Slopylator Nintendo Entertainment System Emulator

Algorithm and Programming Final Project

By Samuel Reagan Purnama (2902671833 - Class L1AC)

Introduction

I have always been interested in low-level programming and computer architecture, which led me into taking computer science in university. When I heard that the final project for the Algorithm and Programming class is making any program with Python, I was delighted. Even though Python is a high-level programming language, I still wanted to sharpen my understanding of computer architectures. After brainstorming some ideas such as making a programming language, I landed on making a Nintendo Entertainment System (NES) emulator. The reasoning was simple; the NES is a relatively old system with a simple CPU, and many people have created emulators and documentation for it.

The idea initially came to me after watching many YouTube videos about the history of game console emulation and also a few videos poking fun at bad NES emulators. After seeing the latter, I wanted to make my own "bad NES emulator". As an excuse to do something relatively tedious, I decided that the emulator will also serve as my final project. By doing this project, I would have an emulator that I fully understand the inner workings of, and I would also better understand how a computer works, since implementing the NES in code requires a good understanding of its architecture.

Project Goal

The goal of this project is to create an implementation of the NES CPU and general architecture that could boot into simple commercial games and at least play the first level of a few games. All of this means that the CPU needs to be functional enough to run game code, the PPU needs to be functional enough to display basic shapes without proper color, and the gamepad needs to be functional enough to accept user input. By making the emulator in Python and making it open source, it allows other people to learn how the system works by reading its implementation in a simple, high-level programming language like Python.

Project Specification - Main Objectives and Features

The project's main features are as follows:

- Emulate the documented (legal) instructions of the 6502-based CPU found in the NES
- Implement enough of the PPU to be able to make out what the game is doing
- Allow the gamepad to be controlled with the computer's keyboard

Project Specification - Running the Emulator

1. Download or clone the GitHub Repo at https://github.com/doomertheboomer/slopylator
2. Install Python 3 (the emulator was tested on Python 3.1.3)
3. Place iNES V1 formatted NES ROM dumps inside the emulator's folder
    a. A test rom called `testrom` (without a file extension) could be run to sanity-check the emulator's CPU implementation
4. Run the emulator using either of these steps:
    a. Open a terminal and type `python main.py romname.nes`
    b. Open a terminal and type `python main.py`. Then, type in the desired ROM file name
5. Play the game. The control scheme is listed in the next section.

Project Specification - Control Scheme

NES D-Pad:	Arrow Keys
Start Button:	Enter
Select Button:	C
A Button:	Z
B Button:	X

Solution Design - General Overview

The project is split into 6 files, each file serving as one emulated component of the NES, and one being the main file. The files are:

- main.py
- bus.py
- cpu.py
- ppu.py
- pad.py
- window.py

Each section below explains which files are responsible for its specific functionality, and what the code does in detail. For consistency, all indexes will start with zero in the explanations.

Solution Design - NES System Bus And iNES ROM Format

In the project, the main file and the bus file act as the NES system bus. The system bus in the NES is responsible for connecting the different components together.

```
# builtin modules import
import sys

# my own modules import
from cpu import *
from ppu import *
from bus import *
from pad import *
```

Figure 1: Imports in Main

To start, the main file imports all the other files that contain NES bus logic and connects to the NES bus. The sys module is imported to read command-line arguments.

```
# python should let this var be used out of the if block
if len(sys.argv) == 1:
    filename = input("Enter the romfile name (enter \"testrom\" for test rom): ")
else:
    filename = sys.argv[1]

# load the rom
with open(filename, "rb") as file:
    romfile = list(file.read())

header = romfile[0:4]
if not (header == [78, 69, 83, 26]):
    raise FileNotFoundError("The file is not an iNES file")
```

Figure 2: ROM File Loading

This code from lines 10 - 22 of the main file loads the main ROM file. It also does a basic integrity check by checking for the iNES header. First, it uses the sys module to detect whether or not the user has inputted a command line argument containing the file name. If the user doesn't supply a file name, the program will explicitly ask the user for a file name and store it in the filename variable. Then, the ROM file is opened and read as a binary (rb). String slicing was done to extract the first 4 characters of the ROM file, which is then checked against the first 4 characters in the iNES format, which should be NES followed by 0x1A, or 26 in decimal. If the file does not meet these criteria, the program quits immediately and raises a FileNotFoundError.

```
prgRom = romfile[4] # number of code banks (*16kb)
chrRom = romfile[5] # number of graphics banks (*8kb)

# control bytes bitmask
ctrl1 = romfile[6] # six
ctrl2 = romfile[7] # sevennnnnnnn
hasTrainer = (ctrl1 >> 2) & 1 # need to skip 512 bytes if this is present
isVertical = bool(ctrl1 & 1) # if false then game mirrors horizontal
is4Screen = bool((ctrl1 >> 3) & 1)

prgStart = 16 + (hasTrainer * 512)
chrStart = prgStart + (16384 * prgRom)

prg = romfile[prgStart:chrStart]
if prgRom == 1:
    prg += prg # mirroring if only 1 rom

chr = romfile[chrStart:] # best if i limit this for sanity
```

