Team N64 Report

Jeremie Kim
Max Regan
Alvin Mao
Prin Oungpasuk

Carnegie Mellon 18-545
Fall 2014

TABLE OF CONTENTS

Section 1 : Introduction

This report details the progress made on the FPGA-based Nintendo 64 project during the semester of Fall 2014. We outline the details of the system, document decisions that were made, provide directions for understanding and using our system, and give insights to future developments that may be made for furthering our project.

Section 2: Purpose, Goals, and Achievements

The purpose of this project is to recreate the modules of a Nintendo 64 (N64) necessary for a cartridge to be runnable with complete input and output. We want to expand the horizons and possibilities of project scopes for this course and hope that future teams won't be deterred to pursue difficult projects. To our knowledge, a N64 has never been written in any HDL, and our goal is to contribute to the expansive but non-comprehensive wealth of knowledge on the N64.

The initial goal for the end of this semester was to be able to have the cartridge for *Namco Museum 64 (*a variety of arcade games on one cartridge including *Pac-Man*, *Ms. Pac-Man*, *Galaga*, *Galaxian*, *Dig Dug*, and *Pole Position*) completely playable by our system with full input (one or two N64 controllers) and output (video and audio). This cartridge did not rely on a graphics processor (the RCP in the case of the N64), which would be a whole other monstrous project in itself. However, this would still require the full implementation of a NEC VR4300 processor, a memory controller and its numerous special sections of memory (a lot of reverse engineering here), VGA driver, audio driver, N64 controller driver, and SSD driver.

By the end of the Fall 2014 semester, each of these individual modules were complete with minor bugs in the memory controller and a slight oversight in the processor that prevented the cartridge from being able to fully run. We were able to get through the entire boot loader, which consisted of several millions of instructions (being able to compare to a working C simulator) and portions of the actual cartridge. However, we came across exceptions in the cartridge for the handling of the drawing of the frame buffer. As this was unexpected, and discovered too late in the semester, we were unable to get *Namco Museum 64* working. However, as a work-around around this oversight, we wrote a game of our own (a custom game resembling *Tron*) that was fully compatible with the system that we had and demonstrated most of the available system's functionalities while also giving us something to show for on Demo Day (I highly recommend having at least something to show for on Demo Day that is interactive).

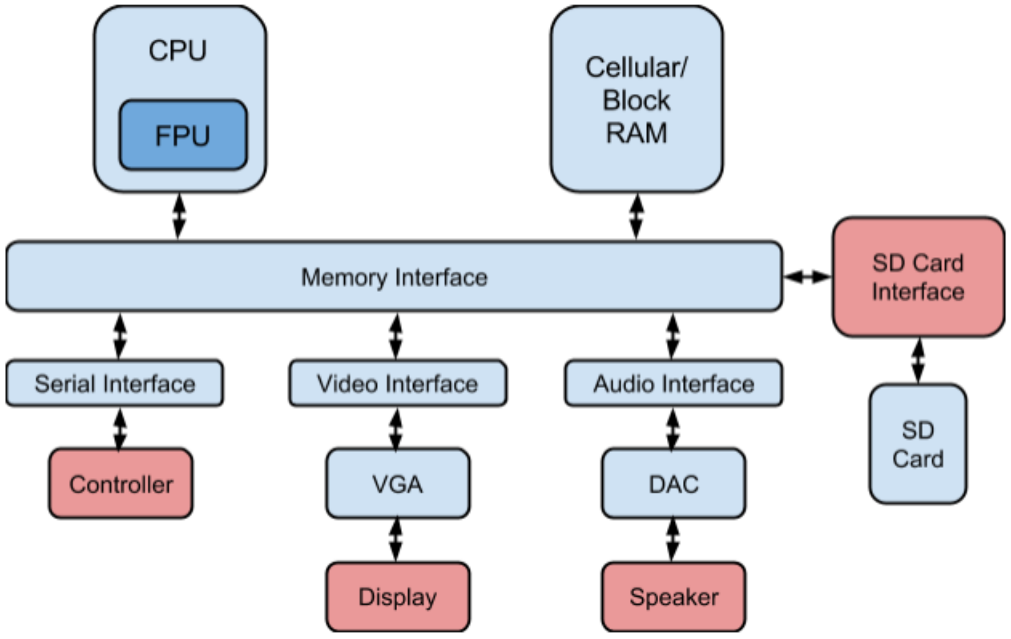
Section 3: System Overview

Our system can be viewed as 6 major subsystems:

1. The main processor that directly interfaces only with the memory controller. This module was created completely from scratch other than small sub-modules including multipliers, dividers, and a majority of the FPU (the FPU is technically considered to be a coprocessor), which were available as hard IP cores.
2. The memory controller that can read or write data to and from any of the memory subsystems and connecting interfaces depending on values written by the CPU to specific memory locations. This also includes the cellular RAM interface, which allows reads and writes to the on-board block RAM. This was written from scratch while referencing the cellular RAM data sheet and a slightly broken open source core.
3. The SD card interface is responsible for handling reads from the SD card. The SD card acts as the N64 cartridge in our system and *Namco Museum 64*. We have a suspicion that some games write to the cartridge for saving in-game data, but we are confident that writes are not necessary for the games to work. However, since SD card initialization requires a couple writes, we do have support to make writes fully integrated very easily. This module was created by fixing a broken open core and adding modifications.
4. The video interface consists of a VGA driver and a frame buffer. The frame buffer holds the data to be displayed onto the monitor where each individual bit is represented in an array in a memory block. Our video interface was tied to an array of memory and the VGA driver was slightly modified from an open source module.
5. The serial interface is responsible for reading inputs from the input controller and writing the controller status to a particular address in memory anytime the CPU writes a command to a particular address as the interface is defined later. The serial interface is very well defined in several documents found in the reference section and was written from scratch.
6. The audio interface takes values generated by the FPU and sends them to the PMOD controller, which generates the sounds, which is also controlled by writing values to particular addresses in memory. The audio interface had to

use a PCM PMOD in order to work around our board's lack of the AC'97 codec.

The figure below shows a diagram of these subsystems and their interconnect.



Section 4: Tools and Components

4.1 Nexys 4 Artix-7 FPGA Board (**Xilinx Artix-7 XC7A100T FPGA)**

We chose this board for both its VGA port and its ability to support SystemVerilog and the Vivado Design Suite. This board gives access to:

- 15,850 logic slices, each with four 6-input LUTs and 8 flip-flops.
- 4,860 Kbits of fast block RAM (a bit small, but was enough)
- Six clock management tiles, each with phase-locked loop (PLL)
- 240 DSP slices (must haves for floating point unit)
- Internal clock speeds exceeding 450 MHz (recommend to create a module that can change clock speeds at the switch of a button: very useful for debugging)
- 12-bit VGA output (A huge plus: VGA is very simple interface for video display)
- 16 MB CellularRAM (similar to SRAM with some caveats)
- 16 user LEDs (very useful for on-board debugging)
- Serial Flash (useful for having a portable system… We were late to demo day because we were trying to figure this out last minute)
- Two 4-digit 7-segment displays (very useful for on-board debugging)
- Micro SD card connector (we used this in place of our cartridge. We have a completely working SD card interface module. Our module only supports SD cards smaller than 2 GB)
- Four Pmod ports (gave us the ability to interface with multiple N64 controllers)

This board was promising because it seemed to have all the benefits of a better Virtex 5, while not being a Virtex 7. This included having access to SystemVerilog and the Vivado Design Suite, as well as being smaller and having an approximately equal number of LUTs and DSP slices. However, it still used a VGA port, which was important for us because we did not want to tackle the Virtex 7's HDMI system. The Nexys 4 also had multiple PMOD interfaces, which we needed if we wanted to interface with multiple controllers.

Although full of qualities, the board may not have been the best choice. It had little BlockRAM and no DRAM, with CellularRAM being used as a substitute. Working with CellularRAM proved difficult and was the source of many bugs, which in retrospect slowed down a lot of our development. However, we now have an available module that allows the usage of the CellularRAM in case future groups are interested in using this board.

4.2 Vivado Design Suite

Vivado was touted as the new design environment, and it seemed to be fancier than ISE. While this quality was true, it also at times proved to be more adversarial than friendly. Vivado's simulator did not play well with SystemVerilog constructs we were accustomed to, such as enumerations and constants. As such, we had to forgo the Vivado simulator for the VCS simulator most of the time, and later use Vivado only for synthesis and implementation.

Vivado also was too smart for its own good. It would optimize out wires and signals that were in use, and throw multiple warnings without much detailed explanation. We spent a lot of time on the Xilinx support forum.

Synthesis and implementation times were also extremely lengthy. However this was not a byproduct of Vivado, but rather of synthesizing any large hardware project.

4.3 Code and Document Repository

We primarily used a private Github repository as our method of source control. The only thing we did not store were Vivado IP cores, as those took up too much space and were easily instantiated. For lab documents and collection of N64 documents, we used collaborative Google documents (a link can be found on the Github README. It is highly recommended to learn how to use Git fairly well, if not just the basics. As much of a pain it may seem to use source control in the beginning, it will pay off for that one time you accidently delete something very important or make a change that breaks everything and can't change it back. Also be sure that everyone in the group is able to use version control. Although it can be helpful to be able to use more advanced techniques, the basics should be sufficient. I would also highly recommend that you comment on each commit very effectively so that you can know very specifically which commit you could revert back to.

