of the Y data bus also connects to some circuitry to implement linear scaling and color intensity. There are several latches needed for these components, in order to ensure that data is changed only when the Vector Generator is executing the respective instruction. VCTR and CNTR signals are also required on the output. These signals control some analog circuitry on the output. Five signals are generated from the intensity latches in the FPGA and are then translated into light intensity changes for the output. Figure 51 shows the general outline of the pin connections and their interactions with the FPGA. The pin numbers will be used in a user constraints file in order to map the FPGA I/O signals to the physical board connection points.

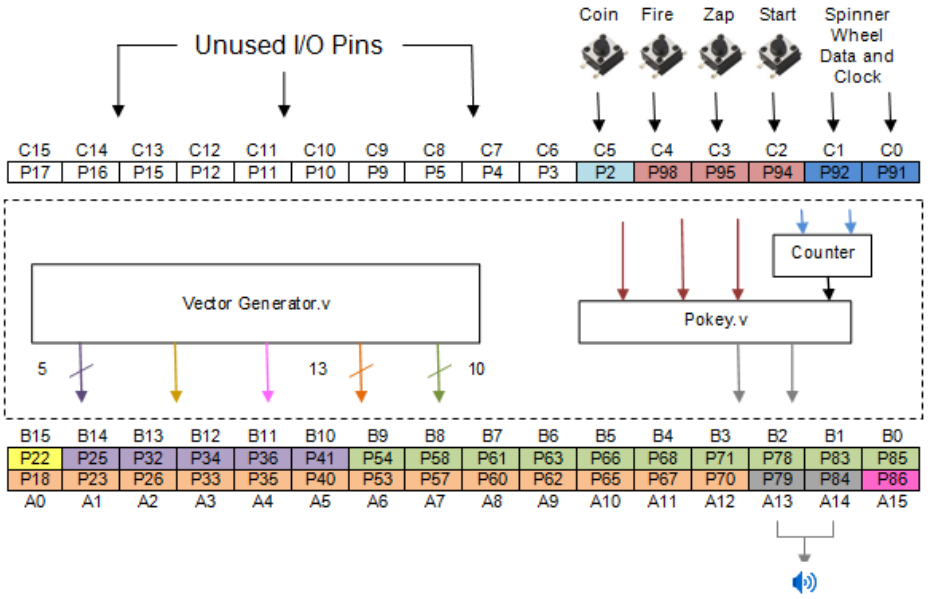


Figure 51: I/O Pin Configuration

The color key for Figure 51 is found in Table 21 and lists the I/O signals.

Color	I/O Signal
	Arcade Spinner Wheel
	Start, Zap, Fire
	Coin Slot
	Data Bus X
	Color Intensity
	VCTR
	Data Bus Y
	Audio
	CNTR

Table 21: Color Key for Figure 50

4.5 Audio

The audio signal was created by the Atari© POKEY chip which handles input from the FPGA. The POKEY chip on the FPGA board can handle 5 volts which is designed to be the output from the FPGA. The 5 volt control lines determine what happens to the frequency that is output from the POKEY chip. The POKEY chip outputs a square wave signal with a magnitude of 5 volts. After entering the PCB, the signal is converted by operational amplifiers to send an output signal from the PCB to the speaker's positive and negative terminal. The terminals take voltages from a range of 0 to 12V; therefore the voltage output from the PCB must be in that range for the speakers to be heard outputting the frequency from the POKEY chip.

4.6 Laser

Similar to the audio signals, the video conversions can take an input operating voltage as 5 volts. The voltage coming out of the FPGA output port is converted into a signal usable by the digital to analog converters. The digital to analog converters create a vector scaling signal and a vector slope signal. Once these signals are sent into the PCBs operational amplifiers, they output as two signals. The two signals are positive and a negative as the galvanometer scanner needs both a positive and negative voltage for it to function correctly. The actual laser that is used to shine at the galvanometer mirrors is a HeNe .5mW 155Asl made by Uniphase.

Section 5: Testing

5.1 Simulation

This section will discuss in detail the methods for testing the various components of the project through simulation. Simulation testing was done in the respective ISE for creating the Verilog or schematic files, which in our case was *Synopsys*. Each module below will detail each test case that needed to be performed and the expected results that will illustrate proper functionality for the module.

5.1.1 Vector Generator

5.1.1.1 Address Decoder Simulation

This simulation will test vg_addr_dec.v, vg_ram.v, and vg_rom.v. Several different scenarios should be tested to verify the design of the VG address decoder. The input addresses will include memory locations to access RAM and ROM. The ROM module will contain to actual data used in *Tempest*. The RAM will contain a few short scripts which execute different instructions. These scripts will be discussed in the complete VG simulation. This simulation will verify that the memory modules are accessed asynchronously. If VG RAM is being written to, it should update its memory on the low Φ2 cycle only. It will also check that proper selection between the VG PC and 6502 address bus is taking place based on the VMEM signal. Several random addresses will be used to ensure that the address decoder is accessing the correct memory locations based on the indexing design.

5.1.1.2 State Machine Control Simulation

This simulation will verify vg_fsm_cntrl.sch. A few scenarios will be tested within this schematic. The correct timing generated from clkgen.v will be tied to the clock inputs within this schematic. This includes a 12 MHz, 6 MHz, 3 MHz, and 1.5 MHz signals. The schematic also requires bits 2 and 3 of the current state,

and VMEM released from the 6502. During normal operation, the State Machine Control logic will not use VMEM, as it is used to reinitialize the circuitry. Therefore, this signal may be tested separately from bits 2 and 3 of the current state. In order to test the correct functionality of these bits, a fake instruction state cycle will be generated. State 9 will be the initial input. Based on the clocking of the next state, once the output state clock goes high, the next state will be available. This may or may not change the value of bits 2 and 3 of the current state. In this fashion, the state value will change on each state clock output. At the end of the instruction cycle, the total time for instruction execution may be observed. This total time should correlate to the timing released by Atari© for each instruction in the VG.

5.1.1.3 State Machine Simulation

This simulation will test vg_fsm.v. Once the State Machine Control has been verified, its outputs will be used to test the State Machine logic. The testing procedure will begin with an initialization of the state machine and LSB of the address bus (AVG0). This initialization happens when a start signal is received from the 6502. The state machine will then default to the State 9, where it will read an Op-Code. Any Op-Code should work. Based on this Op-Code, the next state will be found and decoded. Latch and strobe signals will be observed as outputs of this system. AVG0 may also invert depending on the current state. It will only change values during the four latching states, so this process will be checked. Based on correct clocking of the current state, the next state will be found. Thus, the FSM should cycle through all appropriate states of the given instruction and eventually reside in the idle state. The latch and strobe signals should be compared to the expected values shown in the VG state diagram.

5.1.1.4 Data Latch and Shifter Simulation

This simulation will test vg_data_latch.v. There are basically four latch states that must be verified. The VG data bus is tied to all of these latch/shift registers. The latches should be cleared when Latch 1 is active. If Latch 1 is active, the data bus should be latched to the Op-Code outputs and the upper 5 bits of the output Y data bus. If Latch 0 is active, the data bus should be latched to the lower 8 bits of the output Y data bus. If Latch 2 is active, the data bus should be latched to the lower 8 bits of the output X data bus. If Latch 3 is active, the data bus should be latched to color intensity output and the upper 5 bits of the output X data bus. If at any given time the NORM signal is active, all shifter registers will perform an arithmetic shift left, which affects both the output X and Y data busses. The module may be tested further by connecting its Op-Code output lines to the FSM and the output latch lines to vg_data_latch.v. By doing this, an instruction read may be simulated. This will ensure that the latch signals are operating as intended.

5.1.1.5 Stack and Program Counter Simulation

This simulation will test vg_sp_pc.v. The inputs to this system are the output Y data bus, Op-Codes, AVG0, VG start, and strobe signals. The PC should be cleared when the VG start signal is received. It will increase whenever AVG0 sends a positive edge, meaning the next address in a little endian system is needed. The Y data bus is tied to the load lines of the PC, which are latched when the instruction is JMP or RTS and Strobe 0 is active. All of these functions may be tested with individual test cases separate from the stack. The stack loads the PC at its current pointer location if the instruction JSR and Strobe 0 is active. It is enabled for reading for the RTS instruction. The Op-Codes for these instructions will be provided to verify this input and output functionality. If the instruction is JSR, the stack pointer, implemented with a simple counter, will increase when Strobe 1 is active. If the instruction is RTS, the stack pointer will decreased when Strobe 1 is active. If a reset signal is received from the 6502, the stack pointer is reset to zero. This can be tested independently from the other conditions. Once all these functions have been verified, vg_sp_pc.v may be used in a much larger simulation of RTS, JMP, and JSR instructions involving the FSM and data latches.

5.1.1.6 Vector Timer Control Simulation

This simulation will test vg_vec_timer_cntrl.sch. This schematic should be tested on an individual basis. Each instruction will output different control signals from the data latch, which are fed into this module. Therefore, different scenarios should be tested in the module to verify the functionality of each task executed by the Vector Timer Control logic. The logical operations of this schematic are fairly basic, so the output signals should easily be observed for each test case. For example, if the Op-Code corresponds to a SCAL instruction, the register which stores the binary scaling information, available from bits 8, 9, and 10 of the output Y data bus, should load said data when Strobe 2 is active. If the instruction is SVEC or VCTR, then the vector JK Flip-Flop should drive the output signal VCTR high when Strobe 3 is activated. Similarly, if the instruction is CENTER, then the center JK Flip-Flop will drive output signal CNTR high when Strobe 3 is active. The binary decrementing counter should be tested to verify that output signal SCALE goes high once the counter is empty. Output signal NORM is a function of Op-Code bit 0 is clocked when Strobe 0 becomes active. All of these conditions will have their own set of inputs customized for that particular test case.