Figure 3: iNES Header Parser

In order to figure out how to load the game into memory, certain parameters need to be extracted from the iNES header. The main things that need to be extracted are the number of program and character ROM code banks, whether a trainer is present or not, and the mirroring type of the game. This code from lines 24 - 41 parses the iNES header. Bytes 4 and 5 respectively are the number of program and character ROM banks. Bytes 6 and 7 contains the control bitfield. In this program, only the first control byte at index 6 is used. Several bitwise operations are done to extract bits 2, 0, and 3, which contain the trainer flag and mirroring information. Bit 0 contains the vertical/horizontal mirroring flag, and bit 3 contains the special 4-screen mirroring mode flag.

After all these are parsed, the start addresses for the program and character ROMs are calculated. Since the program ROM lies first, the offset is calculated by offsetting by 16 for the iNES header length, then skipping an additional 512 bytes if a trainer is present. After the program ROM offset is calculated, it is then used as a base to calculate the start of the character ROM. Since each bank is 16KB, the character ROM's start is offset further from the program ROM's start according to how many 16KB banks are present.

These two offsets are then used to splice the ROM into the program ROM array. If only a single bank is present, it is duplicated to emulate the NES bus mirroring behavior. The character ROM starts from the start offset all the way to the end of the file.

```
bus = dmrambus(isVertical)
cpu = dm6502(bus, 0) # has ram mirrored by bus
ppu = dmppu(bus, 5) # has its own ram too
pad = dmjoypad(bus)

# load prgrom into cpu
bus.cpumem[0x8000:0x10000] = prg
cpu.pc = cpu.getIndirectAddress([0xfc, 0xff]) # needs to point to reset vector
if filename == "testrom":
    cpu.pc = 0xC000
    cpu.testmode = True
print(f"Program ROM loaded with entrypoint {hex(cpu.pc)}")

# load chrrom into ppu
bus.ppumem[0x0:0x2000] = chr[0x0:0x2000]
bus.isVertical = isVertical
ppu.buildPatternTable()
print(f"CHR ROM and mirror data loaded into PPU")

# input("Press ENTER to start emulation!")

breakpoints = []
stepping = False
```

Figure 4: System Bus/ROM Loading Code

After parsing the ROM file, the bus is then initialized and the ROM is loaded into emulator memory according to the NES memory map as seen by this code in lines 42-65. First, the bus object is instantiated, with the isVertical variable to toggle between the mirroring behaviors. Then, the CPU, PPU, and gamepad objects are instantiated, passing the bus object into them so they could communicate with each other via the bus. The second parameter of the CPU and PPU constructors correspond to the debug log level of each component.

After the hardware is initialized, the prg variable that contains the program ROM data is then copied into memory location 0x8000. The copy is limited to 0x10000 in order to prevent an out of bounds error if the extracted ROM is somehow too big. Then, a function is called from the CPU to get the entrypoint address which is stored at 0xFFFC. However, if the user loads the test ROM, which is checked with the file name, it hardcodes the entrypoint to 0xC000 and enables the test mode, which tells the emulator to dump the CPU program trace into a file. The CPU program counter (the address of what to execute) is set to the entrypoint address.

After the program ROM is loaded, the first 8192 bytes of the character ROM is then loaded into the PPU's memory. This is stored inside the bus object, and is separate from the CPU memory. Then, the mirroring behavior is loaded into the PPU and the pattern tables are built in order to fully initialize the PPU.

For debugging purposes, a list of breakpoints and a toggle for the stepping / paused mode of the debugger is created. In the production code, the breakpoint list is empty. However, this can be freely changed to any CPU memory addresses.

After this block of code, additional logic is present in order to step through the fetch-decode-execute cycle on both the CPU and PPU. However, this will be discussed in the CPU section.

