

18545 Final Report
Fall 2013
Team OneMoreThing

Project:
Super Nintendo Entertainment System

Team Members:
Saketh Pothireddy
Niharika Singh
Delvis Taveras
{spothire, niharika, dtaveras} @andrew.cmu.edu



We give Professor Nace and the 18-545 staff permission to post this report online.

Contents

[Contents](#)

[The Original System](#)

[The SNES](#)

[The Original Block Diagram](#)

[The Video System](#)

[The Audio System](#)

[The Cartridge](#)

[The Lock-Out Chip](#)

[The Memory Addressing](#)

[The Controller Inputs](#)

[The CPU](#)

[Our Implementation](#)

[The Modified Block Diagram](#)

[The Modified Modified Block Diagram](#)

[The Video System](#)

[The Audio System](#)

[The Cartridge](#)

[The Controller Inputs](#)

[The CPU](#)

[Memory Accesses](#)

[DMA and HDMA](#)

[Logistics](#)

[Partitioning the Work](#)

[Scheduling](#)

[Tools Used](#)

[Design Methodology](#)

[Testing and Verification Methodology](#)

[Status](#)

[Lessons Learnt](#)

[Pivotal Decisions and Their Results](#)

[Words of Wisdom](#)

[Individual Pages](#)

[Saketh Pothireddy](#)

[Niharika Singh](#)

[Delvis Taveras](#)

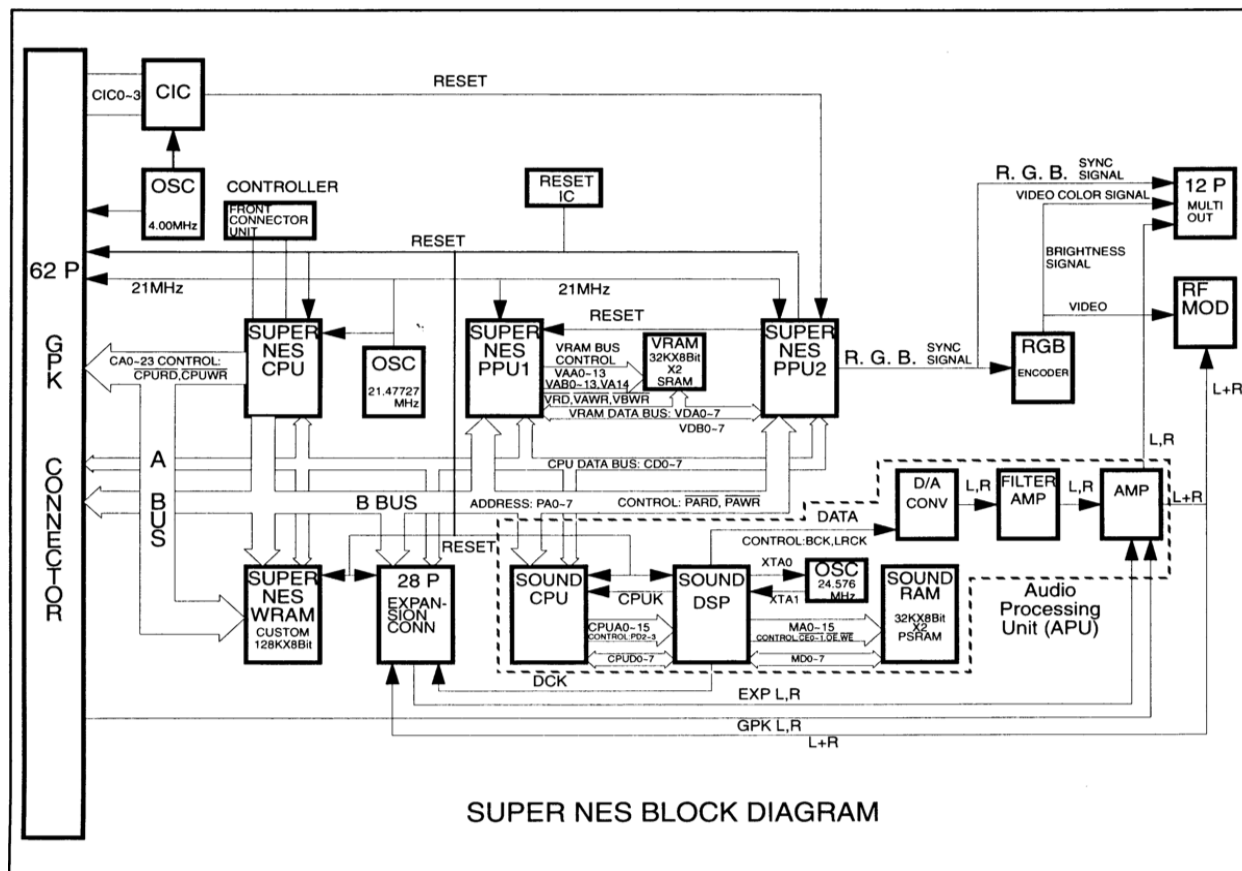
[Code](#)

The Original System

The SNES

The Super Nintendo Entertainment System was released by Nintendo in 1990-1992 in different parts of the world. This is a 16-bit gaming console that can support two players. It also supports a variety of cartridges

The Original Block Diagram



The Video System

The on-board video is implemented by two Picture Processing Units (PPUs). One PPU is primarily responsible for the background while the other is responsible for sprites. These PPUs are supported by three types of RAM: 64 kB of video RAM (VRAM), 544 bytes of object attribute memory (OAM) and 512 bytes of color generator RAM (CGRAM).

The maximum resolution possible is 512 x 478 pixels. However, smaller resolutions are possible. Super Tennis, for example, has a resolution of 256 x 240 pixels. The colors are represented in memory using 15 bits with 5 bits each for red, green and blue. Graphics on screen consist of a background with sprites drawn on top of it. The background tiles are stored in VRAM, from which colour information can be located by the PPU. At the beginning of every game, every time the screen changes, and every time the PPU isn't actively outputting colours to the screen (i.e. in video blanking modes) DMA and HDMA transfers take place on a maximum of eight channels to load new video data into the VRAM. This colour information points to specific colours present in the CGRAM (which contains a palette of colours). The colours can be added or subtracted to form a layering effect. These colours are then drawn on screen. The background is implemented with many 8x8 pixel tiles. Background is present on one of two layers: foreground or background. Scrolling registers in the PPU can be set to translate the background slightly each frame to create a scrolling effect for the user.

Sprites can be of many sizes, and use data stored with VRAM and OAM. The OAM is a small table containing pointers into the VRAM where data about sprites is stored. The OAM also contains some other small pieces of information related to rendering the sprites, like whether the sprite should be flipped vertically or horizontally. Sprites can be drawn on top of each other for depth effects. There are four layers that sprites can be present on. There is a limit to the number of sprites that can be drawn on screen at a time, and there are certain idiosyncrasies as to how they overlap with backgrounds depending on which has a higher priority.

