

# 6.205 6502: NES Emulator

Benjamin Bajd  
dept. of EECS  
Massachusetts Institute of Technology  
Cambridge, MA, USA  
benbajd@mit.edu

Holden Hall  
dept. of EECS  
Massachusetts Institute of Technology  
Cambridge, MA, USA  
hhall314@mit.edu

**Abstract**—We present a system for emulating the Nintendo Entertainment System on an FPGA. An FPGA provides significant advantages over traditional software emulation, allowing for more direct emulation of the hardware and, as a result, higher accuracy with significant improvements regarding latency. The system consists primarily of a central processing unit, which is a modified 6502, and the picture processing unit, which handles displaying the graphics.

**Index Terms**—FPGA, video games, emulation

## I. SYSTEM OVERVIEW

The system consists of the following primary modules:

- Central Processing Unit. A special 6502 processor which executes the game code.
- Picture Processing Unit. The NES graphics driver. Controlled by the CPU, this allows the system to output graphics via an HDMI port.
- Memory Controller. Routes CPU memory to different locations based on the address.
- ROM. Provides the game code that would be included on a cartridge on a real NES. Also contains the PPU memory.
- IO. Provides a memory-mapped interface with the physical controller.

These fundamental blocks interface with each other primarily through memory addresses. These connections are depicted in Fig. 1.

Of these modules, the two most substantial are the Central Processing Unit and the Picture Processing Unit.

## II. CENTRAL PROCESSING UNIT

The Central Processing Unit, or CPU, processes instructions which describe the game logic by interacting with values in registers and memory.

The CPU maintains six registers that allow it to perform arithmetic, manage the stack, and keep track of flags and the program counter. It also has wires connecting it to the memory controller for memory access to the CPU's RAM and ROM, as well as controller input. Specific memory locations also allow the CPU to interact with the PPU.

The logic of processing instructions is divided into multiple steps, some of which are implemented as modules. First, an instruction is fetched from memory based on the program counter and decoded. Next, the instruction's addressing mode is used to obtain the operand from memory if the instruction

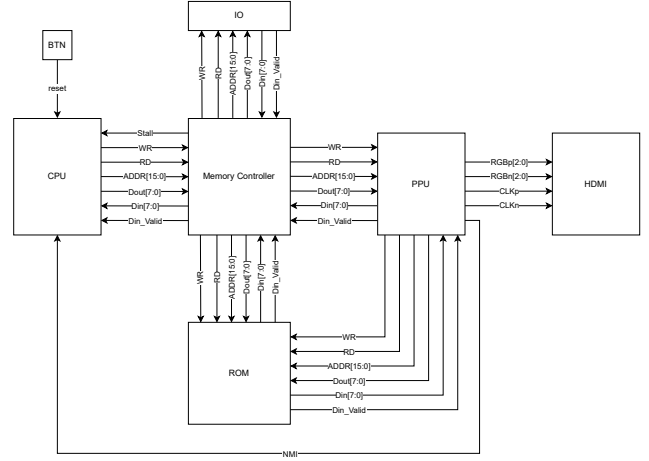


Fig. 1. Top level overview.

uses one. The instructions are then divided into two groups: those that do not jump or operate on the stack and those that do. The first group is processed by the ALU which performs all the necessary computations and register updates. The second group is processed by the jump and stack module which modifies the stack and performs the necessary jumps. Lastly, the result of the instruction is written to memory. CPU's block diagram is shown in Fig. 2.

