F15 18-545

MX-Butterfly: Design Document

Ashish Shrestha, Hui Jun Tay, Peter Pearson

CONTENTS

STATEMENT 3
INTRODUCTION 4
Objective 4
Platform 5
SYSTEM OVERVIEW 6
System Architecture Diagram 6
Hardware Components 6
6502 CORE 7
Implementation 7
Interfacing 8
MEMORY MAP 11
Address Decoder 11
Memory Map Table 12
MATH BOX 13
Motivation and Research 13
Implementation 14
GRAPHICS 15
Battlezone's Analog Vector Generator 15
Analog Vector Generator Instruction Set 15
Emulated Analog Vector Generator Architecture 17
Line Register Queue 18
Rasterizer 19
Frame Buffer 20
VGA Controller 21
INPUTS AND OUTPUTS 22
POKEY 22
Controllers 23
Audio 24
PLANNING AND DESIGN 26
Project Approach 26
Schedule 28
Tools 29

Testing 29
Lessons Learned 30
Acknowledgements 32
Individual Reflections 33
Ashish Shrestha 33
Hui Jun Tay 34
Peter Pearson 35

STATEMENT

STATEMENT OF USE

The members of Team M-X Butterfly hereby give permission for anyone to use the information in this report and the project source code in an academic, educational, or otherwise non-profit generating manner, so long as the original authors are given credit for their work.

The members of Team M-X Butterfly also give Professor Bill Nace, Professor Brandon Lucia and the 18-545 course staff permission to post this report and the project source code online.

The code repository can be found at: https://github.com/MXButterfly/F15_18545_BattleZone

December 12, 2015

INTRODUCTION

Objective

The goal of this project was 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 modeled our software behavior on the MAME emulation of the original game, 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 also had to construct an AVG to VGA decoder that interfaces smoothly with the original 6502 Core and POKEY design. As a stretch goal, we also built a working arcade cabinet with joysticks and a coin acceptor mechanism.

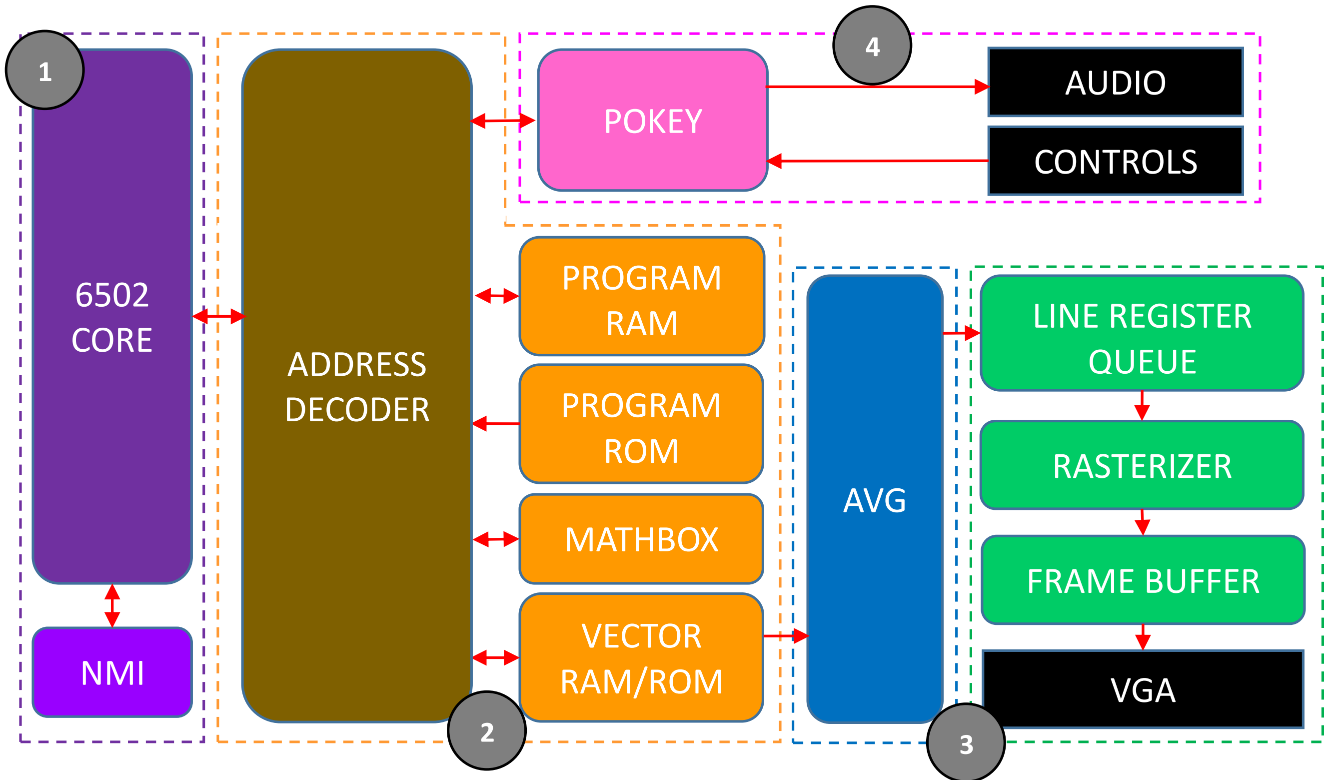
Fig 1. Final Product: Arcade Cabinet with working game, controls and coin slot on Public Demo Day

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 sufficed for most of Battlezone's simplistic sound effects. For the remaining sounds and signals that required analog components, such as the controller inputs from the arcade joysticks, we used the Pmod pins along the side of the board. Our final implementation took up approximately 61% BRAM usage on the board, with a majority of it coming from the frame buffers used for VGA.