4.4 CEN64

With the help of Tyler Stachecki, the creator of Cen64, we were able to acquire two version of the Cen64. One of which was a fully integrated version with the capability of playing full cartridges. The other was a standalone version of the CPU. Initially we were only able to get the complete simulator and that was extremely helpful in itself. We were able to learn how the entire system was integrated and what each module was responsible for. We were able to learn a great deal about the N64 by reading through the code countless times and running the code with print statements scattered through the code.

When working on the CPU, we wanted a better method for testing. This required a standalone NEC VR4300 processor, that didn't run the entire system during simulation, but since the CPU was already very integrated with the rest of the system in the original Cen64, it would be a fairly substantial task to separate it from the rest of the system. I decided to email Tyler Stachecki about seeing whether he had a standalone version of the CPU, and he responded within the day with a standalone tarball of the NEC VR4300, which was a lifesaver. Always reach out to people for help when you can; the worst thing that will happen is they don't respond. With this NEC VR4300 C simulator, we were able to thoroughly test our processor with high confidence of ideal behavior. We were able to make a live tester that would take in a test suite and generate the final register values and compare with our results. With much thorough testing that involved hundreds of test vectors including randomly generated test vectors, we were able to thoroughly verify our Verilog processor before integrating with the rest of the system. This resulted in a very smooth integration with very few bugs coming from the CPU itself.

The Cen64 was the main reason we were able to get as far as we did. Due to the lack of N64 documentation, whenever we had any questions that could not be resolved in specifications, we would boot up the emulator and scrounge for the proper behavior. Many thanks to Tyler Stachecki for his Cen64 Simulator.

Section 5: Sub-modules

5.1 CPU

Before reading any further about my implementation of the CPU, I would highly recommend the reader to have looked through the ISA at least twice and have a fairly strong understanding of it. After understanding the ISA, ready this section of the report and then look through my code to understand the basic layout. Just as a warning, be very careful about any changes you make since most of the code is there for very particular reasons.

In the next few paragraphs, I will explain the organization of the repository related to the CPU. From the top of the directory, the NECVR4300/ testbenches/ and standalone/ are the most important directories. In the top-level directory, N64_top_no_mem.sv is the top module that instantiates the system. I placed the multipliers and dividers in the top-level module because they are IP cores that were instantiated in Vivado. However this was before I thought of using macros for to differentiate between simulation and synthesis. I have two top-level modules: one was specifically used for simulation while the other was for synthesis only. N64_top_no_mem.sv being the module for synthesis while testbenches/ram_tb.sv is the module for simulation. Because ram_tb.sv has a macro SIMULATION defined, I was able to scatter ifdefs throughout my code that would cause the code to act differently based on whether I wanted to simulate or synthesize.

Inside of the NECVR4300 directory is all the Verilog code for the processor and the filenames are fairly self-explanatory as to what exists inside. Further in the NECVR4300 directory is the ROM folder, which holds very useful python scripts as well as some N64 ROMs that can be loaded into the SD card.

- Hex_to_opcodes.py takes in a hex file and prints out the list of assembly instructions that correspond to the hex values.
- Live-disassembler.py allows you to type in hex values into the command prompt and it will return the assembly instruction corresponding to the value.

Inside of the testbenches/ directory exists the files related to testing the processor as well as several more useful scripts.

- Compare_outputs.py is used to compare the outputs of my processor against the output of the cen64 output.

- Coefy.py is used to transform a regular hex file into a .coe file. A .coe file is a simple format that is allowed to be loaded into Vivado block ram cores.
- Diff.py compares two files and prints out the first line that differs along with its line number. This was used for tracking our progress in the boot sequence (millions of lines long). We had a golden output generated by the cen64, and we would diff our system's output against the golden output and check to see how many lines we got through successfully.
- Gen_random_test.py was used to heavily test the processor with 1000+ line programs of random instructions. This was extremely helpful and caught the strangest edge cases that would have been very hard to determine once the CPU was integrated with the system (the more complex your system the harder it is to debug).
- Mips-assemble.py was a homemade assembler for the NEC VR4300. The ones that I found online were all terrible, so I decided to make my own. Mines not the best, but for the purposes of this project, it was satisfactory and well worth the time.
- The programs/ directory holds assembly files that were used for testing the entire system put together. Anywhere from testing the controller input, frame buffer, to testing the memory controller and running *Tron*.
- The cen_tests/ directory holds assembly files for most of the instructions and tests edge cases for these instructions in the processor.

The standalone directory holds the cen64 standalone CPU, which was used to compare outputs of my CPU and verify the processor. Also reading through this code was very effective in learning how MarathonMan implemented certain parts of the processor.

The CPU for the Nintendo64 was a 93.75 MHz RISC 5-stage scalar in-order execution 32/64 bit processor with integrated floating-point unit and internal direct-mapped L1 instruction and data caches. This processor (NEC VR4300) is a variant of the MIPS R3400i.

This project required me to read and reread the NEC VR4300 ISA (655 pages) several times, and I learned how to parse technical documents effectively. I pretty much followed the ISA with very little deviations so you could just as easily find all the information about how I implemented the processor from the ISA documentation.

However in order for this blurb to be useful for others attempting to build up from our code or start from scratch, I will mention several things that will hopefully be useful to those in the future. Firstly, I would say that it would not be as impossible as prior team PSX claims to reach a point where the system can boot up from the N64 PIFROM to an N64 ROM and play a simple 2D game such as Namco Museum 64. I would say that we were very close and we only needed to implement one feature for the CPU that was unforeseen (interrupts) and further debug the DMAs in the memory controller. I would not discourage future teams to continue our work and possibly even integrate parts of the RCP.

The CPU consists of the five usual stages: Instruction fetch stage, register fetch stage, execute stage, data cache fetch stage, and write back stage. The instruction fetch stage interfaces directly with the instruction cache and the data cache fetch stage interfaces directly with the data cache. The difference between the two caches is that the data cache can be written to while the instruction cache cannot be written to. There are also two memory segments that represent whether the data is cached or uncached, so depending on the address of instruction or data, we must go through the respective instruction or data caches to get the data. Currently I don't support virtual memory, but only segments, as this is the only requirement for the PD ROMs that we intended to run. The cacheable virtual memory address fall between 0x8000 0000 0x9fff ffff and the uncacheable addresses fall between 0xA000 0000 0xA03f ffff. Using this information we can determine whether to access the cache or directly request from the memory controller. Between any accesses to the memory controller, however, I have my own memory interface module that receives requests from either the instruction cache fetch stage directly, the data cache fetch stage directly, the instruction cache, the data cache, or the write-back buffer. The write-back buffer is a module that holds up to 32 bytes of data. This is used so that memory writes can be consolidated so that we don't overload the data bus. Any time each of these middle modules want to access memory they set a ready bit to high that gets read by the memory interface module and the interface module must prioritize in a way that allows the maximum throughput of the pipeline.

Because the databus is only 64 bits wide, and the cachelines for the instruction and data caches are 32 bytes and 16 bytes respectively, we need to have buffers that coalesce the data before writing it into the cachelines. The interface between the memory and the CPU is not simple and has many cases, so it must be studied very carefully before changing if need be.

What even further made memory interaction difficult was the availability of byte, halfword, word, word left, word right, doubleword left, and doubleword right stores and loads. I had to implement every type of store and load which would have different results based on the address as specified in the ISA.

There are also four separate register files that must be heeded. The first is the general-purpose register file for the CPU. The only subtleties are that register 0 is constantly held at zero regardless of whether a write occurs to register 0. Typically NOPS are encoded as instructions that write to the register 0. This was slightly tricky when I was implementing data forwarding, because I never considered the case of one instruction down in the pipeline writing to register 0 while a newer instruction read from register 0. This would cause the newer instruction to believe that there was data in register 0 when there wasn't. The rest of the registers are typically used for special purposes, as mentioned in the ISA, but they can be used for anything. The next register file is the coprocessor0 register file, which holds that status of the processor. The different registers represent different statuses of the processor and can be accessed with MTC0 and MFC0 instructions. There are also some instructions that modify specific bits in certain registers of the COP0 register file, but I have only implemented the seemingly important ones. Be careful of data forwarding in here as well, as since there are instructions that can modify the registers directly, and instructions that cause the modification of certain bits in the register file at different pipeline stages, you must be very careful of whether to write values in order simply forward them and write other values in. However, I have set up a nice infrastructure for simple addition of further modifications of the COP0 register file. The third register file is the floating-point register file, which can be accessed with MTC1 and MFC1. The FPU simply puts the data through the same pipeline as the regular instructions so one must be wary when integrating the FPU. I have set up the entire infrastructure here as well with decent commenting on where certain things should go. I decided not to integrate the FPU simply because we were not at the point of needing it in the project. An FPU can easily be generated using the IP Cores available. However, I have thought about the future integration of FPUs, and because the FPU is pipelined in order to maximize the clock speed, the execution stage should drive the clock enable of the FPU low after the FPU instruction completes so as to not start off another FPU instruction. If it does start another FPU instruction after it completes, it is possible that the pipeline will stall further and wait for the instruction to re-execute, or it is possible that the next FPU instruction will take on the previous calculation of data depending on implementation. The fourth register file is the coprocessor 1 control status register file. These registers hold information about the most recent FPU instruction execution, but have not been implemented either.