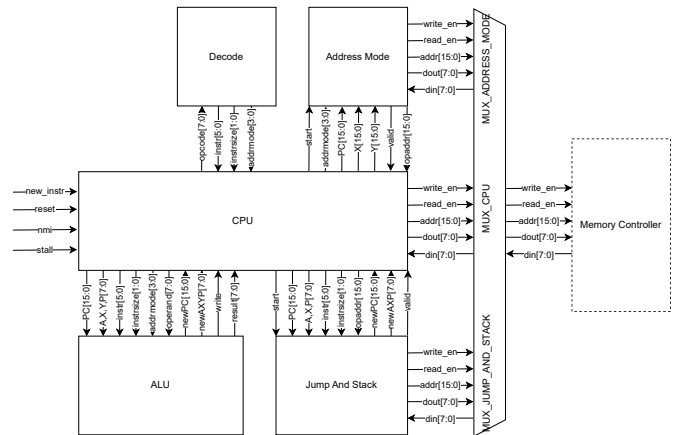


Fig. 2. CPU overview.

The CPU also has input wires corresponding to interrupts and stalls which can arise from the PPU, instructions, or DMA. Interrupts push and/or pull values from the stack (usually the program counter and flags), make updates to some of the registers, and jump to an address at a predetermined address in ROM based on the type of the interrupt. They are implemented as a special case of the jump and stack module, and take up an entire instruction cycle.

#### A. Decode

Decode is a combinational module that takes an opcode of the instruction as an input and produces the instruction's type (of 56 total instructions), size (immediate or memory operand might follow the opcode), and addressing mode (that determines how the operand is obtained from memory).

#### B. Address mode

Address mode is implemented as a minor FSM that has wires connecting it to the memory. It takes the instruction data and necessary registers, and produces the memory address of the operand based on the addressing mode if the instruction uses one.

#### C. ALU

The ALU processes all instructions than do not jump or modify the stack. It is a combinational module that reads the instruction and operand data, and computes the updated register values and a result of the instruction.

#### D. Jump And Stack

The jump and stack module processes instructions that jump or modify the stack, as well as interrupts, and processes reads and writes to the stack. It is implemented as a minor FSM and has wires connecting it to the memory.

1) *Instructions*: These instructions push or pull multiple values like the program counter on the stack, or jump to a different instruction in memory. The module performs the necessary stack modifications and computes the updated register values.

2) *Interrupts*: All interrupts first push the program counter to the stack to jump back to it later, as well as optionally the flags register, and then read the address of the interrupt handler that is stored at a predetermined address in ROM based on the type of the interrupt.

### III. PICTURE PROCESSING UNIT

The Picture Processing Unit, or PPU, is responsible for displaying the graphics. It can display two types of graphics: sprites, and background tiles.

#### A. Background Tiles

The NES contains a 64 by 60 grid of 8-pixel by 8-pixel tiles. A 32 by 30 window can be placed anywhere in this grid, including extending past the edges, in which case it will wrap to the beginning. In reality, only half of this grid is unique, with the other half being a copy. The exact ways in which this copying works depends on the game. For example, Super

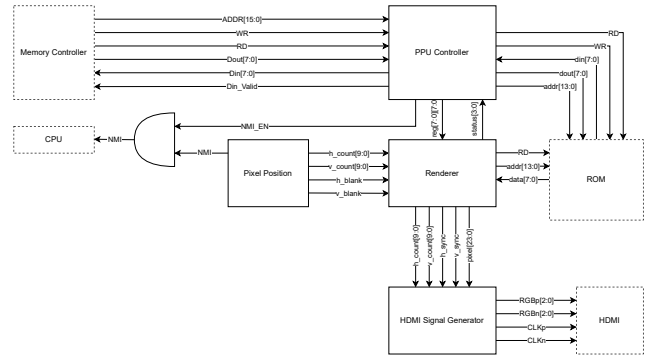


Fig. 3. The PPU overview.

Mario Bros. uses the top 64 by 30 area, with the lower 64 by 30 area being a copy of it.

Drawing each tile requires loading four bytes from memory. The first byte is a tile index which says which image to use for that tile. The second is the attribute data which says which colors to use. The third and fourth make up the actual pixels of the tile.

The processes for loading and displaying tiles is depicted in Fig. 4.

