Nintendo Entertainment System Hardware Emulator

CE 392 Capstone Final Report

Group 2: Veronica Gong, Tara Shirvaikar, James Williams, Megs Yadav

Professor David Zaretsky

May 27th 2020

Table of Contents

List of Figures

Executive Summary

Our capstone project is centered around an FPGA-based implementation of the original Nintendo Entertainment System (NES). We implemented our NES using a modular approach to break the system down into smaller blocks which are implemented as Verilog modules or VHDL entities. These modules were then instantiated inside of a larger top-level entity to give rise to the full system. Our design is versatile in that it can be adapted to fit a variety of video standards such as VGA, HDMI, NTC, PAL, and controller standards such as the original NES joypad, PS2 keyboards or other user-specific interfaces. The primary limitation of our design is its lack of support for mappers, which limits it to only early Nintendo games which fit inside of the 32kB of program memory and 2kB of character memory.

1 Introduction

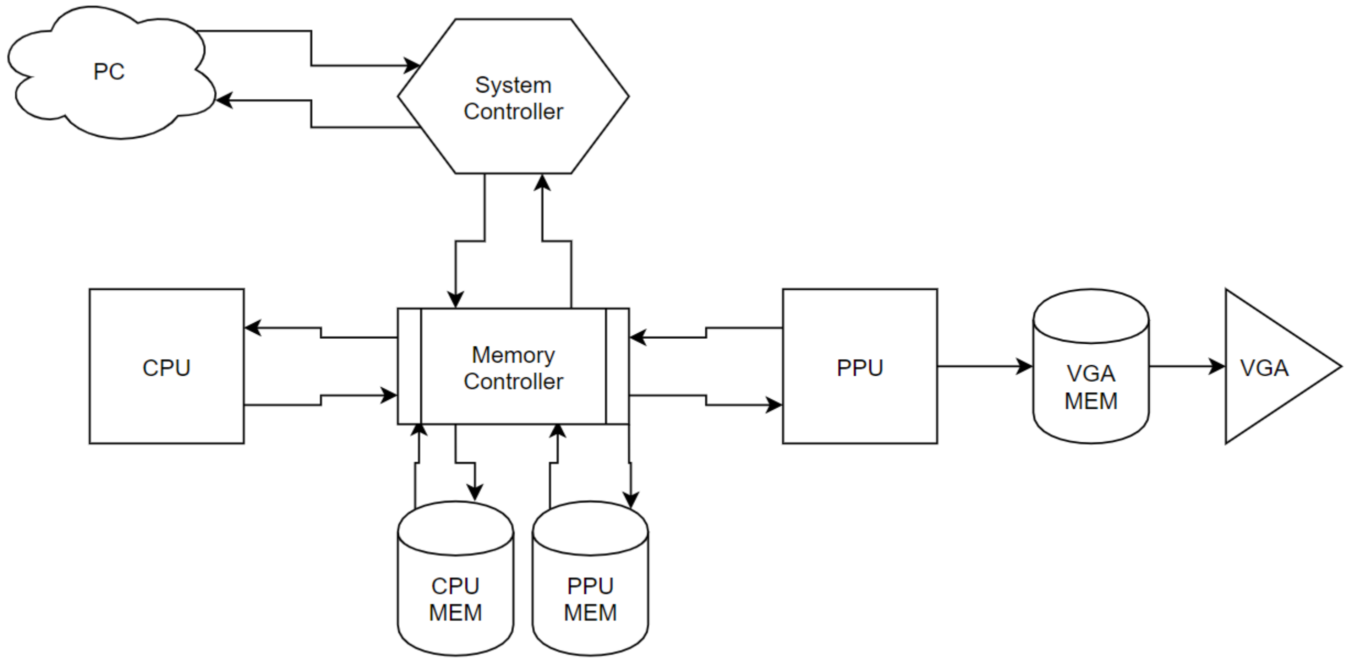

Figure 1: A block diagram of our NES implementation.

