F15 18-545

MX-Butterfly: Design Document

Ashish Shrestha, Hui Jun Tay, Peter Pearson

CONTENTS

INTRODUCTION

Objective

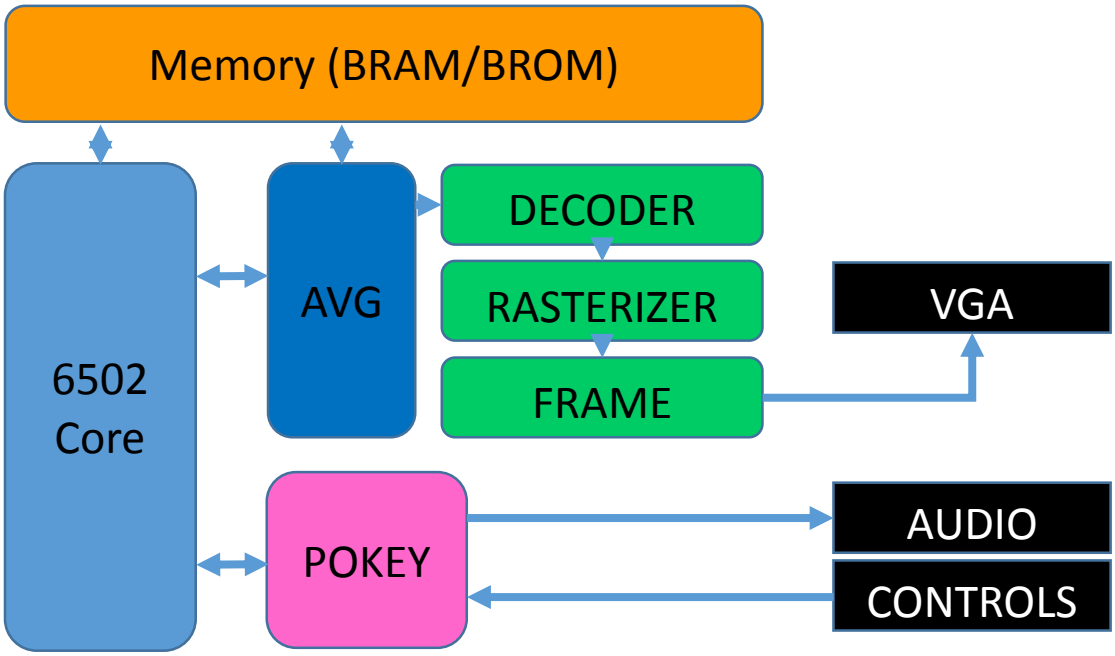
The goal of this project is to recreate an emulation of the Atari 1980 Battlezone arcade game on the Nexys4 FPGA. It is one of the oldest and most popular games to utilize the Atari Analog Vector Generator (AVG) for its graphics engine. We shall be modeling our software behavior on the MAME emulation of the original game and keeping most of the hardware I/O and CPUs the same, with the exception of using a VGA instead of a CRT for our graphics. Besides the challenge of reverse engineering such old hardware, we intend to construct an AVG to VGA decoder that will interface smoothly with the original 6502 Core and POKEY.

Platform

We chose the Nexys4 board as we felt it was sufficiently lightweight for our needs. As the original game used very little (~16KB) of memory, we did not see the need for excessively powerful (and potentially more complex) boards. I/O wise, the Nexys4 provides a VGA port that fits our graphics requirement, and a mono-audio jack that we feel is sufficient for Battlezone's simplistic sound effects. We also have the option of using the Pmod pins for analog signals from the arcade joysticks we intend to use.

SYSTEM OVERVIEW

System Architecture Diagram



Hardware Components

Each of the boxes in the architectural diagram indicates a primary hardware component of our system. The four main pieces are: the 6502 Core (a 8-bit microprocessor that runs on a 1.79MHz clock); the Analog Vector Generator MathBox (which we have replaced with our own VGA decoder), the Potentiometer Keyboard Integrated Circuit (POKEY) that handles controller inputs, audio output and the Random Number Generator; and the Memory blocks that store the value of certain hardware switches (coin inputs and settings) and the program and vector RAM/ROM.

Software Components

We do not have any actual software components in our system. We do however, have a number of disassemblers and python scripts used to convert the ROM code from .hex to .coe files for use with Vivado's Block RAM generator.

