

Field Programmable Gate Entertainment System

Sasha Fefelova

Department of EECS

Massachusetts Institute of Technology

Cambridge, MA

fefelova@mit.edu

Joshua Oki

Department of EECS

Massachusetts Institute of Technology

Cambridge, MA

jolting@mit.edu

Dominic Perno

Department of EECS

Massachusetts Institute of Technology

Cambridge, MA

dperno@mit.edu

Abstract—We present an incomplete design for a Nintendo Entertainment System emulator implemented entirely in hardware on a Xilinx FPGA. This emulator recreates the key components of the Nintendo Entertainment System (NES), such as the Central Processing Unit (CPU) and Pixel Processing Unit (PPU), and supports HDMI output, and is controlled via UART. This emulator is cycle accurate, and is capable of running games such as Super Mario Bros. or Donkey Kong as the original NES ran them. We evaluate the emulator’s performance through determining the slack from Vivado and how far it can run games such as Super Mario Bros. or Donkey Kong. Finally, we will discuss areas of expansion or improvement.

Index Terms—Digital systems, Field programmable gate arrays, Emulation,

I. DESIGN OVERVIEW (DOMINIC)

At the top-level, the Field Programmable Gate Entertainment System (FPGES) consists of the following:

- A 6205 CPU, responsible for processing any instructions, this is the main driver of the NES, and will be discussed in further detail in the next section.
- A 2C02 PPU, responsible for rendering and displaying video from the NES to the TV. This will be discussed further in its own section.
- Two memory buses for the CPU and PPU, respectively. They each supply the outputs from the CPU/PPU (addr or data) to whichever other component will use that information, and take in data to send to the CPU/PPU.
- The cartridge. This consists of a **PRG ROM** and a **CHR ROM**, both in BRAM memory, that is preloaded onto the FPGA, currently by flashing the contents of the ROM onto the FPGA. It also contains a mapper, which is a module which maps the address supplied by the CPU to a larger range, for bigger games. At the moment, we only have the default mapper implemented, which doesn’t change the address. This is hooked both into the CPU bus and the PPU bus.
- Working BRAM Memory, which is hooked up to the CPU bus.
- Video BRAM Memory, mostly used for pattern tables, which is hooked up to the PPU bus.
- The NES Controller, which sends eight bits into the FPGA serially, and is converted into a full byte of data for use by the CPU. This is hooked into the CPU bus. We did not have time to implement the actual controller, so it is implemented

II. CENTRAL PROCESSING UNIT (SASHA)

A. What the CPU needs to do

For a given instruction, we need to load, decode, and simulate it. Let’s go through what this means.

- **Load instruction:** All instructions live in memory as bytes. We need to access them to be able to simulate them. We keep a program counter register that tells us where the next instruction is in memory. Depending on an instruction, an instruction can be one byte, just containing the opcode and addressing mode, or it can also have an 8 or 16 bit argument. We need to load these bytes from memory before doing anything else with the instruction.
- **Decode instruction:** We need to extract bits of information from the instruction byte and look up the instruction’s opcode and addressing mode. We also need to look up other helper information, such as if there are argument bytes based on the opcode or how many memory accesses are needed based on the addressing mode.
- **Simulate Instruction:** We need to update the register values based on the current register values and the instruction.

B. CPU Design

At a high level, our CPU will be a state machine alternating between 6+ states (the states will be explained in a lot more detail later). It will store values of CPU Registers (A, X, Y, PC, S) as well as flags (C, N, etc.).

Advancing through states happens using two separate always-ff blocks, one operating on the slow cpu clock and the other on the faster fpga clock. We decided to use two different clocks because cpu timing, including how many cycles instructions take to run in total, is based on the slow cpu clock. However, since we are sending memory requests to a BRAM, connected to the cpu through the memory bus, and this BRAM operates using the fast clock, we may miss a memory output if we only used the slow clock. So we use the fast clock for states where we need to receive a memory output. As a result, different states advance using different clocks.