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: 1) the 6502 Core (a 8-bit microprocessor that runs on a 1.793 MHz clock); 2) the Address Decoder that interfaces with the program RAM/ROM, vector RAM/ROM and memory mapped I/O components such as the options switches and Mathbox; 3) the Graphics core, which we have replaced with our own pipeline consisting of an AVG decoder, line register queue, rasterizer and frame buffer to VGA output; and 4) the Potentiometer Keyboard Integrated Circuit (POKEY) that handles controller inputs, audio output and the Random Number Generator.

6502 CORE

The Battlezone game runs on a 6502 core, a common core for many games developed in the 1980s. The 6502 has a 16-bit address output port, although the Battlezone game does not have enough memory to support a 16-bit address space. 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.

Implementation

For our project, we did not write our own SystemVerilog 6502 Core description. Instead, we used an available open source core written by Arlet Ottens. The 6502 description came with a verification suite which we used, and revealed no bugs with the core. Upon further inspection, however, we identified a single bug involving a lesser used functionality the core offers. The 6502 core offers a decimal flag, set and cleared with the SED and CLD commands respectively. However, the microcoded open source version of the core we used did not set the decimal flag immediately upon receiving the SED command. Instead, it set the flag only during the DECODE state. There are two other flags, adj_bcd and adc_bcd which determine, respectively, whether the 6502 is doing a decimal add and subtract, or a decimal add. These flags are set at different times, and set dependent on the decimal flag. As a result, there was a rare situation in which the 6502 would incorrectly believe it was doing a decimal subtraction instead of a decimal addition due to the decimal flag being unset, but scheduled to be set at the next DECODE state. We believe we have fixed this bug by making updates of the adj_bcd and adc_bcd flags dependent upon both the decimal flag as well as the SED flag, which schedules the decimal flag. We make no guarantees that this bug is entirely fixed: in particular, we have not determined the way the PLP instruction interfaces with the adj_bcd and adc_bcd flags, as it also (conditionally) sets the decimal flag.

Interfacing

The 6502 processor interfaces with several differing subsystems, each in different ways. These are: the graphics subsystem, the sound/controls subsystem, the memory subsystem, and the Math Box co-processor. All interfacing between the 6502 and major subsystems (those noted above) goes through an address decoder (see below), which routes read and write requests to the correct subsystem and offsets the addresses as necessary.

The graphics subsystem comprises of the Analog Vector Generator (AVG), line register queue, rasterizer, frame buffers, and finally the VGA controller. The game processor runs the game for a period of time, after which it determines what needs to be drawn for the current frame. It then writes instructions to the AVG's memory (Vector RAM) with a set of instructions, which the AVG executes to draw a frame. As noted above, the 6502 is an 8-bit processor, and so writes 8-bit instructions to Vector RAM. However, the AVG is a 16-bit graphics processor, meaning it uses primarily 16-bit instructions (with one 32-bit instruction). This presents complications for the graphics memory, which we address in the graphics subsection of the memory layout description. When the game processor is ready for the graphics processor to begin execution, it sends a pair of signals to the graphics processor: the first, VGRST, resets the AVG, and the second, VGGO, begins execution. We take a moment here to clarify that we do not guarantee a full, new frame is written into graphics memory prior to the VGRST and VGGO signals. This peculiarity is explored below. It is also worth noting that the game processor is able to read from graphics memory, a feature the game does make use of.

As noted in the graphics section of this paper, the AVG has a halt signal indicating it has completed drawing. We take a moment here to make a clarification in the system specifications. It is true that the game processor is able to determine whether or not the graphics processor is in the process of drawing. As a result, we initially assumed that the game processor refused to write to graphics

memory while the graphics processor was running. Our initial design codified this specification through the construction of a memory store queue, or a queue which would delay all writes to graphics memory until the graphics processor was finished drawing a single frame. This was to ensure synchronization. Unfortunately, and to our surprise, the developers of Battlezone did not adhere to this specification and, in fact, consistently broke this synchronizing invariant. We are fairly certain that this is a feature of the game as our analysis of writes to graphics memory in MAME showed that writes occurred regardless of whether or not the halt bit in the AVG was set. Removing the memory store queue resulted in fewer graphical glitches, and in particular fixed a problem with a stuttering background.