The goal of our project is to design hardware which can accurately emulate the original hardware of the Nintendo Entertainment System (NES). Our approach is to break the system down into four main components: the CPU, the Picture Processing Unit (PPU), memory controller, and system controller. The CPU is responsible for running the game and updating the game state. The PPU is responsible for rendering each game frame based on information provided by the CPU. The memory controller manages the memory access and mappings for the CPU and PPU, and the system controller allows a program running on a PC to upload and run games on our hardware design. Several other FPGA-based NES designs exist [1][2][3]. These approaches are less modular in general and only offer support for the NES joypad. They do offer additional features not found in our design including SD card and mapper support.

2 Broader Considerations

An emulator, broadly speaking, allows the user to run game software on a platform for which it was not originally intended. Because of the original non-intention, there has been talk about the ethical and legal gray areas surrounding the creation of commercial emulators, as console manufacturers claim it to be ”copyright infringement” [4]. However, there has been strong pushback from the gaming community and tech enthusiasts that emulators such as ours (not intended for commercial use) are good not only because it provides an enhanced and personalized gaming environment (for example, translation of games into foreign languages) as well as promoting nostalgia, but it also emphasizes the importance of developing for backwards compatibility. Because emulators keep this in mind, they allow the user to play old games (which new manufactured hardware have rendered obsolete), new games, as well as competing games, thus allowing a degree of flexibility and user awareness unmatched by larger manufacturers. Although the commercial prospects of emulators in industry aren’t good due to issues mentioned above, the impact that an NES emulator can have on society and the broader technical community is huge, as it allows for a far greater audience to have access to a unique and fun gaming experience.

3 Design Constraints and Requirements

3.1 CPU

Instructions for the 6502 CPU can be between 1 and 3 bytes in length. In the original 6502 CPU, these instructions were used as inputs to a ROM chip which would then set the CPU control lines accordingly. Our design will use a separate approach where we use an Instruction Fetch (IF) and Instruction Execute (IE) module. Our design must be able to flexibly decode any one of the 151 valid opcodes as well as be able to properly handle interrupts coming from itself, the PPU, and the user [5].

3.2 PPU and Video Path

The PPU has two constraints. First, it must be able to draw the entire frame within the dead time of the VGA controller (7.5ms). Second, the PPU must render the sprites and background simultaneously, it cannot render them independent of one another. This is because the CPU needs to know which line of the frame the PPU is currently rendering so that it can change pointers within the PPU. This allows a display in which one section of the screen scrolls and one section of the screen is stationary.

3.3 Other Bugs and NES Quirks

There are several known bugs and hardware issues present in all NES systems. Most games for the NES are written with these bugs in mind, therefore our hardware must reproduce these bugs accurately. One such bug is buffered VRAM results whenever an access to the nametables or pattern tables is made by the CPU. Some games, including Super Mario Bros, rely on this bug when loading background data from the pattern tables. Other bugs include strange mirroring configurations for the color pallets and complex behavior of the PPU's internal address register.

4 Design Description

4.1 CPU

Figure 2 displays the block diagram of our CPU implementation. The CPU is broken down into two major sections: instruction fetch and instruction execute. The instruction execute block contains the ALU, which performs operations and sets status flags.

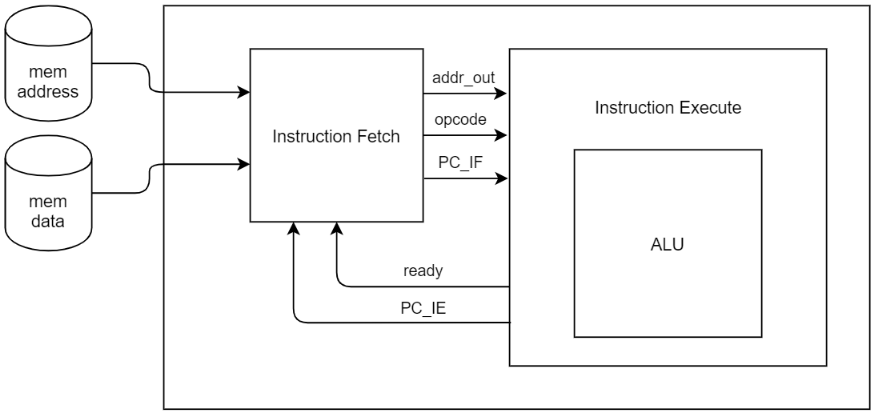

Figure 2: A block diagram of our implementation of the CPU.