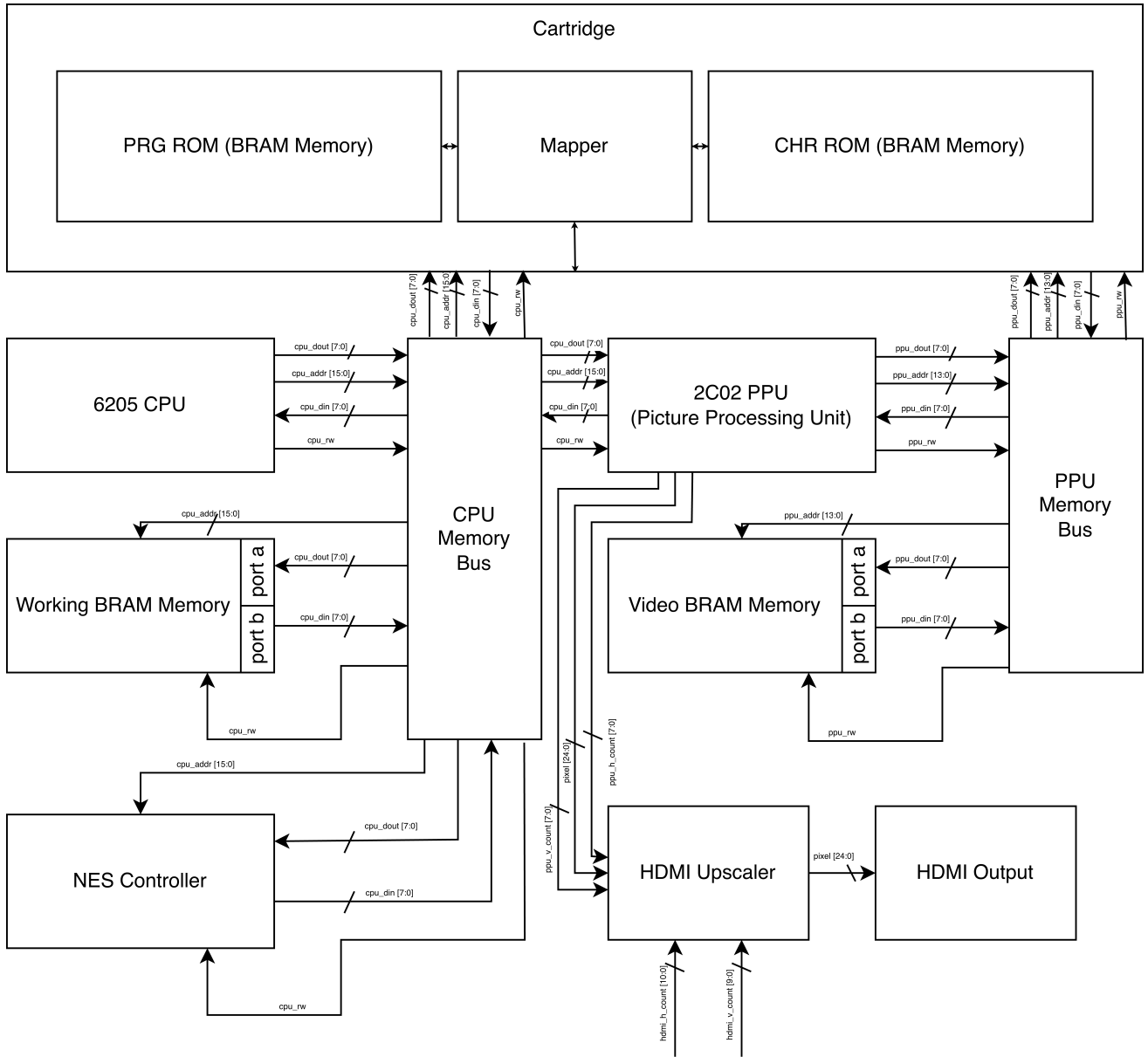


Fig. 1. The top level diagram for our design.

C. Always comb blocks that pre-compute information

We wanted to separate most of the instruction-specific logic from the instruction load and memory access logic. So, we have three always-comb blocks that compute information about the instructions.

One is the decoding block that extracts addressing mode and opcode from an instruction.

Another uses the addressing mode to determine how many bytes of an instruction we will need to load and how many memory requests we will need to make when reading a value from memory.