The sound and controls subsystem are grouped into a single subsystem because they are both primarily attached to a single chip, the POKEY. The POKEY's interfacing is a simpler memory mapped I/O system. For controls, the game processor is able to request the set of buttons which have been pressed, and the POKEY forwards that information along. The sound subsystem is also on the POKEY, and so the game core writes sound information to the POKEY, which the POKEY then plays. There is a secondary sound subsystem which is independent of the POKEY, and is a combination of discrete digital and analog parts. This is interfaced with similarly, in that the game processor sends writes to this sound chip, which then plays the associated sounds.

The memory subsystem is the simplest of the subsystems with regards to interfacing. All memory buses to the 6502 are 8-bit wide. Program ROM is static and read-only, while program RAM is free to be written to. The game processor begins setup for the game on boot by clearing program RAM. We leave the implementation details of the memory subsystem to its own section of this paper.

The final subsystem is a hardware accelerating 16-bit co-processor known as the Math Box. While we have a lot to say about the Math Box, we again save the details for its own section of

this paper. The interfacing between the 6502 and the Math Box is comparable to other memory mapped I/O, insofar as the game core writes instructions to the Math Box and reads the results out later. In particular, the 6502 writes a set of commands, one at a time, informing the Math Box to perform a specific set of calculations. The game core then polls the Math Box's status register for its completion. Upon reading that the Math Box is finished with its calculations, the 6502 then reads data from two different memory mapped registers: a Math Box High register and a Math Box Low register. The former represents the high 8 bits while the latter represents the low 8 bits of the result.

While not major subsystems, the 6502 interfaces with several other minor systems. The first is the non-maskable interrupt (NMI) counter, which sends interrupts to the game core. The NMI counter resets at 2, and counts up to 15, at which point it sends an NMI. It is clocked at 3KHz, while the game core is clocked at 3MHz, and so an NMI is sent once every 13,000 game cycles. The other subsystem is the coin door. While in the start screen, the core polls memory location 0x800 for the Right Coin acceptor. This memory mapping is handled by the address decoder that maps the memory location to a Pmod input pin on the board. We then use the CH-932 Coin Acceptor to send an active-low 50ms 3.3v pulse to the Nexys4 board via this Pmod pin. The CH-932 takes 12 volts of power and outputs a 5v pulse that we lower to 3.3v with a voltage divider. As the core polls at 3MHz, we can be certain that each coin insert will be registered with its 50ms pulse.

We have purposely ignored discussion of the power and watchdog subsystems, which are shown in the schematics of the original Battlezone. The power subsystem is unimplemented as we did not face the same power constraints as the original game cabinet. The watchdog subsystem was meant to reset the game in the case that the game ceases to send watchdog clears, implying that the game core is somehow improperly stuck in a section of code. While we could have implemented a watchdog, we felt it was unnecessary as we did not have the same safety concerns as the original developers.

MEMORY MAP

Address Decoder

The Address Decoder is the main interface between the 6502 Core and the other hardware modules in our design. It muxes reads/writes to different memory locations based on a mapping of their memory addresses to their function. While most ROMS and RAMs were generated using Vivados' Block Ram Generator, memory-mapped I/O locations such as the halt signal and option switch settings were hard-wired in the Address Decoder itself.

We chose to partition our memory blocks based on function rather than maintain a single large BRAM as 1) not all memory locations between the blocks are used, and we wanted to reserve space for the frame buffer and 2) the way the AVG and 6502 CPUs access the memory map are different. Partitioning our design also allowed us to add/remove components over different iterations which streamlined our integration and testing process.

While the 6502 accesses Vector memory as single-byte addresses starting at 0x2000, the AVG CPU accesses them as 16-bits addresses starting at 0x000. This shared access is how the main program offloads the task of drawing objects to the AVG processor. The main processor writes the necessary instruction codes to the Vector RAM, and controls the start/reset of the AVG by writing to the Vector Go/Reset locations. The AVG processor then loops on the RAM instructions to draw the screen while the main processor continues handling the game.

A significant portion of the RAM instructions are JSRs to locations on the Vector ROM, most of which are sections of opcodes that specify how to draw specific objects in the game, such as letters, tanks and the moon. This means that it was necessary for both CPUs to access the same memory locations independently of each other.

We tried various designs, including: using a memory store queue buffer to coordinate read/writes; using two copies of the Vector RAM/ROM for the 6502 and AVG CPUs each; and using a dual-ported BRAM with 8-bit read/writes on one port, and 16-bit read/writes on the other. The simplicity and efficiency of the third design made it our choice for our final design.