GRAPHICS

Battlezone's Analog Vector Generator

The graphics system for Battlezone uses an analog vector generator (AVG). An analog vector generator, in essence, is an electron gun which moves around the screen and draws lines from point to point. The electron gun can be set to different intensities (including off) and can output different colors as well. This allows the game to draw lines over the screen in order to draw images for the game. However, because the AVG is an analog system, it is not one which is re-creatable on a digital system like an FPGA. As a result, we had to emulate the AVG using exclusively digital parts.

Analog Vector Generator Instruction Set

The AVG is, itself, a small microprocessor with a nine-instruction ISA. These instructions are as follows: VECTOR, SVEC, CNTR, HALT, STAT, SCALE, JMP, JSR, and RET. All 9 of these instructions have to be emulated in a digital system. Aside from the VECTOR, SVEC, and CNTR instructions, this ISA is directly comparable to any other microprocessor.

The STAT and SCALE instructions are both special register write instructions, with STAT writing to the intensity and color registers while SCALE writes to the scaling registers. The intensity and color registers indicate the strength and color of the AVG's beam, making certain lines brighter than others (intensity) or just different colors. The intensity register can also be set to 0, meaning nothing is drawn. The SCALE instruction writes to two different registers: the LINSCALE and BINSCALE registers. The LINSCALE register represents linear

scaling (multiply by a value), and was handled digitally on the actual AVG. We are handling it in the same way for our emulation. The BINSCALE register represents exponential (negative exponential) scaling, and was done in an analog manner on the AVG. We are handling this digitally as well. When the AVG is told to draw any line (vector), this line is automatically scaled by these two registers.

The JMP, JSR, and RET instructions are all basic control flow instructions. JMP is an unconditional jump instruction, which changes the program counter to the value encoded in the instruction. JSR changes the program counter in the same way as JMP, and pushes the value of the next instruction (PC + 2) into the return stack. The return stack is a four-deep stack storing program counters. The RET instruction operates in conjunction with the JSR instruction, popping a program counter off the return stack and changing the program counter to the popped value.

The remaining four instructions are VECTOR, SVEC, CNTR, and HALT. The VECTOR and SVEC are both instructions which draw vectors. They are relatively indexed, meaning they move a certain distance (encoded in the instruction and scaled by the scale registers) from the previous position. As a result, it is necessary to keep the values of the current X and Y positions in registers, which are included in the architecture. The VECTOR and SVEC instructions also have intensity values encoded into them, where a zero-valued intensity means the vector is blank, an intensity value of one means we use the value in the intensity register, and any other intensity value is used as the intensity of the vector. The encoded X and Y positions (deltaX and deltaY) are two's complement signed values. VECTOR and SVEC differ in that VECTOR is a 32-bit instruction while SVEC is 16-bit (where deltaX and deltaY are multiplied by two). As a result, VECTOR is used for longer vectors while SVEC is used for shorter

vectors. The VECTOR instruction is also the only 32-bit instruction (the rest all being 16-bit). The CNTR instruction is used to center the electron gun. This means that the X and Y registers are both set to 0 (center of the screen). The final instruction is the HALT instruction, which sets to high the halt signal of the AVG. This stops execution of the AVG until it is told to resume execution via the VGGO signal from the game 6502 game processor. On receival of the VGGO signal, the AVG restarts execution at its base address (0x2000).

Emulated Analog Vector Generator Architecture

The architecture for the emulated analog vector generator is relatively simple. As has been discussed, we have six specialized registers: LINSCALE, BINSCALE, INTENSITY, COLOR, X, and Y. The LINSCALE and BINSCALE registers are for scaling, the INTENSITY register maintains the intensity of vectors, the COLOR register holds the color of the vectors, and the X and Y registers hold the current X and Y positions. The PC register holds the current program counter, which is an output to BRAM. BRAM returns the instruction, which the emulated AVG passes into the decode unit. The decode unit then decodes the instruction, and passes various flags back to the emulated AVG. For control flow operations (JMP, JSR, RET), the decode unit returns flags overwriting the next PC with the jump PC. For JSR, the decode unit also passes out a flag to write to the return stack. For HALT, the decoder passes out a flag to halt execution fo the AVG. For CNTR, the decoder passes out a flag to center the X and Y registers. For STAT and SCALE, the decoder passes the values to write and the flags to write to the registers. For VECTOR and SVEC, the decoder passes out the deltaX and deltaY values, along with flags to write to the X and Y values.