The Audio System

The audio system for the SNES consists of two chips: the SPC700 and a DSP. The SPC700 is an independent CPU responsible entirely for running and processing the sound program. This chip then transfers the actual sound data to the DSP which applies different effects to the sounds that make it into music. This data can then be output to the user. These two chips share a working RAM sized 64kB.

The audio system only communicates with the CPU using four registers shared by both systems. These registers are used by the CPU to copy the sound program files (in the .spc file format) from the game ROM to the SPC700. The SPC700 can then take over execution and work independently to create and output the sounds. This lets the CPU handle other tasks instead of having to continuously output sounds.

Using the DSP allows the game designers to save ROM memory. The game designers can simply store some basic sound samples and rely upon the DSP to mix them and apply filters to create numerous tunes with the same basic samples. The DSP can support effects like ASDR envelope control, Gaussian filters, echos, frequency and pitch modification, delays and volume control. The DSP can also mix sounds on up to 8 channels simultaneously which allows for richer sounds.

The Cartridge

Cartridges come in different shapes and sizes depending on their memory size. They are broadly divided into two categories: LoRAM and HiRAM. LoRAM cartridges have a smaller connection area with the SNES and a smaller memory. Cartridges contain a small battery.

The Lock-Out Chip

Cartridges contain lock-out chips designed to ensure games only run on consoles when the location for both match. That is, American games can only be played on American consoles. The cartridge lock-out chip must complete an undocumented secret handshake with the console chip before the rest of the chip can be accessed.

The Memory Addressing

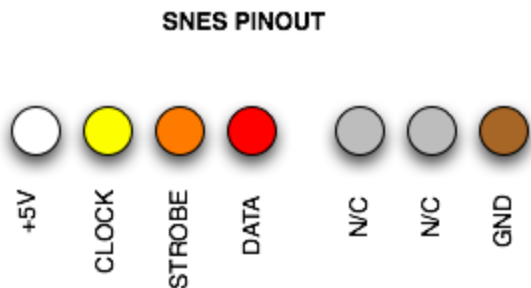
The CPU has access to memory from a variety of other modules and inputs. The memory addressing ensures that each possible source of information has a specific address that the CPU can refer to. The CPU memory has 16-bit addresses, and can be present in one of 256 banks.

- All addresses between 0x8000 and 0xFFFF in all banks refer to information present on the game cartridge. Different banks translate to different offsets on the cartridge, and the translation scheme is different for LoROM and HiROM games.
- Addresses between 0x0000 and 0x1FFF in all banks refer to the stack used by the CPU, which is usually initialized to 0x1FFF.
- Addresses between 0x2100 and 0x213F refer to PPU registers. This includes registers used to setup the video effects, and registers used to read from and write to the various RAMs inside the PPU.
- Addresses between 0x2140 and 0x2143 are used to communicate with the audio units and transfer the sound programs to them.
- Addresses between 0x2180 and 0x2183 are used to write to and read from specific addresses in the CPU RAM.

- Addresses of the form 0x42xx are used to setup various parts of the CPU, to read from the game controllers and access the multiplication and division units.
- Addresses of the form 0x43xx are used to setup DMA and HDMA channels.
- Addresses in the banks 0x7E and 0x7F are used to access the CPU RAM.

The Controller Inputs

The controller pins are laid out in the following manner:



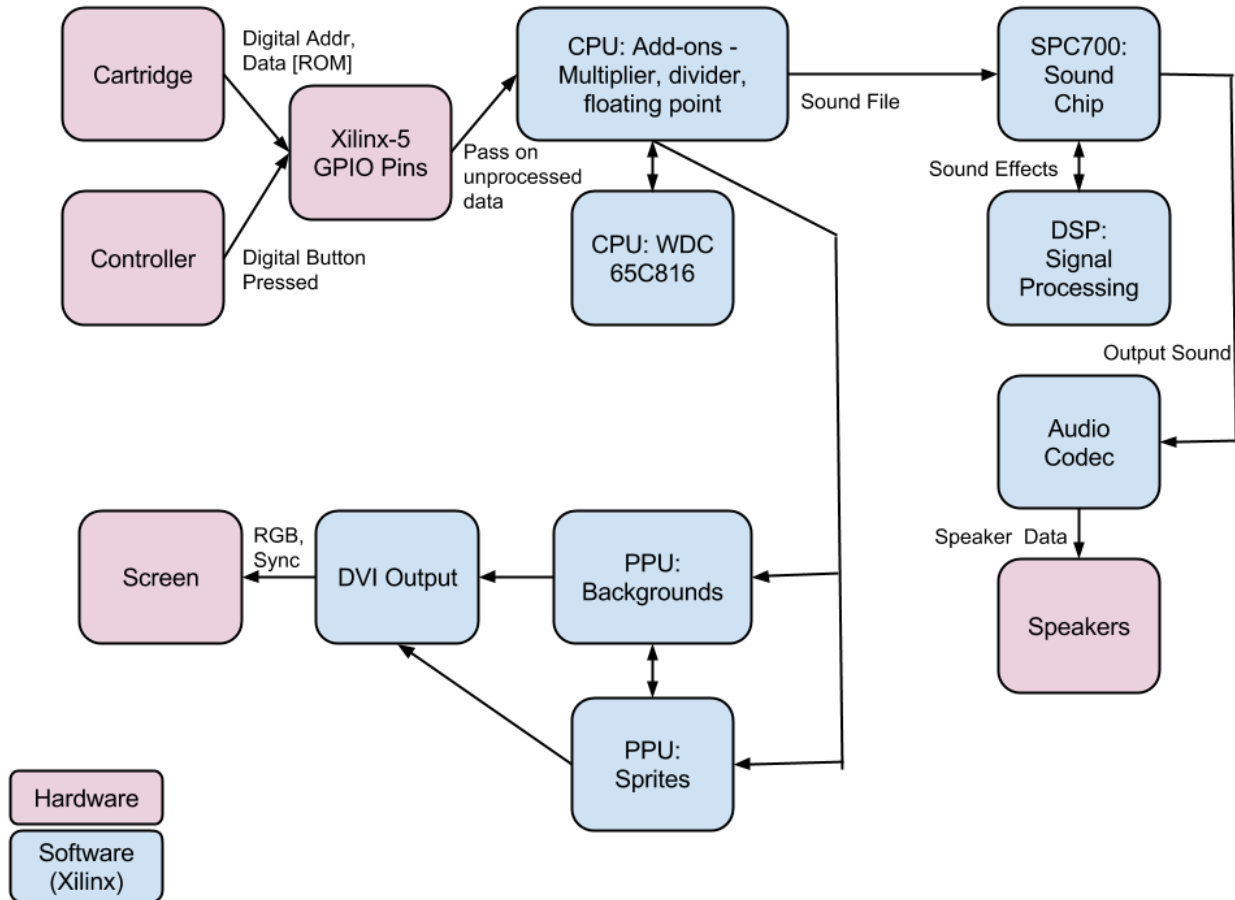
The controller must be fed a special clock by the CPU. This special clock consists of a latch signal followed by 16 quick pulses. The controller will then pull down the data pin voltage from high to low for the cycles corresponding to the buttons pressed on the controller.

