CSEE 4840 Embedded Systems
NES Emulator

Spring 2023

Otito Darl-Uzu (ood2103), Jason Lam (jjl2247), Tyler Manrique(tm3147)

Table of Contents

Design Overview

The goal of this project is to successfully emulate the NES core functionality to play a ROM on a 256 x 240 screen.

The scope of our emulation is as follows:

1. CPU (8-bit 6502)
2. Addressable memory space (16-bit)
3. Picture Processing Unit (PPU): Our emulation will render on a 256x240 screen with support for as pixel-level scrolling
4. Controllers: Keyboard inputs will be primary mode of control with possible support for NES controllers
5. Cartridge boards: We will limit our emulation to ROMS that dynamically map ROM/RAM into CPU and PPU memory space. We will not support cartridges that have their own battery-backed RAM, or audio processing unit.

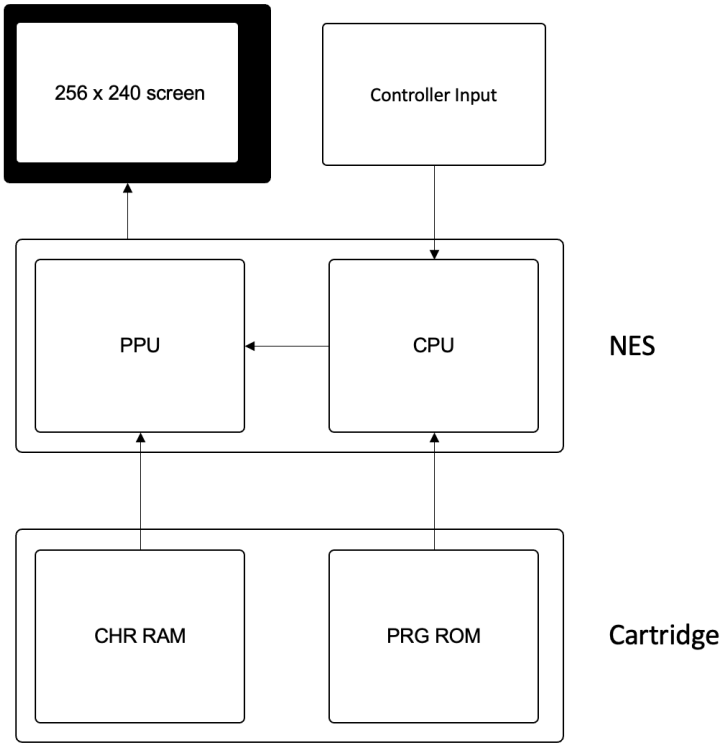

Figure 0: NES Emulator Architecture

Hardware Interface

PPU block diagram

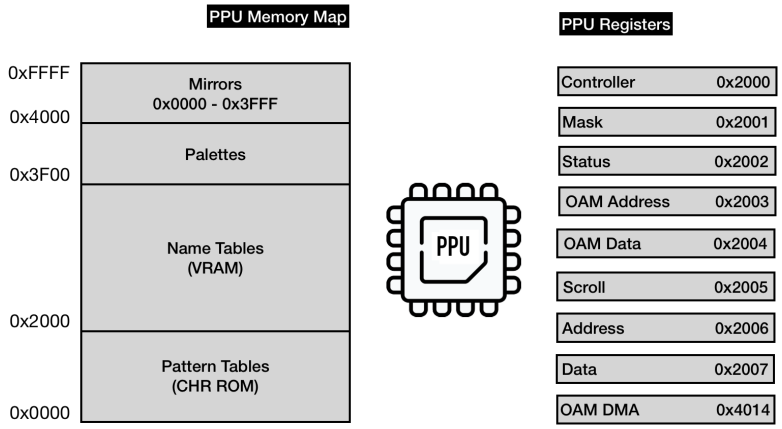

Figure 1: PPU block diagram from NES Ebook (Bugsmanov).

Like in a real NES, nine registers are used to control the current VGA output. Controller, mask, and scroll are used for PPU output, determining which bit is being drawn at a time, and scrolling if necessary. OAM addresses/data/DMA are used to access sprite memory. Address and data access the PPU memory map, shown on the left. This stores much of the game itself in memory- namely palettes, names, and patterns.