Each instruction has a constant instruction latency. In order to emulate this instruction latency, we use a counter register and enforce the following specifications. For each instruction, the decode unit outputs the latency of the instruction. On counter value 0, the counter changes its value to that of the latency of the new instruction. The counter then decrements every cycle. On the cycle where the counter is 1, the program counter changes. Due to the BRAM delay, the instruction then changes on the next cycle (counter value 0). In addition, the instruction is executed on the last cycle of the instruction (counter value 1).

These timing specifications hold for all instructions except the VECTOR instruction. This is because the VECTOR instruction is the only 32-bit instruction. The instruction width is only 16 bits (dual-ported 8-bit BRAM to keep specification with the 6502 processor), and so it is not possible to get all 32 bits of the instruction at once. As a result, in addition to the previous specifications, the program counter temporarily increments by two to get the second word of the instruction at the first cycle of the VECTOR instruction (counter value 7). This allows us to get the remainder of the instruction and store it into an instruction register, which gets passed to the decoder as the second 16 bits of the VECTOR instruction. Aside from these specifications, the VECTOR instruction executes as normal (executing on the last cycle of the instruction).

Line Register Queue

The output of the emulated AVG is two pairs of points representing a line segment, along with a color and a write signal. These connect to a line register queue (LRQ). The line register queue is a queue of line segments, which are passed into the hardware rasterizer. When the write signal of the emulated AVG is high, the line register queue reads in the data about the line

segment, along with the color of that line segment. One thing to note is that the LRQ is clocked at 100MHz, while the emulated AVG is clocked at ~6MHz. As a result, the emulated AVG would naturally write many times to the LRQ. In order to remedy this issue, the LRQ does not accept writes unless the write signal goes low between them. When the LRQ is given a read signal from the rasterizer (also clocked at 100MHz), the LRQ moves to the next line segment.

Rasterizer

The rasterizer serves as the bridge between the output of the AVG processor and the BRAM that feeds the VGA output, by rasterizing vectors into pixels. This module takes as input two coordinate pairs on the xy-plane and a signal to indicate the coordinates are valid. It outputs a series of addresses, each of which corresponds to a pixel in the BRAM, and a signal that alerts the rest of the system when it has finished processing a line.

Our implementation is built around using Bressenham's line algorithm. This algorithm relies on the fact that for a given line on a discrete xy-plane, at least one pixel will be activated on the line in every row and every column, and assumes that if a line is taller than it is wide that it will have exactly one pixel lit per row, and likewise for wide lines and columns.

Specifically, we:

1) Determine which of delta x and delta y is larger.
2) If delta x is larger than delta y for the line, then we know that exactly one pixel will be lit in every column that intersects with that line.
3) Calculate the slope and iterate along the columns starting from one endpoint of the line.
4) At every column, increment a counter by the value of the slope.

5) When this counter crosses 1, advance the index of the row where the next pixel will be drawn.
6) This continues until we reach the other endpoint of the line.
7) For each pixel, we write the address of that pixel out to the BRAM.

Because we've implemented this process of incrementing the line in purely combinational logic, each additional pixel takes only one additional clock cycle to calculate. While this is reasonably fast, it is possible that lines which are too long may not get entirely drawn in time, since time to draw is directly proportional to line length. We have yet to determine if this is an issue we actually need to be concerned about, or if we have sufficient down time between frames to draw all the necessary lines.

Frame Buffer

The frame buffer controller coordinates the switching between two 640x480 BRAMs by muxing the inputs and outputs to and from the two BRAMs. Each BRAM has a 4-bit width data_in/data_out used to transmit (I)RGB pixel data, and a 307200-bit addressing depth. At any one point, one BRAM is being written to by the Rasterizer, while the other BRAM is being read by the VGA controller. The BRAM being written to takes in a 19-bit write-address, 4-bit pixel data and an enable signal from the rasterizer. The BRAM being ready from takes a 19-bit read-address and an enable signal from the VGA controller. This allows us to convert the intended CRT output of the AVG to a 640x480 VGA output without worrying about the specifics of the Hsync and Vsync used by the VGA. The basic FSM can be summarized as follows:

1. WRITE_A: write to BRAM_A and read from BRAM_B until a *done* signal is asserted from the Rasterizer or a *pixels_done* signal is asserted from the VGA controller. After reading from a specific memory address in BRAM_B, clear the pixel data inside on the next clock cycle to "refresh" the frame buffers.
2. DONE_A: buffer state used to wait for the Hsync and Vsync pulses to finish synchronizing. The same time period is used by the Decoder and Rasterizer to calculate the pixel coordinates needed for the next frame. No pixel data is read from or written to the BRAMs in this state. When first transitioning into this state, *vggo* is asserted to signal the Decoder.
3. WRITE_B: same as WRITE_A, but instead writing to BRAM_B and reading the previously written data from BRAM_A.
4. DONE_B: same as DONE_A.