For branching, since there was a branch delay slot, it wasn't terribly difficult, but there was a slight modification to branching instructions that involved the likely flag. If likely followed any branching instruction, then if the branch conditions are not satisfied the instruction in the branch delay slot must be discarded.

One of the more complicated instructions was the cache instruction. Depending on the opcode within the cache instruction, it would modify the instruction cache or the data cache. This required heavy propagation of wires to the instruction and data caches. I initially tried to optimize for the number of wires that had to be sent, but I decided against that later and changed my implementation to something more understandable. Pre-optimization is bad and just made my code confusing to read even to myself. Some of these opcodes even did some strange things such as changing the tag bits for the cachelines or having a hit write back option for the instruction cache. Considering instruction caches can't be written to, I don't see why anyone would want to write back the cacheline to memory.

One strange thing that I did was with the multiply and divide instructions. I instantiated cores for each of the 6 different mult/div instructions (MULT, MULTU, DMULTU, DIV, DIVU, DDIVU) and because these were each pipelined, I would cause a stall in the execution cycle during one of these instructions. The strange part came in when the pipeline was stalled and the execution stage couldn't be propagated but the mult/div instruction had completed. Because these instructions manipulated the reghi and reglos only, I wrote directly to them after the instruction completed, afterwards I would invalidate the execution stage so that another mult/div would not be kicked off. This really helped to improve the throughput of the processor, but this made it a little confusing when debugging, as no multiply/divide instructions were ever being retired and when comparing against the golden simulator, we had to keep this in mind. One thing that I found was the usage of macros for simulation and synthesis. Since there are different top modules depending on whether you are synthesizing or simulating, you can define a macro at the top of the simulation module and within the submodules, wrap certain things in 'IFDEF SIMULATION tags depending on whether the module is used only for simulation or not. This was a very elegant solution to a problem we faced of having to have separate branches for simulation and synthesis and we were able to control our repository with less pain.

I built several tools including an assembler, disassembler, system state comparator, as well as several other tools that helped to debug the program much more efficiently. I am very glad I spent the time up front to creating these tools as they were constantly put to use later in the project.

5.2 FPU

The FPU in the NEC VR4300- though it is mentioned as Coprocessor 1 (CP1) in the processor specifications, this is in name only; the FPU is implemented in the same datapath as the main processor, Coprocessor 0. However, as it is easy to logically separate the FPU from the main CPU, we decided to develop them separately and integrate them later. The main reason for the combined datapath in the original N64 was to reduce hardware cost, and this was not a concern for us.

The FPU is 64-bit unit, and upholds the IEEE-745-1985 standard. This includes format conversion, multiple rounding modes and standard operations. To facilitate floating-point operations, there are registers specially allocated for usage by the FPU. These are the floating-point general registers and floating point control registers. All arithmetic floating-point operations are done with values in the floating-point general registers, and all status and control for the floating-point unit is done through the floating-point control registers.

Our implementation of the FPU contained only a subset of all the FPU instructions, as we had decided early on that we should only implement the necessary instructions for our target game, Namco Museum 64. As mentioned above, the FPU was designed as its own module, where it would receive instructions from the CPU and later output a done signal for the CPU to latch the results after a number of cycles. It had a clock enable signal to allow for stalling, and was fully pipelined.

We originally started writing the FPU module based on the FPU that was used in 18-340, though we quickly found that it was less than optimal. The 340 FPU was single-cycle and implemented only add, subtract, and multiply, which was far from complete. Although effort was made to flesh out this FPU, it was later made apparent that an easier way to handle all of the FPU instructions was to instantiate Vivados numerous FPU IP cores for each operation and to write a wrapper to connect all of them. This simplified our logic immensely and helped our verification, as these IP cores came with their own testbenches that verified correctness. The only verification we needed was to verify that the output matched our expected behavior, particularly in regards to rounding modes and cycle time.

5.3 Memory

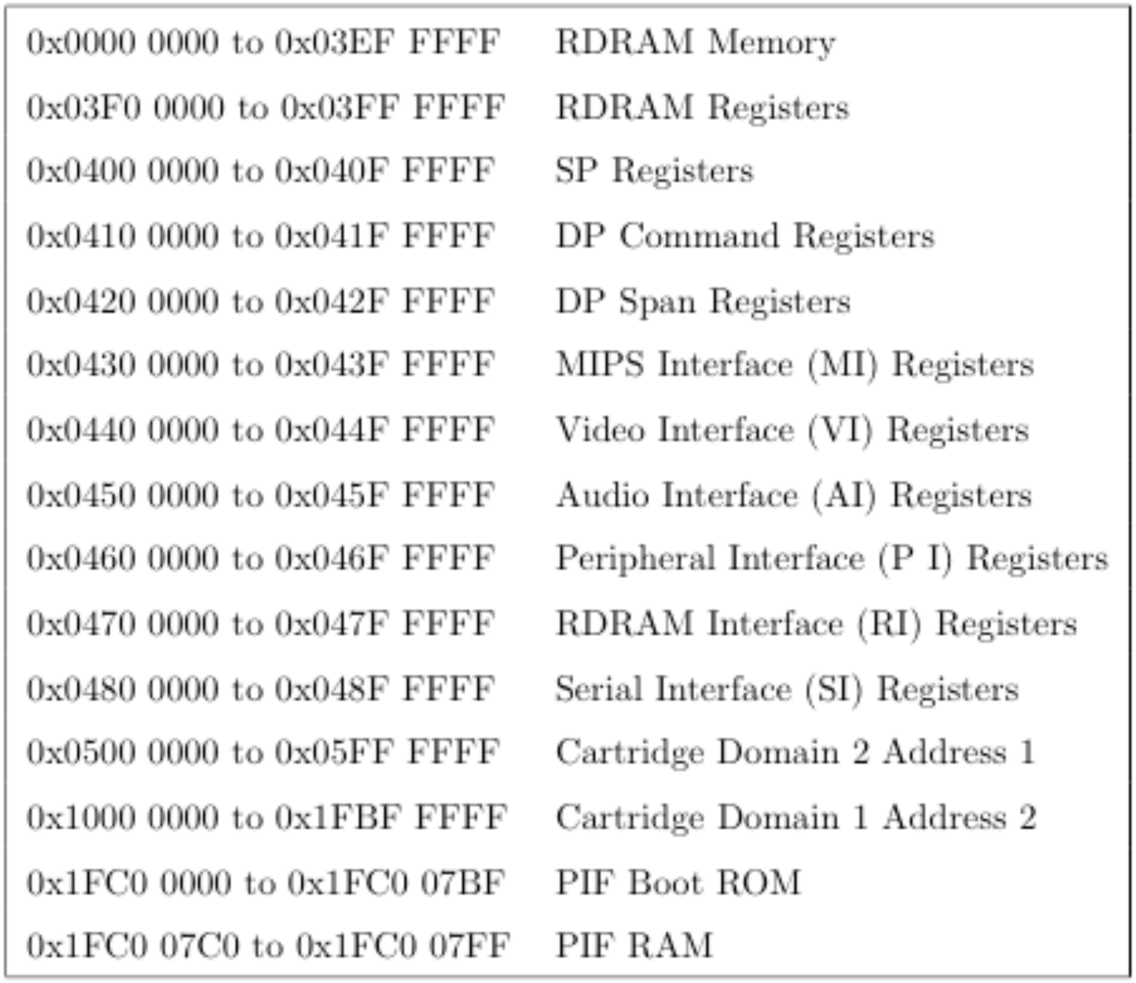
5.3.1 Overview

The N64 featured a unified memory, with incredibly high throughput but long latencies. We chose to use the Nexys 4 because of the development tools and the VGA port, not the memory. As a result, the tools were on the board were less than a perfect fit to match what the Nintendo 64 had. Our board holds a 16Mbyte Cellular RAM (Micron part number M45W8MW16) has a 16-bit bus that supports 8 or 16 bit data access. It can operate as a typical asynchronous SRAM with read and write cycle times of 70ns, or as a synchronous memory with a 104MHz bus.

5.3.2 Memory Segments

5.3.2.1 RAM

The following is an overview of all of the most important memory regions that we implemented in our design. This is taken from the Nintendo 64 Toolkit Part H, which can be found in various places around the Internet, by Anarko.