Memory Map Table

We acquired the scans of the original design schematics for the Atari Battlezone game from the Andy's Arcade website. These memory locations were verified against the MAME Debugger emulator. The table below is a combination of data from Schematic Sheet 2A of the original documentation with our own annotations added in brackets:

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 (language, bonus tank, missile, starting tanks)
0C00	Option Switch Inputs (bonus coin, coin mech, coin mech, coin play settings)
1000	Coin Counters
1200/1400/1600	Vector Generator Go/ Watchdog Clear/ Vector Generator Reset
1800, 1810, 1818	Mathbox Ready, Mathbox low-byte, Mathbox high-byte
1820-182F	POKEY I/O
1840	Sound Access
1860-187F	Mathbox Op Access (Writes/Operations)
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)

MATH BOX

The Math Box is a 16-bit coprocessor meant to handle various calculations that were not native to the 6502 core. As noted above, the 6502 Core interfaces with the Mathbox as a memory mapped I/O, being written to with instructions and having the results read via memory access.

Motivation and Research

We briefly discuss the motivation for the Math Box, and discuss the findings of our research. Battlezone is a game with a 3-D world at a time when 3-D games were very rare. The reason for this is that rendering a three-dimensional world requires significantly more complex calculations than a two-dimensional world. In particular, Battlezone requires various rotations and translations in order to display the game world. These calculations are too complex for the 6502, and so they were off-loaded to a separate co-processor, which is the Math Box.

Unfortunately for us, the Math Box was a rather expensive chip which made its use rare and infrequent. Only a handful of other games used it, including Red Baron (a dogfight simulator) and Tempest. As a result, we ran into significant issues finding documentation about the Math Box, with our most reliable documentation being comments by Battlezone developers reminiscing about the need for a separate co-processor to handle various calculations. To make things worse, we found that the Math Box was a proprietary piece of hardware which used unspecified chips labelled as transistor arrays in the schematics. While we eventually found the chips used (various ALUs), we were unable to determine the connections made between the Math Box ROM and the ALUs. Faced with these challenges, we opted not to do a hardware reconstruction of the Math Box as we did for the POKEY. Instead, we relied on the MAME software specification to gain an understanding of the functionality and replicated the functionality as best we could.

Implementation

We constructed the Math Box to meet the specifications of the MAME code provided to us. Unfortunately, this means that our SystemVerilog description of the Math Box reads more as software than hardware and does not have a clear architectural description. The design itself is a many state FSM, wherein various calculations occur in each state and each state has a section of MAME code corresponding to it. We sincerely apologize to any readers who hoped to use this report as documentation for the inner workings of the Math Box, and hope they understand that we faced the same problems they must be facing if they have turned to this report for an understanding of the original hardware. We do offer our meager understanding of the expected results, with the caveat that we make no guarantees about their accuracy. Instructions 0x6B and 0x72 perform fixed point multiplication and addition, multiplying two pairs of registers representing fixed point decimal values, and then adding the results of those two multiplications. Instructions 0x73 and 0x74 perform fixed point division. Instruction 0x71 performs some mixture of the previous instructions, although we are not certain what the result is. We did not implement instruction 0x7C as we found it was unused in Battlezone. We believe instruction 0x7D performs some subtraction and then absolute value, and then performs 0x7E, which is the maximum of two registers added to three-eighths of the minimum of the two registers.

GRAPHICS

Battlezone's Analog Vector Generator

The graphics system for Battlezone uses the Atari analog vector generator (AVG). An analog vector generator, in essence, a hardware module that moves an electron gun around a screen to draw 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. Rather than build an external analog component or CRT, we chose to emulate the AVG using exclusively digital parts that fed to a VGA output.

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.

CATEGORY	**INSTRUCTION**
STAT, SCALE	Special Register Writes
JMP, JSR, RET	Control Flow
VECTOR, SVEC, CNTR, HALT	Vector Drawing/Control

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 handle 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. Our implementation handles 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 (PC) to the value encoded in the instruction. JSR changes the PC 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 PCs. The RET instruction operates in conjunction with the JSR instruction, popping a PC off the return stack and changing the PC 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 receiving 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 for 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. As the instruction width is only 16 bits (dual-ported 8-bit BRAM to keep specification with the 6502 processor), it is not possible to get all 32 bits of the instruction at once. Hence, 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 in 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 on the last cycle of the instruction.

Line Register Queue

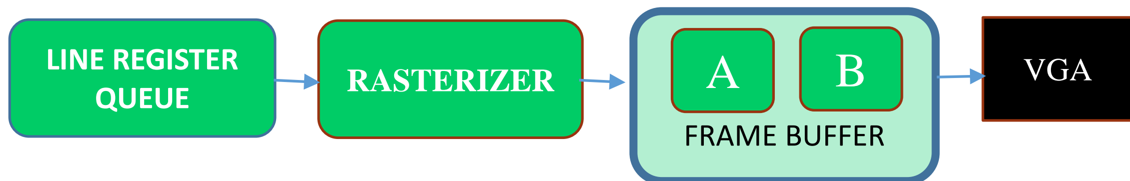


Fig 2. AVG decoder to VGA output pipeline

The output of the emulated AVG is two pairs of points representing a line segment, along with an intensity 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 intensity 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.