5.1.1.7 Vector Timer Simulation

This simulation will test vg_vec_timer.sch. Vector Timer Control (VTC) logic must be simulated and verified as its output signals will be used as inputs to this schematic. The simulation will observe the counting process given a VEC, SVEC, and CENTER instruction. A 12 MHz clock operates the four 4-bit counters within

the schematic. Given the various instructions, a set of test inputs will be sent to the VTC logic in order to simulate the states of the machine and how each state affects the input to the Vector Timer. The STOP signal generated by Vector Timer will be observed given the inputs from the VTC logic. Cases that include binary scaling will also need to be considered. The NORM input should also be tested for several cases of inputs to the VTC logic.

5.1.1.8 Complete Vector Generator Simulation

This simulation will test the overall functionality of the Vector Generator. It will combine all the completed and verified sub-modules and run a set of instructions. This set of instructions will be manually written into VG RAM at the initialization of the memory modules. A few possible test scripts are described here.

To test a simple vector draw, an SVEC, VCTR, and CENTER script will be inserted into VG RAM. The Vector Generator should execute these instructions consecutively and the output busses should reflect the data within these memory locations. It will also be necessary to observe the time it takes to draw each vector. The execution time of these vector draws should correspond with the timing simulated in the FSM and Vector Timer simulations. It should also follow suit with the instruction cycle times released from Atari© when the game was originally designed. The timing of each vector draw is crucial to the proper implementation of our hardware. The goal is to make sure the 6502 and all other components are consistent with the timing of the actual *Tempest* hardware.

Another script to test is one that includes memory manipulation, such as JSR, RTS, or JMP. One script might include VCTR, JMP, and VCTR, in respective order. Therefore, the Vector Generator should draw one vector, jump to the new memory location, and draw the other vector. This will test the functionality of program counter module when connected with the other modules in the design. Similarly, a test script might include VCTR, JSR, VCTR, RTS, and VTCR, in the shown order. This will verify the operation of the stack and stack pointer. We should see a proper transition from the top routine to the subroutine draw and back again. Most of the vectors drawn on the screen are done so by using predetermined scripts stored in VG ROM. Therefore, it is crucial that the stack functions are intended to properly address memory and execute instructions in a meaningful manner.

One final test script should include all instructions within the 6502. For example, the script might include the following order of instructions: VCTR, JMP, SVEC, CENTER, JSR, SCAL, STAT, VCTR, RTS, SVEC, HALT. This test case covers all instructions and proceeds in a similar manner to that of a real instruction script. Besides ensuring that the output reflects the instructions appropriately, it may also simulate memory modification. The script may first be written into the VG RAM via the 6502. This simulation will ensure that the 6502 has access to

the VG and that the VG functions as intended after receiving information from the 6502.

5.1.2 6502 Modules

5.1.2.1 Address Decoder Simulation (T65_Address_Decoder.v)

This simulation will test T65_Address_Decoder.v. Several different scenarios should be tested to verify the design of the 6502 address decoder. The input addresses will include memory locations which access both system RAM and ROM. The system ROM will contain the *Tempest* game data. The RAM can contain any data, as it will only be used to verify it is working. The modules should be accessed asynchronously. Other input address locations should include values used to access external modules, such as Vector Generator RAM. These values will be derived from the Atari© Memory Map. For example, if we wanted to start Vector Generator operations, the address from the 6502 would be 4800h, and R/W will be set to write. With these inputs, the address decoder should activate VGGO, which is the signal used to start the VG. All cases we anticipated using were tested by using the Memory Map information. It was also necessary to verify the functionality of the data bus and its latches. Since the data bus can access the Math Box, Vector Generator, and system memory, the latching given a particular address and R/W signal was crucial to ensure proper dataflow. One final case to observe is the proper indexing of system memory. Since the original address decoder for *Tempest* had a series of smaller, asynchronous memory modules, chip select lines were used to differentiate between each memory block. Our design incorporated an indexing system which was based on these chip select lines. By doing this, the memory was stored in a single, larger block and accessed using the address and index generated from the decoder. This will be tested through simulation so that the address decoder can be used in larger simulations of other components.

5.1.2.2 ALU Simulation (T65_ALU.vhd)

This simulation will test T65_ALU.vhd. The simulation will test all of the valid Op-Codes and observe the result. The list of arithmetic and logical operations are described in the design section. Given an Op-Code, the A and B registers will be operated on to produce the desired function described by that code. It is also important to verify that for a decimal mode enable, the half-carry and carry are processed in BCD format rather than binary. The flag outputs should be updated according to the instruction operation and result. The flags will be set for various operations between two numbers. For example, if a subtraction is performed and the A and B registers are equal, the result will be zero. The Z flag should be set to one given this result. One final parameter that will be tested is the output register. In order to control dataflow, this output register is updated with a new value when Φ2 is high.

5.1.2.3 Microcode Simulation (T65_MCode.vhd)

The simulation for the T65_MCode.vhd module was done through testing of the T65.vhd module. Further testing was also done through the testing of the T65 modules after being integrated with the T65_Address_Decoder module, vector Generator modules, and Math Box modules.

5.1.2.4 Top Module Simulation (T65.vhd)

The simulation for the T65.vhd module was first done through Verilog test benches that were written to test simple functions, while also testing the interrupt handling. Once this set of testing was completed, the Program ROMs were added and the testing continued by simulating the execution of the program code upon receiving the appropriate interrupts.

The next simulation and testing to be done with the T65 module was to test the interfacing of theT65 modules with the T65_Address_Decoder module. This simulation consisted of first using a Verilog test bench to verify that the appropriate enable lines were set when a given memory location was accessed, while also placing dummy values to be replicated data being latched to and from the components accessed by the given memory location.

The final simulation was to test the T65 and T65_Address_Decoder modules interfacing with the Vector Generator and Math Box modules. The first step in this simulation was to write small programs in Assembly Language that would loop through the process of writing and reading to the appropriate locations in memory that are designated for these components. Once successful results were seen from this testing, the Program ROMs were placed into operation and the code was analyzed for its execution of the program code, based on a trace of the execution provided from another emulator.

5.1.3 Memory (ROM and RAM)

The Memory modules will be tested by first loading generated .coe files into a Verilog project. Once this is done, a small Verilog program will be written that will simulate addresses being passed to the ROMs, since all of the individual ROM blocks will be created as one large block. These addresses will have to pass through the indexing component so that the correct ROM is located. This test will be performed multiple times to check that the correct ROM sub-block is selected and has the address forwarded to that block for any given address that is passed for look up within the ROMs.

Once the indexing component is working properly, the actual ROM look ups will be tested. At first, a single address will be passed to the ROM Verilog module by the Verilog script. The script should then look up the memory location of the desired ROM and then output the data at that memory location to the simulation

screen. This data will be compared to the converted .bin files to check for consistency. After the ROM look up is working for a single address, multiple addresses will be tested in succession. Proper functionality after this expanded test will verify that the ROM component is working properly.

5.1.4 Auxiliary Board Address Decoder

The functionality of the module was essential. The Auxiliary Board Address Decoder activates the Math Box, the High Score Memory or the Player Inputs and Audio Output, depending on the specified function from the 6502 microprocessor. The Auxiliary Board Address Decoder reads in the External Address Bus and runs the top four bits through four 2-to-4 decoders. One decoder activates the POKEY which contains the sounds of the game. Another decoder activates the High Score Memory. The last decoder activates the Math Box counter and the ALUs. These addresses are specified in Section 2.5.2. To test this module, we simply wrote different addresses to the External Address Bus, and monitor which sections were activated. Since none of the predetermined address overlap, only one section will be activated at a time.

5.1.5 Math Box

The Math Box simulation tested the separate modules individually and then as a whole. First we made sure that the Math Box was only active when the 6502 had written to the Math Box's address on the External Address Bus. To do this, we wrote to every address which was set aside for the Math Box and make sure that the module is active for every address. Then, we wrote to some of the addresses which were not meant for the Math Box. During this test, the Math Box should be inactive. The addresses assigned for the different modules can be found in Section 2.5.2. Once this test had been verified, we tested Control Block 4. This Block tells the B1 latch when to forward the next address to the Math Box's counter. Since this block consisted of a simple JK Flip-Flop, we then set the device to active and various combinations of J and K. We also needed to make sure the clock input and clear input are working properly. If the clear signal was activated at any time, the output would go to zero.

Next, we checked Control Block 1. This block controls when the next instruction is read and when to advance the program counter of the Math Box. This block is made up of a D Flip-Flop, an AND gate and an inverter. First we will test for basic design of the logic. Then we tested the output with the clock attached. At no time during the test should CLK and $\overline{CLK}$ be the same value. After this module was working, we tested Control Block 2. This block contains a D Flip-Flop, and AND gate and an EXOR gate. This block was only responsible for setting the A10* bit. Finally we tested Control Block 3. This block was responsible for setting the enable line for the counters within the Math Box. Control Block 3 consists of various basic logic components such as NANDs and EXORs.

After the Control Blocks have been tested, we tested the A1 ROM, along with the counters, the E1, F1, H1, J1, K1, L1 ROMs and the B1 Flip-Flop. First we did basic testing by writing a value to the A1 ROM, clocking the counter and reading from the ROMs. We wrote simple hexadecimal values to the different ROMs, and then checked their outputs. A1 provided a proper address for all of the ROMs. The ROMs then write to their correct lines, including writing an address back to the B1 ROM. B1 then forwards this address to the counter.