0x0000 0000 to 0x03EF FFFF	RDRAM Memory
0x03F0 0000 to 0x03FF FFFF	RDRAM Registers
0x0400 0000 to 0x040F FFFF	SP Registers
0x0410 0000 to 0x041F FFFF	DP Command Registers
0x0420 0000 to 0x042F FFFF	DP Span Registers
0x0430 0000 to 0x043F FFFF	MIPS Interface (MI) Registers
0x0440 0000 to 0x044F FFFF	Video Interface (VI) Registers
0x0450 0000 to 0x045F FFFF	Audio Interface (AI) Registers
0x0460 0000 to 0x046F FFFF	Peripheral Interface (P I) Registers
0x0470 0000 to 0x047F FFFF	RDRAM Interface (RI) Registers
0x0480 0000 to 0x048F FFFF	Serial Interface (SI) Registers
0x0500 0000 to 0x05FF FFFF	Cartridge Domain 2 Address 1
0x1000 0000 to 0x1FBF FFFF	Cartridge Domain 1 Address 2
0x1FC0 0000 to 0x1FC0 07BF	PIF Boot ROM
0x1FC0 07C0 to 0x1FC0 07FF	PIF RAM

The first feature of the Nintendo 64 was a 4MB RDRAM module from RAMBUS. The company is now defunct and the spec sheets are no longer available, as they were pulled when the company underwent some patent litigation. Fortunately, the exact specs are not really necessary. We chose to map the 16MB of Cellular RAM, what Micron, the manufacturer, refers to as a pseudo-SRAM. It was the only memory on our board that was large enough to support the whole memory of the N64, as block RAM was far too limited. Cellular RAM has access latencies of about 70ns, which is far faster than the Nintendo 64s 640ns latencies. Though, the maximum burst performance of the Cellular RAM is two bytes per cycle, or 200MB/s, while the RAMBUS module was capable of up to 500MB/s. Again, we are fortunate that we never experienced throughput issues because our final demo did not use Cellular RAM at all. Though, implementing the CellularRAM controller was not very fun. By default, the Cellular RAM operates in asynchronous mode, which is slower than in synchronous burst mode. Determining how to write the configuration register was simple enough, save for the fact that all of the asynchronous timings had to be translated into clocked timings. The other real issue though was one that I would never have discovered if not for the behavioral model the Micron has on their website- the memory will attempt and fail to act synchronously if there is an active clock but the RAM is configured for asynchronous mode. This meant disabling the clock until the RAM was properly configured, then enabling the clock. Another issue occurred later when our simulation times seemed to be increasing at an absurd rate. It turned out that the source of the slowdown was a non-constant access time in the behavioral model Micron provided. We rewrote this part of the logic and reduced our simulation runtimes from three hours to five to ten minutes.

5.3.2.2 Frame Buffer

The second feature of the Nintendo 64 that we had to account for was the frame buffer. In a typical Nintendo 64 system, the frame buffer is not a separate entity. A single hardware register called the VI ORIGIN REG (Visual Interface Origin Register) contains a pointer to the base address of the frame buffer. During the boot sequence, a value is written to this register and that is the address that will be mapped to the frame buffer. However, many games use the idea of double or triple buffering, where the next frame of the game is being rendered at the same time the current frame is being drawn to the screen. Games will write values to particular areas in memory and then change the frame buffer address to that segment with the idea that this will provide smoother transitions between frames. Due to our concerns with the throughput of the Cellular RAM, and the fact that ideally, a 320x240 resolution game will take 4.6MB/s of throughput to push the pixel data to the screen (and would likely be far worse due to its access patterns), we were concerned about this. Additionally, the visual interface may need to read the frame buffer at the same time that the CPU is writing to it. Therefore, we elected to remap this portion of memory into block RAM. Using code we found online, we inferred a true dual-ported block RAM. There would be some overhead for ensuring that data would end up in the correct places when these segments were separated, but we were certain that our games should only require minimal changes. Also because the framebuffer required 2 bytes to represent each pixel, it required a total block of memory of 153 KB. For other games, we would need to go to reasonable lengths with multiple frame buffer modules in order to enable double- and triple-buffering. For *Namco Museum 64*, the frame buffer only switches one time, so we came up with a hack around this “double buffering” and made it so that it would ignore the first frame buffer and simply use the second frame buffer address values that it changes to. Because our frame buffer is not inside of RAM as it is defined in the N64 specification, we were unsure of the side effects that this would have, but after looking at values being written in the original frame buffer, we decided that it wouldn't have any effect on the program. Also, switching to real double or even triple buffering would be slightly complicated. As we are not sure how exactly the frame buffer is used, and whether data values from the frame buffer not being used was unknown to us, we would recommend looking into how the simulators handle the double buffering. Since our buffers are outside of memory, we thought of two ways to prevent coherence issues. Either, every time the frame buffer is switched, we issue a DMA from the frame buffer into RAM, or we find a board with a faster memory module and place the frame buffers back into RAM.

5.3.2.3 PIFROM

The bootloader, also called the PIFROM (Peripheral Interface Read-Only Memory) also has its own segment in memory. Because this region does not need to be large, but should be relatively fast and also must be initialized with the bootloader, we elected to use Vivado to generate a block RAM IP Core. Once we put the bootloader into the XDC file, the PIFROM would populate itself automatically. We did have some issues getting the data to go to the correct place in the block RAM. It may have been a bug on our part, but at first it seemed to be getting loaded into the middle of the RAM rather than the beginning. After playing around with a lot of things, the issue disappeared and we never completely figured out the core cause of this error.

5.3.2.4 PIFRAM

To complement the PIFROM was the PIFRAM. The PIFRAM was a 64-byte section of RAM used by the serial interface and the peripheral interface when interacting the cartridge and performing CIC checks, as well as while sending commands and receiving data from the controller. We elected to use block RAM for this as well, as its dual-ported nature allowed us to easily abstract away the communication between the CPU (which sends commands) and the Serial Interface (which receives the commands and writes back the data).

5.3.2.5 Memory Mapped Registers

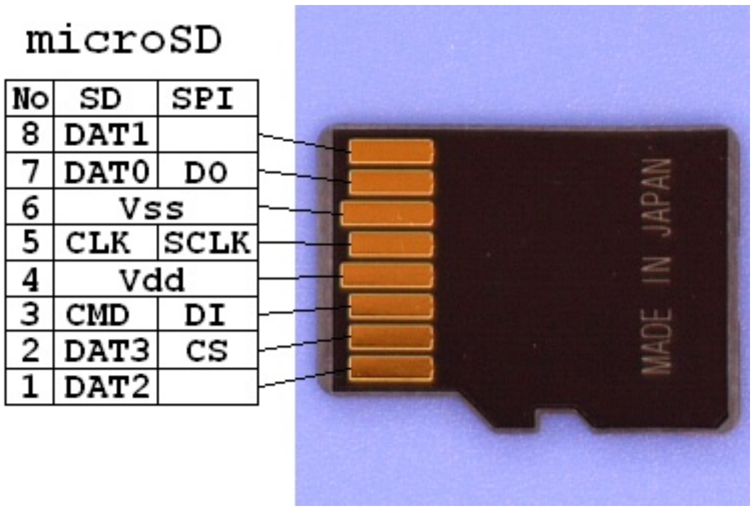
The final internal section of memory that we implemented was the memory-mapped registers. These were by far the most “hacked together” portions of the memory. The each memory mapped register in the Nintendo 64 implements a separate function, from things like the DMA functions to containing the bitrate of the audio samples to the memory bus interrupt status. The majority of these registers were simply implemented as a place that could hold data, while the value stored there performed no actual functionality. There were a few exceptions to this- most notably the DMA registers (containing lengths and addresses), the VI ORIGIN REG and the some of the status registers. These registers were duplicated in the logic, where they performed the actual functionality. The register region also contained the SP IMEM and SP DMEM regions, which are each 1KB regions that are utilized by during the boot sequence and by the RSP. There are many system-specific features implemented in these registers, and it was extremely difficult to determine what should be done with them.

5.3.2.6 Cartridge

Early on, we decided not to implement the cartridge using actual N64 roms given the complexity of the project and that it would be much more effective to load roms onto an SD card. This provided the added benefit of being able to load multiple ROMs onto the card and being able to switch in between games as well as being able to take advantage of the abundance of N64 ROMs available online. This could be done easily by cat-ing the ROMs together and separated with 0s generated using xxd.

Like team PSX, our first attempt at loading a ROM onto flash memory was via SD card. We were at first concerned regarding the same two problems that team PSX mentioned. The first problem was that there were allegedly no SD card controllers that did not require a soft-processor. Given the limited size of our board and our project being a hard core oriented project, a soft core processor was not an option. The first SD card controller we found on opencores operated in the standard four bit SD mode but was designed to be used with a soft processor and initializing and performing reads were very complicated to perform in hardware. We then found another SD card controller on opencores that operated in SPI mode which provided a series of instructions for reading, writing, and initializing and provided a much more simple interface. Prior to being able to get the SD card working, we ran into three problems of note before getting the SD card to work.

The first problem we encountered with this core was that running the provided test bench resulted in an initialization error. This was resolved by combing through the core and SD card documentation and fixing a bit that was meant to be asserted to initialize properly. The second very major problem and most embarrassing problem we solved was that we swapped the dataIn and dataOut lines of the SD card. The reason we did this was because when we read the SD card documentation, the documentation was clearly written with respect to the SD card and not to the core. However, we didn't realize this and assumed the dataIn and dataOut lines corresponded to the same data lines as the core. To prevent future incidences of such errors, we adopted a naming convention copied from SPI that labeled ports MISO (Master in Slave Out) or MOSI (Master out Slave In). This meant that the programmer no longer had to consider from whose perspective they were dealing with and derive from the port name the exact purpose of the port. We lost almost two whole days worth of debugging on this problem and using this naming convention made our lives much easier. The last problem of note we resolved prior to being able to read and initialize properly was that the addresses passed to the wishbone interface had to be multiples of 512 bytes. This wasn't addressed in the documentation or the comments and was discovered via trial and error after realizing we were receiving a read error whenever we had a cache miss on reads that were not multiples of 512 bytes.



No	SD	SPI
8	DAT1	
7	DAT0	DO
6	Vss	
5	CLK	SCLK
4	Vdd	
3	CMD	DI
2	DAT3	CS
1	DAT2	

Figure 5.1 SD Card SPI standard

The second problem team PSX mentioned was being unable to load raw data onto an SD card. This was solved with the discovery of Win32 disk imager which is capable of writing raw images onto a variety of removable devices. This meant that we were able to bypass file systems and additionally rewrite out sd card with whatever ROM we wanted to. Another advantage this offered was that it allowed us to buffer the ends of the files with some null value (in our scenario 100 MB of 0s generated using the xxd command) in case the system attempted to perform accesses beyond the end of the ROM.