VGA Controller

The VGA controller outputs were based on the Nexys4 VGA specification, with one Hsync pin, one Vsync pin and three 4-bit RGB pins. Per specification on the Nexys4 documentation, a 25MHz clock divider was used to with two counters to count clock periods and generate the Hsync and Vsync signals. Row and col values derived from the two counters were used with an *enable* signal to read a 4-bit (I)RGB output from the frame buffer controller. This value was extended into three separate 4-bit Red, Green and Blue registers that output to the VGA pins.

Timing information was based on the 60Hz 640x480 specification from TinyVGA.com instead of the Nexys4 documentation, as using timings from the latter led to the two top pixel rows being cut off on our test screen.

Challenges

Interpreting the AVG instruction set: While we had accessing to scans of the hand-written AVG instruction set, the documentation was limited in how much it explained. Online resources by hobbyists also provided minimal information besides a few cryptic remarks about endianness and byte/word addressing.

Different clock speeds: Our AVG has three components that run at three different clock speeds: 100MHz, 6MHz and 25 MHz. Many of our initial bugs were caused by missed signals due to mismatched clock speeds across our interfaces. In one case, the synchronous reset signal was not being properly asserted as the clock divider was out of phase with the main clock. This was eventually dealt with using a separately clocked registers to buffer key inputs and outputs. As the main game processor runs at an even slower speed, we anticipate similar such problems occurring further in the project.

MEMORY MAP

We acquired the scans of the original design schematics for the Atari Battlezone game from Andy's Arcade. **Schematic Sheet 2A** details the memory map used by the game:

Memory Map Table

ADDRESS (Hex)	**FUNCTION**
0000-03FF	Program Ram (1k)
0800	Coin Switches, Slam Switch, Self Test Switch, Diagnostic Switch
0800	MSB is 3KHz clock, 2nd MSB is HALT
0A00	Option Switch Inputs
0C00	Option Switch Inputs
1000	Coin Counters
1200	Vector Generator Go
1400	Watchdog Clear
1600	Vector Generator Reset
1800-187F	Auxiliary PCB Enable
2000-27FF	Vector RAM (2k)
2800-2FFF	Vector RAM/Vector ROM (2k)
3000-3FFF	Vector ROM (4k)
5000-5FFF	Program ROM(4k)
6000-7FFF	Program ROM(8k)

Currently, we have verified the Program RAM, Vector RAM, Program ROM, Vector ROM and Auxiliary PCB Enable locations using the MAME Debugger simulation. We intend to preserve the structure of the memory map while abstracting some of the signals such as the VGGO signal to registers in other parts of our design. For most part, the ROM locations will remain untouched.

Of particular interest is the way the AVG and main game CPUs see the memory map. Current research supplemented with our own testing using the MAME Battlezone simulation suggests that the main CPU sees memory as single-byte addressed starting at 0x000, while the AVG CPU sees it as 16-bits starting at 0x2000.

This shared access is how the main program offloads the task of drawing objects to the AVG processor. The main processor writes the necessary op codes to the Vector RAM, and controls the start/reset of the AVG by writing to the Vector Go/Reset locations. The AVG processor loops on the RAM instructions to draw the screen while the main processor is handling the game. The Vector ROM contains opcodes that specify how to draw specific objects in the game, such as letters or mountains.

Further research later in the semester would be to expand the table to include more details about the tasks accomplished by different sections of Vector ROM and Program ROM.

BROMs and BRAMs

We have both the MAME simulator memory dumps and what we believe to be the original copies of the ROMs used in the Atari Battlezone arcade machine. Schematic sheet 2A shows how these BROMs are connected to the microprocessor. Currently, we are using an 8-bit read/write dual-port BRAM to store snippets of the Vector RAM/ROM code that the decoder accesses to generate outputs for our AVG to VGA converter. We intend to stay true to the original partitioning of memory, mostly due to the addressing complexities created by differences between the vector processor and main game CPU.