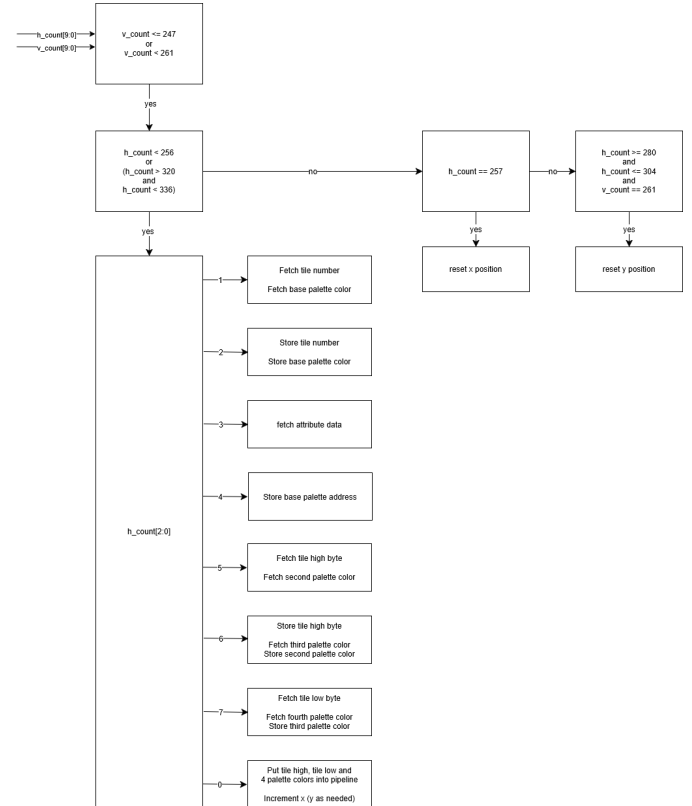


Fig. 4. The tile renderer

## B. Sprites

Unlike background tiles, which are fixed to a grid, sprites are objects that can appear anywhere on screen. The NES can hold up to 64 sprites, and it can render up to 8 of them per line. Sprite data consists of 4 bytes: the x position, the y position, the sprite index, and the attributes. The attributes choose the color, the layer, and allow the sprite to be flipped. The tile index points to the location of the two bytes that make up the actual image, similar to the background tiles.

To render the sprites, on each line, the first 8 sprites that appear on the next line are loaded into 8 sprite render units. These output the sprite pixels whenever the horizontal count falls in the range of the sprite.

## C. Controller

The PPU memory is not directly accessible by the CPU. Instead, the CPU interfaces with the PPU by writing to a few memory mapped registers to control various aspects of rendering. In addition, a DMA transaction can be performed to copy sprite data from CPU memory into PPU memory every frame. The controller module implements these registers and controls the PPU.

## IV. MEMORY CONTROLLER

The memory controller is responsible for maintaining the NES's memory and providing read and write access to it to the other modules.

The address space is 16-bit wide, so the size of the memory is 64 KB. The memory is mapped as follows:

- Addresses 0x0 to 0x2000 map to RAM.
- Addresses 0x2000 to 0x4020 map to IO registers.
- Addresses 0x4020 to 0x6000 map to expansion ROM.
- Addresses 0x6000 to 0x8000 map to SRAM.
- Addresses 0x8000 to 0x10000 map to ROM.

Some address ranges are further mirrored, specifically, only the bottom 11 out of 13 bits determine the RAM address, while the 8 PPU registers are mirrored from 0x2000 to 0x4000.

The RAM is only utilized by the CPU, while the ROM is also accessed by the PPU. As such, the memory controller provides the CPU with wires to the entire address space, while the PPU is only provided with wires to the ROM. Since some of the IO registers are used for CPU-to-PPU communication, the wires from CPU to the memory controller are extended to the PPU so that it can respond to CPU accesses to such registers. Memory controller's block diagram is shown in Fig. 5

The memory is implemented as BRAM on the FPGA.