The third always-comb block also uses the addressing mode, and also how many memory requests we've already done, but

this time to calculate the memory address we need to request the memory from.

Having this logic in these always-comb blocks allows our always-ff blocks to primarily focus on sending memory requests and receiving memory values. We don't need to worry about how many requests to make since this information will already be available to us.

One exception is the simulation of instructions. We decided to keep this logic in a sequential block since we want to directly update register values and also may transition to different states (Mem Val Requested or Mem Val Received) based on what instruction we are working with.

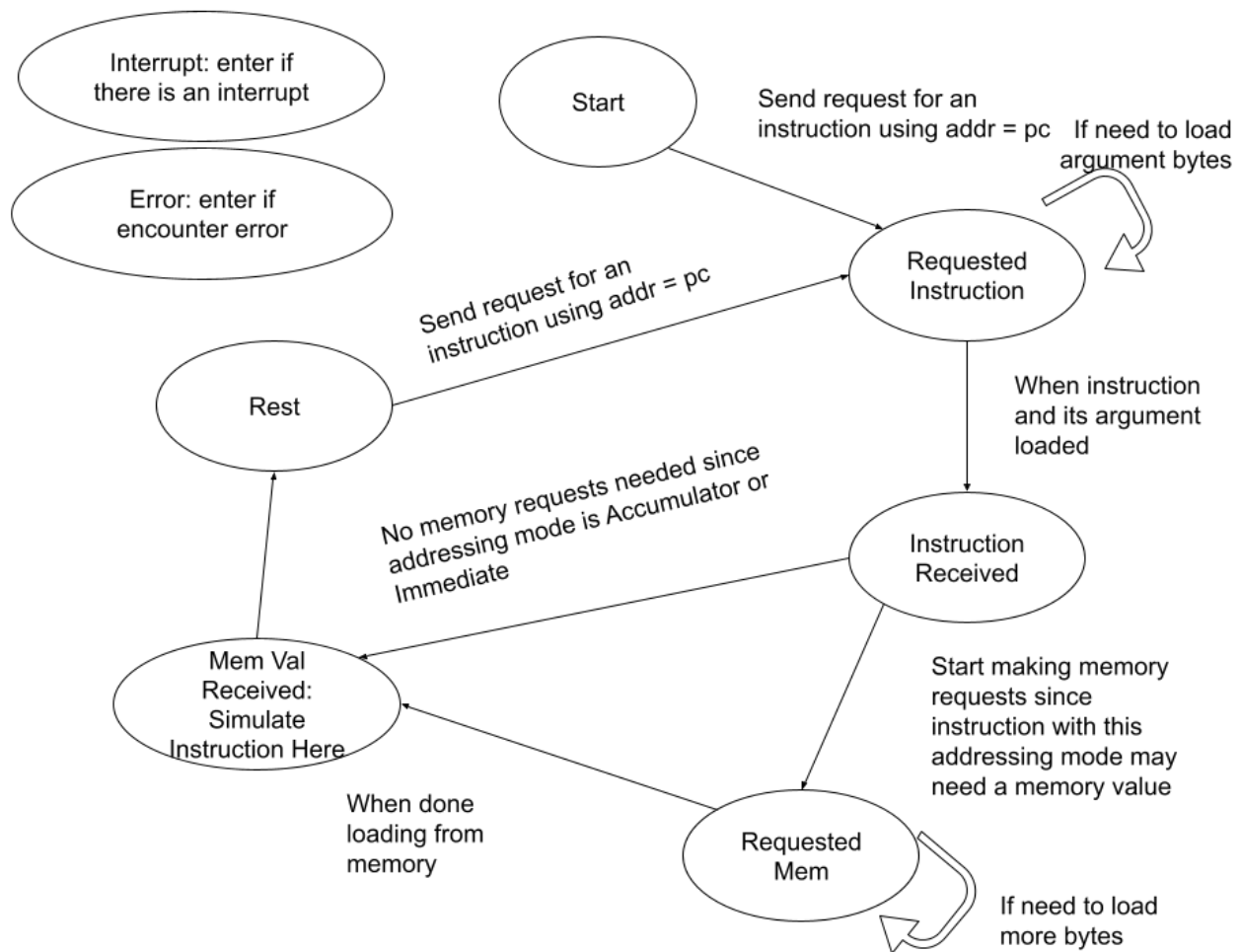


Fig. 2. The CPU state machine diagram

D. CPU State Machine

- **Start:** The start state advances using the slow clock. This is the state the system starts in after a reset. We request an instruction by sending a read request to a memory bus with $\text{addr} = \text{pc}$. We also re-initialize all the counters used for an instruction fetch and simulation. Note that we expect to get back the result of this instruction request from din after two fast clock cycles since our BRAM uses the fast clock. So, if we waited for the next slow clock cycle, we would miss this instruction. Therefore, we need to “catch” this instruction using the fast clock cycles. We transition to Requested Instruction state that uses the fast clock to catch this instruction.
- **Requested Instruction:** This state advances using the fast clock since it directly works with memory outputs of BRAM, which operates off the fast clock. Here, we always store the first byte of the instruction, but also sometimes request and store one or two more bytes depending on what the instruction is because it may have a one or two-byte argument. We sequentially store the first byte of the instruction, but then we combinationaly

determine the number of instruction requests we need to make in total, taking into account the number of arguments we need to make. This is because on the next fast clock cycle, we will have correct instruction stored, and the always-comb block uses this instruction to correctly determine instruction-requests-needed. Once the cpu receives all bytes needed for a given instruction, it transitions to an Instruction Received state.

- **Instruction Received:** This state advances using the slow clock. We are in this state when we know that we received the full instruction and its arguments. We have an always-comb block that decodes our instruction, extracting an opcode and an addressing mode from it. In this stage, we store the results for the opcode and addressing mode so we can access them later even if the instruction signal changes. Then, we start making memory requests since most instructions reference or write to a memory value. We know how many to make based on the addressing mode. If we don’t need to make any, meaning the addressing mode is accumulator or immediate, we go straight to Mem Val Received state. If