Next, the bit-slice ALUs needed to be tested. They should read data from the External Data Bus. The individual 4-bit ALUs should behave as one 16-bit ALU. We tested this behavior by testing each ALU independently on simple instructions such as add, subtract, and bit-shifting. Then, we tested all four together. If the K/L2 ALU, which controls the lowest bits, had a carry, that carry should have been added into the next ALU. This forwarding of the carry bit occurred for all of the ALUs that are connected together.

Finally we needed to test the entire Math Box. To do this, we wrote data to the External Bus and wrote to one of the Math Box's addresses on the External Data Bus. The lower five bits of the address were read into the A1 ROM and read as an instruction. This instruction was passed onto the counter. When Control Block 2 sets the CLK and $\overline{PCEN}$ lines are set to high, this address was read into the Math Box ROMs. These ROMs will choose the functions of the four bit-slice ALUs. The ALUs will read data from the External Data Bus, and performed the function specified by the Math Box ROMs. This cycle continues as long as Control Block 1 does not signal that the instruction is over.

5.1.6 High Score Memory

Since the High Score Memory only needs to store high scores and display high scores, rigorous testing was not as essential for this module. To test this module, we wrote scores to the External Data Bus and specified where to store them. Then we read the memory and confirmed that the values were successfully stored.

5.1.7 Input Simulation

The input schematics were tested using Multisim 11 or similar circuit design software programs to ensure the theory behind the circuits was correct. For the input schematic, the first important piece of the puzzle is the parallel RC circuit. These circuit components served to put the current 90°s out of phase with the voltage to create an easy way to determine which direction the spinner wheel is rotating. The circuit is shown to work as expected in Figure 45. Both signals from the spinner wheel, the direction and clock signals, are then passed through the FPGA. As shown in Figure 52, when the spinner is turned in the clockwise direction, the data line (yellow line) is on when the clock line (blue line) is on.

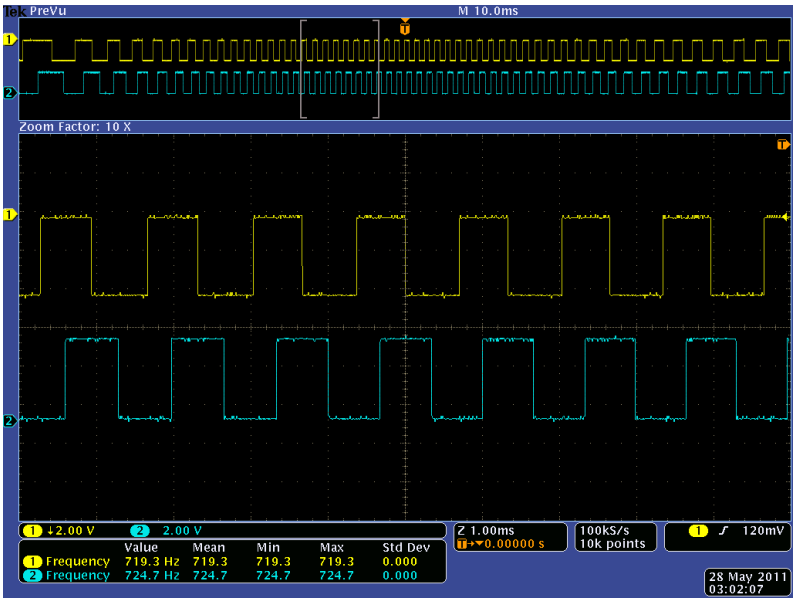


Figure 52: Proper Output for Clockwise Spin

The next element of the circuit that both the direction and clock pass through is a low-pass filter connected with an inverter gate. This part of the circuit serves to turn the input voltage from an analog signal into a digital signal that the FPGA can use. The voltage input and output ports on the FPGA are limited to 5 volts DC, so the circuit must also output that 5 volt requirement. The simulated circuit response is shown in Figure 53 to output the expected result. In this figure, the input analog voltage and the DC conversion are shown on the same graph.

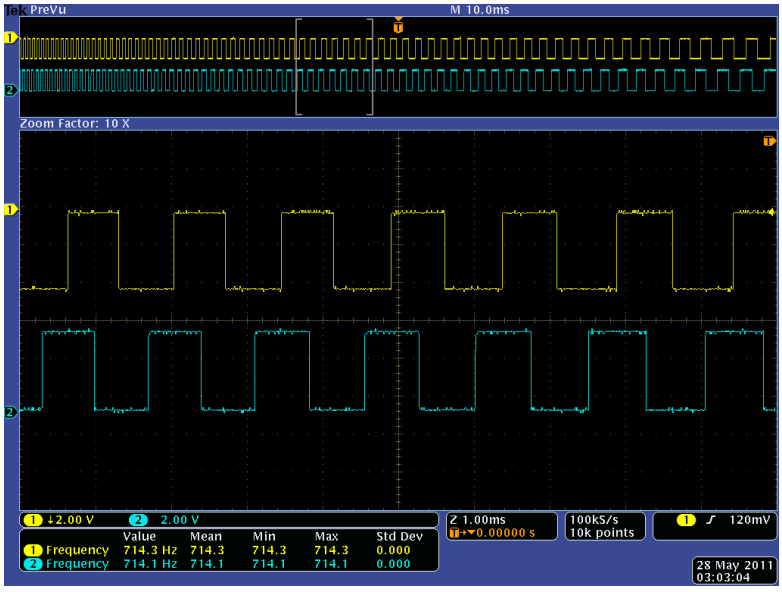


Figure 53: Proper Output for Counterclockwise Spin

The other important feature of the controller is the buttons. When the buttons are pressed, the micro-switch sends impulses of current into the system; these

impulses must also be converted into DC signals for processing. The individual circuits for the buttons were designed so that the current impulses could be converted into a voltage the FPGA can utilize.

5.1.8 Output Laser Simulation

The output video laser schematic could only be partially tested using Multisim 11. Multisim is limited to using analog devices and therefore we could only test some of the circuit theory behind the output schematic. The output from the FPGA enter directly into the digital-to-analog converters, the DACs take the data from the FPGA and decide how long each vector should be and the slope of the vector to be drawn. The output currents from the DACs are sent through an integrating operational amplifier for the vector to be drawn. By using an integrator, a straight line can be drawn. Without the integrator, the capacitor has a normal exponential curve that was not conducive to our application. The difference between the vector drawing of the capacitor versus the integrator circuit is shown in Figure 54.

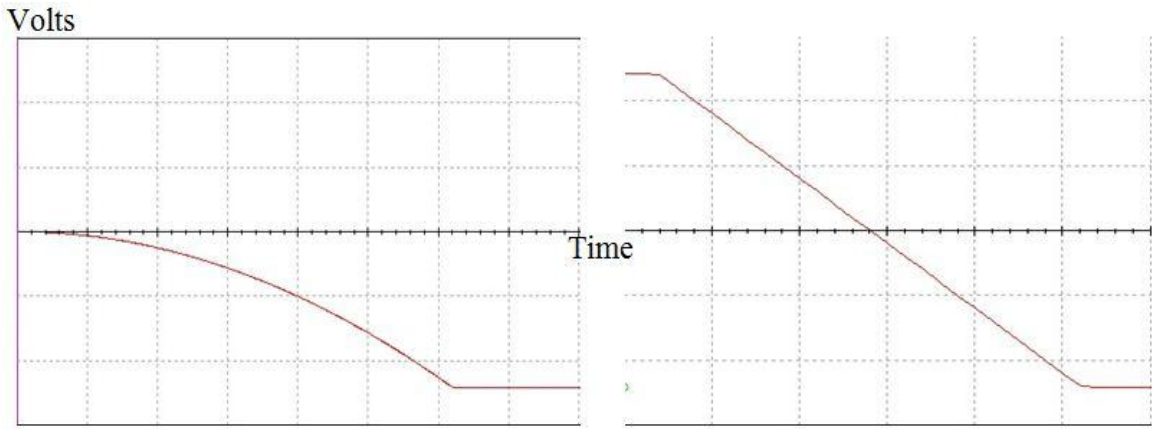


Figure 54: Vector Drawing with Capacitor vs. Integrator Circuit

5.1.9 Output Audio Simulation

As with the output laser schematic, the output audio schematic was partially tested using Multisim11. Multisim is limited in that not many digital components can be used for simulations; as a result some circuit concepts were tested, but not fully guaranteed. However, the input into the system was simulated as the output audio signal is a square wave analog signal. This analog signal output by both the POKEY1 chip and the POKEY2 chip and added together by using a summing amplifier. The output of this amplifier is sent directly to the positive terminal of the speaker present in the system. The output of the summing amplifier is also sent to an inverting operational amplifier (Op-Amp) to invert the signal so it can be used on the negative terminal of the audio device. The positive signal output to the speaker can be seen in Figure 55.