4.1.1 Instruction Fetch

The goal of the Instruction Fetch (IF) module is three fold. First, it must determine the length of the instruction and increase the program counter (PC) by that length. Second, it must simplify the opcode by determining the correct memory address that this instruction requires. The 6502 CPU has 13 different addressing modes, each one providing different ways to access memory locations and contents of registers. The IF module is responsible for consolidating the giving this information to the Instruction Execute (IE) module. Third, this module sets several flags based on whether the current instruction uses immediate addressing modes, needs to load or store a value, and what functions it requires the ALU to perform.

The general structure of IF is as follows. After receiving a signal from IE that it is idle and awaiting an instructions, it enters the load portion of its FSM. The next PC generated by IE is sent out on the address bus to retrieve three consecutive bytes of data, each of which are then stored in a local register. Because of the variability of instruction length, the actual instruction length and location of the next instruction is determined after three instruction bytes are loaded. After the instruction has been loaded, it is decoded based on the addressing mode, the length of instruction, and finally the flags are set as well to alert IE to the specific characteristics of that instruction.

Inside IF, we initialize a module that, given two addresses in memory, will return the data stored in both locations. This is used for the indirect addressing modes, namely the pre-indexed indirect and post-indexed indirect addressing modes. Because of the latency incurred when accessing memory twice, instructions that use this mode are forced to wait in a separate state in the state machine until the memory retrieval is done so as to prevent bus contention.

4.1.2 Instruction Execute and ALU

Instruction Execute (IE) is responsible for executing instructions loaded by the IF module. IE takes in the simplified opcodes and their corresponding flags from IF, then uses a state machine to determine the path of the instruction. IE also instantiates the ALU and the interrupt controller.

IE has four main states: load, ALU operation, store, and interrupt check. The load state is triggered whenever the contents of a memory location is needed to perform an operation. The load state first loads the address bus with the address provided by IF, and then waits two cycles for the data to be available on its input. During the ALU operation, IE decided what inputs should be provided to the ALU based on the flags provided from IF. During the store operation, the results of the ALU operation are stored in their appropriate destinations. Before starting the IF on fetching the next instruction, IE starts the interrupt handler to check for interrupts. The details of the interrupt handler are covered in section 4.1.4.

Some instructions such as jump to subroutine (JSR) and return from subroutine (RTS) have their own state machine paths as these instructions don't involve ALU operations and require several loads and stores to push and pull the program counter from the stack.

4.1.3 ALU

The ALU performs the arithmetic and logical functions within IE and sets the flags for the processor status. It takes in 2 arithmetic inputs from Instruction Execute, the opcode of the instruction - an ALU specific opcode - and the current processor status. Its outputs are the result of the arithmetic/logical operation, an updated processor status based on the operation and/or opcode, and a flag that indicates to IE whether it should disregard the result from the ALU. This last input is used to ignore the result of compare instructions which are only used to modify the processor flags.

The ALU first looks at the ALU opcode to see which of the 6 operations the instruction is performing. These operations are add, subtract, shift, AND, OR, and XOR. It then looks at the opcode of the instruction to perform the specific operation for the instruction. While some opcodes perform the same operation and can be implemented together, such as increment x and increment y, due to the inputs being loaded into the ALU from IE, some of the instructions have specific nuances that must be implemented individually, such as add with carry and subtract with borrow. Since all instructions that set flags or have an operational ALU output will go through the ALU, an instruction with an ALU opcode that does not correspond to the 6 operations mentioned earlier but can still edit the processor status will still pass through the ALU but skip to where the flags are set.

Individual flags of the processor status are set after the operation assuming the opcode can edit the flag. There are also many cases where specific instructions set or reset flags depending on the operational output. These are handled separately by instruction if the logic differs, but some logic for the flags, such as the zero flag, do not differ across instructions and can be implemented for all instructions together.