The CPU

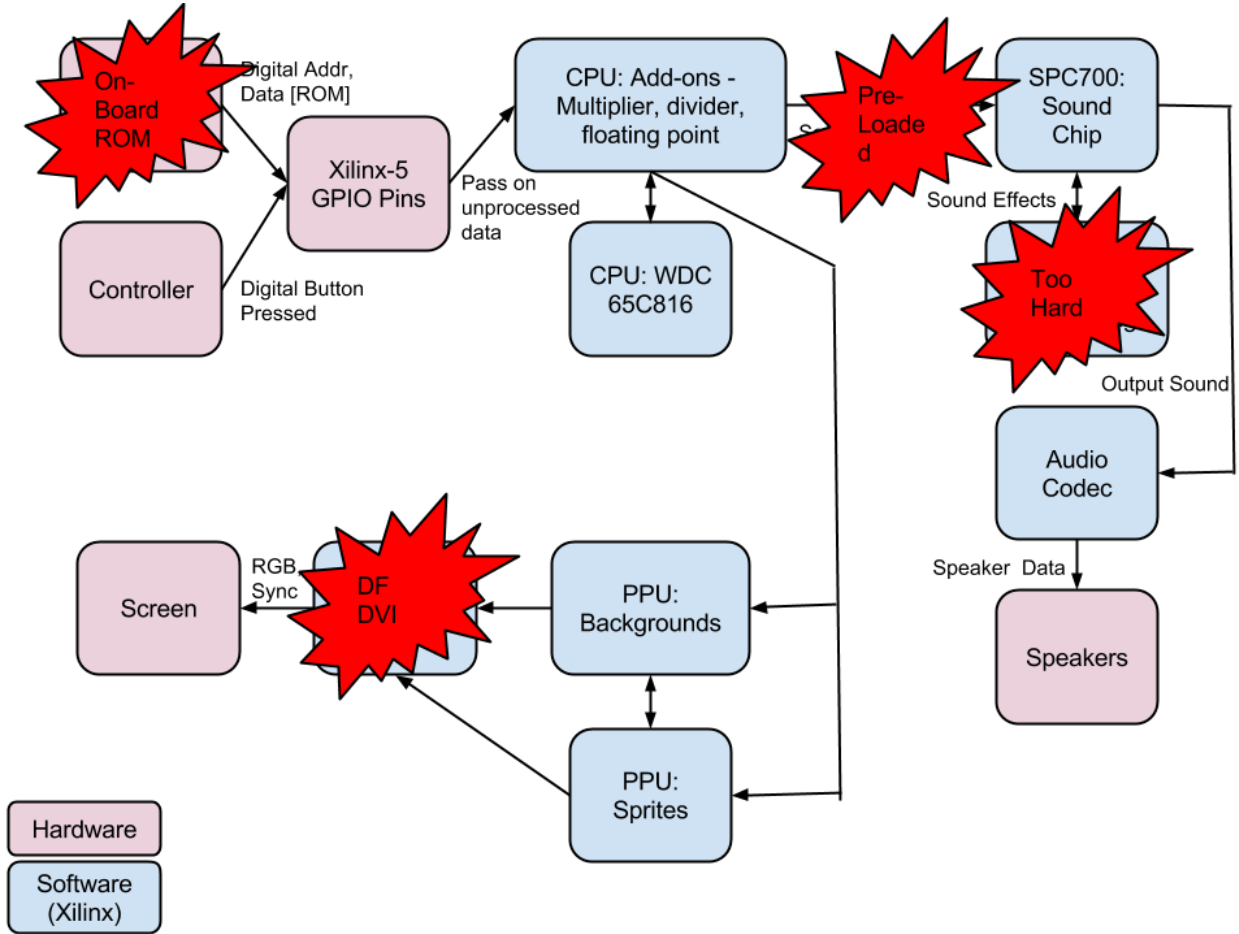
The CPU used in the SNES was a Ricoh 5A22, which was based on a 16-bit processor named 68516 created by the Western Design Center. The underlying design by WDC is responsible for implementing opcodes, interrupts and execution. The parts added by Nintendo to this underlying processor serve to implement computation units for multiplication and addition, provide RAM and stack space, take care of memory management to peripherals and other chips, provide address and data buses, provide DMA and HDMA channels and implementations, and have registers needed to provide specific functionality within the SNES.

Our Implementation

The Modified Block Diagram



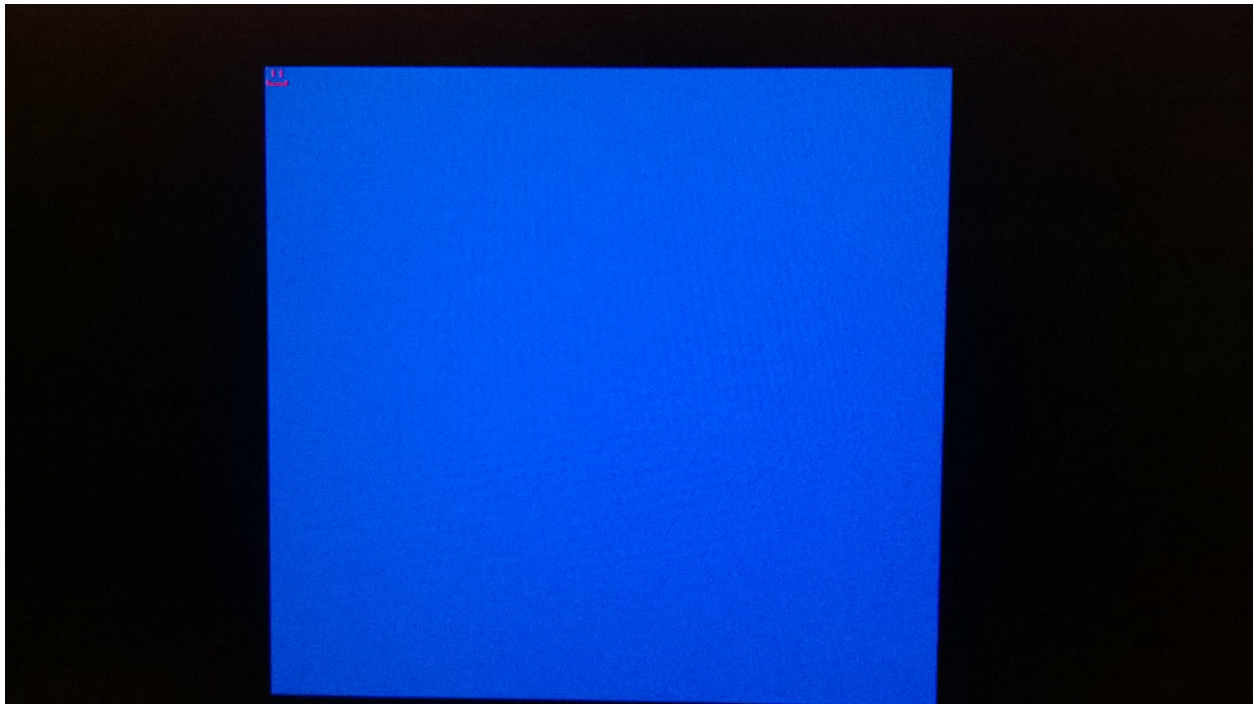
The Modified Modified Block Diagram



The Video System



Test bars output using just the DVI module



Background with smiling face in top left corner generated using PPU

Our team created a working DVI module using the Chrontel datasheet and other specification and timing documents. However, we found out in the middle of the