### A. RAM

The RAM comprises the lower 8 KB of the address space and can be read from and written to by the CPU. The addresses 0x0 to 0x100 correspond to the zero page and the addresses 0x100 to 0x200 correspond to the stack. The actual size of the RAM is 2 KB which is then mirrored three times.

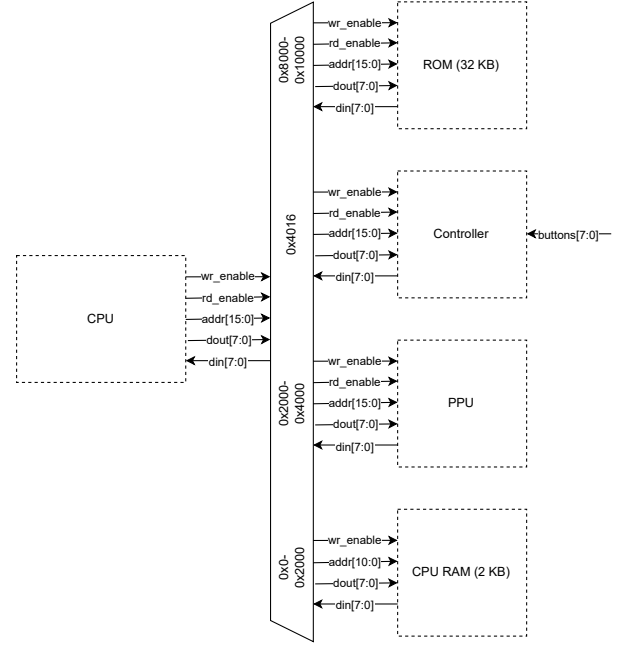


Fig. 5. Memory controller overview.

## B. IO registers

The IO registers are special registers the CPU can access to interact with other components of the NES like the PPU, the APU, or the controller. Addresses 0x2000 to 0x2008 (mirrored to the entire 0x2000 to 0x4000 range) are used for interaction with the PPU, while the addresses 0x4016 and 0x4017 are used to read the controller input. The register 0x4014 is used for direct memory access.

## C. DMA

Since copying larger amounts of data from CPU to PPU through IO registers requires many instructions, the register 0x4014 provides access to DMA. A single write to 0x4014 copies 256 B of data from the CPU to the PPU's sprite RAM in one instruction. A wire leading back to the CPU is set to high while the DMA is in progress to stall the CPU.

## V. ROM

A physical NES cartridge consists of a ROM chip with the game code and a ROM chip with the PPU graphics data. The ROM module in this system consists of BRAMs to implement these features, and it also features BRAMs for the different types of PPU RAM. The reason for adding these features to the ROM module is to simplify the PPU, since it just needs to read and write to the ROM module for all of its memory accesses instead of needing to manage separate memory locations. In addition, NES cartridges can control how the PPU memory is mapped, so this approach allows more flexibility in implementing different cartridges, since only the ROM module needs to be changed and not the PPU module.

as well. For testing the PPU, the ROM module is flashed with the game code in the BRAMs. In the future, a UART interface will be implemented to allow games to be transferred to the FPGA without having to regenerate the bitstream.

The ROM module has 7 ports:

- The PPU renderer's ROM port
- The PPU renderer's palette (color) port
- The PPU renderer's OAM (sprite memory) port
- The PPU controller's ROM port
- The PPU controller's palette port
- The PPU controller's OAM port
- The CPU's ROM port

The renderer and controller each get their own port for accessing PPU memory for simplicity. On the actual NES, there would only be one port for the renderer and controller, with control being decided by whether the frame is in the blanking region. We have opted to allow two ports for simplicity, and it shouldn't affect standard behavior because games should only try to access PPU memory when the renderer is not using it (during the blanking region).

## VI. IO

The IO module maps the physical inputs and outputs, from either FPGA buttons or a controller to the memory address 0x4016.