we do, we make the first memory request and transition to the Requested Mem state to receive this request and possibly make further requests.

- **Requested Mem:** This state advances using the fast clock since it works with the BRAM. Different addressing modes will need to make different numbers of memory requests to access the memory value. Most need just one, but Indexed Indirect and Indirect Indexed use two memory values to determine the memory address of the value we are actually interested in, so these addressing modes would need three memory requests. Our always-comb blocks determine how many memory requests are needed and calculate the correct address to request memory at based on a counter of how many requests we have already done. Using this, our Request memory stage can just request memory at address `addr-comb` computed by this always-comb block. When we have received the actual memory value we are interested in, we can transition to Mem Val Received. Note that after we've done this, we will know the actual memory address we need for implied addressing modes and when writing to memory we would just need one request instead of three.
- **Mem Val Received:** This state advances using the slow clock. Here is where we actually simulate the instruction. Based on the instruction opcode we decoded, we update the register values. If needed, we also send a memory request to write to some address.
- **Rest:** This state advances using the slow clock. Here we wait any extra cycles to ensure instructions take predictable amount of time to run. Currently we set all instructions to run in 5 cycles, but this value is fairly easy to change, and we originally planned to look up
- **Error:** If any error occurs while loading or running any instruction, such as if the opcode + addressing mode is not supported, we enter the error state and stay there. CPU cannot advance, and the programmer should try to understand why the error occurred.
- **Interrupt:** Unfortunately, we did not have a chance to implement interrupts.

E. CPU Testing

The main cpu testbench is in `test-cpu.py`. We wrote a dummy memory using a python dictionary. We keep looping, letting the cpu run, and when it requests to read or write to memory, we do this using this dummy memory. This dummy memory also contains a simple program (INX, INY), so by running the cpu on this memory we test instruction loads as these simple instructions are loaded from memory. We also test instruction decoding and simulation for these simple instructions. We debugged and confirmed correctness of cpu execution of this simple program using a waveform. If we had more time, we would have wanted to test more complicated programs, but probably using the same dummy memory concept, since it allows us to primarily focus on testing correctness of the cpu assuming the memory operates correctly. We could also add

artificial delays in the dummy memory to see if the cpu can handle unexpected delays.