semester that this DVI module only worked intermittently. Hence, we switched to using the DVI module that was created in a previous year by Team Dragonforce. Comparing these two modules showed us that our module had been correct apart from the constants we used for the blanking intervals in the sync signals. The intervals we had coded were identical to those provided in the Xilinx and Chrontel documentation; however, they did not work with the monitors provided in lab. The game we attempted to build had a resolution of 256 x 240 pixels, so we created a wrapper for the video module that output a black color to the DVI module outside of the game screen, since the Chrontel DVI module was configured to output a resolution of 640 x 480 pixels.

The main PPU system code was found in partially completed form online. The code we found was given in VHDL and was only about half complete. We were able to take this code and use it as a foundation to implement backgrounds completely and sprites partially. Apart from fixing holes in the code itself, we also had to add in memory arbitration units to account for the fact that the CPU, OAM and the drawing unit would all want to access VRAM and CGRAM simultaneously.

To provide RAM to the PPU, we synthesized Block RAM cores using the Xilinx Core Generator. Using this module, we were able to display various images on screen when using just the video module. However, we had difficulties with integrating this video unit into the entire system due to the video unit running on a different clock which caused some errors when the CPU tried to write to the video unit.

The Audio System

Early in the semester, we were able to code the entire SPC700 sound processor CPU instructions in Verilog by using emulator code as a reference. However, after implementing all the instructions, we realized that the SPC700 relied heavily on the DSP chip and would not be able to output any coherent sounds without using the DSP. The DSP chip had to implement sound effects for the sound to be created from the programs loaded onto the SPC700 chip. Unfortunately, none of the team members had enough experience with sound or DSPs to be able to code the DSP in a reasonable amount of time since it was a very complex and specialized chip with little documentation. We were also not able to find any reference code online for the DSP.

Hence, we decided to change the design to implement sound in our system. Instead of loading the sound programs and running them to generate the sound, we found the music data online and loaded this into flash memory on board. Due to the size of sound

files, we only used the sound for one channel (of left and right channels) and used a reduced loop time to be able to fit it into the given memory.

We were then able to simply run this music using the AC97 sound module created in Lab 2. Similar to the DVI module, our team created its own audio module which was fully functioning and used to output the sound.

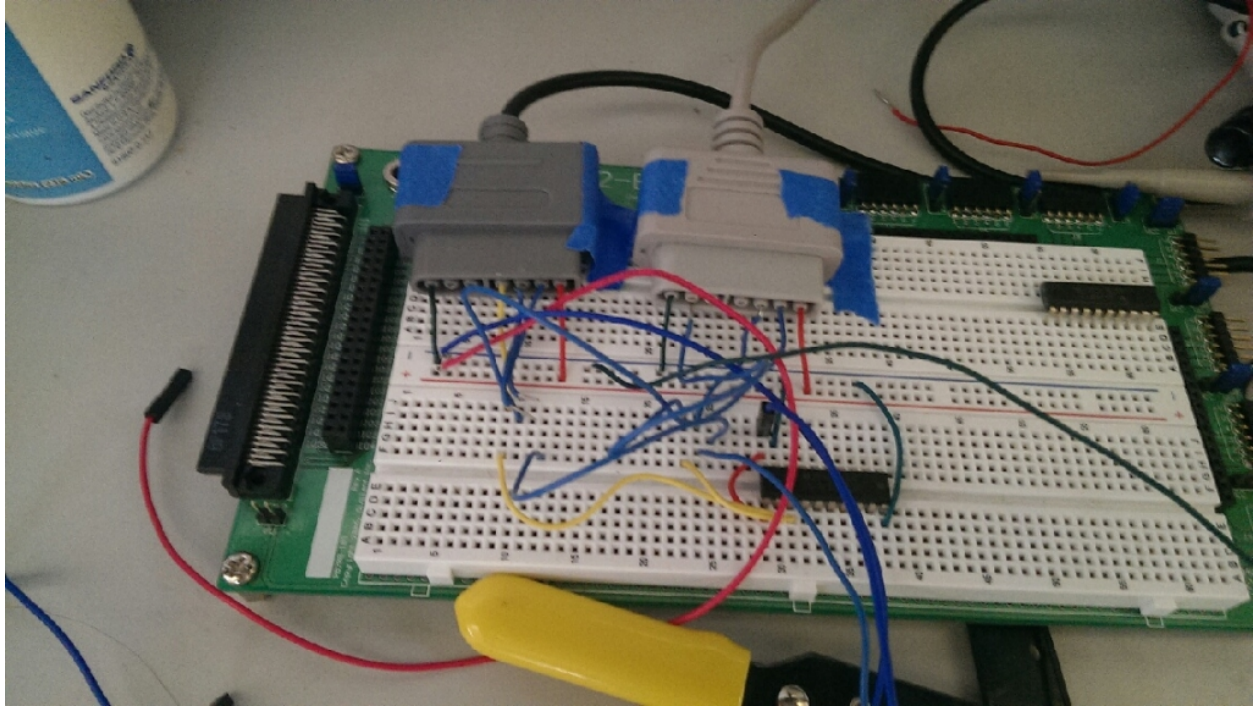
The Cartridge

Our team initially wanted to read the game ROM data directly from the cartridge. We were able to create a cartridge reader circuit. However, when we attempted to power up this circuit the game cartridge would become excessively hot. The documentation we had offered conflicting information as to which pins were connected to what on the cartridge. Since we were not able to find a combination of connections that accessed the cartridge without almost burning it up, we changed our design to load the game ROM data onto on board flash.

To do so, we first obtained the game data online, where it was quite easily found as these same files are used by emulators. We then converted this binary file into a .mcs file that could be loaded onto flash memory. Finally, we created a memory accessing module that would access the same flash memory chip to obtain both sound and ROM data without causing excessive latency for either request.

During debugging, we also loaded the game ROM into block RAM so we could see the memory access results during simulation and ensure we were executing the correct instructions.