6502 CORE

The Battlezone game runs on a 6502 core. The 6502 has a 16-bit address output port, although the Battlezone game does not have enough memory to support a 16-bit address port. As a result, only the lower 15 bits of the address port are used. The 6502 has an 8-bit data port, which allows for reads and writes from memory. This is also the port from which the 6502 reads instructions, meaning it reads instructions in 8-bit groups. Note that the data port width of the 6502 is what requires the emulated AVG to do two consecutive reads for the 32-bit VECTOR instruction.

There are two submodules that need to be added to interface to the 6502 processor. The first is the non-maskable interrupt counter. This counter increments from 2 to 15, and when it reaches 15 it outputs a non-maskable interrupt to the 6502 core. This module is clocked at 3KHz, while the 6502 is clocked at 3MHz, meaning there is a non-maskable interrupt once every 13,000 cycles.

The other submodule that needs to be constructed is motivated by constraints placed by the emulated AVG. In particular, the AVG uses 16-bit and 32-bit instructions, meaning it would be difficult to use only a single 8-bit port for the AVG to read instructions. As a result, both ports of the BRAM are consumed by the AVG, and so there is no port dedicated to allow the 6502 to write to the vector RAM. In order to allow the 6502 to write to vector RAM, we plan to implement a store queue. This allows the 6502 to write to the store queue instead of the vector RAM. When the AVG is halted (meaning the AVG is not using memory), the store queue can then write to vector RAM.

INPUTS AND OUTPUTS

POKEY

The POKEY is a chip made by Atari that focuses primarily on keyboard inputs and audio output. The full POKEY implementation has a 4-bit memory space and includes features such as a random number generator (RNG), four audio channels, a keyboard scanning system, eight IRQs, and a potentiometer reading system with an optional fast scanning mode. Based off the hardware schematic and behavior of the MAME Battlezone simulation, we have determined that we only need to implement some of these features, namely the POT reading system, the RNG and the Audio.

POT scanning: The POKEY has 8 pins that it uses to read analog voltages off of potentiometers. Each pin has a digital counter and a DAC associated with it. Upon writing to a specific register in the POKEY's memory space, these counters will begin counting up from 0, and stop counting when the output of the DAC fed with the counter passes over the voltage on the pin. This forms a crude ADC that takes a fairly long time to execute.

An alternative mode is available for these pins, which allows a "fast scan" option. This option increments the counters in extremely large increments, which allows scanning to complete quickly but with a much smaller resolution. This is useful for reading digital voltages attached to the POT pins, all eight of which can be read off in a single memory access to the POKEY. Battlezone uses the POT pins exclusively for this fast scanning mode, and has its digital user inputs attached directly to the POKEY. Our implementation intends to emulate this with a register.

Random Number Generator (RNG): The RNG is implemented with a polynomial counter that can be set to either 17 bits or 9 bits, depending on certain flags in a control register. When a read request is made to the memory address associated with the RNG, the value of the top 8 bits of this counter are read off and returned to the user at that time. Again, implementing this is fairly straightforward, once a design is chosen for the counter.

Controllers

Currently we intend to use the RetroAracde's JS9 Joystick for controller input, which is the closest we could find to the original Battlezone arcade controls. We intend to add a peripheral board and possibly a coin operator as additional polish later in the semester.

Audio

According to its datasheet, the POKEY has four semi-independent audio channels, and a single audio output pin. Each channel has fully independent volume control, 8-bit frequency division, and noise production. The four channels can all be clocked from the same source, allowing for four channels of 8-bit resolution. As an alternative, some channels can be clocked by other channels, allowing for zero, one, or two channels of 16-bit resolution.

The channels may sample random noise from one of three polynomial counters, one of which is the same counter used to implement the RNG. Some channels may have a low pass filter and/or a high pass filter appended to them, options which are controlled by flags in other registers. The four channels are combined at the end and output on a single pin as an analog

voltage, which we will use PWM to replicate. This module will be the largest of the three, meaning it will both be the most time-consuming and have the highest potential for bugs.

PLANNING AND DESIGN

Project Approach

Our project can be divided into three phases – design, implementation and integration. Our first two weeks were spent on the design phase, largely looking for existing schematics and references for the Atari Battlezone machine. We attempted to identify the different elements needed for a working Battlezone game, acquire the tools required, and form a rough plan of attack for each of them.