Originally, we tried to implement this process of incrementing the line in purely combinational logic, such that each additional pixel takes only one additional clock cycle to calculate. While this was reasonably fast, we ran into timing issues during placement on the board due to having too much combinational delay. Our final design pipelines the calculations in the rasterizer by adding a third "buffer" state. While this means we draw at a 2 cycle delay, the rasterizer is so much faster relative to the VGA output that the delay is barely noticeable.

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 store intensity 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 read 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 it is safe to switch frames and start clearing A. Since pixel data flows from the AVG core to the line register queue to the rasterizer, we should only switch when all three are idle. *avg_halt* indicates that the AVG core is done processing the vector instructions for that cycle, after which *empty* and *idle* signals from the line register queue and rasterizer are checked in order. Once all three signals are asserted, the frame buffer switches frames.
2. CLEAR_B: read from BRAM_A, clear BRAM_B until *vggo* is sent from the 6502 Core. Clearing 307200 memory locations on a 100MHz clock requires approximately 3ms. Rendering

a single 640x480 frame takes 16.7ms. We thus assumed that there would be sufficient time for the frame buffer to fully clear BRAM_B before *vggo* is asserted to start drawing a new frame.

3. WRITE_B: same as WRITE_A, but instead writing to BRAM_B and reading the previously written data from BRAM_A. from WRITE_A in WRITE_B.
4. CLEAR_A: 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 intensity output from the frame buffer controller. As the original game had hard-coded color values based on the position of the screen, we simply set the intensity of the RGB output pins based on the row value and the 4-bit intensity data. 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.

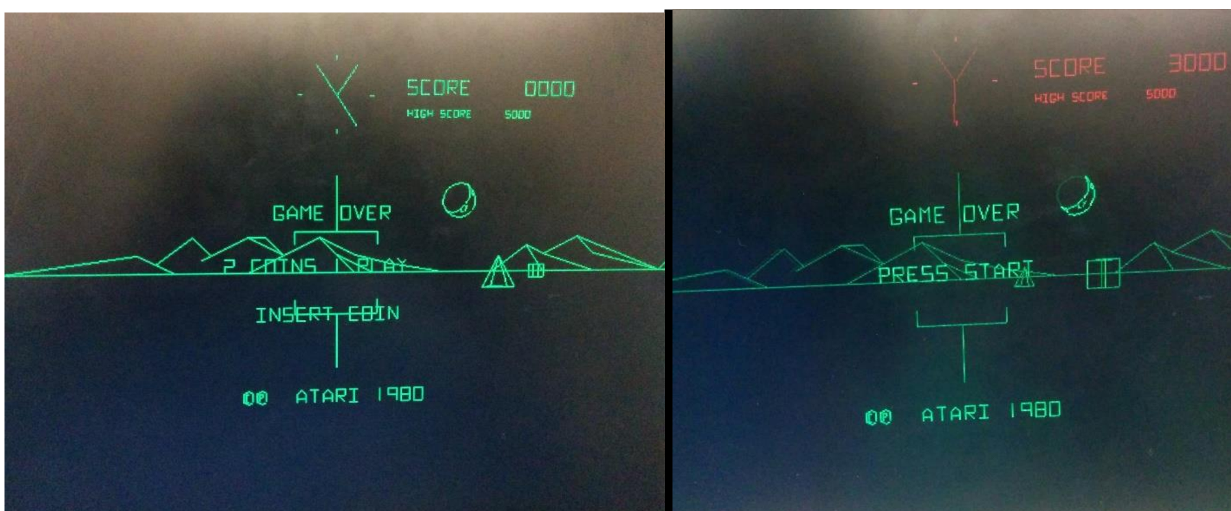


Fig 3. Comparison of early (left) and late (right) snapshots of the AVG-VGA pipeline. Intensity and color were added in the latter half of the project.

INPUTS AND OUTPUTS

POKEY

The POKEY is a chip made by Atari that focuses primarily on keyboard and potentiometer 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 determined that we only needed to implement some of these features: the fast 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 emulated this functionality by writing the scan pins to a single digital register in a single clock cycle upon a write to the appropriate control 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. Implementing this was achieved with a pair of shift registers, a few XNOR gates, and a multiplexer to switch between the 17 bit and 9 bit modes.

Audio: Most of the information necessary for implementing the POKEY audio systems can be found under section two of the POKEY's datasheet. Some details, however, are unclear with just the information found in that section. For example, the details of the implementations of the noise circuits are not detailed anywhere in the datasheet except in the schematic diagram at the end. All three noise generators come in the form of linear feedback shift registers of different lengths. It took a fairly long time to unravel the connections on the feedback gates in these sections of the circuit because of the poor scan quality of the only copy of the document that we could find. To complete the audio implementation we simply chained together each of the components specified in section two of the datasheet for each of the four mentioned semi-independent audio channels.

