Sega Master System

18-545: Advanced Digital Design

Fall 2016

Team Swag Master System

Celeste Neary

Jeremy Sonpar

Suzz Glennon

Table of Contents

1. Statement of Use

We, Celeste Neary, Jeremy Sonpar, and Suzz Glennon, will allow this document to be posted on the 18-545 course website. We will also allow any part of our code to be reused or edited in future semesters of the course.

Github Repository: https://github.com/celesteneary/sms

2. Project Overview

2.1. Goal

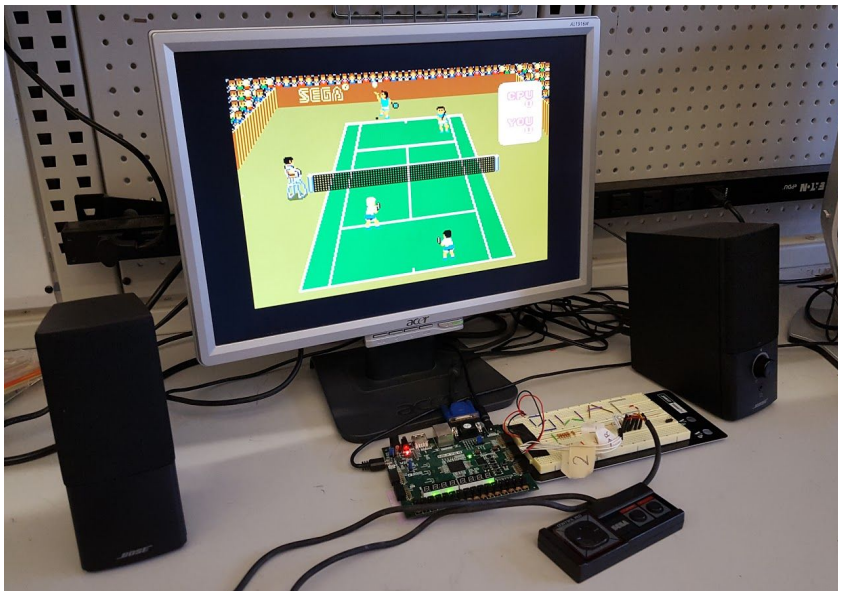
Our goal at the beginning of the semester was to create a Sega Master System that ran *Sonic the Hedgehog* (and other arbitrary games) and supported two players. Our demo would have video, audio, a small game selection, and two controllers.

2.2. Motivation

Our team agreed on creating a video game console from the beginning, since we wanted to have a project that would be fun to play at the end of the semester. We liked the idea of doing a console similar in scope to the NES, and we wanted to make a console that had not been attempted before in 18-545.

2.3. Result

We made a Sega Master System that runs multiple games and supports two players. Our demo had working video, audio, and two controllers. We chose four, fully-working games for our demo: *Super Tennis*, *Great Soccer*, *Transbot*, and *Ghost House*. We also ran *Sonic the Hedgehog*, which runs but has graphical glitches (we believe that *Sonic the Hedgehog* uses a different graphics mode than our implementation).



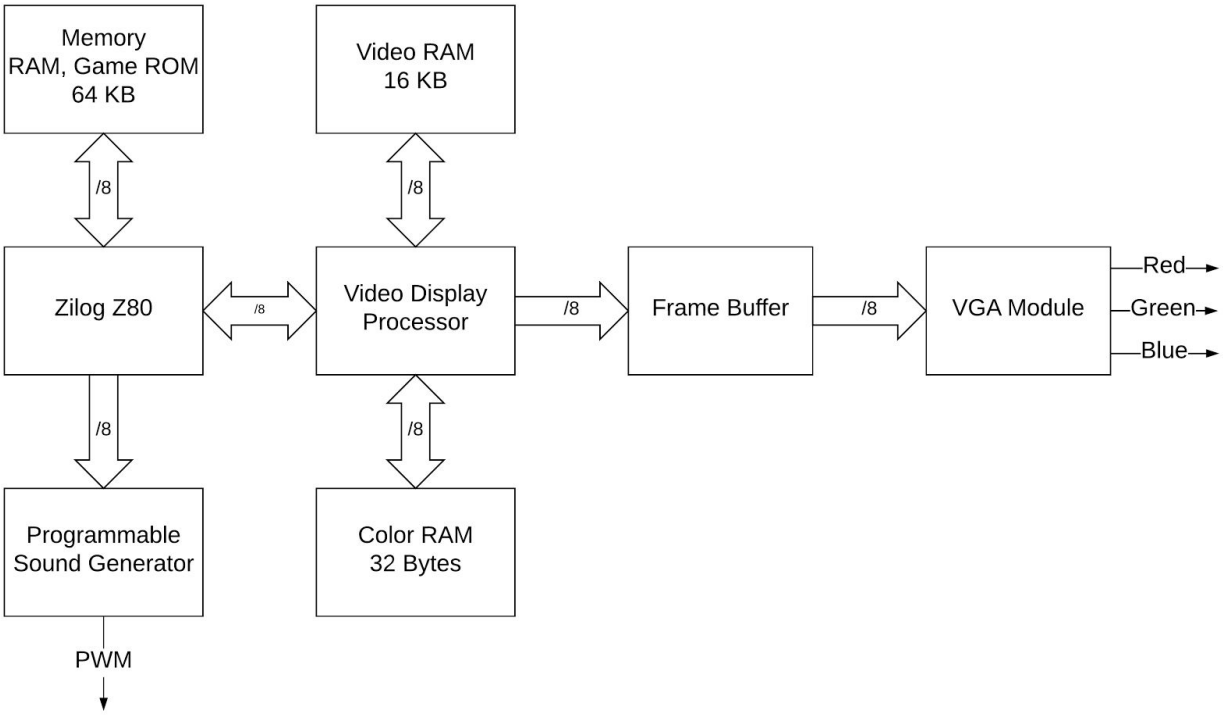
2.4. Development Tools

We developed our Sega Master System in SystemVerilog with Vivado. Our console is programmed on a Nexys 4 board (which has the Artix-7 FPGA). We chose the board because it had enough resources for the Sega Master System, had VGA and PWM audio outputs, and had enough Pmod connectors for two controllers.

We used spreadsheets to create our own patterns and tables for the Video RAM (VRAM). One spreadsheet converted patterns that we designed to the 32-byte pattern format, and another spreadsheet held the contents of VRAM memory so it could easily be converted to a coefficient file. We also had a python script for converting game ROMs to coefficient files.