III. PIXEL PROCESSING UNIT (JOSHUA)

A. Memories

The pixel processing unit (PPU) is responsible for generating the pixel output of the NES. The actual NES has 16 KB of addressable memory that is used to carry out its major functions. The pattern tables contain the visual info for tiles made up of 8x8 pixels each that can be drawn on the screen. One pattern table contains background tiles and the other contains tiles for sprites. These 16KB include the two pattern tables which take up 8KB of RAM, the name table which takes up 4 KB of addressable space but 2 KB physically, and palette RAM which takes up 256 bytes of addressable space but 32 bytes physically. Due to the relatively large amount of memory the pattern table and name table require, they were both implemented with BRAM. Name tables in the NES are 1 KB memory spaces that explain where background tiles should be placed on the screen. The NES has two nametables, but internally, the PPU sees it as four and mirrors addresses accordingly. Due to this added complexity, a dedicated module was created for it. The module takes in the read address given to it by the PPU and internally combinationally maps the address to its real physical space in memory which the BRAM takes as input. The exact mapping scenario depends on the cartridge so the module takes in whether the name table should have horizontal or vertical mirroring. The module outputs the output of the BRAM to the rest of the PPU to use accordingly. The palette RAM was implemented in a similar manner since it, but with registers instead and more simple mirroring as palette RAM is constant and not dependent on the cartridge. The palette RAM describes which index in the system palette a pixel should correspond to. The system palette was represented solely using registers. A Python script was created that would convert a .pal file into a .mem file that could be used to load any arbitrary palette into the emulator. The .mem file contains 64 entries that each have 24 bit RGB values which HDMI will output. This .mem file was then loaded into registers that are never changed afterward. The output from the palette RAM was fed into the system palette .mem which was fed into the HDMI module to actually output an image on the screen.

There is also OAM (object attribute memory) that contains a list of sprites to display for the foreground. This is not within the addressable memory of the PPU. This has been implemented partially, but was not in a good enough state to include in the final iteration. The code is still included in the final version of the but commented.

To test the pattern memory and palette system in isolation, a pattern memory output system was created with an arbitrary palette RAM selection and it produces accurate results from NES ROMs. Additionally, to test that the loading system was accurate, a testbench was created to prove given preloaded memory, it outputs an accurate frame of NES games from running a frame of the PPU module, including correct colors and background tiles, though it can't output most foreground

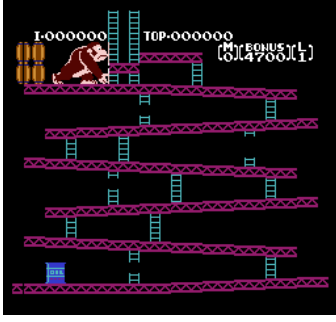


Fig. 3. Image of Donkey Game in game produced from the PPU module in simulation

components at the moment. This frame was attempted to be created on the PPU as well, but was unable to do so for unknown reasons.

B. Timing and Other Rendering Details

The actual NES's PPU runs at 5.32 MHz, and this time is used to render scanlines, each horizontal line in the screen and pixels. There are 262 scanlines processed by the NES per frame and 341 NES PPU cycles per scanline. Since the Xilinx FPGA runs much faster than the NES, there is a PPU clock trigger register that the PPU module uses to advance to the next simulated PPU cycle. This is done so that the FPGA can still make use of the additional cycles allowed to retrieve from BRAM and do other operations but to still make the system cycle accurate. This is driven by the top level and is triggered roughly every 18-19 cycles.

The PPU also looks for commands from the CPU and based on the commands, writes to or reads from internal PPU registers that are also used to determine the pixel. These registers in the original NES are PPUCTRL, PPUMASK, PPUSTATUS, OAMADDR, OAMDATA, PPUSCROLL, PPUADDR, PPUDATA and OAMDMA and have all been replicated here. These are all 8 bits except for PPUSCROLL and PPUADDR which are 16, though not all bits are necessarily used. PPUCTRL determines how much to increment VRAM addresses by, selects the scroll bit, determines which tile should be selected for both the foreground and background, shows whether a non-maskable interrupt is enabled, and has the sprite height. PPUMASK has whether the sprite and background are enabled, whether a color should be emphasized, whether the system is grayscale, and whether to enable the left column of the sprite and background. PPUSTATUS contains the blank, sprite overflow, and sprite 0 hit bits, and otherwise are unused. OAMADDR and OAMDATA are used to represent the OAM read and write address and data, respectively. PPUADDR is used to know which address the PPU should read from instead within its internal memories, but when the CPU writes to it, PPU puts the data into two 16 bit registers, `temp_vram` and `vram` which eventually receives values from `temp_vram` that represent the current `vram_address` to use and work seamlessly with scrolling. Both these registers include the same information: the fine x value, coarse x, coarse y, and base name table