Following this, we were able to perform reads from any sd card less than 2 GB at 24 MHz. While the module seems to suggest that there were only two acceptable frequencies for SPI system clock, we found out that these requirements werent very strict as were were able to vary the frequency from 25 MHz up to 50 MHz and still perform reads properly.

This means that groups in the future that wish to store memory off board will be able to do so using the SD interface that we wrote. We verified this by taking the hexdump of our target ROM and were able to read the exact same data values from the FPGA. Conveniently for us, Win32 disk imager writes the image from address 0.

Section 6: Interfaces

6.1 DMAs

The Nintendo 64 supported DMA capability for copying data between RAM and other sections of memory, like the cartridge and the Serial Interface. A full diagram of the pathways can be seen below. Because most of our documentation was from emulators that implemented DMAs being performed instantly, we stalled all transactions with the CPU while a DMA was occurring in order to replicate this. Here, we sacrificed some speed for simplicity and accuracy. However, while running the bootloader on the FPGA, the stall caused by the DMA was not even detectable.

DMAs were made simple to implement once we had really refined the interface that each memory segment used. Then, it became almost trivial to simply connect the two modules to each other, command one to read while the other wrote data, and then they managed themselves. The DMA engine simply broke the transactions into 32-byte chunks and triggered reads and writes while incrementing the addresses.

6.2 Boot Sequence

Booting the N64 was not a simple process, but one that we at least completed in simulation. It exercised many of the regions of memory as well as many of the memory-mapped registers. It was tough to determine what the memory-mapped registers should do, and because there are so many, it didn't make sense to attempt to implement all of them. So, we tried to work out which ones should be implemented by running the bootloader and comparing the results to CEN64. The worst hack of these was performed by CEN and MESS whereby a register changes value when the value of another register is read. The particular registers that these apply to are fairly easy to find in the cen simulator with a simple grep on memory-mapped registers. These register value changes were done in order to make the bootloader believe that the bootloader had passed some CIC security checks. The CIC is a proprietary chip on every cartridge that returns values to certain inputs, and the bootloader verifies these inputs, and there are about five different CICs with different model numbers. Currently, our console would only have supported the CIC-NUS-6102, the most common CIC. If other games were to be implemented, the correct seed value would need to be placed into the PIFRAM before the bootloader began. This would require either using some switches to switch the seed value, correctly implementing the CIC checks (good luck!) or checking the cartridge against a static list to determine which CIC it uses. Many status registers were also twiddled with, and sometimes value simply had to be written to them without any real understanding of what the value was meant to do. If you want a very basic idea of what the boot loader is doing, open up the file 'boot_patterns' in the top-level directory of the repository and you will have a semi understanding of how complicated it is. On top of this, the boot loader generates a list of instructions, writes it to some portion of memory, and then jumps to it to execute it for some reason. For a full understanding, please run the cen64 simulator printing every time an instruction retires along with the register it modifies and the address and instruction.

6.3 Serial Interface (Controller)

The serial interface is responsible for sending and receiving requests to and from the controller. The serial interface does so by reading from requests written to a 64 byte block of RAM sent via the CPU, sending these instructions to the controller, and writing the controller response back to the block of ram. The 64 byte block of RAM is partitioned into 4 byte chunks and is portrayed in this table:

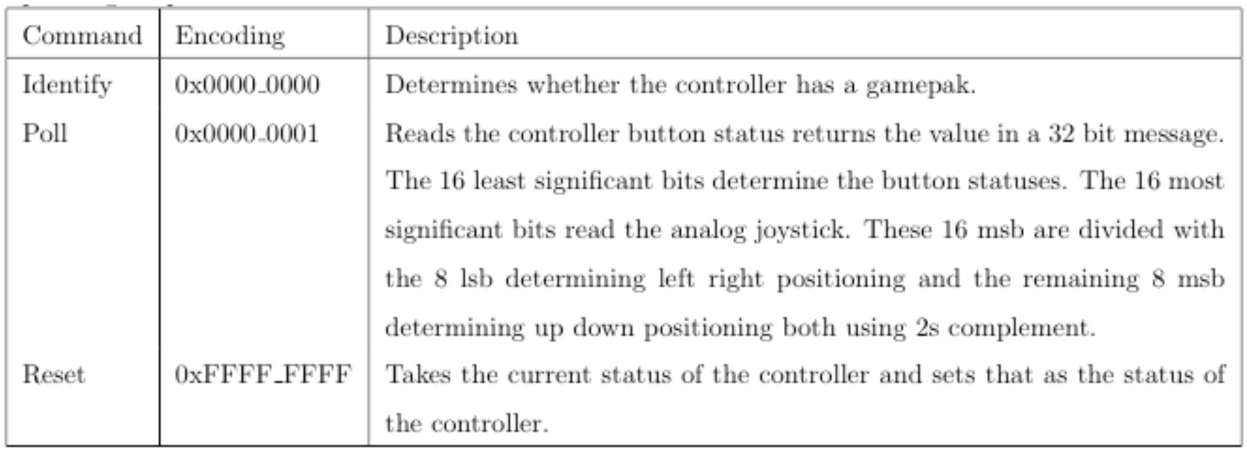
Memory Location	Purpose
0-3	Command to controller 1
3-7	Response from controller 1
8-31	Repeat of the above for controllers 2-4
32-35	Unknown
36-39	CIC Seed
40-59	Unknown
60-64	PIF Status, including the lowest bit indicating controller command status

The N64 wiring is very simple as there is only a single serial data line in addition to power and ground. The system poller operates at 4 MHz while the controller operates at 250 KHz and takes 4µs to send a bit other than the stop bit, which takes 3µs (all commands responses have a stop bit appended). The bit encodings are detailed in this table:

Bit Type	1μs	2μs	3μs	4μs
0 bit	0	0	0	1
1 bit	0	1	1	1
Stop bit	0	1	1	N/A

This bit configuration was handy as the data line was asserted high meaning a message was coming in upon seeing the first 0. In order to determine whether a bit was a one or a zero, I sampled after 7 4MHz cycles to check the middle bit. This worked fairly well other than the fact that I had to tweak the clock by a marginal amount, as the controller wasn't sending data at exactly 250 KHz.

The controller we configured accepts three different commands, which consists of all commands that don’t require a gamepak. The commands are encoded in this table:

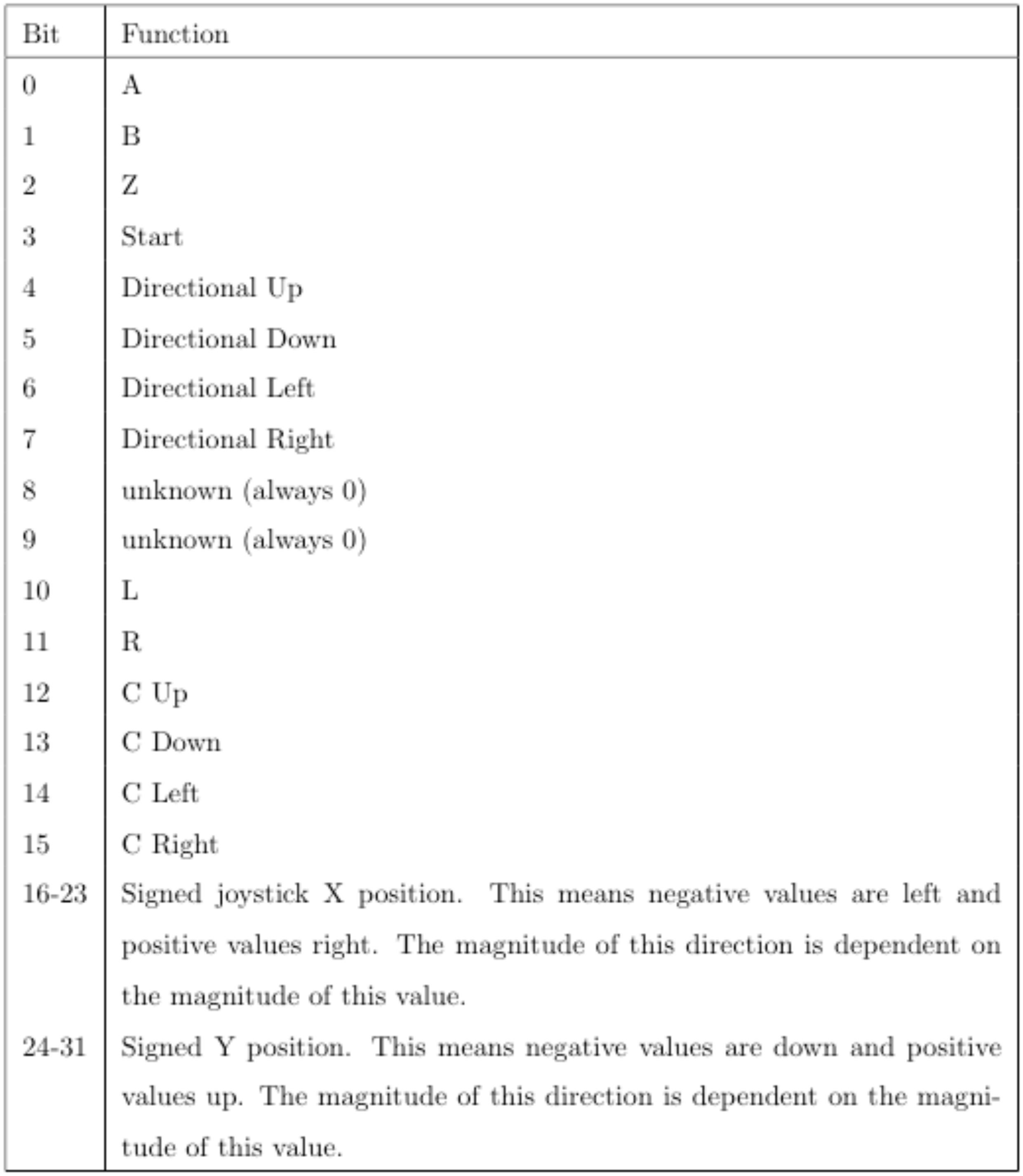


Command	Encoding	Description
Identify	0x0000_0000	Determines whether the controller has a gamepak.
Poll	0x0000_0001	Reads the controller button status returns the value in a 32 bit message. The 16 least significant bits determine the button statuses. The 16 most significant bits read the analog joystick. These 16 msb are divided with the 8 lsb determining left right positioning and the remaining 8 msb determining up down positioning both using 2s complement.
Reset	0xFFFF_FFFF	Takes the current status of the controller and sets that as the status of the controller.