3. Project Design

3.1. System Block Diagram



The Zilog Z80 is the CPU for the Sega Master System. It has its own RAM and loads game data from a ROM. The Z80 sends commands to the Video Display Processor (VDP) and the Programmable Sound Generator (PSG), which handle graphics and audio respectively. The VDP uses data from the Video RAM (VRAM) and Color RAM (CRAM) to determine the color of each pixel on the screen, which is stored in the frame buffer. The VGA module reads from the frame buffer to create the red, green, and blue VGA outputs. The PSG uses commands from the Z80 to produce a PWM audio output.

3.2. Z80 Processor

3.2.1. Z80 Overview

The only CPU used in the Sega Master System is the Zilog Z80. The Z80 is a small CPU with an 8-bit data bus, 16-bit address bus, and 23 registers designated for specific tasks. Rather than write the Z80 ourselves, we were able to locate an open source Z80 processor on OpenCores, called the A-Z80 (linked here: https://opencores.org/project,a-z80). Naturally, we still had to verify the A-Z80 to make sure that it would function correctly and would not introduce any bugs. Tracking down bugs in a CPU that we did not write ourselves would have been time consuming and difficult, so we wanted to avoid doing that later in the semester.

3.2.2. Verifying the A-Z80

The A-Z80 came with a few toolsets to verify its own correctness. It came with multiple test files to simulate every documented instruction code on the Z80 to verify that they ran correctly. We believe these were intended for in case the Z80 was modified for some reason to verify its behavior. Naturally we could not completely trust that the testbenches that came with the A-Z80 meant that it was correct (it would have been very surprised if they didn't) so the next step was to instantiate the A-Z80 in Verilog and verify it ourselves.

3.2.3. Instantiating the A-Z80

Before we could run my own assembly programs on the A-Z80 (from here on referred to as just the Z80) we needed to instantiate a RAM to hold the instruction opcodes and the data used by the program. We also required FSMs to monitor the signals coming from the Z80 to ensure that the rest of the Sega Master System was interpreting them correctly. For example, the WR_l signal dropping could mean either a write to memory, a write to memory-mapped I/O, or memory refreshing after an instruction fetch.

3.2.4. Instruction Fetching

Below is a timing diagram for the Z80 instruction fetch. Instruction Fetching takes 4 clock cycles to do, the first two for fetching the instruction and the second two for doing a memory refresh of the RAM. This is not necessary to do on an FPGA, however to stay as faithful as possible to the Z80 we wait those two cycles. Going into the Z80 to remove that, even if done correctly, could have introduced bugs in other locations.

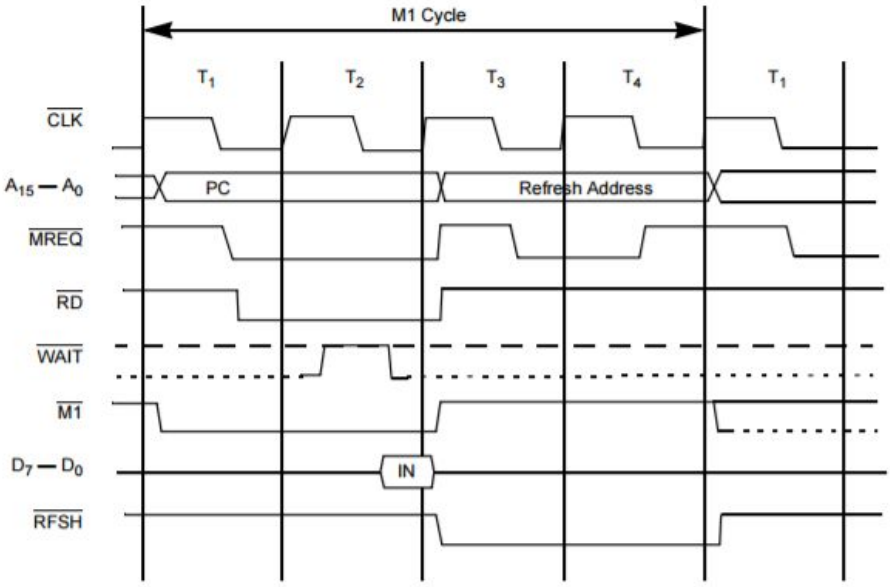


Figure 5. Instruction Op Code Fetch

3.2.5. Memory Reads and Writes

Memory Reads and Writes with the Z80 are very similar, they both take 3 cycles, but with memory reads we needed to be careful to only drive the data into the Z80 on the last cycle of the read. Write also drops a cycle later than read does when comparing the two timing diagrams, however the Block RAMs provided by Vivado are single cycle write so that is enough time to do the write before the end of the cycle.

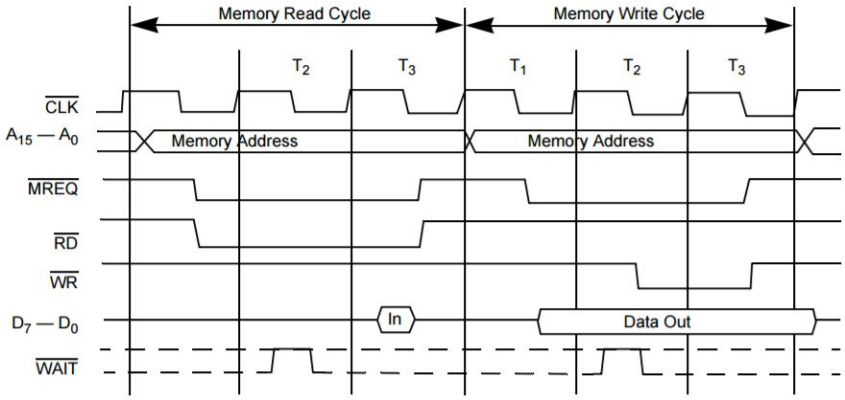


Figure 6. Memory Read or Write Cycle

3.2.6. IO Reads and Writes

The last main operation required by the Z80 was IO reads and writes. These were used by the Z80 to communicate to other components or chips in the hardware system it is in. The Z80 had 8-bytes devoted to IO ports, so it is capable of doing reads and writes with up to 256 other components. Similarly to memory interfacing, reads and writes are very similar, even with the same restriction to only drive read data from another device on one clock cycle.