4.1.4 Interrupt Controller

The interrupt handler (IH) is implemented as a finite state machine which checks the three possible sources of interrupts at the end of execution for each instruction. When IE is done executing the current instruction, it gives the start signal and provides it with the current PC, stack pointer, and processor status. If IH has detected that an interrupt has occurred, it first loads the interrupt vector from the appropriate location. It then pushes the current PC and the processor status to the stack and provides IE with a new PC and stack pointer. This method of handling interrupts ensures that IE can be totally agnostic to the process of handling and returning from interrupts, allowing IH to handle all of the details of interrupts.

4.2 PPU and Video Path

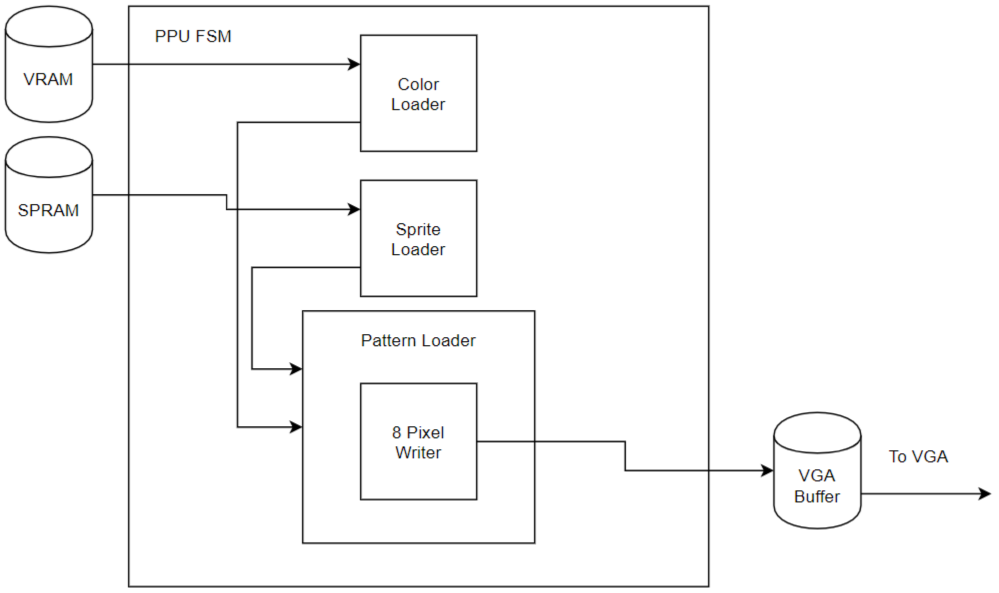

Figure 3: A block diagram of our implementation of the PPU.

Figure 3 displays the block diagram of our PPU implementation. Our PPU design is broken down into five state machines: the color loader, the sprite loader, the pattern loader, the 8 pixel writer, and the top level PPU state machine. At the beginning of the frame, the PPU state machine first triggers the color loader to load in the sprite and background color pallets from VRAM. Once the color loader has finished, the sprite loader begins to load sprites which are visible on the line to be drawn next by the PPU. The sprite loader loads up to 8 sprites and provides the two sprites which could be drawn in a group of 8 pixels to the pattern loader. Once the sprite loader has finished, the pattern loader begins to render the first 8 pixels of the screen.

The pattern loader may start on a negative column so that it is always loading an entire background pattern each cycle instead of having to load two and then stitch them together. After the pattern loaded has loaded the appropriate background and sprite patterns, it provides this information to the 8 pixel writer which then writes the composite 8 pixels into the VGA buffer. The process of triggering the pattern loader is repeated across 256 columns of the screen, advancing 8 columns after each trigger. Once the PPU has finished with one line, it triggers the sprite loader again so that the sprites visible on the next line can be cached for the pattern loader. Once the PPU has finished rendering a frame, it waits until the VGA controller begins to render the frame and then resets itself.

The VGA controller is a standard VHDL-based VGA timing generator used in 355 with two additional modifications. First, a signal denoting when the VGA controller has exceeded line 240 is generated by checking the current VGA row. This is needed to indicate to the PPU when it is safe to draw in the video buffer. Second, a color decoder was added to convert the color pallet entries of the NES into RGB values.