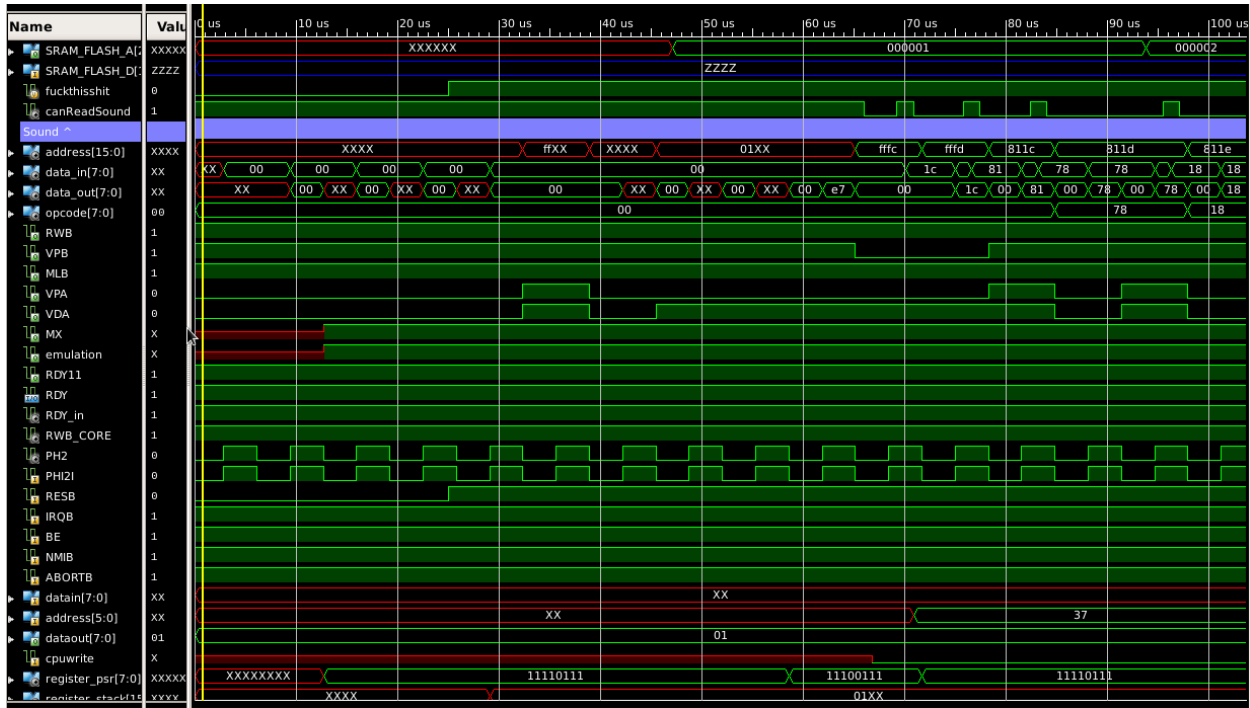
The Controller Inputs



Controller circuit

We used wires and a breadboard to create a circuit that would be able to read controller data. A voltage shifter was needed to translate the controller voltages to board voltages. The wires were connected to GPIO pins on board. A controller reader module then stored the button information in the appropriate register. We were able to use an oscilloscope to test the controller signals and verify they looked as expected.

The CPU



CPU sending out reset vector, labelled address[15:0]

Our team was lucky enough to receive the 68516 processor’s Verilog description from WDC after reaching out to them and agreeing to sign a NDA. We did not modify this code apart from some small syntax changes to make this code run well with the Verilog parser we used. To setup and use this CPU, we had to ensure the following:

- The clock had to be constant before the CPU reset could be de-asserted.
- The CPU sent out addresses when the clock was low, and expected data back when the clock was high, so we had to ensure we would meet these timing deadlines.
- We had to turn the CPU off when executing DMA and HDMA accesses to prevent address and data bus collisions.
- We had to send the CPU the correct interrupt and enable signals to ensure it ran.

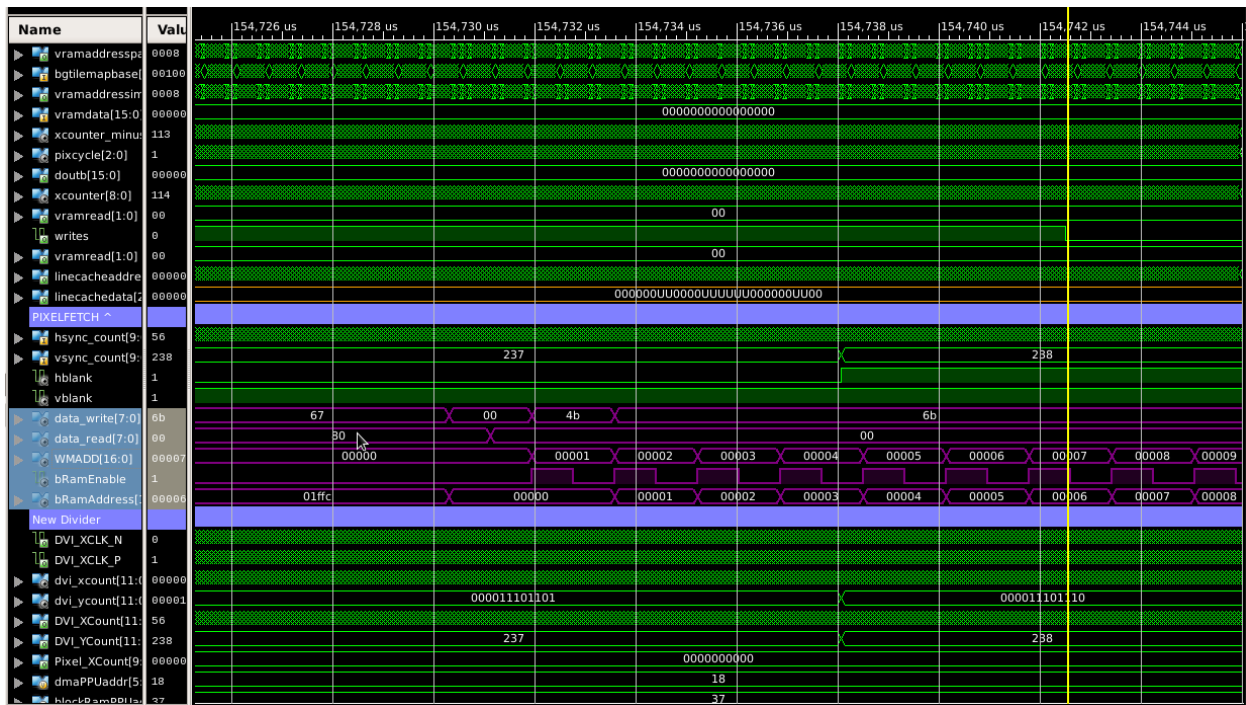
Memory Accesses

The memory accesses for the CPU were handled in a wrapper module for the CPU. They were implemented by using various flags to direct addresses and data to the appropriate destinations, and wait state in the FSM to ensure we met the CPU’s timing requirements. Specifically:

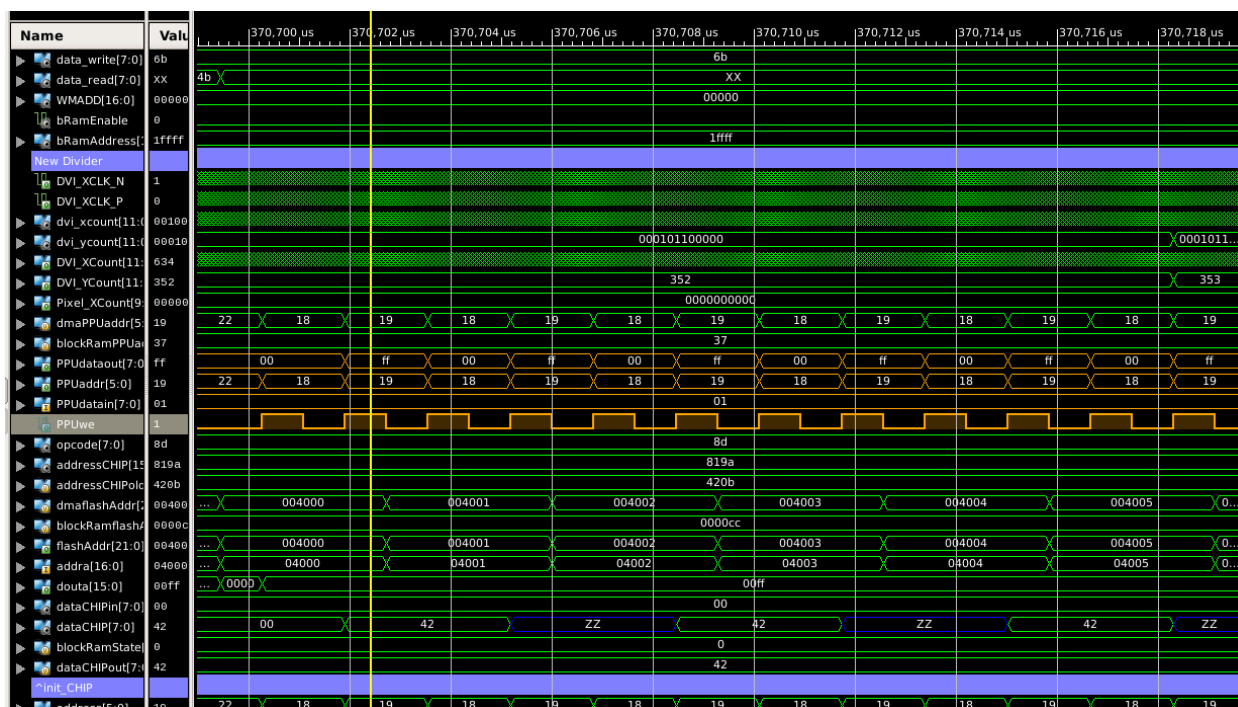
- The stack and RAM for the CPU was implemented using Block RAM.
- The game ROM addresses were translated to their location on the flash memory. We had to account for the fact that the flash memory was stored on 16-bit word boundaries, and the CPU could operate in both 8-bit and 16-bit modes.

- The PPU registers were implemented internally within the PPU module, so those addresses and data were simply passed on to the PPU.
- CPU registers were implemented using Verilog registers and continuous assignments to ensure all related registers would change appropriately (for example, the multiplication result register would change when the multiplicand registers were written to).
- To be able to use variables to select register bits, we had to use a new Verilog compiler within ISE. This meant that we had to change the syntax of some other statements, especially those involving arithmetic in all our modules.

DMA and HDMA



DMA transfer happening to CPU RAM (purple signals)



DMA transfer happening to PPU (yellow signals)

The DMA and HDMA channels were an integral part of the system as they were the modes used to transfer information from the game ROM to the PPU and the SPC700. The peripheral modules could not work without having information transferred to them. DMA and HDMA were used instead of CPU instructions due to their greater speed. The CPU was paused during these instructions to allow them to execute with the greatest speed possible. The difference between these two types of transfer was that DMA happened as soon as instructions asking for DMA were executed, while HDMA happened only when the PPU wasn't actively drawing to screen (i.e. during the screen blanking intervals). HDMA transfers took precedence over DMA transfers. Both types of transfers could happen on one of eight channels, and hence eight such transfers could be queued by the game.

These instructions were implemented with a very large FSM that took into account the pausing of the CPU and the priority of HDMA and channels. The FSM also took into account the fact that certain registers related to DMA could be written to by both the CPU and the DMA controller, and hence would need to be appropriately shared and updated to ensure no new information was lost. DMA was tested in simulation to send information to both the PPU and the CPU RAM from the game ROM, PPU and CPU RAM.

Logistics

Partitioning the Work

Due to the changing design of the project as the semester progressed, the responsibilities of each team member changed. This is indicated in the chart below, the first two columns of which were included in the mid-semester report.

Team Member	Initial Responsibility	Actual Responsibility
Delvis Taveras	PPU, Continue DVI development	PPU
Niharika Singh	SPC700, DSP, DVI output module	SPC700, Flash memory, Sound output, CPU, Memory accesses, DMA/HDMA, Initial DVI, AC97, Reports
Saketh Pothireddy	CPU, CPU modifications, Hardware interface	Flash memory, sound and video integration, Debugging, Cartridge HW, Controller HW and code

Scheduling

We created a schedule in the first week of class to guide our work. We had learned from Professor Nace's introduction to the class that the most common mistake teams made was not beginning work early, and we really wanted to avoid that mistake. We had also learned from the introduction that to do a good job on the project, we would need to set aside a lot of time to polish and debug the project at the end. With this in mind, we created the following schedule:

Date	Milestone
13 October	Controller communications code finished, PPU research finished and coding begun, SPC700 CPU code finished
14 October	Design Review Due

4 November	Coding and Unit Testing Finished
21 November	Integration Testing Finished
2 December	Polishing Finished

This schedule was shared using an online calendar we all used. The shared calendar also contained milestones for the class such as due dates for the labs and reminders to submit weekly reports.

Towards the end of the semester, we needed tighter scheduling controls, so we created a shared spreadsheet with a list of tasks yet to be completed, who was responsible for completing them, when they would be completed by, and reasons for why the deadline had not been met (if applicable). These tools greatly helped us organize our time.

Tools Used

- ★ Board: Xilinx Virtex 5 110T
- ★ Design Environment: ISE 14.2
- ★ Code Repository: Github
- ★ Document Repository: CMU Box
- ★ Collaborative Assignments (ex: Reports, Presentations): Google Docs

Design Methodology

Since this was a project with well defined specifications to work towards, we had a waterfall approach for our code. Each team member worked on their own section of the code, with collaboration occurring when discussing how to implement tricky sections of the code or how to debug malfunctioning code. We used documentation about the SNES found online, along with emulator descriptions to guide our coding.