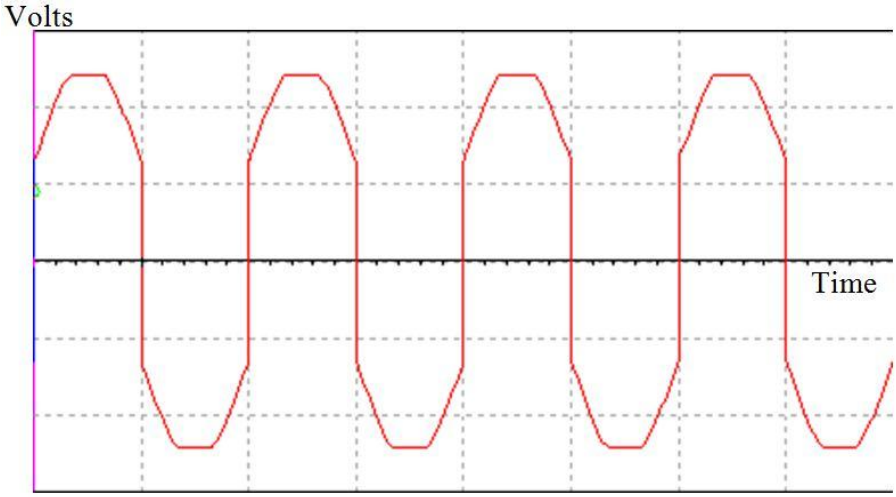


Figure 55: Positive Output Signal for Speaker

5.2 FPGA Synthesis and Testing

This section specifies the test cases for the various components of the system after being synthesized to the FPGA or implemented on the PCB. Each section will provide all information about the test cases to be executed and the results that are expected.

5.2.1 Vector Generator Synthesis

Once all modules in the Vector Generator Verilog design were tested and verified, the design was synthesized to the FPGA for individual testing. We tested various input scenarios and observe the output pins on the oscilloscope. This allowed us to execute small scripts within the Vector Generator and verify that the design was working as intended on a real FPGA chip.

Inputs included a push button tied to the VG start and reset signals. We initialized the machine by resetting the internal registers of the VG module and starting operation. Normally, these signals will come from the 6502. However, for the sake of testing the Vector Generator as a single entity, the input signals were manual inputs. The initial address was set to position 0 when the VG receives the reset signal. From there, the start signal will be sent and the script will begin executing at the first location in VG RAM. Several scripts were used to test all the sub-modules of the VG. The same scripts as described in simulation were used in synthesis testing so that a consistency from simulated results to actual results can be observed.

It was difficult to observe all of the output data busses at once, since the data released from the FPGA was digital information. A simple DAC circuit will be used to convert the digital output vectors to an analog voltage. Based upon this

voltage, we were able to verify the output vectors that were generated from the VG.

Some additional tests were performed to check timing within the VG. The Finite State Machine (FSM) has specific timing that corresponded to a particular instruction. For this test, each individual instruction was executed, one at time. The state values for every instruction were observed, as well as the time that it takes for each state to change. The FSM should change based on the State Machine Control logic, which was observed on the output. The total execution time corresponded to the Atari© times released with their original hardware. More importantly, the timings matched the simulation times for a given instruction.

Once the Vector Generator was tested and verified as a single component, we connected it to the output PCB to observe its interaction with that part of our design. A short script was written to produce some output voltages which correspond to changes in X and Y vectors. This not only verified the proper functionality of the VG but also the actual conversion circuitry for the VG's output data busses.

It was also important to test the scaling output. This test was for both binary and linear scaling outputs. The binary output affects the amount of time output signal VCTR is high. Given a particular binary scaling value, we observed VCTR and made sure it was high for the appropriate amount of time. Linear scaling information is clocked into an output register and sent to a separate DAC. This DAC converted the linear scale to an analog voltage, which was sent to the X and Y vector DACs. The output PCB circuitry was used to verify the correct conversion of the linear scale and how it influenced the X and Y output voltages. Scaling is used heavily and was a very crucial process to verify.

The final output to test was the color intensity information. For several instructions, the color intensity values Z2, Z1, and Z0 were sent to the output pins. To test this function, a single instruction was executed in the VG. The color intensity information was then available on the respective output pins. Once this process was tested and verified, we were able to use these output pins to test the circuitry which generates color and intensity. Again, this circuitry was on the output PCB and verified that the VG was working as desired.

5.2.2 Math Box Synthesis

After the simulation of the components for the Math Box was completed, the entire Math Box module was synthesized to the FPGA. This allowed for physical testing of the Math Box. Inputs for the Math Box will be tied to a series of push buttons. Based on the configuration of these push buttons, an address will be passed into the Math Box as normally is done from the External Address Bus. This caused the cycle of decoding the address to begin. Followed by respective ROMs, and forwarding the appropriate data to the ALUs.

Throughout this sequence, multiple LEDs and 7-segment displays were hooked up at various test points. These test points served as a way to show that data was being passed to the correct components. For example, eight LEDs will be connected to the output of the A1 chip as mentioned in Section 2.6. These LEDs should turn on when an output line goes high, which the decoded address may then be checked. This type of testing was repeated multiple times by loading synthesizing the modules to the FPGA multiple times. For each test, different output lines were connected to the LEDs and 7-segmen displays each time. This allowed the group to check each component individually and ensure that it was functioning properly.

The final synthesized test for the Math Box was to synthesize the system to the FPGA board with a set of LEDs or 7-segment displays connected to the input lines from the External Address Bus, while also connecting the a set connected to the External Data Bus. Once this was done, we were able to see the address and data that had initially passed into the Math Box. Following its computations, we were able to monitor the final output of the Math Box to the External Data Bus. This test case was performed with the collaboration of individual test case previously mentioned. Since the test results of this test case match that of the individual test cases, as well as the theoretical results calculated from the converted .bin files, it illustrated that the Math Box was functioning properly.

5.2.3 6502 Microprocessor Synthesis

After the 6502 design has been tested through simulation, we synthesized the design to an FPGA. At this point in time, we did not have some of the other components verified, and we need to confirm our implementation separately. This presented a small obstacle, since we needed to create our own software to run on the 6502 that was rigorous enough to test the full capabilities and complexity of the 6502. This software also needed to be simple enough for our group to confidently predict the results. First, we created a simple script which added two registers together. We then wrote these two register values through the built-in switches on the FPGA. Then, we clocked the 6502 just once. We then connected the output of the 6502 to the FPGA's built-in LEDs. Since this simple exercise only tested the decoding logic and the ALU, we needed to move on to more advanced testing afterward.

Next we tried a script that contained multiple lines of code. For example, we may have the 6502 read a value from memory and write that value to a register. We will then tell the 6502 to add that register to another register. The 6502 will then store that result in memory. That result will also be displayed on the FPGA's LEDs or on a set of 7-segment displays. This tested the ALU and decode logic again, as well as also testing the program counter and the memory addressing.

Some of the major components of the 6502 that still needed to be tested were the Stack Pointer, the interrupts, the Program Counter (PC), and the Timing Generation Logic. One way to test the Stack Pointer was to create a function call or sub-routine within our testing script. Then, we were able to try and write to a register that was still in use in the main function of the script. When the Stack Pointer behaves properly, the sub-routine call should perform without overwriting the original value of the register which the main function is using. This also tested the Program Counter's behavior when a function call was made. When the function call is made, the old PC was saved, and the new PC was read into the PC register. Upon completion of the sub-routine, the previous PC should return to the PC register. One difficulty was tracking all of this behavior. Since the FPGA has only a limited number of LEDs, we were not be able to track every memory address, the value of the Program Counter and the value of the registers on which we were operating. However, if the 6502 created the correct output, we knew that the function had been handled properly. If our tests had not yielded the desired result, we would have had to revert back to simulation so we could track all of the inner-workings more closely.

Once we were confident that the 6502 module had passed these tests, we were ready to allow the 6502 to communicate with the other modules of our project. By this phase, the 6502 was able to address the different modules of *Tempest* by writing to their respective memory addresses via the External Address Bus. When a module was accessed, the 6502 also output data to the External Data Bus. The 6502 then processed this data and pass control to other modules as necessary. By this phase in integration, we had tested each component individually. Since there were no other errors that manifested themselves during this phase in testing, we assumed that those errors are coming from connections to each module, not within the modules themselves.

5.2.4 Input Testing

In order to test the prototype for the design project, the individual pieces of the circuit had to be tested to ensure proper performance starting with the game controller. The output signal pins of the TurboTwist 2™ had to be located on the PS/2 interface by the use of an analog meter. Each of the two outputs appeared on the meter as two square wave signals. The two input signals needed to be identified as the clock and direction. The clock signal always output data one way, while the direction signal alters its phase direction when the encoder wheel is rotated in the opposite direction. The spinner wheel should be turned counterclockwise and then rotated in the opposite direction to ensure the output signals invert when the directions are changed. The proper output of the controller should be verified. The signals should be 5V sinusoidal waves. Once the two output signals were located and identified, the output from the spinner wheel had to be hooked up to the circuit in Section 4.1.2 by use of a breadboard. The parallel RC circuit of the input design schematic was tested by the use of an oscilloscope. This output appeared as a 5V sine wave with the current 90 °s out

of phase with the voltage signal. The whole circuit should then be tested by once again using the oscilloscope. The output signal for the input schematic should be a DC voltage of 5V for the POKEY chip input port.

The buttons of the game controller were also tested and compared individually to ensure proper functioning of the micro-switches in the button. Each button should output an impulse current of 2mA. As with the arcade spinner wheel, the buttons had to be connected to the circuit in Section 4.1.2 to ensure the circuit worked as it was expected to during simulations. The output DC signal of the input schematic goes directly into the POKEY chip for processing. For the testing, the processing though the POKEY chip must be confirmed as the data out from the POKEY goes directly into the FPGA input port. Once all of the buttons and the spinner wheel were tested and proven to work properly, the next components of the design schematic could be tested.