Scanlines

The PPU renders 262 scanlines per frame. 240 of those are used for the actual screen, and these are translated into VGA output on the FPGA board. The PPU design is based on the original NES NTSC PPU, which has 341 clock cycles for each scanline. This is equivalent to 113 clock cycles (~3 PPU cycles per CPU clock cycle).

- Cycle 0
    - Idle
- Cycle 1-256
    - Fetches data for each tile from memory
    - Nametable, attribute table, pattern table low and high
    - Fed into shift registers (each 8 pixels has the same color palette as a result)
    - At the beginning of each scanline, the data for the first two tiles is already loaded into the PPU
- Cycle 257-320
    - Fetches tile data for next scanline

    - Garbage nametable bytes
    - Pattern table high and low
    - Loads X positions for each sprite
- Cycles 321-336
    - First two tiles for next scanline are fetched and loaded into memory
- Cycles 337-340
    - Not needed (legacy from old NTSC implementation)

The PPU then idles when all 240 scanlines are rendered, and the Vblank flag is set to actually display the contents through VGA output.

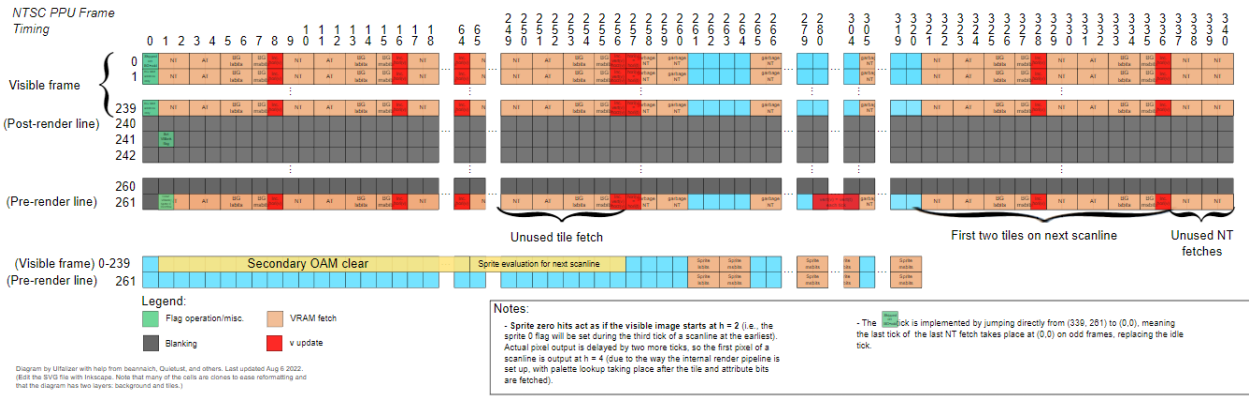

Figure 2: Frame timing diagram for a single scanline, courtesy of nesdev.com

Communication between PPU and CPU

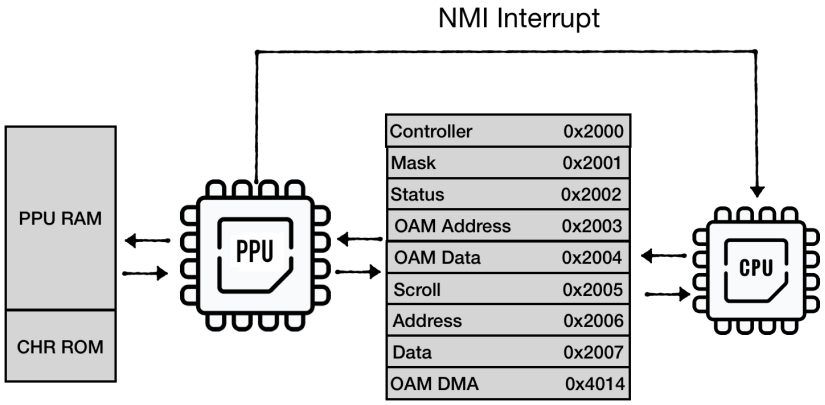

Figure 3: Communication between PPU and CPU from NES Ebook (Bugsmanov).