Testing and Verification Methodology

Our testing followed the following steps:


- ★ Find out what the correct output should be from documentation.
- ★ Ensure code has no syntax errors.
- ★ Ensure code simulates waveforms identical to those given in documentation.
- ★ Put code on FPGA board and connect inputs and outputs to switches and LEDs respectively. Ensure results are as expected.
- ★ Run code on FPGA.

- For video, ensure results from FPGA are identical to those shown by loading identical instructions into an emulator.
- For audio, ensure sound coming from FPGA code sounds like original sound file.
- For memory/CPU, the LCD screen was used to output the current execution address and instruction for further debugging. For certain stages of the project, we would connect the clock to an external button instead of the onboard clock to be able to step through individual memory accesses and learn how the program was running.
- For others, use ChipScope to confirm board waveforms are identical to simulation waveforms.
- ★ Integrate with larger section of design.

If errors were found, we debugged with the following steps:

- ★ Ensure correct setup using documentation.
- ★ Browse Xilinx forums for similar issues.
- ★ Find a similar pre-written working module online and ensure simulation results of both are identical.
- ★ Use Chipscope to see if simulation results match FPGA results.
- ★ Talk to team member to get outsider's perspective.

Status

Device Utilization Summary				
Slice Logic Utilization	Used	Available	Utilization	Note(s)
Number of Slice Registers	788	69,120	1%	
Number used as Flip Flops	772			
Number used as Latches	16			
Number of Slice LUTs	1,106	69,120	1%	
Number used as logic	1,099	69,120	1%	
Number using O6 output only	807			
Number using O5 output only	105			
Number using O5 and O6	187			
Number used as exclusive route-thru	7			
Number of route-thrus	112			
Number using O6 output only	112			
Number of occupied Slices	404	17,280	2%	
Number of LUT Flip Flop pairs used	1,178			
Number with an unused Flip Flop	390	1,178	33%	
Number with an unused LUT	72	1,178	6%	
Number of fully used LUT-FF pairs	716	1,178	60%	
Number of unique control sets	34			
Number of slice register sites lost to control set restrictions	44	69,120	1%	
Number of bonded IOBs	97	640	15%	
Number of LOCed IOBs	94	97	96%	
Number of BUFG/BUFGCTRLs	6	32	18%	
Number used as BUFGs	6			
Average Fanout of Non-Clock Nets	4.60			

Utilization Information



Case for demo day

We were able to play the sound, run the program on the CPU and process controller input simultaneously by the end of the project. This was verified by having sound playing on speakers, while we used the LCD screen onboard to verify the execution addresses and instructions were changing, and the oscilloscope to ensure controller inputs and outputs looked as expected.

We were also able to display background video tiles on the monitor when we loaded the PPU with the relevant information directly. We verified this by displaying a smiling face and a tic-tac-toe board on screen. We were also able to send the correct data transfer signals to all modules, which we were able to verify using simulation waveforms.

However, our team was not able to integrate our PPU module with the CPU module due to timing errors between the two modules. Hence, we did not have working video for our demo. We're fairly certain this was because the two modules used different clocks (the PPU had to run on a clock that allowed it to have a 60Hz refresh rate on the screen, while the CPU from WDC could run at a maximum of 10MHz and had to run at a lesser speed to allow memory accesses to meet timing deadlines).

Lessons Learnt

Pivotal Decisions and Their Results

- Debugging DVI module: Since our DVI module worked intermittently, and matched provided documentation exactly, we had a very hard time figuring out exactly why it didn't work consistently. We spent a week and a half towards the end of this semester trying to fix this problem, and this was time we could have used for more important things.
- Beginning work early: This was a very fruitful decision on our part as our project underwent several design changes (DVI module, pre-loaded sounds, no physical cartridge) that would have been difficult to all implement under a time pressure.
- PPU planning: We would probably have been able to get further on integrating this module if it could have been finished and ready for testing earlier, or if more than one team member had been available to help with this. This base code we found for this was written in VHDL, which meant Delvis had to teach himself VHDL from scratch, and the other two team members couldn't help out much with debugging due to lack of familiarity with the language.

Words of Wisdom

- Do not attempt to do everything yourself: Our team implemented its own sound and video modules, and while this was a good learning opportunity for us and let us have an early success, it also consumed time that could have been better spent elsewhere. Take advantage of previous teams' efforts and ask to access/use their code whenever possible.
 - Use all resources: Don't just limit yourself to online documentation. Ask other classmates for help and learn from their implementation decisions (especially for common modules like sound and video that all teams implement in some fashion). Also, always mail sources for the information you find online, as they may be willing to help answer your questions and provide more personalized help.
- Work together in lab: Ensure all team members are in the lab for all work sessions, even if everyone is working on different modules. It ensures everyone is on the same page, and that questions about the interfaces between modules can be resolved quickly and correctly.

Individual Pages

Saketh Pothireddy

The first few weeks of the semester were spent on the introductory labs. I worked with the rest of my team on getting the LCD screen working, as well as getting the sound working. I generated the Block RAM, and built the Verilog interface for the block ram while my teammates worked on the audio controller. We ran into a few issues outputting sound, so all three of us were involved in debugging the audio controller. During this time, we also spent a large amount of time researching various components of the SNES, and determining how best to approach each component.

As we began dividing up the work, and beginning to design individual components, I started to work on the controller interface. The controller needed a specific clock and specific pulses at the beginning of each clock cycle to latch the state of the buttons. I initially wrote the timing module, and then tested the controller output on an oscilloscope. Because of the voltage differences between the controller and the FPGA, I then implemented the circuit for the level shifter, and wrote a module in Verilog to convert the serial data into bit vector that the CPU can use.

I then moved on to work on the cartridge interface for the SNES. I found lots of documentation for the cartridge online, but there was a lot of conflicting information on the proper wiring of the cartridge. I tried various configuration of wiring up the cartridge but I found out that the cartridge overheats within a couple of seconds every time power was applied. This might have been due to the CIC chip that is usually present on the console that communicates with the cartridge.

Around this time Neha was also having trouble with building the sound controller and the DSP chip, and she suggested that we place both the ROM and the sound files on Flash and read it from there. This required reading ROM data most of the time, and reading the sound data at a certain frequency. While we were initially able to get sound working initially while reading ROM data at the same, it took some time to figure out the addressing of the ROM itself. After a little debugging and carefully looking at the ROM file layout, we were able to figure out how to properly address the ROM. Around this time, our DVI module stopped functioning, and we all had to drop everything we were doing to get the module working. It took us a week to get it reliably working again, after we tried everything from debugging our module cycle by cycle, and implementing modules we found online.