To read the inputs, the CPU first writes the value 1 to 0x4016. This puts the input into polling mode which keeps on refreshing the pressed buttons. Once the CPU writes 0 to 0x4016, the inputs are stored in a shift buffer and returned to the CPU one button at a time every time the CPU reads from 0x4016. The CPU can then determine which buttons are pressed by reading from these memory addresses. Due to its simplicity, it is implemented as part of the memory controller.

Possible expansions to this module would be to also interface with a physical controller or control the Audio Processing Unit (APU).

## VII. EVALUATION

### A. CPU

The correctness of the CPU was first checked in simulation. The test bench can be run on both specified instructions and arbitrary test ROMs created by the community. Next, the CPU code was tested on hardware by displaying the register and memory values on the FPGA or through UART.

1) *Simulation:* The wires connecting it to memory are simulated in Python so all reads and writes that the CPU performs can be checked. For each instruction the simulation first outputs the program counter and the instruction at that program counter (if it was written with the instruction writer script), then simulates the entire instruction cycle while outputting all memory reads and writes, and lastly outputs the values of registers. A custom instruction writer interface was also added to simplify testing custom instructions.

2) *Test ROMs in simulation:* The testbench works with arbitrary ROMs, so community created test ROMs [2] were used to compare the resulting memory values in simulation.

3) *Testing on hardware:* Lastly, the CPU was tested to work on hardware. The seven segment display and the LEDs are used to display register values. For easier testing of memory values, one of the buttons starts a UART transmission of the entire RAM to the PC.

The execution stops after every instruction, and one of the FPGA buttons is used to execute the next one so that registers and memory can be checked for every instruction.

A simple set of instructions was used for testing, and community test ROMs could be used to further test the correctness, but the functioning integration test ROMs indicate that the ones they use seem to be working correctly.

### B. PPU

The validity of the PPU was first evaluated using a testbench. Using a known-good software emulator<sup>1</sup>, the PPU memory was dumped to a hex file. A fake ROM module was used in a testbench to interface with the PPU, and a fake display was used to convert the RGB values into an image. In this way, a frame could be rendered, and the rendering pipeline could be tested. Fig. 6 shows the results of a testbench running for a frame of Super Mario Bros.



Fig. 6. The PPU rendering in simulation.

This can naturally be extended to hardware by implementing simple ROM and top-level modules. Fig. 7 shows the results of the PPU rendering on the FPGA. In addition, the top-level module featured some ability to write to the PPU registers, allowing the scroll of the screen to be configured.

### C. Overall system

Fig. 8 shows the results of the CPU and PPU working together to run a test ROM. This simple ROM sets the palette

<sup>1</sup>Mesen RTA: <https://github.com/threecreepio/mesenrta>



Fig. 7. The PPU rendering on hardware.

to a black and light gray, and displays the text “Hello World” in the top left of the screen. To do this, the CPU must correctly write values to the PPU using the PPU controller interface. A small visual glitch is present in the top left of the image, which is to be expected. This is because this test ROM does not setup the sprite data, and with zeros filling the PPU memory upon power-on, this corresponds to displaying the 0 character sprite in the top left corner of the screen, and it remains partially uncovered by the tiles. For this test, the PPU memory was not pre-loaded with values dumped from an emulator.