4.3 Memory Controller

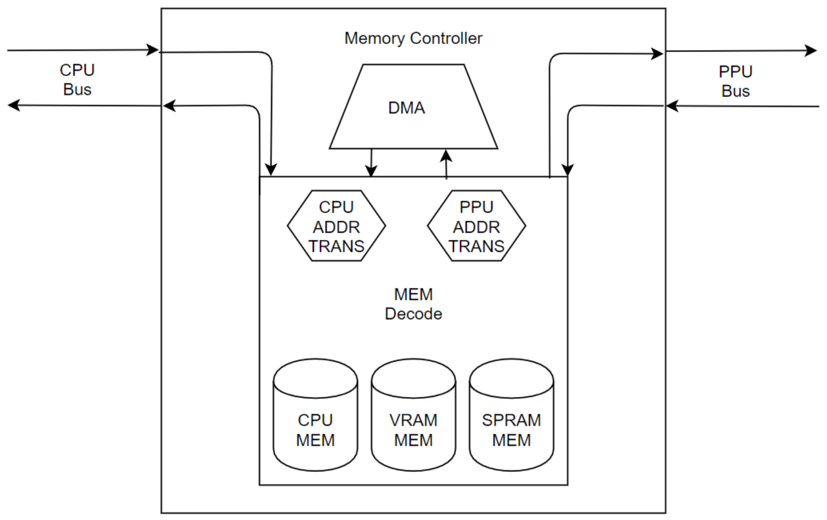

Figure 4: A block diagram of our implementation of the memory controller.

Figure 4 shows a block diagram of our memory controller. The memory controller implements three functions. First, it manages the memory spaces for the PPU and CPU using address translation and block RAMs resized to match the amount of usable memory. Second, it implements the memory mapped registers which are used by the CPU to set various options on the PPU and communicate with the PPU's memory. Third, it implements the DMA controller which is used to perform a fast memory copy operation used by the CPU to quickly move new sprite information into the PPU's sprite memory.

4.4 System Controller

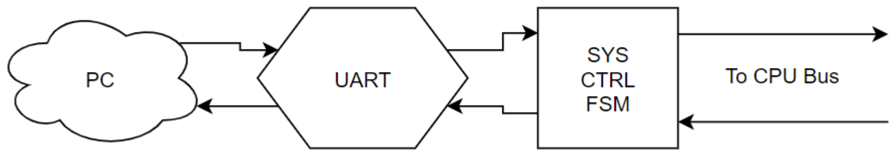

Figure 5: A block diagram of our implementation of the system controller.

Figure 5 shows a block diagram of our system controller. The system controller is implemented as a finite state machine which accepts commands via UART and access the NES via the CPU's memory bus. The system controller can perform five tasks: halting the CPU, resuming the CPU, resetting the CPU, reading a byte from the CPU's memory address space, and writing a byte to the CPU's memory address space. The PPU's memory and sprite memory are accessible from this address space because their access registers are mapped to the CPU's memory.

4.5 Keyboard Module

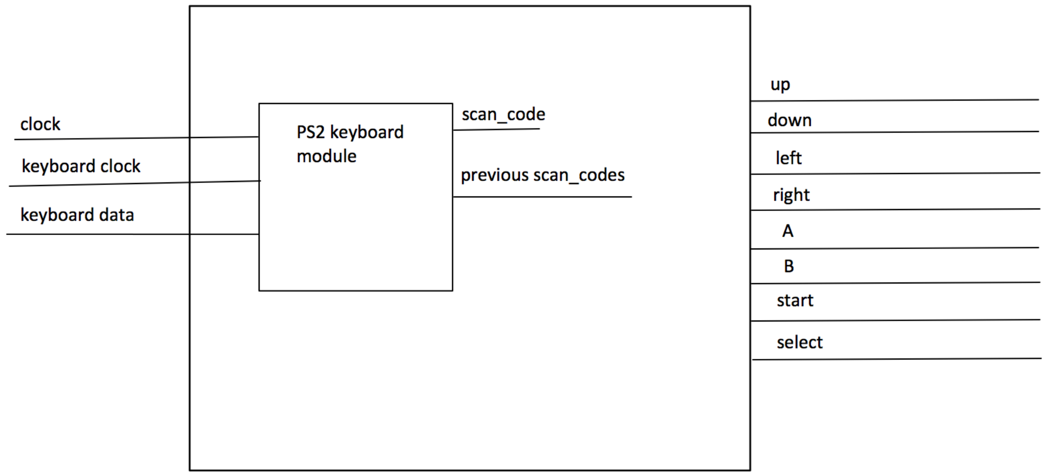

Figure 6: A block diagram of our implementation of the keyboard module.