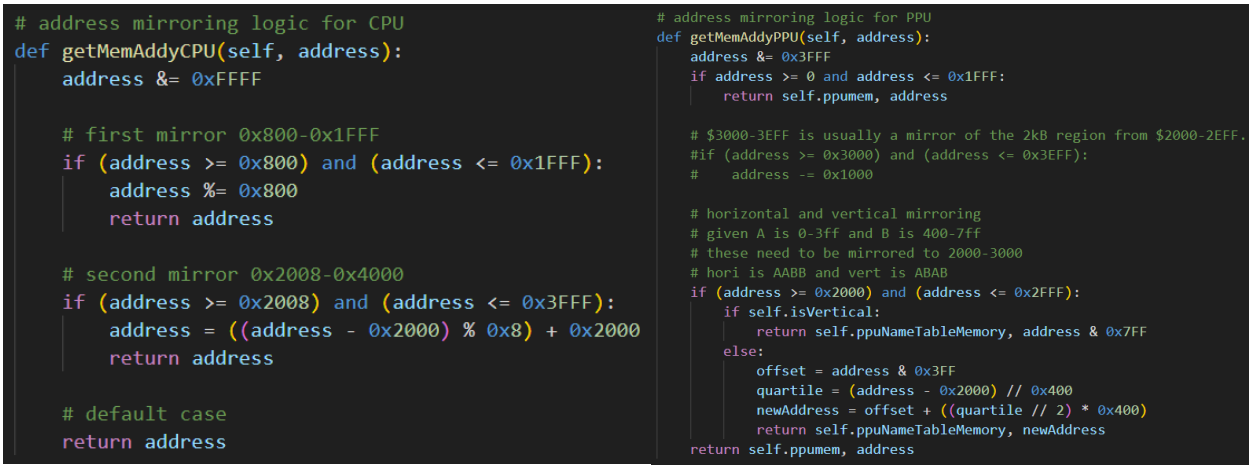
```
class dmrambus:
    def __init__(self, isVertical):
        # up to 0x10000 for both cpu and ppu
        self.cpumem = [0] * 0x10000
        self.ppumem = [0] * 0x4000

        self.ppuNameTableMemory = [0] * 0xffff

        # bus rw log, can be reset by PPU/CPU i guess
        self.cpuLastRead = 0xFFFFF
        self.cpuLastWrite = 0xFFFFF

        # internal PPU flip flops
        self.ppuintlAddrHigh = True
        self.isVertical = False
        self.ppuInterrupt = False

        self.isVertical = isVertical

        self.readHooks = []
        self.writeHooks = []
```

Figure 5: System Bus Constructor

The main file mostly initializes everything and connects all the components to the system bus. However, the dmrambus (short for Doomer's RAM Bus) object itself is stored in a separate file. The code shown here is the constructor that spans from lines 1 - 21. Both the CPU and PPU have 16-bit address spaces, so they could address up to 0xFFFF. However, the PPU's memory is limited to 0x3FFF and any following addresses are mirrored. So, a list of length 0x4000 is created for it. The PPU's nametables are stored in a separate list in order to reduce confusion.

A couple internal variables are set to allow the bus to access certain internal states. These are the last accessed read/write memory addresses for the CPU, a flip-flop for the PPU's 16-bit read behavior that uses two reads/writes to an address, interrupt flags, and finally the vertical mirroring flag.

The readHooks and writeHooks lists store references to functions that could later be accessed when a read or write is done. More on this later.



```python
# address mirroring logic for CPU
def getMemAddyCPU(self, address):
    address &= 0xFFFF

    # first mirror 0x800-0x1FFF
    if (address >= 0x800) and (address <= 0x1FFF):
        address %= 0x800
        return address

    # second mirror 0x2008-0x4000
    if (address >= 0x2008) and (address <= 0x3FFF):
        address = ((address - 0x2000) % 0x8) + 0x2000
        return address

    # default case
    return address
```

```python
# address mirroring logic for PPU
def getMemAddyPPU(self, address):
    address &= 0x3FFF
    if address >= 0 and address <= 0x1FFF:
        return self.ppumem, address

    # $3000-3EFF is usually a mirror of the 2kB region from $2000-2EFF.
    #if (address >= 0x3000) and (address <= 0x3EFF):
    #    address -= 0x1000

    # horizontal and vertical mirroring
    # given A is 0-3ff and B is 400-7ff
    # these need to be mirrored to 2000-3000
    # hori is AABB and vert is ABAB
    if (address >= 0x2000) and (address <= 0x2FFF):
        if self.isVertical:
            return self.ppuNameTableMemory, address & 0x7FF
        else:
            offset = address & 0x3FF
            quartile = (address - 0x2000) // 0x400
            newAddress = offset + ((quartile // 2) * 0x400)
            return self.ppuNameTableMemory, newAddress
    return self.ppumem, address
```

Figure 6: CPU and PPU Memory Mirroring Logic

The following code from lines 23 - 38 and 80 - 102 respectively are the wrapper functions to get an address from the NES CPU and PPU memory. Every read and write uses this function to resolve the address, as the NES mirrors certain memory regions as part of how the chips are wired in the bus.