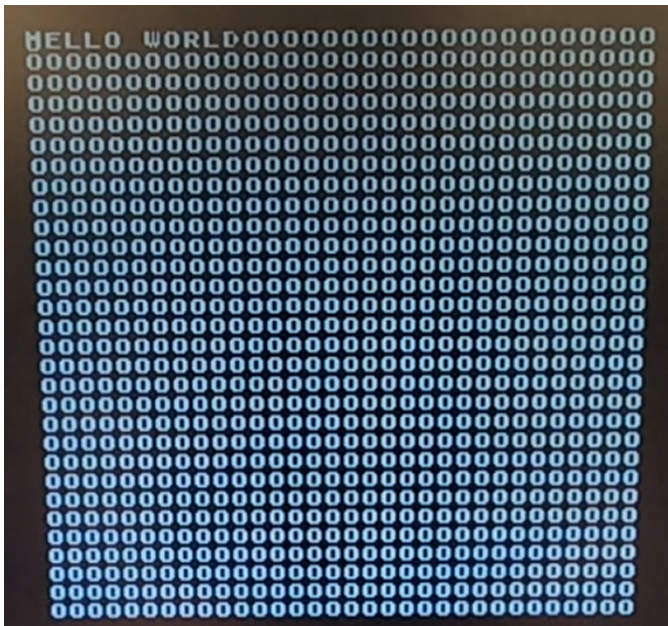


Fig. 8. The CPU and PPU working together to run a test ROM.

Finally, fig. 9 shows the system running a simple test ROM that displays the state of the input devices. The top row of the first array is the relevant block here. It shows the state of

the standard 8 buttons for the first player, which match the rightmost 8 switches on the FPGA.

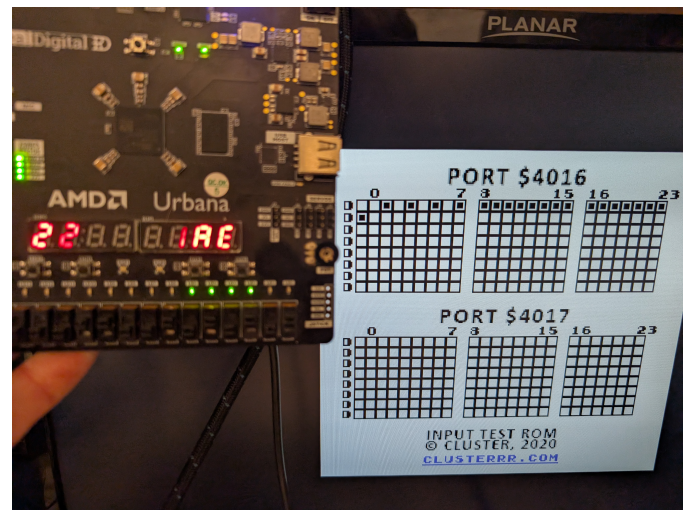


Fig. 9. An input test ROM working.

Although the system was able to demonstrate working for a simple ROM, it was unable to work on more advanced ROMs (such as Super Mario Bros.). This is likely due to some bugs in the interaction between the CPU and PPU which could not be fixed in time. In addition, DMA, which is necessary for rendering sprites for almost all ROMs, was not implemented. As it stands, the system should likely be able to handle simple ROMs which rely on background graphics only and do not rely on reliable timing. As it stands, the system runs in a fixed number of cycles for each instruction, which may cause issues for more complicated interaction with the PPU.

The system does pass timing. The main constraint is the PPU clock which runs at 85.913 MHz. Most of the system runs off of this clock. The system does not include any especially complex calculations, so timing is not an issue. In terms of throughput, the system is capable of outputting slightly over 60 frames per second (very close to the framerate of the actual NES). With a latency of about 1 frame due to the frame buffer used for crossing between the pixel clock domain and the system clock domain.

The system uses 25 block rams of the 75 available. The frame buffer contributes a significant portion of this, and some advanced techniques could theoretically be used to save on this space, but with so much memory to spare, these are not currently necessary.