Controllers

We used RetroAracde's JS9 Joystick for user input, which is the closest we could find to the original Battlezone arcade controls. These joysticks feature a trigger button on top of two-axis digital switches. To reduce each joystick to the single-axis specification needed for Battlezone, we laser cut small acrylic pieces to replace two of the switches. These pieces prevented the joystick from moving along one of its axes. We also put a button on the final cabinet that the user presses in order to start playing the game. For this, we just grabbed one of the available arcade buttons out of the cabinet in the lab. As an interesting side-note, the inputs are attached in the Battlezone schematic as being active low, but the MAME ROM we used had them listed as active high in the documentation. This caused some confusion when we first attached the buttons to the FPGA. We eventually configured the buttons to be active high to match the ROM.

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 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 used PWM to replicate. This module was the largest of the three, meaning it was the most time-consuming to implement. This was partially due to the fact that we didn't have a complete schematic of the audio subsystem available, and had to infer what we could from a very poor-quality scan of the POKEY datasheet.

Beyond the POKEY audio, the schematic notes that three other wires are attached to the line that goes to the game's speaker system. These three lines represent the sounds for firing a shell, an explosion, and the revving/putter of the tank's engine. These sounds are distinct from other sounds in the game because they must change volume smoothly (for example, the sound of a shot being fired starts loud and slowly gets more quiet). Perhaps for this reason, they are implemented externally to the POKEY. Instead, all of these sounds are produced by charging and discharging a capacitor to create a smoothly changing voltage. Unfortunately, capacitive charging/discharging follows an exponential equation. By the time we realized we would need to implement an exponential adder to have these sounds on the FPGA, we knew that we did not have the necessary time to do so. To get around this, we did as Atari did, and made a 1:1 replication of the necessary circuits on external breadboards. To convert

the 3.3 volts from the board to the 5 volts needed to drive the external circuits, we used inverting MOSFET amplifiers. These external sounds are subject to extreme static interference and for a time were able to pick up an AM radio signal being broadcast from the roof of Hamerschlag Hall. We also had a tendency to get extremely loud bursts of static while testing, and sometimes the circuits would cut out entirely. We were eventually able to achieve a relatively stable configuration that made the correct sounds during both demos, though we never did identify the source of our problems in the earlier circuit.

PLANNING AND DESIGN

Project Approach

Our project was 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.

Based on our initial research we split the project tasks for implementation into four distinct parts – the Core CPU, the AVG (graphics), the memory map interface and the remaining POKEY 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. It was only later in the project that the fifth major component, the Mathbox, was recognized as a significant and separate component from the POKEY.

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 split the remaining three sections amongst ourselves, with one member handling each section. Once each member had a barebones version of their respective sections, we intended to spend the remaining weeks up to Thanksgiving integrating them together. While the AVG itself was developed smoothly in time for mid-semester, the remaining integration approach had to be modified due to our late realization of the Mathbox as a fifth, undocumented component. Thankfully, initial integration of the AVG and 6502 Core components was relatively smooth, which freed up two of our members to focus on the Mathbox. Integration of the POKEY however, had to be delayed to the last two weeks before the final demo, which we had thankfully allocated as buffer time for full integration.

Ultimately, this approach led to the development of team ‘specialists’, where only one member really understood how a specific component work. On one hand, this caused several issues at times where progress could not be made with integration as the team member responsible for that component was unavailable. On the other hand, implementation was fast, while modifications and debugging was very efficient as specialists had a deep and thorough understanding of their area. To mitigate this later in the project, we tried to have every subsystem have a secondary ‘specialist’ who understood enough of the primary specialist’s code to perform integration. We found that this allowed for a good balance in a team of three - at any meeting there would always be at least one primary and secondary specialist for any one component, even if one team member was unavailable at that time.

We intended our schedule to be fairly aggressive, with the aim of finishing the main project by Thanksgiving. With this in mind, we attempted to front load 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. However, we overlooked the MathBox component in the POKEY, which resulted in the actual schedule becoming slightly warped. The table below illustrates the changes between our initial planned schedule, and the final actual schedule we followed;

Schedule

DATES	MILESTONES (PLANNED)	MILESTONES (ACTUAL)
9/2/ – 9/16	INITIAL RESEARCH + DESIGN	INITIAL RESEARCH + DESIGN
9/17 – 9/28	AVG DELEGATION + LABS	AVG DELEGATION + LABS
9/29 – 10/4	INDIVIDUAL IMPLEMENTATIONS DONE + LABS	INDIVIDUAL IMPLEMENTATIONS DONE + LABS
10/5 - 10/11	INTEGRATION 0 – VGA + RASTERIZER	INTEGRATION 0 – VGA + RASTERIZER
10/12 - 10/18	HARDWARE PARTS ORDERED, IMPLEMENTATION PHASE 2	HARDWARE PARTS ORDERED, IMPLEMENTATION PHASE 2
10/19-10/20	AVG-VGA DONE (DECODER)	AVG-VGA DONE (DECODER -> VGA)
10/21	DESIGN REVIEW	DESIGN REVIEW
10/22 - 10/25	CORE VERIFIED	CORE VERIFIED
10/26 - 11/1	MICROPROCESSOR DONE	AVG + CORE INTEGRATION
11/2 - 11/8	INTEGRATION 1 - ROM + CORE	AVG + CORE INTEGRATED (MOVING DISPLAY)
11/9 - 11/15	INTEGRATION 2 – AVG + CPU (DISPLAY SMOKE TEST)	JOYSTICK INPUTS COMPLETE, MATHBOX STARTED
11/16 - 11/22	CONTROLS/AUDIO I/O DONE	MATHBOX INTEGRATED
11/23 - 11/24	BASIC GAME WORKING	MATHBOX DEBUGGING
11/25	THANKSGIVING	THANKSGIVING, MATHBOX DEBUGGING
11/26 - 12/6	DEBUG + POLISH	POKEY INTEGRATED, 6502 CORE BUG FIXED
12/7	FINAL PRESENTATION	FINAL PRESENTATION, MATHBOX WORKING
12/9	INLAB DEMO	INLAB DEMO, COIN DOOR + SOUND INTEGRATED
12/11	PUBLIC DEMO	PUBLIC DEMO, CABINET COMPLETE