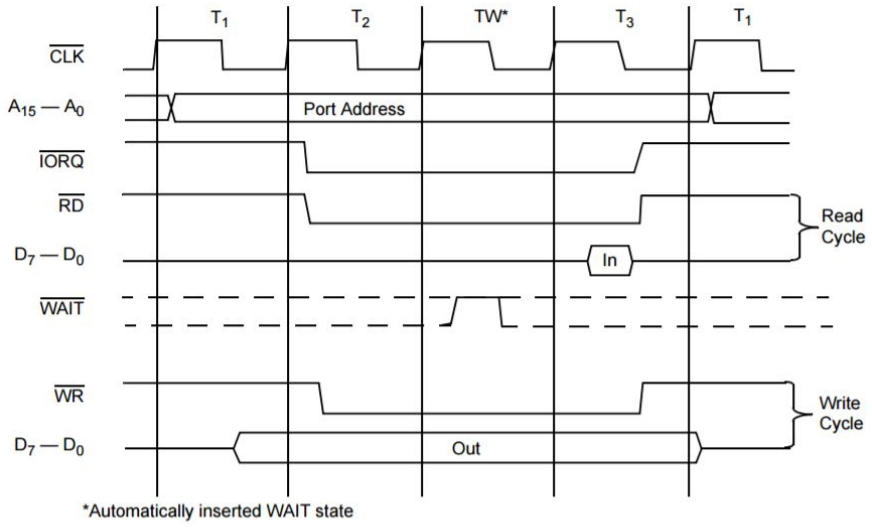


Figure 7. Input or Output Cycles

3.2.7. Interrupt Switching

In general, the Z80 is capable of handling three interrupt modes (modes 0, 1, and 2). However the Sega Master System only required one, Interrupt Mode 1. The Z80 samples for the interrupt signal on the last clock cycle of every instruction. If it is enabled to accept interrupts (they can be disabled during portions that the CPU could not recover from, like when it is already handling an interrupt), then it triggers a special instruction fetch which lasts for extra cycles. It retrieves the location of the interrupt handler and during the refresh portion (not in diagram) stores the address of where the instruction handler needs to return to. After switching to the handler, the responsibility is left to the programmer to save all other program state, handle the interrupt as necessary, restore the program state, and call a return instruction to resume the normal execution.

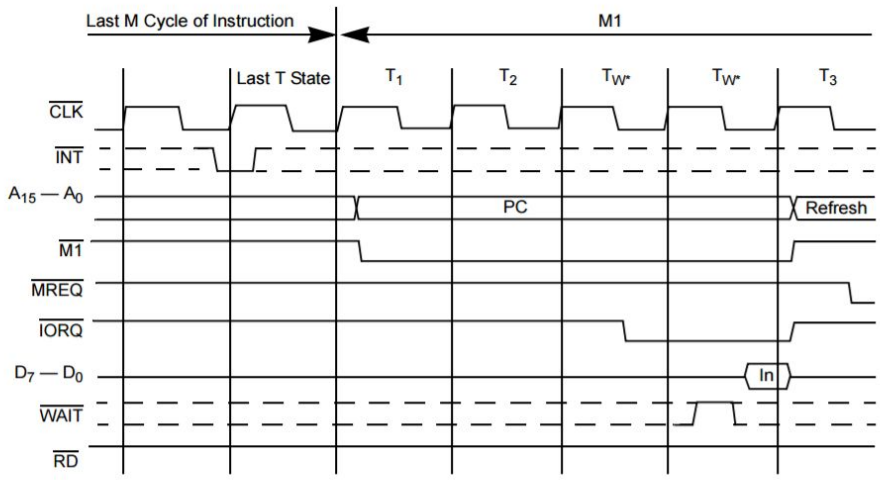


Figure 9. Interrupt Request/Acknowledge Cycle

3.2.8. Verifying the Z80 In SystemVerilog

So after instantiating and verifying the timing of the FSMs for controlling the Z80 we needed to write some assembly programs to verify that everything was working correctly with the Z80. Luckily for us everything was working exactly as expected. The A-Z80 met all timing diagrams and consistently produced the correct data values at the correct time. The one issue that became apparent at this point with the A-Z80 was that on reset its internal registers were reset to X (undefined) rather than a 0, as I and the Sega Master System programmers assumed. To circumvent this, on reset before executing any actual code I wrote a small assembly program to manually set all of the register values to 0 and then switch into whatever needed to run next (usually the BIOS).

3.2.9. Z80 and the Sega Master System

The Sega Master System used a right out of the box Z80 with no modifications. However it did have a specific memory map and designated specific memory mapped IO ports for different purposes. Finally, due to the Z80 only having a 16-bit address bus to access 1 byte values, the maximum cartridge size would have been 2^{16} = 64 KB. The Z80 needs to address some RAM on its own though, so without any paging system the maximum game size is only 32 KB. The Sega Master System implemented a simple paging system that would allow for game cartridges of up to 512 KB, which allows for much bigger more expansive games.

3.2.10. Memory-Mapped IO Ports

We used five memory-mapped IO ports for the Z80 in the Sega Master System. Their directions and a high level description are provided in the table below.

Port	Direction	Function
3E	Out	Memory Enables
7F	Out	PSG Data
BE	In/Out	VDP Data
BF	In/Out	VDP Status/Command
DC	In	Joystick Port
DD	In	Joystick Port

3.2.11. Port 3E

Port 3E manages the memory enables of the Z80's memory system. That way, it can address using the same addresses and do things like disabling the BIOS code when a game is running (why would a game ever care about the BIOS code?) or enabling the card versus cartridge port on the Sega Master System.

The exact format of port 3E and what values correspond to what is provided below.

Bit	Value 0	Value 1
0	Don't care	Don't care
1	Disable Init ROM	Enable Init ROM
2	Enable Joysticks	Disable Joysticks
3	Enable BIOS ROM	Disable BIOS ROM
4	Enable 8K RAM	Disable 8K RAM
5	Enable Card ROM	Disable Card ROM
6	Enable Cartridge ROM	Disable Cartridge ROM
7	Enable External Port	Disable External Port

A note on the Init ROM, it is a short assembly program that exists to initialize all of the values in the A-Z80 to 0, as they reset to X's. This was done separately as doing so in the BIOS would ruin function calls to specific addresses. This did not exist in the original Sega Master System, and so bit 1 was also a don't care. Everything else is replicated exactly as the original console.

3.2.12. Port 7F

This is the interface to the PSG, and is described in the PSG section.

3.2.13. Port BE/BF

This is the interface to the VDP, and is described in the VDP section. In addition to these ports for communicating with the VDP, we also added the interface signals from the original TMS9918, which were csr_l, csw_l, and mode.

3.2.14. Port DC/DD

These two ports are for interfacing with controllers and are described in the Controllers section.

3.2.15. Interrupt Handling

The Z80 normally uses three different interrupt modes, IM 0, IM 1, and IM 2. The Sega Master System only requires use of IM 1, which will always jump to address 0038 to handle interrupts.

3.2.16. The Sega Master System Memory Map

Below is a diagram of the Sega Master System's memory on bootup. It reserves the bottom 8 KB for the BIOS instructions, and reserves the top 16 KB (with a few exceptions that I will discuss) for RAM usable by the Z80. However, this is really only two 8 KB blocks because the Sega Master System mirrors all memory writes in the range of addresses C000 to DFFF to the range E000 to FFFF.