The keyboard module, shown above in Figure 6 , is responsible for providing the user inputs to play the game. Eight keys on the PS2 keyboard represent up, down, right, left, A, B, start, and select. The ps2 module (also containing keyboard and oneshot modules) was used to read the keyboard codes corresponding to which keys were pressed. The top-level module used the outputs from ps2 (scan code, hist0, hist1) to generate one-bit outputs for each key corresponding to the status of the key. The signal is set to '1' if the key is pressed and '0' if it is not. The logic was implemented independently for each key so that multiple keys' signals can be set to '1' at the same time.

5 Design Optimizations

Several operations were done to reduce the amount of memory usage. The CPU's memory was reduced from 64kB to 55kB to avoid wasting the logical memory space used by the NES to mirror the PPU control registers. The PPU's memory was reduced from 16kB to 12kB through a similar technique designed to reuse mirrored addresses. The VGA buffer size was also reduced from requiring 8-bits per color to 6-bits per color.

The rest of the optimizations were performed using the synthesis tools. Register relocation was used to combine redundant registers and dangling logic removal was used to remove unused registers and combinations logic.

6 Testing and Simulation

6.1 CPU

6.1.1 Instruction Fetch

In order to test IF, a simple bubble sort program was written in 6205 ASM and assembled into a list of bytes. Since the memory is byte-addressable, this was made into a txt file structured as a 16 bit hex address word and an 8 bit data halfword on each line. The IF testbench first reads this into memory via a generic RAM module and then uses this simulated memory to test whether IF can successfully access memory, store it, and index it in its instruction register. It is also important to note that when the testbench was trying to write to memory initially, it was necessary to make sure reset was low to prevent IF from reading unititialized memory.

6.1.2 ALU

To test the ALU, many different combinations of inputs, opcodes, ALU opcodes, processor statuses, and abilities to edit the processor status were used as inputs to the ALU to observe the ALU's outputs. Each operation has been tested with every possible combination of inputs that would create a different operational outfit, such as inputA > inputB and vice versa and accounting for overflows. Each operational output has given outputs as expected, but this was the simpler aspect to test. The processor status flag testing has been the more nuanced aspect of testing that has raised questions and led to changes in logic. The reason for this is that certain flags are set differently depending on the current processor status and the instruction meaning there are many cases to test for every opcode that enters the ALU.

6.1.3 Instruction Execute

The IE module was instantiated along side IF in order to create a working 6502 CPU. This CPU was then loaded with a program sourced from Github [6]. This program allowed us to test the functionality of all 151 op codes. It also allowed us to uncover subtle 6502 quirks and requirements such as certain bits being set when the processor status is pushed and pulled from the stack. This program also performs a memory check to ensure the memory is not corrupted during the program's execution.

6.2 PPU and Video Path

In order to test the performance of the PPU and video path, two separate tests were developed which integrate the PPU, VGA controller, and memory controller so that all three can be tested at the same time. In the first test, a test bench is used to instantiate the PPU, VGA controller and memory controller with the appropriate connections. The PPU's memory is the loaded with a known graphics design and the PPU state machine is allowed to render one frame. After the PPU has finished rendering the frame, the memory is inspected to see if the correct graphics were rendered.

In the second test, the PPU, VGA controller and memory controller are created inside of the FPGA itself. This allows us to test the hardware in operation and verify that timing is met. Testing using the FPGA itself also allows us to detect subtle graphics bugs which may not be apparent from looking at the raw contents of the VGA buffer.

7 Implementation and Synthesis

The inputs of our design consist of the 50MHz clock supplied by the on-board crystal oscillator, the reset button, the CPU halt button, the PPU reset button, the CPU soft reset button, and the keyboard clock and data lines. The outputs of our design are the VGA color and timing signals along with the signals that drive the 7-segment displays on the DE2-111 board. These 7-segment displays are used to display the current address of the system controller and the current program counter of the 6502 CPU.