Due to the sudden introduction of the Mathbox, our integration of the POKEY was pushed back, with majority of our time spent on debugging Mathbox errors due to a lack of documentation. Thankfully, as we front-loaded our schedule, we had ample buffer time to complete the Mathbox debugging, although many of our intended polish features such as the coin door and the cabinet were not added until a few days before the final demo.

Tools

We used Vivado for programming our board, including the IP Block Design for creating and managing the BRAMs. We also relied heavily on the MAME Debugger 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.

We also wrote a number of disassemblers and python scripts used to help build and understand our project. Functions provided by these scripts include:

1. Converting ROM code from .hex to .coe for use with the Block RAM generator.
2. Disassembly/Assembly of the Atari AVG instructions based on the instruction set
3. Visualization of the AVG vector instructions
4. Mathbox simulation used as a reference for debugging Mathbox behavior

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 was employed to isolate the issue. While this was effective early on when we were debugging the AVG core, it became extremely time consuming during the integration phase as the full game contained long cycles of initialization and memory checks. Simulations of the game later in the project schedule had to be run for at least 2 seconds of simulation time with a 100MHz clock, taking up over 5GB of memory which required many simulation runs to be done overnight.

To verify that our assumptions about the behavior of the game were correct we used the MAME emulators' debugger mode. Getting used to the MAME debugger interface was difficult, though the help menu at *balleyalley.com* was invaluable for understanding some of the subtler commands like performing trace dumps of watchpoint accesses or modifying memory locations on the go. Many errors in the Mathbox were found by simply comparing values in the MAME program execution trace with the simulation outputs. Battlezone's built-in self-test mode was highly useful in this regard, as it would run tests on the Mathbox and set a flag that indicated errors in high/low return bits. Checking the results of the self-test simulation with the MAME trace and our own C code was a tedious and time-consuming process, but one that eventually yielded results. The difficulty of this debugging process was the main contributor to the Mathbox taking up so much of the later weeks of our revised project schedule

As mentioned in the 6502 section, we found and fixed a bug in the 6502 Core decimal mode that was causing our high scores to be displayed with large, hexadecimal offsets. We had originally believed this bug to be from our own modules, as the 6502 Core has been used by multiple 18545 teams in the past, several whom have identified a number of helpful and roust test-benches that we used to verify our Core. Testing with the ILA however, showed that the source of the error came from the Core's decimal addition mode. It was only by using the ILA to capture a cycle-accurate snapshot of the 6502 core's inputs/outputs at the time of the error that we were able to reproduce the bug. Vivado's simulator was unsuitable for this task, as there were too many signals with too much information to sift through. We recommend future teams take understanding and using the ILA seriously, as it is a highly useful tool for capturing isolated behavioral edge-cases in a project.

Lessons Learned

The most major lesson we learnt was not to overlook components. Although we had access to the full system architecture early-on, our initial research had dismissed the Mathbox as being a minor part of

the POKEY, when in actuality it was a highly-complex, critical part of the main system. In this sense, we were perhaps a little too eager to start working on one major difficult component (the AVG pipeline), so much so that we neglected to check if there were other, equally difficult subsystems we needed to be aware of. More initial research into the work needed to develop each component in the system should have been done during the planning stage. A checklist of components, with a ranking of priority and importance might have been helpful in ensuring that critical tasks did not get overlooked.

One of the early problems we encountered was conflicting coding styles within the project. Initial integration during the first half of the project had many bugs caused by having conflicting implementations of common modules such as registers and counters. For example, one team member would use reset low for their counter FSMs, while another would use reset high. Errors like this made it hard to discern the true bugs in the system during integration. We would recommend future project groups establish a common coding convention early in the project, as well as a common utilities file for standard modules such as registers and counters.

MAME was an invaluable tool in understanding and debugging our project. For any team that is looking to do an arcade emulation, we strongly recommend understanding how each of the MAME commands work so that you can gain full use out of it. There were a number of times in the project where we developed our own scripts to handle a particular task, only to discover later that MAME could do something similar. Look at the help file on Balley Alley for some good use case examples.