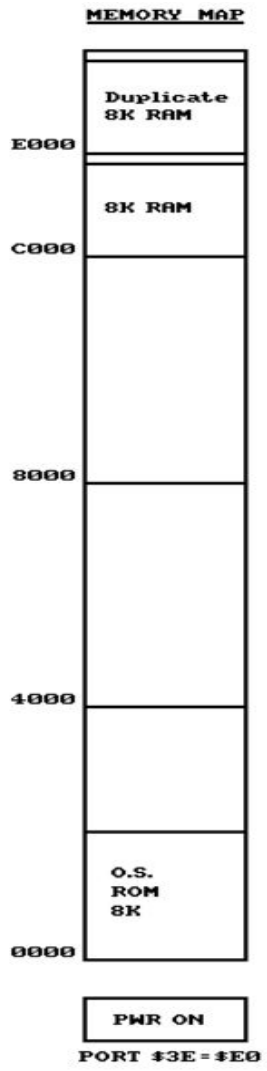


Figure M-1

Image Credit: smspower.org

If the BIOS determines that a 32 KB Sega Card has been put into the System, it will modify its memory map to include the space for the game in the bottom 32 KB. The 16 KBs of RAM from C000 to FFFF still function in the exact same way they do for the BIOS. A diagram of this is provided below.

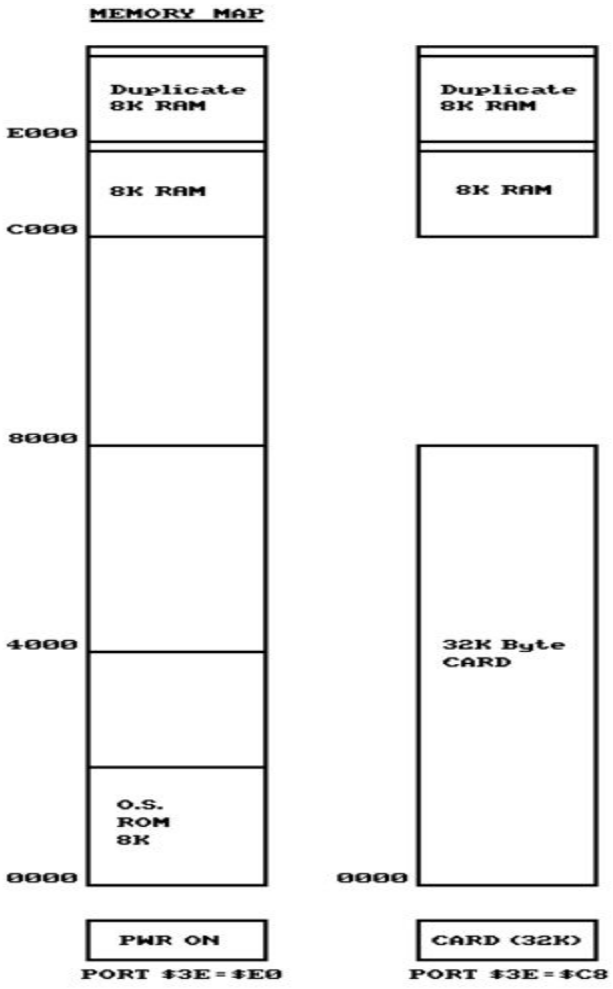


Image credit: smspower.org

3.2.17. Paging Cartridges

To handle paging larger cartridges, the Sega Master System divides its memory into four Frames: 0, 1, 2, and 3. Frame 3 is reserved for RAM as it is with the BIOS and Sega Cards, but Frame 0, 1, and 2 are used to page the ROM of larger cartridges. Each 16 KB Frame requires 14 bits to access throughout, so that gives the bottom 14 bits of the paged address. For any bits higher than that, the Sega Master System cartridges save three registers in their hardware (not in their ROM) to store which bank value corresponds to Frame 0, 1, and 2. This provides up to 8 higher order bits to the cartridge, allowing for paging of up to 2^{22} = 4 MB cartridges. Those three hardware registers are set by writes to addresses FFFD for Frame 0, FFFE for Frame 1, and FFFF for Frame 2. So in our implementation, we moved those three on-cartridge registers onto the console, as our cartridges are purely Block RAMs with no additional functionality.

This comes with a few caveats though, the bottom Kilobyte (from addresses 0000 to

03FF) of Frame 0 is always mapped directly to those addresses on the cartridge, regardless of what Frame 0 should map to. This is so the Z80 always has access to the initialization code and the interrupt handler (which is at address 0038) without needing to worry about fixing its paging.

Finally, the Sega Master System allows Frame 2 to act as ROM for reading game data from the cartridge but also to act as 16 KB of RAM which is stored on the cartridge for saving game data (such as high scores). Whether Frame 2 is to correspond to the ROM or the RAM is defined in memory address FFFC, and similar to the paging is normally implemented in hardware on the cartridges, so that was also moved to the console in our implementation. In practice though, because the FPGA clears all memory on power off, we cannot save high scores after the console is turned off (but they will record until then).

3.3. Video Display Processor

3.3.1. Z80 to VDP Interface Overview

The Z80 to VDP interface can perform five tasks: write to a VDP register, read from the VDP status register, write to the Video RAM (VRAM), read from the VRAM, and write to the Color RAM (CRAM).

To perform these actions, the VDP has a state machine based on the information in the control port and the mode, csw_l, and csr_l signals.

3.3.2. Write to a VDP Register

The VDP writes from the Z80 to a VDP register after receiving 8 bits of data followed by a second byte, which contains the VDP register number and the write to VDP command word “10.” For this transfer, both bytes are sent on the control port.

First Byte	D7	D6	D5	D4	D3	D2	D1	D0
Second Byte	1	0	X	X	R3	R2	R1	R0

There are 11 8-bit VDP registers. Each register contains information for VDP control or VRAM addressing. Some of the registers have no effect, which is a result of the VDP reusing some of the TMS9918 architecture. We kept these registers since the games will write to them.

Register 0	Control signals
Register 1	Control signals

Register 2	Name table base address
Register 3	No effect
Register 4	No effect
Register 5	Sprite attribute table base address
Register 6	Sprite pattern generator base address
Register 7	Backdrop color
Register 8	Background X scroll
Register 9	Background Y scroll
Register A	Line counter

3.3.3. Write to Video RAM (VRAM)

The VDP writes from the Z80 to the VRAM after receiving 2 bytes on the control port, the last of which starts with the command word “01.” Both bytes contain the address information.

First Byte	A7	A6	A5	A4	A3	A2	A1	A0
Second Byte	0	1	A13	A12	A11	A10	A9	A8

Then, the data on data port is written to that address in the VRAM. The 14-bit address register auto-increments, so subsequent data written to the data port is written to the VRAM at the next addresses, as long as csw_l = 0, csr_l = 1, and mode = 0. The VRAM is a two-port Block RAM with 3.58 MHz writes and reads.