The CPU sends information through IO registers (the nine registers seen above), which communicates with the PPU. The PPU pulls from the RAM and ROM respectively, updating the registers which is then sent back to the CPU for another cycle.

Software Interface

INES File Format

The .nes file format is the standard format for nes binary files. It is used in several emulators including official Nintendo ones. Of note to this implementation, it contains a 16 byte header containing the string “NES” in ASCII, the size of the PRG ROM (in 16K units), the size of the CHR ROM (in 8K units), and the specific mapper used. Our emulator will parse this file format and be able emulate the rom present in it.

CPU

The 6502 is a 8-bit little endian processor. It has three 8-bit registers - Accumulator used in heavily by the ALU and X,Y for data processing. It has a 16-bit program counter (PC) register which refers to the current instruction. There is a 8-bit stack pointer which can reference memory addresses $0100-$01FF. There is a 8-bit status register which contains bit flags set by ALU operations such as overflow, negative, zero, carry, and interrupt disable (which disables interrupts).

The NES has all of its components mapped to the same address space which can be divided into three parts: ROM inside the cartridges, the CPU’s RAM and the I/O registers. The data bus is used to read or write the byte to the selected address. The I/O registers are used to communicate with the other components of the system, the PPU and the control devices.

- The NES CPU has 2K of ram which is mirrored every 2K on the address space $0000-$07FF.
- $2000–$2007 maps to eight 8-bit registers exposed by the PPU to the CPU
- $2008–$3FFF are mirrors of PPU registers every 8 bytes
- $4000–$4017 map to APU and I/O registers
- $4018–$401F is typically disabled on NES CPUs
- $4020–$FFFF is mapped to the cartidage.
- The CPU also expects vectors for interrupts and resets at the end of the cartridge.

On system start, we start program execution (program counter register) to the value stored in the reset vector (addresses $FFFC–$FFFD), initialize the stack pointer to $100 and start reading instructions. We will use a reference to convert opcodes into equivalent operations (using basically a large case statement) on allocated memory and registers represented as uint8_t variables in C code. Instructions take varying number of

cycles to complete but in this implementation we do not care about cycle accuracy. Our solution is to complete all operations in one cycle and tick the PPU clock to catch up with the CPU. This is accurate enough for most games and the timing errors will (hopefully) not be too bad.