Vivado has a tendency to flood the user with warnings and information flags, to the point that we started ignoring the 300 or so warning flags we were getting at one point of our project in favor of ‘getting it to work’. Many of these warnings turned out to be minor coding errors that when added up, caused significant issues in the project. Subtle bugs were often simple “off-by-1” port-width errors (e.g.

mapping a 13-bit port as [11:0]) that only turned up during edge-cases that made use of the dropped bits. Eliminating these warnings should be the first, not last thing one does when synthesizing a project.

Lastly, at least one team member should spend some time to understand the various implementation settings and optimization flags Vivado provides. Use of these flags was crucial towards the our success at the end of our project, as several of our modules were not meeting timing under the regular implementation settings. Enabling flags as such post-place physical optimization and aggressive-explore really improved the timings on our system, especially in areas with long logic propagation such as the rasterizer to frame-buffer pipeline. Using the right implementation settings can mean the difference between constant screen flickers and a smooth display experience.

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. Arlet Otten’s 6502 Core implementation saved us a lot of project time, allowing us to focus on the main parts of the project. We would also like to thank the other Atari-based groups for sharing information on obstacles and bugs they encountered themselves. Lastly, we would like to thank the instructors for their continued advice throughout the project.

Individual Reflections

Ashish Shrestha

As the only member who took computer architecture, I spent most of my semester working on system-level design and integration. I began by writing the graphics processor, handling the instruction decode and execution. Afterwards, I planned the game core integration and planned the memory layout for the system as a whole. I spent the second half of the semester designing the Math Box and working on integration.

I would average my workload at a little over a dozen hours a week, although that number is not very representative of my actual workload. My time spent on this class tended to fluctuate heavily week to week, increasing the closer we got to a major milestone.

I thought the class was very loosely structured, which was great for our team because we were on top of our work the entire semester. Our major setback for this semester was how late we recognized we needed to design the Math Box, putting us back around two weeks. Luckily, we allotted enough time at the end of the semester to allow for that slack, but it would have been nice to have an extra set of eyes on our project at the start of the semester. An early meeting to go over system-level design with a professor or TA might have let us on to the fact that we had overlooked a major part on the schematic.

Hui Jun Tay

I would say my biggest responsibility was actually project integration. Most of my early research focused on understanding and using the MAME emulator. I thus functioned as the go-to for checking correct game behavior. This resulted in me eventually becoming the groups 'debugger' later in the project, since MAME was needed to verify everything from memory mappings to instruction traces during integration. While the only modules I wrote individually were the frame buffer and VGA, in the course of debugging and integration I became familiar with the Address Decoder, AVG decoder, Rasterizer and Mathbox modules. I also worked with Peter on the coin box, and was the one responsible for moving all our work from the lab bench into the arcade cabinet (which I painted!). Administratively, I was responsible for a majority of the architecture diagram and poster used in our final demo.

The amount of time I have spent varied from week to week, but at its high points I have easily spent 15-16 hours a week on 18-545. Debugging and integration took up about 80% of the time spent on this project, of which a large portion was simply waiting for synthesis/simulation to finish. At one point, to debug the Mathbox Ashish and I were doggedly stepping through every Mathbox read/write in our Vivado simulation run and comparing them against a ~2000 line trace of the MAME simulation until we found discrepancies that pointed us towards the bug.

Overall, I found 545 a very effortful but satisfying class. The ability to define your own project is both intimidating and liberating. I feel that taking this course has taught me a lot about making good design decisions and managing/scheduling a project team. My only complaint is on the instability of AFS (and Vivado), which cost us a lot of overnight simulation data. I also did not find the TAs very helpful with using the tools like Vivado, though the professors gave excellent feedback and advice on how and where to take our project next.

Peter Pearson

This project went much better than I ever could have expected it to. Hui Jun and I came into class on the first day of the semester with two other people, expecting to spend the semester working on a Game Boy Advance. This plan fell apart almost immediately when Professor Nace told us that the maximum team size was three, and our two other team members left to join other teams. Ashish also found himself without a team, and so M-x butterfly was born.

Given this initial setback I thought that the three of us would never have been able to work as well together as we did. Each of us came in with a radically different skill set, and we were able to use this to our advantage. I was the weakest in Verilog, but I was the only one of us who had taken 18-348, so I was designated as the I/O guy. This translated into me writing the code for the rasterizer and POKEY, as well as constructing the external circuits necessary for the controller input and audio output. When writing the POKEY I tried to keep my design as close to the hardware specification as possible, to avoid any new bugs that I might introduce by translating the description to software.

My hours worked per week varied wildly throughout the semester, so I can’t give an accurate schedule of what I was doing at any given time. I can say for sure that I put in a lot more effort as the semester started wrapping up, in order to get all the sounds implemented and the cabinet finished. My advice to future takers of this class: don’t be afraid to ask for help when you need it. Manage your time wisely, ask questions early, and for the love of all that is holy and sacred, frontload your work. Don’t put off until tomorrow what can be done today.