5.2.5 Output Laser Testing

The circuitry in Section 4.1.3 for the output to the laser was the most difficult component to test. Each digital-to-analog converter (DAC) was tested first to see if they respond as expected. I_{OUT} of each of the DAC chips should read around 4mA for an I_{REF} input of1mA. The output signal must be measured with an oscilloscope, and should be near a straight line with a positive slope. Multiple vectors for different cases of length and slope should be drawn to the oscilloscope. It should be determined how long the vectors can be drawn before their outputs become distorted by the capacitor's storage of charge. This information was vital in knowing how long the Vector Generator can draw vectors before the capacitor. Once the vectors become distorted, the switch on the integrating amplifier should be opened to discharge the capacitor. After it is determined the output of the circuit is in a voltage range usable by the laser galvanometer further testing can commence.

The laser galvanometer was tested by first giving the driver the proper +/- 15 volts signal to its terminals. Also, the power supply for the laser must be powered with 100 volts AC before anything was drawn to a surface. Test points for +/- X and Y coordinates should be tested before vectors are drawn with the galvanometer-scanner. Proper output of the laser was verified visually before continuing with the testing. The laser was tested by applying different DC voltages to its X and Y coordinate scanner inputs. The appropriate voltage was applied to the inputs to make the laser draw shapes it will encounter during the course of normal operation while running *Tempest*. In Table 23, the testing input DC voltage values are picked in order to draw two squares of different sizes. If these voltages can be interchanged fast enough, two concentric squares will be observed.

+X	-X	+Y	-Y
1V	-1V	1V	-1V
5V	-5V	5V	-5V

Table 23: Testing Laser via Input Voltage Values for Square Drawings

As shown in Table 24, testing DC voltages are chosen to draw two triangles of different size. In normal operation of the game, there is always a triangular shape drawn to mark the position of the player's ship at any given time. Drawing the triangular shape is the most important shape the laser galvanometer-scanner will have to draw.

+X	-X	+Y	-Y
2V	-2V	2V	0V
2V	-2V	5V	0V

Table 24: Testing Laser via Input Voltage Values for Triangular Drawings

Once the circuit and the laser have been verified as working correctly, the output from the circuit is sent to the laser galvanometer. Multiple vectors of different lengths and slopes should be tested through the laser by sending the appropriate signals from the FPGA. A testing program within the FPGA was designed in order to verify proper interfacing between the DACs and the digital outputs from the FPGA. This test program required the laser to draw various shapes that it would be required to draw in normal operation conditions. Similar to the testing above, voltages were sent to the laser. However, by using the FPGA, signals were sent to the laser much faster, thereby allowing the testing to be more robust. Several different shapes can be drawn by the laser at the same time. Instead of having to test by drawing a triangular output or a square output separately, both shapes are drawn simultaneously. The FPGA test program will also test the refresh rate of the laser drawing on the wall (screen). The limits of the laser's speed should be determined by making the laser draw increasingly more shapes at a time. The laser should be able to draw at least 15 different shapes at the same time in order for it to work properly for the game. Sample input voltages for the FPGA test program are shown in Table 25 below.

+X	-X	+Y	-Y
1V	-1V	1V	-1V
5V	-5V	5V	-5V
2V	-2V	2V	0V
2V	-2V	5V	0V

Table 25: Testing Laser via Input Voltage Values for Square and Triangular Simultaneous Drawings

The input values in the Table 25 should properly display two concentric squares and two triangles of different heights. The testing table of voltages will grow until it appears visually that the scanner cannot keep drawing the shapes without the

image appearing to blink. This blinking indicates that the scanner cannot redraw the image fast enough before the image disappears from the screen (the surface the laser is drawing on). If all of these shapes are shown properly and the threshold for the number of shapes drawn to the screen is determined, the group can conclude that the laser and the Output schematic work correctly and testing on the next system can commence.

A static image taken from an emulation of the original *Tempest* game is shown in Figure 56. The image was replicated from the original game-play using our project's hardware and vector generation code. This output is displayed on the oscilloscope. The noise in this picture is partially due to some error in our circuit design by relatively inaccurate components and a higher persistence in order to be able to get a readable output.

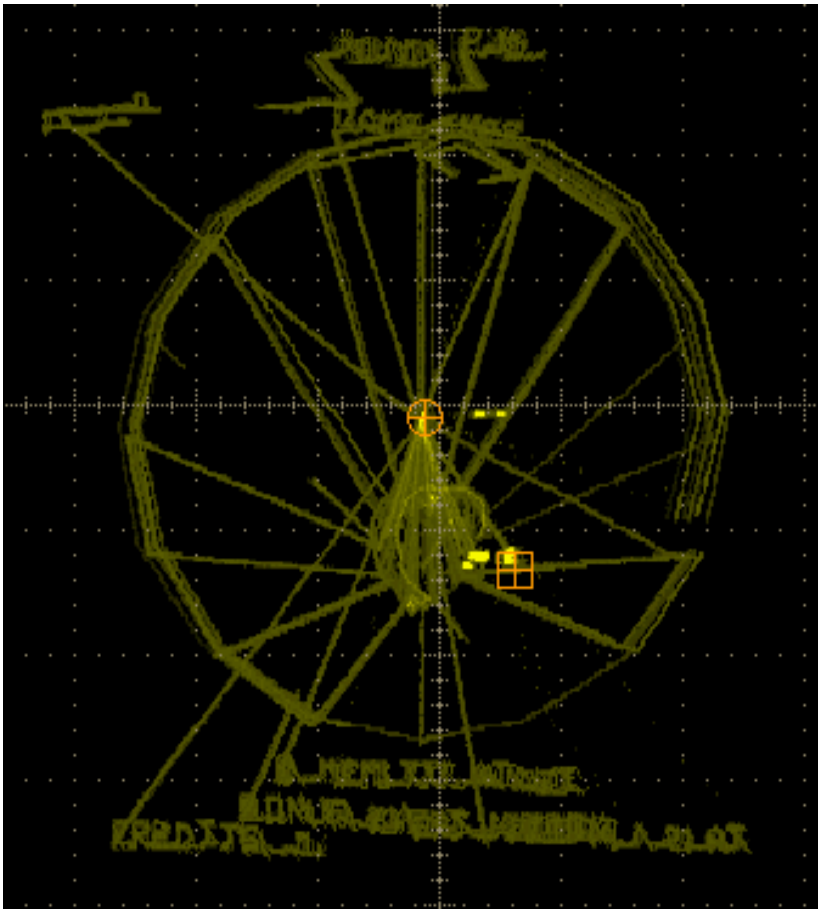


Figure 56: Vector Generator Testing on Oscilloscope

5.2.7 Overall System Testing

Once all of the individual component testing has been completed and each component has passed its respective tests, the final test of the prototype was to test the entire system with all of the components connected together. This meant testing with the user controller connected to the FPGA, which was then

connected to the output via the laser. For testing the system, there are two test cases that were performed.

The first test case was to power on the system and allows the user to attempt gameplay. Attempting gameplay tests multiple components at once. To begin, the user uses the buttons on the FPGA to replicate that action of putting in coins to obtain game credits. If working properly, the displayed number of credits remaining increases as the user presses the respective button. If the user selects a different button other than the coin input button, then the number of remaining credits remains unchanged.

The user is now able test the controller by navigating through the main menu of the game. In the main menu, the user is able to view past high scores, select a level to play, and also select the difficulty. The user navigates each of these, one at a time and in any given order. A successful navigation of each of these options within the main menu signifies that the 6502 is handling the main menu instructions properly, while it is verified that the user controller is communicating properly with the POKEY chips within the FPGA.

The user next tests the actual game play of the system. Once the user has selected there desired level of play and/or level of difficulty, the game begins execution. Game play will continue to test the user controller by reporting the firing of the weapons at the enemies (push buttons), while also testing the navigation of the user's spaceship (arcade spinner wheel). These inputs will then drive the 6502 microprocessor. This will continue testing on the 6502 by forcing it to perform the instructions needed to replicate the user's actions on the screen. In order to replicate these actions, the 6502 will also have to communicate with the Vector Generator and Math Box, again testing more aspects of the 6502. As the 6502 dictates to the Vector Generator and Math Box what instructions need to be performed, each of these modules undergo its own physical testing by carrying out the instructions. Proper results from each of these two modules should result in the appearance of advancing enemies on the screen, rotation of the playing field, and even the enlargement of the enemies' size as they approach the user's spaceship. Through this point, successful results indicate that the system is working properly. This indication is maintained under the assumption that the laser is displaying the output to the screen properly and performing the actions of game play as the user has performed with the controller.

At this point, the first test case has reached its conclusion. An addition test case that was performed was the built in test program for the game. When Atari© created the game, they built in a feature to the program that allows it to test itself. This feature was called test mode. The test mode has a series of instructions that it executes as a small script. Portions of the program are used to specifically test the 6502 microprocessor, Math Box, Vector Generator, and also the output to the screen. This tool was utilized after multiple rounds of testing from physical

game play. It was performed multiple times itself so that we could test the components in the most robust way possible. Successful completion of this test case provided a high level of confidence in claiming that the components of the system worked properly as individual components and also as a collaborative system.

Both of these testing cases for the game were performed multiple times by all members of the group and also by people with no connection to the project. The use of people who had no connection to the project allowed the group to gain feedback about the ease of use for the user controller. They also provided an opinion about how accurately the game play resembles that of the original game. All of the feedback from anyone testing the system helped the group to discover features that could be added or modified to improve the system as a whole. All of this robust testing allowed the group to experience the minor quirks of the system that many people call “cheats”. Some of these cheats include the user getting 40 game play credits. This cheat and many more were obtained various ways such as getting your score in a certain format.