The CPU mirroring logic is simple. There are two mirrors in the CPU memory map. Region 0x800 - 0x1FFF is the first mirror, and it's simply a duplicate of the first 0x800 bytes of memory. This means that in this region, simply taking the modulo by 0x800 will suffice to redirect any reads/writes in that region to the beginning of memory, where it is mirrored to. The second mirror is repetitions of 0x2000 - 0x2007 spanning all the way from 0x2008 - 0x3FFF. This is handled by using a modulo of the size of the mirrored region, which is 8 bytes. Then, it is offset by the start of the mirrored region, which is 0x2000.

Even though the PPU only has 0x4000 bytes of physical memory, it could address up to 0xFFFF just like the CPU. So, any addresses higher than 0x3FFF are mirrored to the start of memory. This is done by using a bitwise AND to purge any bytes past the 14th bit, which will correctly emulate the mirroring behavior found in the PPU for addresses beyond the physical memory size. The first region 0x0 - 0x1FFF don't need mirroring, so it is returned verbatim.

The isVertical variable is used to determine mirroring type for the region 0x2000 - 0x2FFF. In order to avoid corruption, the nametables are stored in a separate list from the rest of PPU memory. As stated by the comments in the code, horizontal mirroring maps this region to nametables A, A, B, and B in order, and vertical mirroring maps this region to nametables A, B, A, and B in order. This means that simply using a bitwise AND by 0x7FF works for vertical mirroring, which mirrors the first 0x800 bytes to the second half. However, horizontal

mirroring requires extra logic, as it repeats the same nametable twice before moving onto the next nametable. So, the address is first split into 4 quartiles. Then, a floor division is used to determine whether the address is in the first or last two quartiles. This then decides on which nametable to map to. Finally, the offset calculated at the beginning is added to get the final address. As the PPU memory is split into two lists in the code, the getMemAddyPPU function returns a tuple of the list followed by the memory address within that list.

```
def memoryReadCPU(self, address, end = None):
    fixAddy = self.getMemAddyCPU(address)
    self.cpuLastRead = fixAddy

    for f in self.readHooks:
        v = f(fixAddy)
        if v is not None:
            return v

    # handle weird PPU edge cases (prepare for yandev quality code)
    # if fixAddy == 0x2002:
    #     self.ppuintlAddrHigh = False
    # elif fixAddy == 0x2006:
    #     # for double address reads
    #     self.ppuintlAddrHigh = (not self.ppuintlAddrHigh)

    if end != None:
        retVal = []
        for i in range(address, end):
            fixAddy = self.getMemAddyCPU(i)
            retVal.append(self.cpumem[fixAddy])
    else:
        return self.cpumem[fixAddy]
    return retVal
```

Figure 7: CPU Memory Read Wrapper

```
def memoryWriteCPU(self, address, value):
    fixAddy = self.getMemAddyCPU(address)
    self.cpuLastWrite = fixAddy

    for f in self.writeHooks:
        if f(fixAddy, value) is not None:
            return

    self.cpumem[fixAddy] = value

    # handle weird PPU edge cases (prepare for yandev quality code)
    # if fixAddy == 0x2006:
    #     # for double address writes
    #     self.ppuintlAddrHigh = (not self.ppuintlAddrHigh)
```

Figure 8: CPU Memory Write Wrapper

The function shown above from lines 40 - 63 wraps all memory reads to the CPU. Wrapping memory reads serves a double purpose of being able to use the memory mirroring logic and also to serve as toggles for certain PPU flags.

The function starts off by simply getting the real location in the memory by resolving any mirrors using the aforementioned wrapper functions. Then, it sets a last read variable in case any other functions need to know where the CPU last read from. Then it iterates through every item in the readHooks list, and calls it as a function using the brackets, passing the fixed address into it. If the function executes and intercepts the read (returns anything other than a None), it executes the hook and completely replaces the functionality of the memoryReadCPU function from that point onward.

Since the read function is supposed to replace instances where the list is sliced, it needs an end value too. This functionality is replaced using a for loop that manually appends the fixed addresses into a list. The function returns either a list or an int.

The code shown in Figure 8 from lines 65 - 78 is the memory write wrapper for CPU memory. It has mostly the same logic as the read wrapper, but in reverse, and without the list logic.

```python
def memoryReadPPU(self, address, end = None):
    data, addr = self.getMemAddyPPU(address)
    if end != None:
        retVal = []
        for i in range(address, end):
            data, addr = self.getMemAddyPPU(i)
            retVal.append(data[addr])
    else:
        return data[addr]
    return retVal

def memoryWritePPU(self, address, value):
    data, addr = self.getMemAddyPPU(address)
    # print(hex(addr))
    data[addr] = value
```

Figure 9: PPU Memory R/W Wrappers

The following code from lines 104 - 118 shows the read and write wrappers for the PPU. It is mostly the same logic as the CPU wrappers.