3.3.4. Write to Color RAM (CRAM)

The VDP writes from the Z80 to the CRAM after receiving 2 bytes on the control port, the last of which starts with the command word “11.” Only the first byte contains the CRAM address.

First Byte	X	X	X	A4	A3	A2	A1	A0
Second Byte	1	1	X	X	X	X	X	X

Then, the data port is written to that address in the CRAM. The address is stored in the same 14-bit auto-incrementing address register as the VRAM address, but only the

bottom 5 bits are used. The Z80 continues to write to the CRAM as long as csw_l = 0, csr_l = 1, and mode = 0 (the same behavior as writing to the VRAM).

The CRAM contains two 16-color palettes. 8-bits are used to store each color in the following format: - - B B G G R R.

3.3.5. Read from VRAM

The VDP sends the Z80 data from the VRAM after receiving 2 bytes on the control port, the last of which starts with the command word “00.”

First Byte	A7	A6	A5	A4	A3	A2	A1	A0
Second Byte	0	0	A13	A12	A11	A10	A9	A8

The data from that address is stored in the read buffer and sent to the Z80 on the data port. The address register is incremented and writes to the data port will be written to the VRAM at the current address.

3.3.6. Read from Status Register

When csw_l = 1, csr_l = 0, and mode = 1, the VDP will send the contents of the status register to the Z80 over the control port. The status register contains the interrupt flag, coincidence flag (when two sprites overlap), and ninth sprite flag (when there are more than eight sprites on a scanline).

3.3.7. Video RAM Format

The VRAM holds three tables: the pattern name table, the sprite attribute table (SAT), and the pattern generator table. Registers 2, 5, and 6 determine the base addresses for these tables.

Register 2:	1	1	1	1	Name Table [13]	Name Table [12]	Name Table [11]	1

Register 5:	1	SAT [13]	SAT[12]	SAT[11]	SAT[10]	SAT[9]	SAT[8]	1

Register 6:	1	1	1	1	1	Pattern Generator Table [13]	1	1

The default addresses are $3800 for the name table (register value FF), $3F00 for the sprite attribute table (register value FF), and $0000 for the pattern generator table (register value FB).

3.3.8. Pattern Collector

Both patterns and sprites are made up of 8 by 8 pixel blocks, which are found in the pattern generator table. Patterns are aligned on a 32 by 24 grid and make up the majority of the display. Each patterns uses 32 bytes in the pattern generator table. Each row of pixels is represented by four bytes, one byte for each bit in the CRAM address. The most significant bit of each byte corresponds to the leftmost pixel. Putting the most significant bits together forms the CRAM address that holds the color of the leftmost pixel in the line.

The pattern collector module uses scanlines to determine the patterns to display on each line. To do this, it reads from a location in the pattern name table that corresponds to the current scanline (a value between 0 and 191, inclusive), which is simply the row for which the pixels to be displayed are currently being calculated. Each entry in the name table corresponds to a single 8x8 pixel pattern and is 2 bytes long, in little endian format. The least significant 9 bits are the index of the pattern in the Pattern Generator Table. The other bits contain control bits for each pattern - vertical flip, horizontal flip, priority, and color palette. When enabled, the vertical flip and horizontal flip bits flip just the pattern in that one 8x8 location vertically or horizontally, respectively. The priority bit determines whether to display the pattern above any sprites in that pattern location, effectively ignoring any sprites, as patterns do not have transparency. The color palette bit determines whether the pattern's colors will be read from the first 16 colors or second 16 colors in the CRAM. Similarly to sprites, after obtaining the index of the pattern, the pattern collector module then reads from the corresponding location in the Pattern Generator Table to obtain the CRAM addresses for the 8 pixels of the pattern in the current scanline. After a line has been completed, the pattern collector waits for the sprite collector to also complete its line, before incrementing the scanline value for both to continue on.

3.3.9. Sprite Collector

Unlike patterns, sprites can be displayed anywhere on the screen. The sprite collector module parses the Sprite Attribute Table (SAT) to output a scanline of sprite pixels. The SAT is set up in the following format (table taken from *SMS Power!* VDP Documentation by Charles Macdonald):

00: yyyyyyyyyyyyyyyy
10: yyyyyyyyyyyyyyyy
20: yyyyyyyyyyyyyyyy
30: yyyyyyyyyyyyyyyy

```
40: ????????????????
50: ????????????????
60: ????????????????
70: ????????????????
80: xnxnxnxnxnxnxnxn
90: xnxnxnxnxnxnxnxn
A0: xnxnxnxnxnxnxnxn
B0: xnxnxnxnxnxnxnxn
C0: xnxnxnxnxnxnxnxn
D0: xnxnxnxnxnxnxnxn
E0: xnxnxnxnxnxnxnxn
F0: xnxnxnxnxnxnxnxn
```

y = Y coordinate + 1
x = X coordinate
n = Pattern index
? = Unused

Our sprite collector module has its own copy of the VRAM and iterates through the table of y coordinates for every line. The y coordinate in the table represents the top row of the sprite. If the sprite falls within the current line (up to 7 lines below the y coordinate), the VRAM address jumps into the table of x coordinates and pattern indexes. Next, it stores the x coordinate. The pattern index points to the very first line of the 32-byte pattern. Using the difference between the line number and the y coordinate, the VDP skips to the right line in the pattern and reads four bytes (one for each bit of the CRAM address).

When a sprite is found, it is added to the sprite collector output, which is five 256-bit signals: one for each bit in the CRAM address, and one that describes which pixels are enabled. We do this by creating a 256-bit mask with the current sprite and existing sprites on the line. When the next sprite is processed, we mask the sprite output signals so the earlier sprites on the line are not overwritten. The enable output allows us to make areas not covered by sprites transparent (patterns will show instead of sprites).

Our sprite collector module will process up to eight sprites per line. If there is a ninth sprite on one line, the ninth sprite flag is set letting the processor know that the VDP has stopped processing sprites for that line. The collision logic is also in the sprite collector module, which occurs when any two sprites overlap, setting the collision flag.

3.3.10. Color Priority

The color priority is determined for each pixel separately. Since we use scanlines to simultaneously determine the patterns and sprites displayed on each line, we can also read the pattern priority bit for each pattern. Normally, sprites are simply displayed on

top of the patterns, so when a sprite is enabled, in any opaque pixel, we write the color of the sprite into that pixel rather than the color of the pattern behind it. However, each pattern also has some control bits, one of which is the priority bit. When this bit is enabled, we display that 8x8 pixel pattern on top of any sprites that may overlap with that location.