Following these test cases, the prototype was fully functional and ready for presentation. Additional features may have been added at a later time following the presentation. This would require the group to repeat these test cases in order to make sure that the original system is compatible with the newly added features and that they collectively perform in the manner desired.

Section 6: Prototyping

6.1 Output Testing

In Figure 57, the final sample of the project’s output is shown. The four stages of crosses represent the different layers of what would be seen on an actual game screen. With this output drawing with the laser, we were able to implement the use of both the arcade spinner wheel and a button press. The spinner wheel controlled the horizontal line that is pointing to the left, and is able to be spun both in the clockwise and counter clockwise directions. The button was used to simply turn off the galvanometer mirrors to cease drawing to the screen.

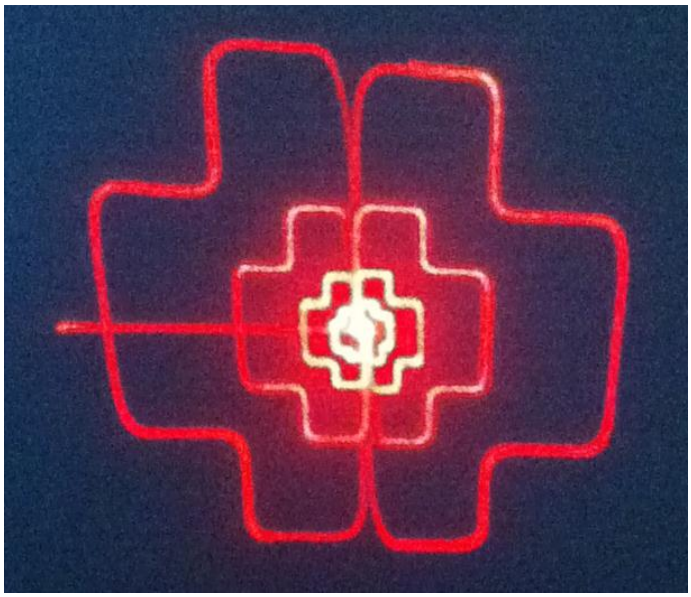


Figure 57: Testing Output from Galvanometer and Laser

6.2 Final Output Schematic Design

In the final output schematic design, both the X and Y coordinate outputs and their related circuitry are shown. In Figure 58, the 8-bit scaling DAC is connected to an op amp that controls a V_{REF} voltage that changes depending on how long the vector is to be drawn. This V_{REF} voltage is sent to the X and Y slope DAC which output +/- 5V signals that are sent directly to the galvanometer-scanner. The analogue switch is also shown in the schematic, the switch allowed greater control over some error that was produced within the circuit by being digitally controlled. The other, more important function performed by the switch is to clear the residual voltages on the capacitor over the integrating op amp.

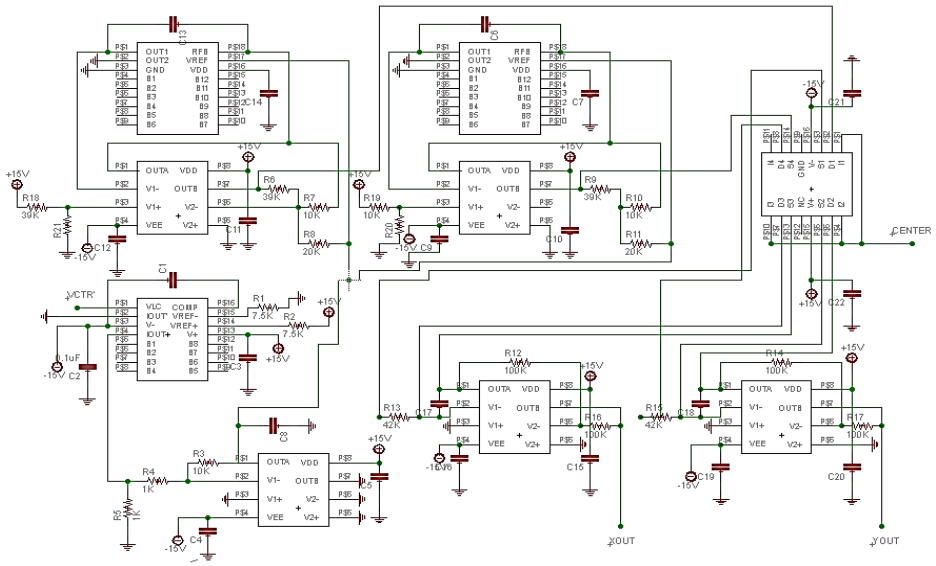


Figure 58: Final Output PCB Design Schematic

6.3 Final PCB Product

Figure 59 shows the final PCB product and attached FPGA board. The FPGA board was connected via two female-to-male headers that were soldered directly to the PCB and the FPGA. This set up allowed for easy removal of the FPGA for safe transportation. There was a modification that needed to be performed post PCB manufacturing that fixed an input offset residual voltage on the positive in terminal to the first op amp connected to the X and Y slope DAC. This modification was done by carefully soldering connections to the op amp and breaking the connection out to an auxiliary board to solder extra resistance values.

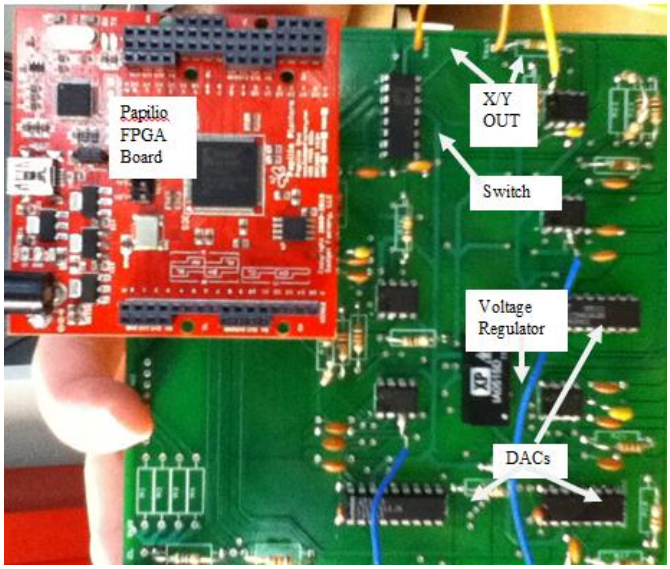


Figure 59: Final PCB Product

6.4 Final Prototype

The final prototype is shown in the Figure 60. The separate components were mounted on an optical board to help stabilize the galvanometer marked on the upper left portion of the picture. The laser and galvanometer were aligned such that the laser points to the X axis of the galvanometer and would shine up, perpendicular to the board. The figure also shows the connections of the X and Y interface for the galvanometer, power supply for the galvanometer, FPGA board, PCB, and auxiliary board.

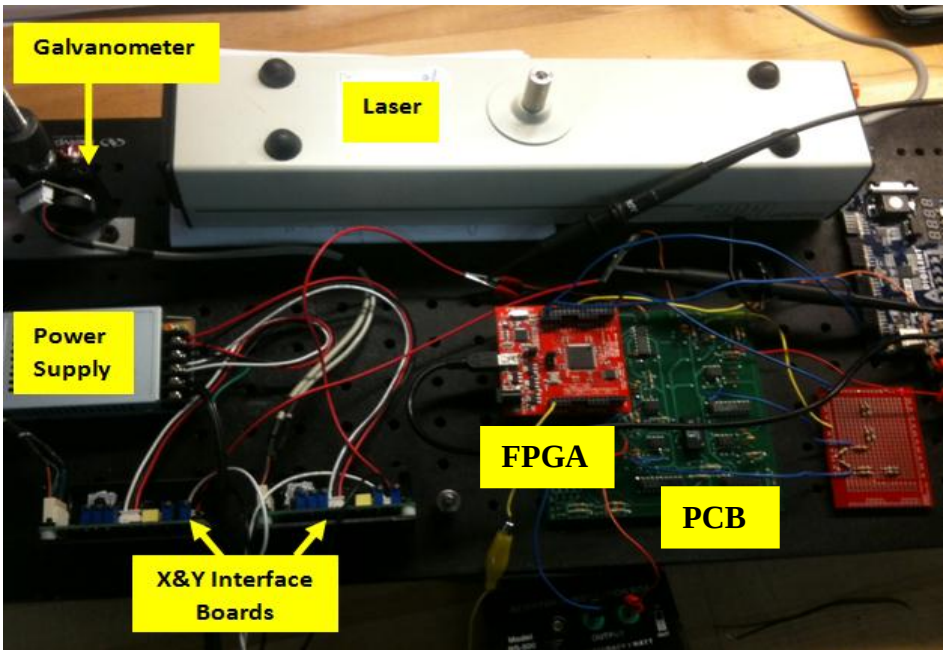


Figure 60: Final Prototype

Section 7: Operation Manual

7.1 Software Setup:

The first step in setting up the software is to create a new project in the user's desired HDL navigator. This process will step through the process with the use of Xilinx ISE 13.1. Once the user has Xilinx open, they should create a new project titled *TEMPEST*. This folder that is created will contain all of the project files needed for the emulator.

With the new project created, the user is now ready to begin adding in the source files. The first source file to be added should be *tempest_top.v*. This file is the top file that packs all of the sub-modules together and allows the sub-modules to communicate with each other. Once this file has been successfully added, the user should now add in *clk_gen_top.v*, *Schematic_3b.v, vg_top.v, T65.vhd,* and *T65_Address_Decoder.vhd*. In addition to these files, the am2901 folder should also be added as a sub-folder to the *Tempest* project folder. The user should now progress by adding the project files representing the sub-modules for each of the modules previously added.