The original checklist had the following commitment:

- Correct results for all standard NES demonstrated via testbenching
- Correct execution of 6502 code on hardware demonstrated using switches and 7-segment displays
- Correct rendering of arbitrary frames including background and sprite data
- Correct HDMI output of arbitrary frames on hardware with dumped PPU memory flashed onto the FPGA

These have all been met. Additional goals included:

- Integration of the PPU and CPU in hardware. Tested via no-input test ROMs that perform a calculation and display the result on the display
- The ability to read input via a controller, as demonstrated by an input display test ROM
- The ability to play Super Mario Bros. at high enough degree of accuracy that the average person could not tell the difference between the FPGA and a real NES
- The ability to send a game to the FPGA via UART

The system passes the first of these as demonstrated by the “Hello World” rom. In addition, it passes the second as demonstrated by the input test ROM. The system was unable to play super Mario Bros. due to its more advanced nature, relying of features which were not implemented properly or at all. The system also does not feature the ability to load a ROM via UART, with the bitstream needing to be generated with the desired ROM.

None of the stretch goals were implemented. This leaves the project in the middle of the “Goal” range.

### VIII. INSIGHTS

A big takeaway from implementing the project was that having several smaller modules greatly helps to simplify code and ease debugging. Originally, the PPU renderer was one big module, but this got complicated quickly, and splitting it into the sprite and tile renderer, with additional modules like the pipeline register and the renderer address manager helped significantly to make the code easier to understand and work on. Dividing the CPU into multiple submodules, some of which combinational and some of which minor FSM also made the modules much less complex and easier to work with.

Adding interrupts to the CPU seemed difficult initially due to their multiple writes and reads to the stack. Implementing it as a part of the jump and stack module, however, ended up being not too difficult due to their similarity to stack instructions.

For CPU testing in simulation and hardware, manually writing instructions to memory initially was tedious due to different opcodes and instruction sizes. Implementing an instruction writer that wrote instructions based on their type, addressing mode, and immediate simplified this process.

One big challenge with the PPU was managing all of the pipelining to ensure everything lined up, especially with the tiles. The tiles are not all read contiguously, with some tiles being read immediately next to each other, and some having time between, so a FIFO was used to buffer them before displaying them. In addition, scrolling of the background tiles required making some changes to the pipelining to allow for tiles that were partially offscreen.

The interface between all of the modules was somewhat of a challenge. In particular, there were many different read, write, address, and data lines to manage, which led to a lot of mistakes and confusion. In doing the project again, it would be useful to create a better system for managing all of the

memory addresses, perhaps using some custom types or a bus manager module.

### IX. CODE

All code is contained the GitHub repository located at <https://github.mit.edu/6205F25/fa25-6205-team58>.

### X. CONTRIBUTION

Broadly speaking, Holden worked on the PPU and its associated testbenching, and Benjamin worked on the CPU and its associated testbenching. Holden also helped with the CPU testbenching. Both of us worked on the memory controller, with Holden starting it, and Benjamin adding some additional features and fixing bugs. Holden wrote the ROM module. The top level was mostly written by Holden, with some additions by Benjamin. This work division applies to both the code and the report.

Documentation was sourced almost entirely from [https://www.nesdev.org/wiki/NES\\_reference\\_guide](https://www.nesdev.org/wiki/NES_reference_guide). Some test ROMs from <https://www.patreon.com/posts/making-your-nes-137873901> were used to test the CPU. Some test ROMs were also taken from [https://www.nesdev.org/wiki/Emulator\\_tests](https://www.nesdev.org/wiki/Emulator_tests). Mesen RTA (<https://github.com/threecreepio/mesenrta>) was used as a fully-featured known-good emulator to compare results and to dump memory.

### XI. ACKNOWLEDGEMENTS

We thank Professor Joe Steinmeyer and the 6.205 teaching staff for their continued assistance and advice on implementing this project.

### REFERENCES

- [1] (2025, Nov.) NES Reference Guide - NESdev Wiki. [Online]. Available: [https://www.nesdev.org/wiki/NES\\_reference\\_guide](https://www.nesdev.org/wiki/NES_reference_guide)
- [2] (2025, Dec.) Making your first NES emulator — Patreon [Online]. Available: <https://www.patreon.com/posts/making-your-nes-137873901>