The moment a command gets sent, the controller responds almost immediately (within 2µs).

The last step involved here was converting the data received from 4 MHz to the 50 MHz of our system. This involved two crossing clock domain transactions: 1) passing the 50 MHz system command to the 4 MHz controller module and 2) passing the 4 MHz response from the controller back to the 50 MHz system. This was done by passing a flag across clock domains and latching the values meant to be sent. Lastly, the controller can’t be polled too quickly otherwise it fails to respond. We found a 0.375 millisecond delay was enough.

Here is a table containing the mapping the 32 bit poll response to the buttons:



Bit	Function
0	A
1	B
2	Z
3	Start
4	Directional Up
5	Directional Down
6	Directional Left
7	Directional Right
8	unknown (always 0)
9	unknown (always 0)
10	L
11	R
12	C Up
13	C Down
14	C Left
15	C Right
16-23	Signed joystick X position. This means negative values are left and positive values right. The magnitude of this direction is dependent on the magnitude of this value.
24-31	Signed Y position. This means negative values are down and positive values up. The magnitude of this direction is dependent on the magnitude of this value.

*The analog joystick is somewhat sensitive as it is occasionally defaulted to -1, 1 in either the X or Y direction. If left as is, this could potentially influence gameplay. Hence, most games have a deadzone of +/-2 to protect against this.

6.4 Peripheral Interface

The peripheral interface is responsible for sending requests for data from the cartridge and sending requests to retrieve that data. There are two parameters of note: 1) a size request ranging from 1 to 63 bytes and 2) the 64 bit data line to send data back to the RAM controller. While the peripheral interface is normally responsible for performing writes back to the cartridge, the games that we targeted for demonstrating did not require writing hence this was not necessary (however, with the infrastructure in place this could easily be implemented).

We also implemented a single 512-byte cache because we were using an SD card in SPI mode that used 512 byte transfers per transaction (more on why we used SD card in next section). Given the variable size requests, it was possible that the CPU could request data that there would partially fill up the returned 64 data bit bus line and require another cache read. Our peripheral interface handles this and populates the 64 data bit bus in a single cycle barring a cache miss.

Additionally, in order to properly use the SD card, the SD card needed to be properly initialized. This included waiting 1ms for the SD card to power up the internal MCU in the SD card before inputting a sequence of commands to the wishbone interface that would trigger the core to perform the initialization. Our peripheral interface performs this initialization immediately meaning the rest of the system needs only check whether the peripheral interface is exerting a waiting signal and from there can request whatever reads it wishes to.

Lastly, we also wrote the hardware for performing a read sequence. This sequence essentially consisted of writing to the wishbone address registers and writing to the wishbone command status registers and before waiting to populate a FIFO with the values retrieved from the SD card. This FIFO would go on to serve as our 512-byte cache, which we cleared whenever we hit a cache miss.

6.5 Audio Interface

Unlike the Virtex5 boards that these labs were designed for, our Nexys 4 boards do not have a soundcard or hardware for the AC'97 codec. The Nexys 4 only had mono pulse-width modulated (PWM) audio output, which we could not use for this lab nor our project. Thus, we ordered a pulse controlled modulation (PCM) pmod to remedy this problem. This pmod is capable of playing 16-bit stereo PCM audio.

The PCM Pmod required us to follow the I2S specification, a common serial protocol for communication. As this was a serial protocol, we had to translate our 16-bit PCM data into serial data. This involved storing the PCM in a FIFO that was then connected to a parallel to serial converter. We found a module for this converter online, though it took a while to figure out as half of it was written in VHDL.

We didn't end up integrating the Audio interface with the rest of the system as the FPU wasn't completely ready to be used and we determined that there were more important parts of the system that needed work. However, we do now have a way for those interested in using our board or any board with only a PWM audio output to have audio easily integrated into their system.

6.6 Visual Interface

We decided on using VGA as our video output of choice. The VGA specification on our board was very simple, requiring only RGB values, and an Hsync and Vsync. From VGA timing specifications found online, we were able to write a working VGA module that output the data we expected. However, we used a lot of magic number parameters in this module to match our intended screen resolution of 320x240. The files relevant to this module are in the visual/ folder, however it also relies on parts of the memory controller as the frame buffers (which are attached to the memory controller) send the visual interface data to be written to the screen.

Our visual interface module provided the communication between memory and the VGA. The main method for displaying visuals for the N64 is through the use of frame buffers. The visual interface would request information from the framebuffer about what to draw every cycle. We allocated a 320x240 segment of dual-ported block RAM to act as our screen, where each segment was represented by two bytes. The two bytes allowed for 5 bits allocated per color (RGB) and one extra bit. Any changes made to this frame buffer would be reflected visually on the monitor. More information on the frame buffer in section 5.3.2.2.

Section 7: Condensed Status

7.1 What Is Done

The CPU, memory, serial interface, and visual interface have all been integrated together, and can communicate with each other. The CPU has been thoroughly tested but requires the implementation of interrupts. The memory controller has all of its memory segments implemented and has minimal bugs in regards to the frame buffer and its endianess of access. The serial interface is completely working and is very extensible for extra controllers to be added. There is a very short specification sheet for the serial interface that can be read for an easy way to add more controllers. As of now, we have support for two controllers. The visual interface is working and takes a frame buffer and displays its contents onto the center of a monitor in a 320x240 screen. Our current visual interface however also causes the *Tron* game title to appear outside of this 320x240 box; so future teams will have to go through the visual interface to remove this title (if they want). The audio interface works, but is not attached to the remainder of the system along with the FPU. The FPU is also completed but still needs to be integrated with the CPU. This should be fairly straightforward, as FPU instructions use the same pipeline, only writing results to a different register file.

7.2 What Needs to Be Done

The next step is to implement interrupts for the CPU. For those looking to continue this project, implement the interrupts for the CPU and then synthesize and implement all of our code onto a board. Currently our CPU deviates from the correct execution path at the first interrupt. We have included a test game, *Pong*, that should be able to run with the parts of the system that we have implemented. Aiming to run *Pong* correctly would be a good direction to take to see what the next step in debugging is.

After that, move on to more complex games. Namco Museum 64 should be your next goal, as it requires the CPU but also requires the FPU. This is when you integrate the FPU into the CPU. Also, Namco Museum 64 uses audio without using the RCP, so integrating audio at this point would also be good.

There also seems to be a small bug in the interface between CPU and memory (at least in the frame buffer), where bytes seem to be getting written in backwards (we believe it is related to endianess). You could either completely rewrite the interface (which may make our code easier to read and make for easier bug fixes in the future) or look for the cause of the frame buffer bug (the complexity of which is unknown).

In order to take this project to the next level, having fixed up the bugs mentioned above, you would have to implement the RCP (reality co-processor). We currently don't have too much information on the RCP and are not very sure of the level of documentation that exists for this module online, but with a talented team, the RCP could be a worthwhile endeavor. With the RCP, most N64 cartridges would be able to be played including *Super Mario 64*. The RCP is composed of the RSP (Reality Signal Processor) and RDP (Reality Display Processor). The RSP is a vector processor designed for high performance 3D calculations and handles some of the audio and most video pre-processing of the N64. It handles real-time edge anti-aliasing, automatic load-management, texture mapping, and real-time depth buffering. The RDP on the other hand is responsible for rendering shaded, textured, and depth buffered geometry to an arbitrary frame buffer in main memory. With this in mind, simulators have been made before, so even with a lack of documentation, reading through the simulator will give a deep understanding of what needs to be done. Tyler Stachecki told us about his current work on the RCP, so by the time you are looking at this, there may be more public information on the RCP. If that is the case, I would highly recommend taking on this project, as getting *Super Mario 64* working would be very fulfilling.