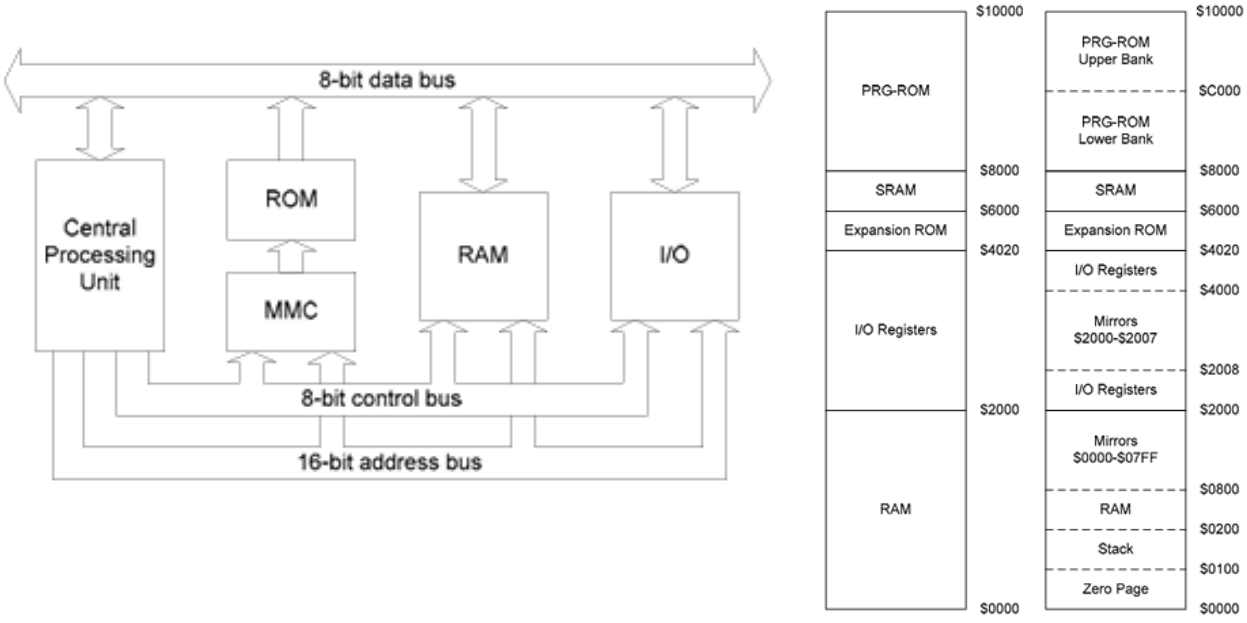

Figure 4: CPU Memory map

Mappers

Most NES cartridges use various internal components to bypass the NES's limitations. These mappers can map memory visible to the CPU and PPU to different parts of the ROM. Historically this was commonly used to save games (through the use of EEPROM) or expand memory. Some of the first cartridges with 16K or 32K of ROM used mapper NROM which mapped its first 16K of ROM to addresses $8000-$BFFF. The next 16K of addresses ($C000-$FFFF) were mapped to the remaining 16K of data. However, if the cart only had 16K of ROM, it mirrored the first 16K to the next 16K. More complicated mappers used registers and more sophisticated memory mapping to get around the NES's memory constraints. We will not be implementing all 256 mappers but will start with NROM and implement a small selection of other mappers to get a wide range of playable games.

Milestones and References

Milestones

3 easy steps

1. Successfully load ROM into memory space using INES format
   Functional Memory space
   Successful CPU Emulation - test CPU using nestest (don't care about cycle accuracy)
2. Implement PPU in C
   Ability to render pixels to 256x240 screen
   PPU doesn't melt CPU
   Render ROM on to 256x240 screen
3. Implement PPU in Systemverilog
   Input handling
   Test and Demo

References

The NES emulation project has been replicated several times, we will leverage the learnings of other attempts to guide or build: https://yizhang82.dev/nes-emu-overview

NES Dev Wiki: https://www.nesdev.org/wiki/Nesdev

NES emulator tests: https://www.nesdev.org/wiki/Emulator_tests

NES opcodes: http://www.emulator101.com/reference/6502-reference.html

System Documentation: Nintendo Entertainment System Documentation

CPU基板回路図

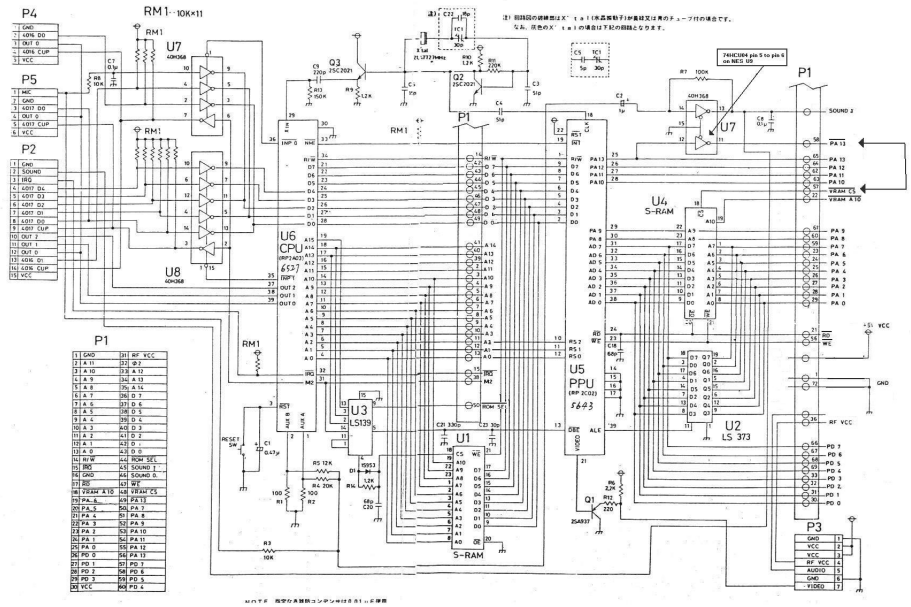

Figure 5: Original NES Block Diagram