3.3.11. Pixel RAM

The Pixel RAM (PRAM) is a frame buffer using a dual port BRAM. It holds the color information for each pixel in the 192 by 256 grid. Since there are 49,152 pixels on the screen, the PRAM has 16-bit addresses. The VDP writes to the PRAM at 3.58 MHz and the VGA module reads from the PRAM at 50 MHz. The 8-bit data stored in the PRAM is organized as follows: - - B B G G R R. After reading the data from the PRAM, it is converted to vgaRed, vgaGreen, and vgaBlue signals.

3.3.12. Interrupts

The VDP provides two different types of interrupts and two flags to the Z80 which are stored in the status register (see: Read from Status Register). The VDP also has a separate output to the Z80 with the interrupt signal. The interrupts, the horizontal line interrupt (HLI) and vertical blanking interrupt (VBlank), tell the Z80 when the VDP has completed displaying a line or the entire visible screen, respectively. The Z80 can enable either, both, or neither of these signals to send interrupts by writing to VDP control registers 0 and 1. When both types of interrupts are enabled, the value in bit 7 of the status register indicates which type of interrupt is currently being outputted when the interrupt signal is high.

3.4. Programmable Sound Generator

3.4.1. Overview

The Programmable Sound Generator (PSG) is the Texas Instruments SN76489. The chip has three square wave tone channels and one noise channel. Channel 3 is the noise channel. Our implementation does not output the noise channel due to time constraints. Each channel has 16 volume levels.

The PSG has an 8-bit data input. The Z80 can write to eight internal registers in the PSG: four 4-bit volume registers, three 10-bit tone registers, and one 3-bit noise register. In the original architecture, the tone registers give counter reset values for the tone generators. Our implementation generates the tones slightly differently because we use a 100 MHz clock instead of the 3,579,545 Hz clock to generate the signals.

3.4.2. Z80 Interface FSM

The first step in the PSG is parsing the data word from the Z80 and writing to the correct register. The module that does this is called psgFSM. There are four states in the FSM called Start, Wait, LatchData, and Data. If databus[7] = 1, the current word

holds register and data information. This is the LatchData state. Bits 6 and 5 of the data choose the channel (channels 0 through 3). Bit 4 chooses whether to write to the volume register (1) or the tone/noise register (0) for the chosen channel. Only one register can be enabled at a time. The remaining four bits in the data is written to the chosen register. If the register is more than four bits wide, the lowest four bits are written. We have two registers for each tone register: a 4-bit register and a 6-bit register. Since there are 11 total registers, the enable signal is 11-bits wide.

If databus[7] = 0, the current word only holds data information. This is the Data state. Any subsequent data is written to the previously chosen register until the next LatchData state. If a tone register is chosen, the six highest bits of the register are always written in this state; the enable signal changes from the lower register to the higher register for that tone. For the volume and noise registers, the enable signal is exactly the same as in the preceding LatchData state.

The PSG uses the 3.58 MHz clock for writes from the Z80. The clock and write enable signal (we_l) are used to change the state, since the Z80 does not write to the PSG every clock cycle.

The outputs of the psgFSM module are the data in each tone, volume, and noise register. If the output enable (oe_l) is high, the volume outputs are 4'b0 instead of the values of the volume registers. Since 4'b1111 is actually the quietest volume level (no sound), the volume outputs of the psgFSM module are inverted to make processing the volume easier in the next steps.

3.4.3. Generating a Tone

A 1-bit wave is generated for each of the three tone registers. According to SMS Power, the equation for frequency given a tone register value is:
Frequency = (3,579,545 Hz) / (2 × register value × 16)

So, the tone register value of a 440 Hz square wave is 254. 440 Hz is 227,272 clock cycles at 100 MHz (the input clock). Since a 440 square wave would last 227,272 clock cycles, there would be 113,636 clock cycles between 440 square wave rising and falling edges. 254 × 447 ≈ 113,636, so the counter to generate the correct square wave is 447 times the tone register value. When the counter reaches tone register × 447, the output signal is inverted.

While writing this report, we realized that we could have used the 3.58 MHz clock from the PSG data writes, but the above paragraph describes how our implementation actually works.

3.4.4. Mixing Tones

Since there is only one output channel and three tone channels, we needed to combine the signals. Recall that each tone has a 1-bit square wave output. To combine the channels, we multiplied each channel by its volume (4'b1111 being the highest volume) and added them together to get a value between 0 and 45 (inclusive) at any given time.

3.4.5. Pulse Width Modulation

The Nexys 4 has a mono PWM audio output. PWM stands for Pulse Width Modulation, which is a type of digital to analog conversion that relies on the duty cycle of a signal. For a speaker, the higher the duty cycle, the louder the sound.

We have a simple module that outputs a signal for the given duty cycle. The module counts up to 100 and then starts over at 0. When the count is less than the duty cycle, the output is a 1. When the count is more than the duty cycle, the output is a 0. The input of this module is simply the added channels. The output goes to ampPWM, which is the audio output of the Nexys 4.

3.5. Controllers

We have two controllers. Each controller has seven wires: one ground wire, and one wire for each of the six buttons (up, down, left, right, button 1, and button 2). We cut the wire on each controller and added sturdy crimp wire terminals to the ends of the wire. Each wire (besides ground) was connected to either the JA Pmod port (player 1) or the JB Pmod port (player 2) and also a resistor connected to the 3.3 V Pmod output.

The game programmers were responsible polling the controllers for inputs and then handling those inputs in the game. A table is provided below of the Memory-Mapped IO ports and what the different values to each port mean.

Port	Bit	Controller Input
DC	0	P1 Up
DC	1	P1 Down
DC	2	P1 Left
DC	3	P1 Right
DC	4	P1 Button 1
DC	5	P1 Button 2
DC	6	P2 Up
DC	7	P2 Down

DD	0	P2 Left
DD	1	P2 Right
DD	2	P2 Button 1
DD	3	P2 Button 2
DD	4	Reset
DD	5	Don’t care
DD	6	Don’t care
DD	7	Don’t care

4. Building the Master System

4.1. Design Partitioning

The high level partitioning is exactly the same as the original Sega Master System: we have a Z80 processor, a VDP, and a PSG. The individual chips are partitioned differently than the original hardware, mainly because we did not have schematics for the chips, only functional descriptions. The VDP is broken up into the Z80 interface FSM, the pattern generator, and the sprite generator. The PSG is broken up into the Z80 interface FSM, the tone generator, the tone mixer, and the PWM output module.

4.2. Testing and Verification

For most of the semester, we used the ILA IP block for debugging our code. When the VDP was displaying pixels, we could also use that for debugging (although the ILA was often more helpful). For integration, we found out about simulation, which cut down on synthesis time by a lot.

Using the oscilloscope was extremely helpful for debugging the PSG (sound generator). Often times, the audio output frequency was too fast for human ears, so debugging would have been so difficult without being able to see the actual waveform. For debugging the PWM waveform, we assigned the output to a pins on the Pmod connector. For debugging the actual audio output (after PWM), we connected a cable to the audio output and probed that.