Knowing this, we split the project tasks for implementation into four distinct parts – the Core CPU, the AVG (graphics), the memory and the remaining I/O. Of these four sections, we identified the AVG as being the most interesting and potentially challenging component of our task. This was mostly due to the AVG being a separate processor from the 6502, one that we believed had less documentation available, especially since no previous teams had attempted a project with the AVG before.

Weeks 3-6 were largely spent on getting the AVG working. Our intention was to have a working AVG pipeline capable of taking in Vector RAM opcodes and outputting a specific display (preferably one from the game itself) in time for the mid-semester demo. For the remainder of the implementation phase, we have split the remaining three sections amongst ourselves, with one member handling each section. Once each member has a barebones version of their respective sections, we intend to connect them together as a ‘smoke-test’ for the integration phase. A cycle of implementation and integration will then take place, with the end goal being a mostly-working product by the time of Thanksgiving.

Schedule

DATES	MILESTONES
9/2/2015 – 9/16/2015	INITIAL RESEARCH + DESIGN
9/17/2015 – 9/28/2015	AVG DELEGATION + LABS
9/29/2015 – 10/4/2016	INDIVIDUAL IMPLEMENTATIONS DONE + LABS
10/5/2015 - 10/11/2015	INTEGRATION 0 – VGA + RASTERIZER
10/12/2015 - 10/18/2015	HARDWARE PARTS ORDERED, IMPLEMENTATION PHASE 2
10/19/2015-10/20/2015	AVG-VGA DONE (DECODER -> VGA)
10/21/2015	DESIGN REVIEW
10/22/2015 - 10/25/2015	CORE VERIFIED
10/26/2015 - 11/1/2015	MICROPROCESSOR DONE
11/2/2015 - 11/8/2015	INTEGRATION 1 - ROM + CORE
11/9/2015 - 11/15/2015	INTEGRATION 2 – AVG + CPU (DISPLAY SMOKE TEST)
11/16/2015 - 11/22/2015	CONTROLS/AUDIO I/O DONE
11/23/2015 - 11/24/2015	INTEGRATION 2 – BASIC GAME WORKING
11/25/2015	THANKSGIVING
11/26/2015 - 12/6/2015	DEBUG + POLISH
12/7/2015	FINAL PRESENTATION
12/9/2015	INLAB DEMO
12/11/2015	PUBLIC DEMO
9/2/2015 – 9/16/2015	INITIAL RESEARCH + DESIGN

We intended our schedule to be fairly aggressive, with the aim of finishing the main project by Thanksgiving. With this in mind, we have been front loading the more complicated and demonstrable elements of our project such as the AVG and the display. Since integration between our modules proved difficult over the first half of the semester, we allocated more time to it for the back end of our project.

Tools

We used Vivado for programming our board, including the Block Design for creating and managing the BRAMs. We also relied heavily on the MAME Deugger to acquire memory dumps of the Battlezone game in order to verify and understand how the game worked. For version control we used GitHub for our code and important documents, while Google Drive was used to store and update team documents such as the schedule, report and presentation.

Testing

For each component we wrote individual unit tests to verify their functionality before integrating them together. Code was then run using the Vivado behavioral simulator to verify its correctness before implementing it on the board. As the behavioral simulator compiled much faster and was easier to use compared to the ILA, we ended up relying on it for most of our bugs. In the case that the simulator was unable to find a problem, the ILA we employed to isolate the issue.

To verify that our assumptions about the behavior of the game were correct we used the MAME debugger memory dumps. However, we realized later on that the MAME debugger was not entirely reliable as the watchpoints often displayed what appeared to be race conditions (as the value in the memory location at the point of the flag triggering might not be the actual value that was put there). Currently, as this is still our best lead on the system, we are proceeding with our assumptions after cross-checking them against the logic designs gleaned from the schematics.

The 6502 Core has been used by multiple 18545 teams in the past, several whom have identified a number of helpful and roust testbenches. Currently, we intend to utilize the testbench found by the Atari 5200 team to verify the core, supplementing it with our own set of tests as necessary.

Acknowledgements

We used a lot of information from Andy's Arcade, which contained both the schematics of the original Atari Battlezone system and the instruction set for the Vector generator. MAME and the MAME community were invaluable in cross-referencing our assumptions about the memory mappings in the project. We would also like to thank the other Atari-based groups for sharing information on obstacles and bugs they encountered themselves.