Once the flash interface and DVI were completed, I moved onto integration while Delvis and Neha continued to work on the PPU and CPU respectively. First I integrated sound and DVI so that we have something to show for the demo. By this time the CPU was finished and I tried to integrate that in the design as well. The CPU required a lot of

debugging since we had only simulated it up to this point. There were a lot of errors and warnings that needed to be fixed for the design to pass synthesis. Soon after we also integrated the PPU into the overall design and began debugging the system as a whole. I helped implement different ways in which we could debug the system, and helped the team step through the program to figure out exactly what was going on. We fixed a large number of bugs, but at the end of the day we couldn't get the entire system working, which was a little disappointing, but as a whole this class was very memorable and rewarding. I learnt a great deal about working on large scale projects in a team.

Niharika Singh

This was a very rewarding project for me personally, because I was able to contribute to my team and pull my weight which I was initially concerned I would not be able to do. Both my team members had taken Computer Architecture the semester before, while the last time I had seen Verilog had been in 18-240. I was also concerned about implementing a system that a previous team had been unsuccessful with. Considering the size and complexity of the system we attempted to implement, I believe we made really good progress. I also believe that the next team to handle this project could definitely make it work completely, using our code as a starting point.

The semester started with creating video and audio output modules, both of which required a lot of debugging and were essential in making me comfortable with coding in Verilog again. I also read books and online articles about coding in Verilog to ensure I could work with the language. After these two basic modules were done, I began researching the SNES system itself to find out what it would require from me specifically.

A good resource to use was emulator code since it very closely mirrored the operation of the SNES and explained how code could be used to implement SNES functionality. I used an emulator as my primary resource when I began working on the sound system. I was able to finish the SPC700 instruction set code, but realized that implementing the DSP would be a lot harder. I hence decided to implement sound by playing pre-loaded sounds and making use of the already functioning AC97 module to output the sounds. I believe this incident illustrated a strength of our team in that we were able to realize when initial design decisions we made weren't working out and switch strategies as we gained more information.

I then began writing the memory management unit for the CPU followed by the DMA and HDMA modules. This was a really big challenge, and I really enjoyed working on this

problem. The simulation results for these modules looked correct, and I was eventually able to verify their correctness by displaying their instruction traces on the LCD screen.

Finally, I was also given a chance to exercise my creative skills during the course of the class by creating the poster, final presentation, reports and the SNES case. I really enjoyed this class since it gave me a chance to fully devote myself to a very challenging problem and solve several sections of it. It was also a great experience working with my team, since one of my initial concerns was that my team and I would procrastinate on the project till the end of the semester, but we maintained a fairly constant workload throughout the semester and stayed on schedule.

Delvis Taveras

When we first began working on the project we decided to read over previous teams project reports, this turned out to be really useful because we were able to gather a lot more documentation as far as the SNES was concerned. In addition by reading from what previous teams had done we were able to avoid some of the pitfalls that they'd gone through. Initially we divided the work into three sections PPU, Peripherals, and Sound. Later on when Neha finished implementing sound she moved on to implementing DMA/HDMA and adding the missing modules to the WDC core. We started off very well we had lots of documentation and we had sound and DVI working.

Early on I was the one focused on discerning how the whole system worked, which turned out to be a bit overwhelming because I thought that I could initially just read through all the documents then just start coding. This would have worked for any other class however reflecting back on this I should have just started off by just focusing on one part which would have been the Register File, then when I finished that I should have moved on to background and sprites. This something that I truly regret I spend so much time understanding the SNES only to find out that after reading the details of a certain protocol I would forget one of the details that I had read before because it was just too much information to digest at once. After struggling with this when I first began coding I started by writing code in verilog, I started with the register file and as we continued to search for resources online Neha found a VHDL description of about 60% to 70% of the PPU. Initially I began by just referring to this code by reading it and then translating it from VHDL to Verilog. A week passed and I continued coding in Verilog ignoring the elephant in the room. Then I consulted with my teammates because I was making fairly slow progress. Thats when it clicked why would I just read VHDL and translate it why don't I just code in VHDL. It took me about 2 days to pick up the synthax and the execution paradigm of VHDL and I was off and running. Why I had not

just decided to pursue this approach two weeks before I cannot explain I basically ignored the elephant in the room.

After this realizing that there was no point in trying to understand the whole system at the same time and deciding that I should code in VHDL things went a bit smoother. One of the key decisions that we had made early on was that the CPU and PPU timing were going to be the same however after Saketh and Neha figured out the interface to the on board SRAM they figured the CPU had to be much slower than the PPU. I was not initially aware of this so my Register File expected a very different interface. This something that we did throughout the whole project rewrite working code because the interface was not correct. Later on Neha and I sat down and discussed the timing once again and this timing we got it almost right. But I wished we could have discussed it much earlier since the register file was the only communication point between the CPU and PPU we should have made sure that the two sides worked as expected. Finally one of things that hurt us during this time was the fact that the PPU was in VHDL and the CPU was in Verilog. Neha was the most familiar with the CPU which was fairly complicated due to the way in which it was coded by the WDC engineers but because she was not familiar with VHDL. I did not know enough about the CPU to go about this part so in the end what we needed to do check the simulation together so that we could solve the problem we did check it together however we did not do it enough times.

Ok above I have highlighted some of the issues that plagued us throughout the semester and eventually contributed to us not completely finishing our SNES. Now I will proceed to highlight some of the accomplishments of our team. In order to test the PPU I wrote a matlab program that would essentially take background/sprite tiles that would be eventually be loaded into VRAM and draw them. In addition we got 8x8 and 16x16 tiles to draw on the screen. We did have the sprite code but we did not finish integrating it due to the fact that we decided to just debug the CPU/PPU communication for the last few hours. In order to test our CPU we did a variety of things first Neha and Saketh loaded both sound and super Tennis onto the SRAM chip and were able to devise a protocol where they could read instructions and sound. But then I asked that we be able to test it in simulation so then they went about loading the program on to block ram, but our normal sized game did not fit in block ram so we used an assembler and linker to generate a small ROM that would fit onto block ram this worked wonders. The CPU and DMA and HDMA worked correctly you could see it in our demo as we output the instructions onto the LCD. The bug that stopped us from having a working SNES was the fact that when DMA executed the Register file received the data and attempted to right the data to Block RAM however it did not hold it for an entire clock cycle hence the data disappeared. Its certainly a fixable problem its just be found it a bit too late. Finally we

had two controllers working and a sound module that played a loop of about 30s of super Tennis.

In conclusion although I'm very disappointed that we did not accomplish our ultimate goal I'm very satisfied knowing that I had some great teammates in Saketh and Neha. We worked our butts off until the last minute and although many of us had other commitments we found a way to always make it to lab and continue to work. I learned so much not just about coding but about project management and how to approach extremely difficult and time sensitive tasks that I cannot wait for the next challenge that awaits me. In the end we got many parts of the SNES working and if not for one last bug in our timing we would have been able to have a very nice demo.

Code

Due to the fact that we signed an NDA with WDC, our code is not available on a public repository. However, since Professor Nace also signed the NDA documents, we will add him to our Github repository to enable him to look over the code if he just lets us know his account name.