Our design uses the block RAM IP modules. These modules are inferred from a parameterized Verilog module which allow the synthesis tool to correctly implement synchronous pass-through logic. This module had to be refined several times until the synthesis tool correctly recognized it as block RAM.

Our target frequency for this project was 25MHz. This frequency was selected because it is the frequency used to drive the VGA DAC. The design met timing constraints without additional modifications being necessary.

7.1 Resource Utilization

Our design was successfully synthesized with 5133 logic elements, 1926 registers, 1148928 bits of block RAM and 98 pins.

8 Final Design Testing

We tested our final design with three games: Super Mario Bros, Donkey Kong, and Pac Man. Super Mario Bros was chosen in particular as it is notoriously tricky to emulate correctly and makes use of a critical PPU feature (sprite 0 hit).

Super Mario Bros was problematic at first. The game seems to be setting the bits which control the sprite and background pattern tables incorrectly, so these values were hard-coded into the PPU. The game also seems to want to set the scroll pointer at an inappropriate time, causing glitches on screen. This was also patched by rejecting scroll pointer updates until after the sprite 0 hit flag had been set. The colors were also wrong at first with the sky appearing black. This was fixed by correctly setting up mirroring for the color pallet. There were two major problems with Super Mario Bros. First, the attribute table bytes are not being loaded correctly. This is causing the PPU to select the colors of the frame incorrectly. This problem was traced back to incorrect bytes being written into the game's attribute buffer. The second problem involves the fire spinners present in the dungeon levels of the game. These fire spinners normally look line rotating lines of fire, however in our emulator they seem to appear in a quickly spinning circle which is impossible to avoid. Both of these problems were traced back to an incorrect implementation of the rotate right and logical shift right instructions.

Donkey Kong seems to work fine except for one issue: the joycon controller. While the joycon controller works fine for Super Mario Bros, it seems to be completely inoperable for Donkey Kong. Furthermore, pressing keys on the keyboard while Donkey Kong is running can cause strange events to happen in the demo screen such as Mario dying unexpectedly. It was also determined during debugging that small modifications to the joycon controller can cause the entire game to crash on a key press.

Pac Man's major issue is its inability to correctly select the background and sprite pattern tables. This leads to visible corruption on the screen due to the background and sprites being rendered incorrectly. Pac Man can still be played however as it responds well to key presses, and a corrupt version of Pac Man can be controlled on the screen.

9 Conclusions

Our design met our primary requirement in that it was able to play Super Mario Bros without introducing bugs which rendered the game unplayable. It did not however live up to all of our specifications as it was unable to play other games such as Donkey Kong and Pac Man. It also currently has no support for games which use mappers to swap memory banks during gameplay.

The most useful feature to add would be support for memory mappers. This would allow a variety of other games to be played including some of the more modern NES games which can exceed 1MB of program ROM and 150k of character ROM. Another useful feature would be support for the audio processor. The original NES had a small PWM-based audio processor located inside its 6502 CPU. This audio processor was not implemented in our design due to time constraints.

References

[1] B. Bennett, "Brian bennett fpga nes," Jul 2012. [Online]. Available: https://github.com/brianbennett/fpga_nes

[2] Strigeus, "strigeus/fpganes," Jan 2014. [Online]. Available: https://github.com/strigeus/fpganes

[3] BW0ng, "Bw0ng/fpga-nes." [Online]. Available: https://github.com/BW0ng/FPGA-NES

[4] J. Conley, "Use of a game over: Emulation and the video game industry, a white paper," April 2004. [Online]. Available: https://scholarlycommons.law.northwestern.edu/cgi/viewcontent.cgi?article=1022&context=njtip

[5] P. Diskin, "Nintendo entertainment system documentation," August 2004. [Online]. Available: http://nesdev.com/NESDoc.pdf

[6] Klaus2m5, "Klaus2m5/6502-65c02-functional-tests," Jan 2020. [Online]. Available: https://github.com/Klaus2m5/6502_65C02_functional_tests