4.3. Status and Future Work

Currently, we have a fully-functioning Sega Master System that runs some games. Future work could include adding the three other graphics modes: Text, Multicolor, and Graphics II. Graphics II is probably the most complex and most important out of

the three, since we expect that many later and larger Master System games use this display mode.

5. Lessons Learned

5.1. What We Wish We Knew

We wish that we knew about Vivado's simulation tool before the last few weeks of the semester. We could have saved so much synthesis time and ILA debugging time by simulating the output of the VDP instead of always using an ILA.

5.2. Good and Bad Decisions

5.2.1. Good Decisions

It was a good decision to use a Z80 from OpenCores instead of writing our own. Using a Z80 from OpenCores allowed us to finish the console and also have time at the end to make our final product more polished (by fixing graphics bugs, adding an additional controller, integrating sound, making a case, etc.).

One of the first major design decisions we made was deciding the clock speed for the Z80. Online sources were not consistent; some said 3.58 MHz and others said 4 MHz. Originally, we were going to use 4 MHz because we thought it would be easier to implement. We also thought that the clock was 3.58 MHz only because the original VDP was, and that it would not matter which speed we chose. Eventually, we realized that 3.58 was more accurate, and clocking the game at 4 MHz would make it run faster than intended. Also, it turns out that making a 3.58 MHz clock with the clock wizard IP and one clock divider was not difficult to do.

As soon as we realized the scope of the VDP, we had two people working on it instead of just one. It helped to have two people who understood how the VDP worked. Also, we made more intelligent design decisions by talking through the system and agreeing on an implementation before starting the code.

5.2.2. Bad Decisions

It was a bad decision to base so much of our VDP on the TMS9918, especially the CPU interface. The TMS9918 has one bus used for both data and control. It uses csr_l, csw_l, and mode to tell whether the information on the bus is an address, a command, or data. The Z80 has two separate memory-mapped IO ports for data and addresses, so we wrote a module to combine these into one bus and make the csr_l, csw_l, and mode signals. Really, we should have just used two ports from the beginning and only interpreted the commands in the VDP, rather than converting it one way in the Z80 and then converting it back in the VDP.

Initially, our pattern generator iterated through the pattern table and wrote one whole pattern to the frame buffer at a time. The real Sega Master System VDP writes to the screen one line at a time. We thought this made more sense for us because we had a frame buffer, so we didn't need to write the output by scanline. It also cut down on address calculation, because each pattern address only needed to be calculated once per frame instead of eight times per frame. Once we finished the pattern generator, we moved on to the sprite generator and realized that the best way to write sprites was by scanline. Because patterns sometimes have priority over sprites,which is determined in the pattern generator logic, the way we wrote the pattern generator would make it harder to correctly display sprites. So, we made the decision to rewrite the pattern generator so it could work better with sprites, even though we had a working version.

5.3. Words of Wisdom

If there a module or part already exists, use it! Starting with the Z80 from OpenCores allowed us to focus on other parts of the system, which worked out really well for us.

Stick to your schedule as much as possible. We were able to pull it together in the last week, but you will feel better if you have your project done earlier.

Design your interfaces early. If we thought about the Z80 to VDP interface earlier, we could have avoided the awkward interfacing that we have now.

If a component looks challenging, don't be afraid to assign two people to work on it.

Use simulation of some kind! It will cut down on synthesis time.

6. Personal Statements

6.1. Celeste Neary

At the beginning of the semester, my job was to make the VDP. I quickly realized that the VDP was more complex than we expected, so I pulled in Suzz to work on it with me. I worked on the VDP for the majority of the semester. A lot of work I did at the beginning of the semester was a bit misguided. I spent the first few weeks reading about the chip the VDP is based on: the TMS9918. While the TMS9918 documentation was very thorough, it was different than the VDP in more ways than expected, including the way that colors are stored, the number of VDP registers, what each of those registers contained, etc. I also spent a few days learning about Cellular RAM, before I knew that Block RAM existed.

Suzz and I wrote the FSM to interface with the Z80 together. Because I read the TMS9918 manual and SMS Power documentation so much, I was able to answer or clarify most design questions having to do with the VDP while we were working on it. The first thing I did that made it to the final product was writing the interface between the frame buffer and VGA module. It read the individual pixels from their addresses in

the frame buffer and output the correct RGB values for the current VGA pixel. I also wrote the first implementation for patterns that read data from the VRAM. Suzz and I worked on patterns and sprites together until mid-November.

In the middle of November, I started working on sound. I decided to use the mono PWM audio output on the board instead of the stereo I2S Pmod module. I wrote all of the PSG except for the noise output. I took a few breaks from sound to help finish the VDP and do integration debugging, but I finally got the PSG working during the last week of class. During most of the last two weeks, however, I was constantly looking at Z80 to VDP interface waveforms and debugging with Suzz and Jeremy.

Throughout the semester, I also took on the role of organizer. I started all of the presentations and reports and made the outlines for each one. I also checked the schedule often and reminded the team when labs, presentations, and reports were due.

At the start of the semester, I probably spent about 12 hours a week in lab. In the last two weeks of the semester, I think I spent over 40 hours a week in lab.

I really enjoyed 18-545. It is probably the best class I have ever taken. It was a lot of work, and at times it felt overwhelming, but the end result and the skills I have learned were definitely worth it. For most of the semester, the goal felt unattainable or barely-attainable for me. During the last week, when things were finally coming together, I wanted to be in lab all of the time to make the project the best it could be. I am proud of what we accomplished this semester, and I had a wonderful experience with my team and in the class.

6.2 Jeremy Sonpar

After the initial division of labor I was in charge of handling the CPU for the Master System, largely because I had taken 18-447, Introduction to Computer Architecture, in Spring 2016 with Professor Hoe. I was prepared to write the Z80 myself from scratch if I needed to, although following Professor Lucia's advice about open sourcing and my own personal opinions that I would not learn as much from this class if I needed to write another CPU coming right out of Computer Architecture, it would eat up a very significant portion of time and would have been very easy to introduce other bugs by writing our own CPU. So the first thing I did was some researching on the Z80 and if there were any open sourced versions of it online. Luckily enough for me there was the A-Z80 on opencores which looked promising.

After downloading it and experimenting with it (as described above) I was confident by around the start of October that the A-Z80 would work for us as our CPU. Next steps were to instantiate it and figure out how exactly to use it.

I sat down with the Z80 manual at this point and really started digging into what it was doing in it's different states of operation. It took one, maybe two weeks to write the necessary FSMs to interface my block RAMs with the Z80 so I could communicate to the other chips what was happening and what they needed to do.