Section 8 : Words of Wisdom

- Have at least one person on the team with a complete understanding of how the system modules will come together. Even better, have everyone understand it.
- Understand all the modules that will need to be created for the system to work and then create a schedule based on that. Make sure that the schedule is compact and try to finish the bare minimum modules required for hard progress to be seen as fast as possible. It can be very discouraging to have worked half the semester with nothing to show for it, so finish the bare minimum and then work up from that basis.
- The faster you have something to demo, the better your team will feel about your team’s progress. The first time we had something drawing on a monitor was a huge boost in our team’s hope for completion and the team worked harder in anticipation for more major checkpoints to be met.
- Make the schedule fast-paced, but also reasonable so that the group does not get discouraged when nobody makes the schedule. Expect your partners to fall behind schedule, but don’t let them fall too far behind. Stay on schedule as best as you can as that is your best chance of getting a complete project.
- Not only should you read the assigned reading for this class, but use what you learn from it and apply it to your team. This project is a huge endeavor and should not be taken lightly. If you want to have a good chance at project completion, you should expect a high workload.
- Get all the help you can from people around you and online. If there is a module that you might think exists but you cannot find it, ask around on forums and people. The time that you could save by finding a pre-made module could save your project.
- Start early and decide on your project as fast as you can. A large part of our incompletion can be attributed to us having changed our project four or so times in the first two weeks as we were trying to find a project of the proper scope and much research had to be done in order to finally find a project that would be reasonable. Even better yet, do your research before the semester even starts. Once you finalize your project decision, begin as soon as possible and work as fast as possible.
- Fix all the warnings that Vivado gives you, or else you will have unexpected behavior. (You might still have unexpected behavior afterwards too).

Section 9 : Personal Statements

9.1 A Note from Jeremie

While the ISA specified certain ways of implementation, I scaled them down to reduce their complexity in order to be able to finish the processor in time. The largest change that deviated from the specifications was that of cache access protocol. The way it was specified in the ISA required the address to be checked in cache in the cycle before the actual cache access and then to actually check whether the cache line matched the address in the following cycle. If the cache line does not match, an exception occurs that resets the program counter while filling in the cache with the appropriate data for the request. The cache is then probed again after the exception completes. I considered nearly every option for maximizing processor speed, throughput, and completion. After weighing all possibilities, I came to the conclusion that it would be best to simply stall in the cache stages whenever the cache line requested changed from the last request. While the processor seemed initially a daunting task, I was able to cut some corners by basing off of the golden simulator provided by Tyler Stachecki (MarathonMan). I put golden in quotations because although it was able to run a given number of games, it did not match the ISA in several aspects. The NECVR4300 supported switching between 32-bit and 64-bit mode. However, for the games we intended to run, I only had to implement the processor as if it were only running in 64-bit mode. There were also numerous processor modes that gave the program access to different mappings to memory. Using the simulator, I determined the segments that were accessed for our games and only implemented those particular segments. I also managed to reduce the number of instructions implemented by referencing the C simulator. I determined about 100 instructions that were required to run the games.

During this semester I learned several things. I implemented an instruction cache and data cache for the processor and was able to see first-hand how they are being used and how they improve throughput of the instructions by crazy amount. At times, debugging was very enjoyable. After you understand a system large enough such as the one we created, it is cool to see the interaction of parts and be able to determine exactly where in the source code causes the error just by thinking about the effects. Seeing processor work was interesting and it taught me to really appreciate the work that went into larger cores. After putting in lots of hard work, it is really nice to see all the things your hard work can achieve. Implementing such a large design taught me a lot. I had to expand my capability of retaining information of massive amounts of information in my head in order to become an effective debugger of the processor. Towards the end of the semester, I had to be very careful in every change that I made because of all the subtle impacts that it could potentially have on my core. I really learned to appreciate the golden simulator here as well as the tools that I created and that I found. It is essential to go into this project with a very extensive set of tools or at least be willing and able to create them all in reasonable spans of time. If you decide to build a large core for this project rather than use a premade one, it is essential that you have a golden simulator that you can compare your processor against. Having this was incredibly helpful and I probably would have never finished the core without it. I created an extensive list of tests with every instruction and created a mechanism that would run both the RTL as well as the golden simulator and compare the outputs. This tool was amazing and helped me debug the core to near perfection in a very short period of time. Another very useful thing was usage of random testing, where you generate a suite of a random string of instructions and use them as test cases. This really helps to comb out the edge cases in the processor that may come up later in integration (where debugging is 100x harder). During integration, I only found about 4 minor bugs in the processor, which really helped speed up the integration process. Finding bugs during integration is so much harder just because it becomes unknown whether the error stemmed from the processor or from other parts of the machine. It also takes a ridiculously long time to synthesize the system, so in that aspect as well it is very time consuming and time is not on your side in this class. I would also recommend to keep a handy-dandy notebook and write down everything that you do in order to keep track of the things that you have thought of, are in the middle of implementing, are designing, tricks to overcome certain Vivado bugs that have been solved previously etc. It really comes useful and you will not waste time fixing the same bug over and over again. Another recommendation would be to propagate all wires through the pipeline that are generated even if they are not used later in the pipeline in case they may be used in the future. This will solve a lot of headache later down the road, as it is very likely that you are going to require a wire later. Wire propagation sucks when you have to transfer one several sub-modules deep to another module. This early propagation will be very useful and help in debugging. The synthesizer will optimize these out if you end up not using them anyways so don't worry about that. Also, do not use the (.*) for declaring modules but rather indicate every single wire independently. Even if it may look ugly, it will save a lot of time later when debugging.

This processor is not complete. If you want to take it further, a lot will have to be done. However it is a great starting point if you choose to take on the RCP of the N64. Most of the infrastructure is in place for completing the ISA. However one large task that you will have to take on is the implementation of interrupts. This was one thing that I missed while designing the CPU and its need was found two days before the demo day. For this reason I chose to go ahead with what was available to go ahead and create my own demo. I wrote an assembly program compatible with my CPU, which was modeled after the classic arcade game Tron. In this game, there were two players that controlled their respective lightcycles that could move around in a square map. As the players moved, a contiguous trail would be drawn mapping their paths. However, if any player is to hit either players paths or the four square walls around the map, the other player would win the round. The problem in writing this arose when we found a problem with the frame buffer accesses. I suspect that it has something to do with flipping the endianness of the data in the frame buffer when reading from it, but am not completely sure as to what the problem was.

This was probably the biggest project I have undertaken at CMU. I've never pulled more all-nighters in one semester nor written so many lines of code. I really enjoyed the entire process of creating a CPU from only an ISA. I enjoyed the perseverance required to perfect this one component as well as the creation of numerous tools required to bring it to full fruition. As disappointing as it was to not be able to get a real N64 ROM to boot up, it was enjoyable to be able to put together an entire system from scratch and watch how the parts interacted. I really learned about the bottlenecks of the memory hierarchy and this project helped me to appreciate the hard work that has been put into building modern systems.

9.2 A Note from Alvin

This project, while extremely taxing, was an extremely rewarding experience for me. When I first signed up for the project, I was worried about my readiness and how much I would be able to contribute. I hadn't touched Verilog in a year and using a Nintendo 64 as a Verilog 101 crash course didn't sound like the greatest idea. However, by the end of the project I was able to produce both the peripheral interface and the serial interface. In particular, I'm proud of having gotten the ROM to work on an SD card as no other group in the past had interfaced with an SD card using a hard core and other future groups could use this core for their own projects.

At the start of the semester, I wasn't sure where to begin so went about working on three tasks that would help me ramp up as well as get a good grasp of what was going on with the overall system. The first task I worked on was creating a wiki in our Github that would consolidate and organize all documentation we found. We previously piled all of our findings into a very large Google doc and from moving all the documentation to the Github wiki, I was able to get a better overview of the overall picture of the N64 and this also served as a reference for others in the group. The second task I worked on was creating a utils file that would consist of all commonly used Verilog modules such that others could simply reuse such modules instead of writing their own every time. This helped me regain my Verilog familiarity but wasn't as useful for others as I hoped given towards the end of the semester most of us just created modules on the go. The third task I worked on was really looking into Chipscope and figuring out how to debug on board. This step definitely paid off as we ran into numerous on board problems that weren't present in simulation.

From there, I then started working on the controller which I chose because it was the most straightforward component and would allow me to run through the entire design process from searching through documentation to verifying on the FPGA. This proved to have its pros and cons. The pros being that I was able to get acquainted with Vivado, learn which warnings to ignore, use chipscope, and help others in my group. The con was that the serial interface wasn't needed until much later in the semester while my group needed the peripheral interface much sooner.

Lastly, I worked on getting the SD card to work on board. This step was quite interesting as it was my first time working with open source cores. Additionally, the core was written back in 2008 and there were several bug reports filed about there being problems initializing and writing the card that were unresolved. I went against conventional wisdom here and ignored the warnings and pushed ahead after seeing how straightforward it was to initialize and read from the card.

In conclusion, I learned lots and really enjoyed the challenge of taking on something completely foreign. Max was an amazing partner and his passion for the project was contagious and was a great source of information on anything N64 related. Jeremie worked relentlessly and pushed me to harder every week and motivated me to pull more all-nighters than I have in my life. Both Max and Jeremie pushed me to continue working the SD card when I wanted to load the ROM onto a Raspberry Pi telling me that I was very close. Looking back, I’m very grateful that they did.

9.3 A Note from Prin

I was unprepared for the amount of work this class would entail. I had heard stories, read other peoples personal statements, but it was not until we were finishing up our research that I understood the scope of the journey we were about to undertake.