The first module to add sub-module project files for is the T65 module. This module contains three files that need to be added, which are *T65_Pack.vhd, T65_ALU.vhd,* and *T65_MCode.vhd*. Once these files are added, the user should move to adding the sub-module project files for *vg_top.v* and *Schematic_3b.v*. The sub-modules for these modules can be seen in Table 26.

Top Module	Sub-Modules To Be Added
vg_top.v	data_out.v vg_state.v vg_fsm_cntrl.v vg_data_shifter.v vg_sp_pc.v vg_vec_timer_cntrl.v vg_vec_time.v vg_addr_dec_sim.v
Schematic_3b.v	MB_Address_Decode.v POKEY_TOP.v asteroids_pokey.vhd MB_Top.v MB_Timer_Control.v MBControl1.v MBControl2.v MBControl3.v MB_A1.v MB_B1.v MB_Program_counter.v ALU_Top.v

Table 26: Sub-Modules for *Schematic_3b.v* and *vg_top.v*

Since some of the sub-modules for the *Schematic_3b.v* were VHDL modules with custom libraries, the user should now create the custom libraries needed. To do this, the user should right-click on any file on the left panel. Then select “Add Source.” This will bring up a menu. In the menu, select “Add VHD Library.” In this case, the user should be sure to name the library as “work.” Finally add the needed VHDL code, which are the following files listed below:

- *alu_NoBuff.vhd*
- *am2901_comps.vhd*
- *am2901_comps_DataOut.vhd*
- *am2901_comps_NoBuff.vhd*
- *counters_pkg.vhd*
- *mnemonics.vhd*
- *q_reg_KL2.vhd*
- *ram_regs_E2.vhd*
- *regs_pkg.vhd*
- *synch_pkg.vhd*

At this point, the software is set up with the exception of adding in the ROM and RAM instantiations. This can be done by using steps for the CORE Generator as previously described in Section 3.3.1. Table 27 details which .coe files to use for the ROMs, what names to call the ROMs, and the remaining details. The process for each of the ROMs and RAMs in Table 27 will need to be done separately.

ROM Name	.coe File	Read Depth	Read Width	Enable Type	Type
RAM	N/A	2048	8	Use ENA Pin	Single-Port RAM
vgRAM	N/A	4096	8	Use ENA Pin	Single-Port RAM
ROMX	ROMX.coe	4096	8	Use ENA Pin	Single-Port ROM
ROM1	ROM1.coe	4096	8	Use ENA Pin	Single-Port ROM
ROM3	ROM3.coe	4096	8	Use ENA Pin	Single-Port ROM
ROM5	ROM5.coe	4096	8	Use ENA Pin	Single-Port ROM
ROM7	ROM7.coe	4096	8	Use ENA Pin	Single-Port ROM
vgROM	vgROM.coe	4096	8	Use ENA Pin	Single-Port ROM
ROM_MB	CombinedROM.coe	256	24	Always Enabled	Single-Port ROM

Table 27: ROM and RAM Instantiation Settings

The final step is to set the output constraints. This may be done by clicking on the “Files” tab on the left side of the Xilinx window and then double-clicking on the *contraints.ucf* file. Figure 61 illustrates what outputs should be assigned to the desired I/O pins.

```
# Crystal Clock - use 32MHz onboard oscillator
NET "clk_32MHz" LOC = "P89" |
IOSTANDARD = LVCMOS25 | PERIOD = 31.25ns ;

# Wing1 Column A
NET "DVY_out<7>" LOC = "P18" ;    #0
NET "DVY_out<6>" LOC = "P23" ;    #1
NET "DVY_out<5>" LOC = "P26" ;    #2
NET "DVY_out<4>" LOC = "P33" ;    #3
NET "DVY_out<3>" LOC = "P35" ;    #4
NET "DVY_out<8>" LOC = "P40" ;    #5
NET "DVY_out<9>" LOC = "P53" ;    #6
NET "DVY_out<10>" LOC = "P57" ;   #7
NET "DVY_out<11>" LOC = "P60" ;   #8
NET "DVY12inv" LOC = "P62" ;      #9
NET "DVY_out<12>" LOC = "P65" ;   #10
NET "linscale<3>" LOC = "P67" ;   #11
NET "linscale<2>" LOC = "P70" ;   #12
NET "linscale<1>" LOC = "P79" ;   #13
NET "linscale<0>" LOC = "P84" ;   #14
NET "CENTER_not" LOC = "P86" ;    #15

# Wing1 Column B
NET "" LOC = "P85" ;              #0
NET "" LOC = "P83" ;              #1
NET "DVX_out<11>" LOC = "P78" ;   #2
NET "x0" LOC = "P71" ;            #3
NET "y0" LOC = "P68" ;            #4
NET "DVX_out<8>" LOC = "P66" ;    #5
NET "DVX_out<9>" LOC = "P63" ;    #6
NET "DVX_out<10>" LOC = "P61" ;   #7
NET "DVX_out<11>" LOC = "P58" ;   #8
NET "DVX12inv" LOC = "P54" ;      #9
NET "DVX_out<12>" LOC = "P41" ;   #10
NET "DVX_out<7>" LOC = "P36" ;    #11
NET "DVX_out<6>" LOC = "P34" ;    #12
NET "DVX_out<5>" LOC = "P32" ;    #13
NET "DVX_out<4>" LOC = "P25" ;    #14
NET "DVX_out<3>" LOC = "P22" ;    #15

# Wing2 Column C
NET "" LOC = "P91" ;              #0
NET "" LOC = "P92" ;              #1
NET "" LOC = "P94" ;              #2
NET "" LOC = "P95" ;              #3
NET "" LOC = "P98" ;              #4
NET "" LOC = "P2" ;               #5
NET "VCTR_not" LOC = "P3" ;       #6
ET "" LOC = "P4" ;                #7
NET "linscale<4>" LOC = "P5" ;    #8
NET "linscale<5>" LOC = "P9" ;    #9
NET "linscale<6>" LOC = "P10" ;   #1
NET "linscale<7>" LOC = "P11" ;   #1
NET "linscale<3>" LOC = "P12" ;   #1
NET "linscale<2>" LOC = "P15" ;   #1
NET "linscale<1>" LOC = "P16" ;   #1
NET "linscale<0>" LOC = "P17" ;   #1
```

Figure 61: *TEMPEST* I/O Pin Assignments

At this point, the user is ready to select “Generate Programming File” just below the list of files on the left side of the Xilinx window in the Implementation view for

the Design tab. Once the file has been generated, the user is ready to use the .bit file created in the *TEMPEST* project folder to load through any FPGA loader for implementation onto the user's choice of FPGA that meets the specifications and requirements of this project.

7.2 Hardware Setup:

The first step to ensure proper operation of Vic Vector is to verify the solder connections for the controller inputs, these connections are clearly marked on the PCB. The marked connections for the buttons are NO (normally open) and Common; each button should have their appropriate pins connected from the corresponding button terminal to the board. Additionally, the NC (normally closed) pin needs to be grounded on the auxiliary board provided. The order for the connections of the Common on the board should be fire, zap, start, and coin. The order for the connections of the NO terminal on the board should also be fire, zap, start, and coin. The input lines from the clock should connect from Pin A and Pin B on the interface board to the DL and CL holes marked on the board via the connector on the interface board and soldered on the main PCB. Once the input connections are verified, the two output connections must be checked. The two output connections are marked X_{OUT} and Y_{OUT} on the PCB and must be connected via a wire to the input for the galvanometer. The X_{OUT} needs to be connected to the X_{OUT} galvanometer interface board and the Y_{OUT} needs to be connected to Y_{OUT} galvanometer interface board. Additionally, the HeNe 155Asl laser must be aligned properly to reflect off of the mirrors of the scanner galvanometer. Once the connections are made properly and the laser aligned, the FPGA and the galvanometer can be turned on by plugging their power connections into the wall.

7.3 Gameplay:

Once the steps for software setup and hardware setup are complete, the game is ready for playing. Once the components are powered, the laser and galvanometer-scanner will display the Start Menu. At this screen, the user should proceed by hitting the COIN button once to provide a game credit. The credit can be acknowledged by the counter displaying the number of coin credits at the bottom of the screen. Since, 1-Player mode is the only mode available, the user may hit the START button following the system's acknowledgment of one coin credit.

After pressing the START button, the game will display the choices for the various levels to be played. The user may navigate side-to-side to choose the level by spinning the Spinner Wheel. Once the user has their desired level highlighted on the screen, which is illustrated by a box surrounding the level, the user should press the START button again to begin playing the selected level.

Once the user sees the screen for game play, the user is able to navigate clockwise and counterclockwise around the level by spinning the Spinner Wheel in the appropriate direction. The user may press the FIRE button to fire their weapons at the approaching targets, which initially appear in the level at the center of the level and gradually move toward the outside edges of the level and also toward the user's ship. The ZAP button may also be depressed by the user in the event that the user would like to kill all enemies that are currently on the level. This ZAP function may only be used once per level, while the FIRE button has infinite use.

The user may now continue between the levels and insert more coins as needed to continue gameplay upon losing all of their lives. Enjoy playing the Atari© Tempest game through your own Vic Vector.

Appendix A: References

1. "Video Game Spending | Gaming Accounts for One-Third of Entertainment." Web. 21 Mar. 2011. <http://www.npd.com/press/releases/press_090520.html>.

2. Vendel, Curt. "Atari© Coin-Op/Arcade Systems." *Welcome to the Atari© History Museum*. Atari© Historical Society. Web. 11 Mar. 2011. <http://www.Atari©museum.com/videogames/arcade/arcade80.html>.