After the Z80 controlling was done I needed to track down the BIOS, make that into a coe file for Vivado to interpret, and get that up and running. The BIOS I found on *SMS Power!* was a .sms file, and when I tried to open it in vim I got literal garbage. I needed to convert the .sms file into the actual hex codes, so I wrote a python script to open them up and convert the raw data into the format "de ad be ef" for example. Then I just needed to go in and add the "memory_initialization_radix=16;" on top, set that long data stream as the memory_initialization_data and it was done. Once I converted my .sms file to a .coe file, I did a quick check on the *SMS Power!* to verify the correctness of the BIOS, and then loaded it up.

At this point I noticed the issue with the Z80 and how it was initializing its internal registers to 0 rather than X, and wrote my own assembly to fix that.

So now I had the BIOS running, but quickly realized that it required VDP interrupts to function correctly. This was right before integration started, so I could not verify the correctness of anything in the BIOS past that point, although I verified the BIOS was working correctly up to that point. I started working on paging bigger cartridges, which I was working on up until integration started.

Once we started integrating, there were a few interfacing issues immediately regarding the timing with the VDP, so I helped to resolve those. As the Z80 was running the BIOS, a lot of the early integration problems were on my end and issues I needed to take care of as neither Celeste or Suzz had touched anything related to the Z80. After we were reasonably confident that the Z80 was working and shifted into debugging the VDP and PSG, I bounced between trying to get paging working and acting as a "rubber duck" a lot of the time for debugging the other systems, as I had little understanding of them (and even after finishing, I still have a lot of gaps in my understanding of the VDP in particular). I got paging working during the last week, and then we put everything together and that was it.

At the start of the semester, I was balancing job searching with my academics, and so I did not put as much time into this project as I would have liked. I'd estimate in the beginning of the semester I spent around 5 hours a week working on anything related to the Sega Master System. This number grew over the course of the semester, especially in November, when I was finished job searching. At the start of November, I was probably spending about 10-12 hours a week working on this project. After

Thanksgiving happened, we started integration, and I more or less spent every free minute I had in lab, probably about 60 hours a week for the last two weeks.

Overall though, I very much enjoyed this class. It was really satisfying seeing what we did come together, involving multiple chips, real controllers, and real games with accurate sound and graphics. I believe that the Sega Master System is an appropriate level of complexity for a group that wants to get a totally complete console. Anything more complex runs the risk of having to leave parts incomplete or buggy. If for some reason I needed to repeat my senior year and take another capstone, I definitely would take 18-545 again.

6.3. Suzz Glennon

When we first began this project, we knew very little about the amount of work each part of the project would entail. We decided early on that each of the three of us would take charge of one of the three main chips - Z80, VDP, and PSG. I was assigned to work on the PSG. In the first week, I spent most of my time researching and writing notes on how the PSG worked, as I tend to like to research and understand something more than most people before I jump into writing it. I had not begun implementing the PSG by the time Celeste realized how much work the VDP was, and we as a team decided my efforts would be better focused working alongside her on that instead. As she had already had some time to begin researching and understanding the VDP, I quickly felt fairly behind in my understanding and for a while I felt that I was not especially helpful or useful, as I was often just doing research or asking Celeste questions.

When we started writing code for the VDP, I felt a bit more helpful. Celeste was able to answer my questions while I was able to edit and clarify the FSM she designed for the VDP-Z80 interface, and soon implement it in SystemVerilog. I think our system of working together on the different sections, rather than fully dividing up the work, worked well, as our different strengths complemented each other and together, I think we were able to fully understand and avoid many possible bugs that could've come from one person's misunderstanding.

After we wrote the Z80-VDP interface, we did more research on how the graphics worked. We spent a lot of time (too much, probably) studying the well-documented TMS9918 rather than the less well-documented, but more accurate, SMS VDP documentation. This led to us wasting a lot of time learning about things that we ended up not having to implement in our final design. Eventually, however, we did realize this and were able to move on to implementing reading patterns from the VRAM and displaying them through VGA.

After patterns were working fairly well, we implemented sprites. At this point, we had a pretty good system worked out, and sprites were handled fairly smoothly. We were,

however, very behind schedule and had not even begun to implement audio, so at this point, Celeste let me take over finishing up the VDP while she handled the PSG. She was still available for me to bounce ideas off of or ask questions, so this didn't slow us down much and I was able to finish up the small things such as pattern transformation. However, I also ended up having to go back and rewrite our pattern handling module, as the way we had written it was actually incompatible with the way we had implemented sprites.

At this point, in the last two weeks of the semester, we were finally able to meet back up as a team and integrate the VDP with the Z80. I spent many hours fixing bugs, graphics or otherwise, alongside Celeste and Jeremy, and our hard work in the last weeks paid off. The last part I was in charge of was integrating controllers and designing and building the case and frame for our system, which, while not technically very difficult, did take up some time, although I am still glad I did it as it made our project more polished.

Timewise, at the beginning of the semester, I don't think I was as good of a team player as I should've been. I spent very little time in lab in the first couple weeks, and Celeste was very helpful in motivating me to work together with her. I also had job interviews and was very busy on the executive board for an extracurricular organization, so I was often unable to come to lab even when I wanted to. As time went on, I spent more and more time working on the project, but that time was still highly variable - I would often be unable to work at all outside of class from Monday-Thursday, and then spend 8 or more hours in lab at once on Friday to try and reach a certain goal. I don't think this was necessarily a bad thing, as I'm very goal-oriented and having specific, attainable things to complete on a certain day was very motivating for me and allowed me to complete all sorts of different parts of the VDP. I think I steadily increased from about 6 hours a week in lab to 18 hours a week, until the last two weeks, where I spent closer to 40 or 50 hours a week working on the project.

I had a great time in 545 and learned a lot. I was very worried going into it, as I don't tend to have the best time management skills, and usually prefer to have regular, small assignments, as opposed to long, self-managed projects. I am very grateful we decided on a project that I enjoyed working on, as I don't think I would've been as motivated or able to complete things if I hadn't enjoyed it as much as I did. Because of this, and because I was so excited about seeing the end product, I was even more invested in making each part of the VDP work, and it was so satisfying each week to watch a new piece working and displaying correctly. I was incredibly proud of our final project and I'm very happy to be able to say I contributed to making it work so well.

7. Acknowledgements

We wouldn't have been able to do this project without *SMS Power!*, a website with an "international force of enthusiasts" interested in the Sega Master System and other related consoles. In particular, these documents were indispensable:

Software Reference Manual for the Sega Mark III
http://www.smspower.org/Development/SMSOfficalDocs

VDP Documentation by Charles Macdonald:
http://www.smspower.org/uploads/Development/msvdp-20021112.txt

SN76489 Documentation by Maxim
http://www.smspower.org/Development/SN76489