address. The fine x value determines which bit of the tile to consider, the coarse x says which tile horizontally should be considered, the coarse y says which tile vertically should be considered, and the name table select address determines which name table should be used. Like how PPUADDR does not write to PPUADDR directly, when PPUSCROLL is written to, the coarse x and coarse y change accordingly. When PPUDATA is read, gets the value from the pattern table, name table, or pattern RAM at the address the vram address indicates depending on what the vram register value is. The PPU only accepts 8 bits at a time from the CPU, so there is also a w register that represents whether the most significant or least significant byte is being processed. Finally, OAMDMA has the high address for object attribute memory.

An NES outputs a resolution of 256x240, so only 256 of the cycles per scanline are actually used to render the screen and only 240 scanlines are actually output to the screen. While pixels are being output to the screen, the system follows a consistent structure every 8 cycles; it will take 2 simulated PPU cycles to get the nametable byte, then it will take 2 cycles to get the attribute table byte for the next set of 8 pixels, 2 cycles to get a byte containing the most significant bit of the palette RAM for the next 8 pixels, and 2 cycles to get a byte containing the least significant bit of the palette RAM for the next 8 pixels. This continues until all pixels are processed. While this is happening, the PPU uses this info to determine the palette index for the current pixel and uses the system palette to get the appropriate value to output to the screen assuming the background rendering bit within PPU Status is output. At scanline 241 cycle 1, the system internally sets the `vblank` flag and indicates to the system that the system is not currently rendering and sends a non-maskable interrupt to the CPU to tell it this.

On the foreground rendering side, it follows a similar process. There is a variable called `secondary_oam` that contains up to 8 sprites that can be output in the current scanline. In the first 64 cycles of a scanline, this reads as 0xFF by default. Afterward, from cycles 65 to 256, it follows a pattern. Every odd cycle it will read a byte of a sprite, starting with the first one (the Y position), from the primary OAM, and every even cycle, it will do one of the following steps. If the first byte is being read, it will determine whether the Y position is in range of the scanline based on its size and orientation. If it is, in the next cycles it will move on to each byte of the sprite until all 4 bytes are done and add it to secondary OAM memory. Otherwise, it will move on to the next sprite. If secondary OAM memory has 8 sprites already, it won't add any more and will set the sprite overflow flag.

IV. EVALUATION (DOMINIC)

Unfortunately, we were not able to finish working on the system, so it does not work as intended at the moment. If we had more time, we would first and foremost implement the interrupt feature on the CPU, as that's one of our biggest missing features, and implementing that would allow us to

complete our commitment of a working CPU and PPU, in addition to additional debugging.

V. CONTRIBUTIONS (DOMINIC)

Sasha was responsible for the design, development, and documentation of the CPU component, Joshua was responsible for the design, development, and documentation of the PPU and HDMI output, and Dominic was responsible for much of the miscellaneous tasks, such as integration, the other modules like the cartridge and the controllers, creating the high-level diagram, writing the high-level overview, and editing the report video.

VI. GITHUB REPOSITORY

<https://github.mit.edu/6205F25/fa25-6205-team42>

REFERENCES

- [1] “NES reference guide,” NESdev Wiki, 2025.
https://www.nesdev.org/wiki/NES_reference_guide
- [2] N. Parker, “The 6502/65C02/65C816 Instruction Set Decoded,” Llx.com,
<https://llx.com/Neil/a2/opcodes.html>
- [3] “Virtual 6502 / assembler,” Virtual 6502 Assembler,
<https://www.masswerk.at/6502/assembler.html>