3. "*Tempest* (arcade Game)." *Wikipedia, the Free Encyclopedia*. Web. 11 Mar. 2011. <http://en.wikipedia.org/wiki/*Tempest*_(arcade_game)#cite_note-amuseum-0>.

4. “Field-programmable Gate Array." *Wikipedia, the Free Encyclopedia*. Web. 21 Mar. 2011. <http://en.wikipedia.org/wiki/Field-programmable_gate_array>.

5. Nasr, Amr. "Microprocessor AM2901 4 Bit Microprocessor Slice – Top Module." *VHDL and Verilog Designer*. Web. 07 July 2011. <http://vhdldesign.blogspot.com/2010/12/microprocessor-am2901-4-bit_5311.html>.

6. "T65 CPU :: Overview :: OpenCores." *Home :: OpenCores*. Web. 19 July 2011. <http://opencores.org/project,t65>.

   "FPGA ARCADE - Asteroids Main Page." *FPGA ARCADE - Main Page*. MikeJ, 2003. Web. 22 July 2011. <http://www.fpgaarcade.com/ast_main.htm>.

   NOTE: Reference 6 refers to the same code and same authors, but was acquired from two different sources

7. McGregor, I. "The relationship between simulation and emulation," *Simulation Conference, 2002. Proceedings of the Winter.* Vol.2, pp. 1683- 1688 Vol.2, 8-11. Dec. 2002. <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1166451&isnumber=26292>.

8. Burris, Harrison R. "Instrumented Architectural Level Emulation Technology." *ACM Digital Library*. Web. 21 Apr. 2011. <http://portal.acm.org/citation.cfm?id=1499402.1499570>.

9. Chertov, R., Fahmy, S., Shroff, N.B. "Emulation versus simulation: a case study of TCP-targeted denial of service attacks," *Testbeds and Research Infrastructures for the Development of Networks and Communities, 2006. TRIDENTCOM 2006. 2nd International Conference.* pp.10-325. <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1649164&isnumber=34577>.

10. "DatObase ::: M.A.M.E. Rom Information for *Tempest*." *ArcadeHITS² - Emulation Arcade Et Roms MAME -intro*. Web. 06 Apr. 2011. <http://www.arcadehits.net/datObase/rom.php?zip=*Tempest*>.

11. "*Tempest* - Videogame by Atari©." *Arcade, Videogames, Pinball Machines, and Antique Coin-Operated Machines -- The International Arcade Museum*. Web. 06 Apr. 2011. <http://www.arcade-museum.com/game_detail.php?game_id=10065>.

12. Margolin, Jed. "The Secret Life of Vector Generators." *Jed Margolin*. Web. 22 Apr. 2011. <http://www.jmargolin.com/vgens/vgens.htm>.

13. "File:Pincushion Distortion.svg." *Wikimedia Commons*. Web. 06 Apr. 2011. <http://commons.wikimedia.org/wiki/File:Pincushion_distortion.svg>.

14. Mingjie, Dr. Lin. “Lecture 1: Introduction (Objectives, Expectations, Logistics), FPGA's History and Future.” EEL 5722C: FPGA Design. <http://www.cs.ucf.edu/~mingjie/EEL5722/index.html>.

15. "*Tempest* Trouble Shooting Guide." *Atari©, Inc.* Web. 24 Mar. 2011. <http://www.vectorvga.com/Download/Atari©%20*Tempest*/Atari©%20*Tempest*%20Troubleshooting%20Guide.pdf>.

16. Jacobs, Andrew. "6502 Reference." *Obelisk Home Pages*. Web. 24 Apr. 2011. <http://www.obelisk.demon.co.uk/6502/reference.html>.

17. "Instruction Format." *Department of Computer Science*. Web. 18 Apr. 2011. <http://www.cs.umd.edu/class/spring2003/cmsc311/Notes/Mips/format.html>.

18. "How MOS 6502 Illegal Opcodes Really Work « Pagetable.com." *Pagetable.com*. Web. 17 Apr. 2011. <http://www.pagetable.com/?p=39.>.

19. "SIGGRAPH Computer Graphics Newsletter - Memories of a Vector World - May 98." *ACM SIGGRAPH News — Siggraph.org*. Web. 09 Apr. 2011. <http://www.siggraph.org/publications/newsletter/v32n2/contributions/rubin.html>.

20. "Vector Graphics." *Wikipedia, the Free Encyclopedia*. Web. 27 Mar. 2011. <http://en.wikipedia.org/wiki/Vector_graphics>.

21. Sayer, Nicholas. "*Tempest*: Detailed Theory of Operation." Web. 08 Apr. 2011. <http://www.kfu.com/~nsayer/games/*Tempest*.html>. Copyrighted Content – Permission Verification in Appendix B

22. "TurboTwist 2™ Arcade Spinner Control Product Information." *GroovyGameGear.com*. Web. 20 Apr. 2011. <http://groovygamegear.com/webstore/index.php?main_page=product_info&%20products_id=268&zenid=c2619c1529e3cb46c86264b212b42257>.

23. "Welcome to Ultimarc, the Ultimate in Arcade Controls." *Welcome to Ultimarc.com Home Page*. Web. 20 Apr. 2011. <http://www.ultimarc.com/SpinTrak.html>.

24. "POKEY." *Wikipedia, the Free Encyclopedia*. Web. 20 Apr. 2011. <http://en.wikipedia.org/wiki/POKEY>.

25. "How Laser Shows Work - Scanning Systems - www.LaserFX.com." *Home Page - Www.LaserFX.com*. Web. 21 Apr. 2011. <http://www.laserfx.com/Works/Works3S.html>.

26. Lee, Darry. "CW20 Galvanometer Scanner." *Laseric.com*. Web. 22 Apr. 2011. <http://www.laserpic.com/Ebay/CW20%20galvanometers.pdf>.

Appendix B: Copyright Permissions

Content Permissions for Reference #19:
Email Conversation Between Robert Higginbotham and Nicholas Sayer

We will [have] to post everything on a website, so when we get that website going, I will be more than happy to email you the link. That link should have all the information about our project. Thank you for this help!

--
Robert Higginbotham
Email: robert.higginbotham0@gmail.com

On Fri, Apr 8, 2011 at 2:09 PM, Nick Sayer <nsayer@kfu.com> wrote:

Sure! Glad to help. When you get done, I'd love to hear about the completed work.

On Apr 8, 2011, at 11:05 AM, Robert Higginbotham wrote:

> Nicholas Sayer,
>
> I am a student at the University of Central Florida and am a member of a group that is doing a Senior Design project that involves creating an FPGA Emulator for the Atari Tempest game. We came across your write-up about the Tempest game and were hoping you would be willing to give us permissions to use some parts of your document within our Research Report that we have to create. It would be a great help, if you don't mind us using the document as a resource. Below is the link to the document that I was referring to. Thank you for your time!
>
> http://www.kfu.com/~nsayer/games/tempest.html
>
> --
> Robert Higginbotham
> Email: robert.higginbotham0@gmail.com

Content Permissions for Reference #5:
Email Conversation Between Drew Hanson and Amr Nasr

From: amrnasr <amrnasr@gmail.com>
Date: July 16, 2011 6:14:15 PM EDT
To: Drew Hanson <drewhanson.ucf@gmail.com>
Subject: Re: AM2901 VHDL code

yes sure why not?
Can i see your project too? i like your idea..

Best Regards,

Amr

On Sat, Jul 16, 2011 at 12:33 AM, Drew Hanson <drewhanson.ucf@gmail.com> wrote:
Mr. Nasr,

I am a student working on my senior design project. Part of our project involves recreating the 'math box' from Atari's Tempest. One of the math box's components is a set of 4 am2901s. I was wondering if I could get permission to use the code you posted in our project.
Thank-you for your time,

Drew Hanson

Content Permissions for Reference #6:
Email Conversation Between Robert Higginbotham and MikeJ

Sure, you are welcome to do what you want with it.

/Mike

- Hide quoted text -

----- Original Message -----

From: Robert-Gmail

To: MikeJ

Sent: Tuesday, July 05, 2011 4:49 PM

Subject: Re: T65 VHDL Code

Mike,

I just wanted to make sure that I have permission to use the code you have posted for the asteroids project in the project that I am working on, due to the copyright laws on the code. Thank you for your time!

Sincerely,
Robert Higginbotham

Figure 4: Pincushion Distortion

This image is listed online as having the copyright removed and access to use the image being given freely to anyone. The image is also available in the Wikipedia Commons, which is a freely licensed media file repository.

Details on the image, as were mentioned above can be found at that following website: http://en.wikipedia.org/wiki/File:Pincushion_distortion.svg

Figure 12: Vector Graphics vs. Raster Graphics

This image is listed online as being a screenshot of a copyrighted image, yet it is also associated with licensing. According to the licensing associated with this figure, it is licensed under the Creative Commons Attribution-ShareAlike 3.0 License. This license states that users are free to Share (to copy, distribute and transmit the work) or Remix (to adapt the work) the work under the following conditions:

- **Attribution** — You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

- **Share Alike** — If you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar license to this one

This image is also found in the Wikipedia Commons, which is a freely licensed media file repository. Under the Wikipedia Commons, it is found that the author of the image has released the image into the public domain so that anyone may use the image in any way the desire.

More information on this license can be found at the following website: http://creativecommons.org/licenses/by-sa/3.0/

Details on the image, as were mentioned above can be found at that following website: http://en.wikipedia.org/wiki/File:VectorBitmapExample.svg