The N64 was simply a massive project. I will admit that at the beginning I had some pretty unrealistic expectations. With four people, I thought we would be able to churn out code and squash bugs left, right and center. While the code churning was true, having more people churn out more code simply meant we would have more bugs to deal with, as none of us were perfect. Along side this, debugging was made a lot harder because our code was more stratified. During integration, when one of us found unexpected behavior, it was almost impossible to simply fix it. Rather, we would have to ask the person that wrote the module why certain behavior was exhibited, because we had partitioned our works such that no one knew exactly the inner workings of the others.

I started the project working on the floating point unit, as I was the only one who had had experience with in in a previous class. I could not find a suitable unit from OpenCores, so I spent a lot of time researching this unit, and trying to implement all of the features. It quickly grew to be a daunting task. Much of this time turned out to be unproductive, however, as we soon discovered the wonders of Vivados IP catalog, which had already implemented the majority of these floating point instructions. Three weeks of work was quickly overwritten by learning the existence of a tool that I should've known about through more extensive research. Lesson learnt: don't try to implement everything by yourself, especially when there are implementations that are already better.

I then moved on to working on the audio subsystem. While working on this, I seemed to be fighting Vivado more often than I would be writing code. I was road blocked for a week and a half due to faulty documentation on how to initialize block ram, leading to most of the sounds I was testing sounding like static. This was probably the most aggravating period of the semester, as there is an acute sense of despair when the code you write, test, and simulate disagrees with the reality of the implementation. However, ChipScope proved to be my saving grace and allowed me to solve this issue handily.

We did not end up using the audio system I had worked out, though. It was around this time, nearing the third quarter of the semester, that it seemed that we had fallen off schedule. Bugs were popping up everywhere, causing delays upon delays. I was put on help everyone else duty, because the parts I was working on prior would be of no use if the system would not even boot up. To this end, I became the group rubber duck, though I primarily sat through and helped Max. It was only at this point that I started to feel like I was really part of the team, because until then I had been working solely on my own code. Here, I would like to give a shoutout to all of my teammates. Jeremie was an absolute monster, implementing the better half of a CPU from scratch. It was not until we were integrating that I realized just exactly how much effort he had placed into this class. Alvin was relentless in the number of hours he would spend in lab working, which motivated me to stay in lab more. Max was excellent at working the leader role, inspiring all of us with his enthusiasm. I was very fortunate to have worked with these guys.

Even though we were unable to finish our original project goals, working on the project provided valuable life lessons and learning experiences. I can say I have never worked this hard in my life before, starting from 12 hours a week at the beginning of the semester and converging on 18 hours a day during the final week before the demo. Would I repeat the same experience if given the choice? No, I would choose a more reasonable project with more reasonable documentation and used a more reasonable design suite. However, without this experience, I would never have learned the camaraderie of working on a big project with a team of dedicated individuals. My advice to future groups is, unless you want to dedicate your semester to this class, to pick something reasonable and fun. Most importantly though, pick people you would like to work with, because you'll be spending an ungodly amount of time with them over the semester.

9.4 A Note from Max

This project was by a huge margin the most time consuming of any class I have taken at CMU. I'm glad that I had planned for this during this semester, because otherwise I would have really struggled to keep up with my other classes. That being said, we knew from the beginning that this was an ambition project that would not be easy. I took 18-447 in a previous semester, and found that I felt like there was a lot more work that was put in with fewer results. Understanding the complexities of the system were very difficult, and no one is there to spoon-feed you what it is that you have to do. As a result, the first month or so were spent doing as much research as we could manage and determining if the project was viable at all. We were fortunate to find Tyler Stachecki as a resource, who already understood how difficult it would be, and further guided us in finding many of the materials we would need, such as public domain ROMs that did not utilize the RCP and RSP.

When it comes to working effectively, I cannot emphasize enough the benefits of working in simulation. Eventually your design may take a half hour to synthesize, and good luck debugging. If you can keep your model in simulation, do it! Find behavioral models or write your own for things that don't have any. This will help to quickly validate all parts of your design. Also, our decision to use the emulator as a golden reference made identifying bugs much easier and made the whole process faster. When you can identify a fault to a certain instruction with certain operands, or a write to a specific address, working becomes so much more efficient. Though, verify early and often that what you are doing in simulation matches synthesis. Vivado is heartless and will claim that you aren't using parts of you design, then optimize them out. Go through all of the synthesis warnings before you do any else, they usually fix the problem.

Overall, I found this to be an entirely worthwhile project. I would not advise others to stay away from such ambitious projects, I would just advise that if you want to do something as new and poorly documented as the Nintendo 64, be aware that it must consume you and the rest of your team for the whole semester. I know I spent about 30 hours per week on this at the beginning of the semester, and in the last week or so spent about 50-60 hours in lab. Start as early as you can. If you are thinking about doing some- thing like this and the class hasn't started, consider beginning your research, it will save you sleep in the future!

I recently saw that Digilent has announced a Nexys 4 with DDR2 RAM instead of Cellular RAM. If you decide to pick up this project where we left off, save yourself the hassle and get that board! This unreal amount of bandwidth means you would not have to fix the hacks we put in for the framebuffer.

Section 10 : Class Impressions

This class is a unique opportunity to learn project management skills on top of applying everything you’ve learned during your time at Carnegie Mellon to a single endeavor. This will probably be the largest project you will have worked on and it is best not to underestimate the time required to make this project successful. This class could be very fun if your entire group is really excited about the project. If you can’t find a dedicated team you should definitely scale down the project fit for the team members, otherwise you will not have a good time and this class could be very painful for you. This is a self-directed project and requires a lot of individual research to figure out exactly what you need to do to make it work. The lectures are mostly about group management and progress reports; so don’t expect to be told when to start working. Just start working as soon as possible. Good luck!

Section 11 : References

Nexys 4 Artix-7 FPGA Board specs:
http://www.digilentinc.com/Products/Detail.cfm?NavPath=2,400,1184&Prod=NEXYS4
http://www.digilentinc.com/Data/Products/NEXYS4/Nexys4_RM_VB2_Final_5.pdf

Xilinx Designing High-Performance Video Systems in 7 Series FPGAs with the AXI Interconnect:
http://www.xilinx.com/support/documentation/application_notes/xapp741-high-performance-video-AXI-interconnect.pdf

Crossing Clock Domains:
http://www.fpga4fun.com/CrossClockDomain1.html

Cycle Accurate N64 Emulator:
https://github.com/tj90241/cen64

How Emulators Work:
http://stackoverflow.com/questions/448673/how-do-emulators-work-and-how-are-they-written

PIF Information:
https://code.google.com/p/mupen64plus/wiki/SoftResetNotes

Introduction to N64 SI Devices:
http://level42.ca/projects/ultra64/Documentation/man/pro-man/pro26/26-01.html

Cartridge Interface / Replacement (USB?):
http://bit.ly/1rRlCx8
http://64drive.retroactive.be/64drive_hardware_spec.pdf

Audio/Video Interfaces (VGA and RCA):
www.pumpingstationone.org/2013/04/nerp-fpgaok/

Controller Specifications:
http://afermiano.com/index.php/n64-controller-protocol
http://www-inst.eecs.berkeley.edu/~cs150/fa04/Lab/Checkpoint1.PDF

RCP Things:
http://www.dragonminded.com/n64dev/Reality%20Coprocessor.pdf

RCP Chip Information:
http://en.wikipedia.org/wiki/RCP_(chip)
http://en.wikipedia.org/wiki/Reality_Coprocessor

RDP Test (draws a rectangle):
https://github.com/PeterLemon/N64/blob/master/RDP/32BPP/Rectangle/FillRectangle/Cycle1FillRectangle320x240/Cycle1FillRectangle32BPP320X240.asm

RSP Stuff:
https://github.com/cxd4/rsp

Thread and Collection of Documents:
http://www.emutalk.net/threads/53938-N64-tech-documentation

Collection of Hardware Docs N64:
http://hwdocs.webs.com/n64

Full Paper on NES in FPGA:
https://entropia.de/wiki/images/4/47/GPN13-Nintendo_Entertainment_System_on_FPGA.pdf

N64 Programming:
http://en.wikibooks.org/wiki/N64_Programming
https://www.dropbox.com/s/urjq83d0fn3ikk3/Nintendo_Ultra64_Programming_Manual_and_Addendums_text.pdf?dl=0

System Architecture:
http://sourceforge.net/p/n64dev/code/HEAD/tree/trunk/n64dev/docs/
http://sourceforge.net/p/n64dev/code/HEAD/tree/trunk/n64dev/map.txt

N64 Architecture Overview and Programming Manuals:
http://n64.icequake.net/doc/n64intro/kantan/step1/index.html

R4300i Assembler/Disassembler:
http://viper.shadowflareindustries.com/index.php?downloads=

N64 Technical Specifications:
http://en.wikipedia.org/wiki/Nintendo_64_technical_specifications

R4300i overview:

http://www.futuretech.blinkenlights.nl/prod_overview_book.html
http://people.cs.clemson.edu/~mark/R4300i.pdf
http://www.futuretech.blinkenlights.nl/n64.html
ftp://cftp.clevo.com.tw/MG100/Datasheet/Nec/ch03.pdf

R4300 ISA (complete):
https://www.dropbox.com/s/o5xlib1m66lukcm/N64%20CPU%20ISA.pdf?dl=0

N64 ROMS Interrupts not required (first thing should get working if continuing):
https://github.com/PeterLemon/N64/tree/master/CPUTest/CPU/ADD