Solution Design - NES CPU Implementation

```python
import sys
class dm6502:
    def __init__(self, rambus, loglevel = 3):
        self.loglevel = loglevel
        self.loglevels = ["[fatal]", "[error]", "[warn] ", "[info] ", "[debug]", "[trace]"]
        self.testmode = False

        # general purpose registers
        self.a = 0  # accumulator
        self.x = 0  # x register
        self.y = 0  # y register
        # special registers
        self.pc = 0x0000 # program counter
        self.sp = 0xFD # stack pointer
        self.sr = 0b00100100 # status register
        self.__srFlags = {
            'n': 7,
            'v': 6,
            'b': 4,
            'd': 3,
            'i': 2,
            'z': 1,
            'c': 0
        }
        self.cycles = 0

        # more layers of abstraction please sir
        self.rambus = rambus

        # master list of all opcodes
        # opcode: [func, size (including first byte), cycles, stars]
        self.__opcodes = {…

        self.log("6502 CPU initialized", 3)
        pass
```

Figure 10: CPU Constructor

```
0x69: [self.__adc69, 2, 2, 0],
0x65: [self.__adc65, 2, 3, 0],
0x75: [self.__adc75, 2, 4, 0],
0x6D: [self.__adc6D, 3, 4, 0],
0x7D: [self.__adc7D, 3, 4, 1],
0x79: [self.__adc79, 3, 4, 1],
0x61: [self.__adc61, 2, 6, 0],
0x71: [self.__adc71, 2, 5, 1],
```

Figure 11: Part of the CPU Instruction Lookup Table

The following code from lines 1 - 228 (with a truncated lookup table) is the constructor for the CPU object. The CPU file imports sys in order to exit the program in the testing scenario. The CPU object is called dm6502, which is short for Doomer's 6502.

The constructor starts with initializing the logging parameters of the CPU object. This is especially useful during debugging to ensure that the CPU is executing the correct instructions and that those instructions are being executed properly. The loglevels variable gives strings to prepend before printing into the console, and testmode is a flag to determine whether or not the emulator is executing the test ROM.

The 6502 has 3 general-purpose registers and 3 special registers. The general-purpose registers are called A, X, and Y, and the special registers are the program counter, stack pointer, and status register. In lines 8 - 15, the code initializes these registers with default values. The general-purpose registers are used to store values in very short-term memory that instructions will read and write from. The program counter is the memory address that the CPU is executing, the stack pointer is an offset in memory page 1 (0x100 - 0x1FF) of where the CPU stack currently is, and it is moved left and right. The status register is a bitfield that contains flags of the CPU's current status, which is modified accordingly to different instructions. The status flag bit positions are shown in the __srFlags variable. In order from bit 0 to 7, the flags are the carry flag (if an addition/subtraction passes the 8-bit boundary), zero flag (if the result is zero), the interrupt disable flag (whether the CPU should ignore interrupts), the decimal flag (this has no effect on the NES), two unused flags, the overflow flag, and the negative flag (if the result is negative according to signed 8-bit values).

After the registers have been initialized, a cycle count is initialized to ensure that the PPU runs at the proper speed (3x the CPU's speed), and the rambus object is connected to the CPU so it could communicate with other devices on the bus. Then, the dictionary of instructions and their subsequent opcodes is defined. The key for the dictionary is the opcode, or the first byte of the instruction. The value is a list that contains a pointer to the function that handles the opcode, how long the instruction is in bytes, how many cycles it takes to execute, and whether it should add an extra cycle depending on certain conditions. However, the emulator isn't intended to be cycle-accurate, so the last entry is never used.

```
def printStack(self):
    stack = self.rambus.cpumem[0x100:0x200]
    for i in range(0xFF):
        print(f"{hex(i)} {hex(stack[i])}")

def log(self, *args):
    level = args[-1]
    va_list = args[0:-1]
    if (self.loglevel >= level):
        print("[cpu]",self.loglevels[level], *va_list)
```

Figure 12: Debug Printing Functions

The two functions above from lines 230 - 239 are functions used for debugging. The printStack function prints the stack. It simply slices the entire CPU memory for just the stack memory and prints everything with the format of address and value in hex.

The log function uses an asterisk to make an argument list in order to print many things at once just like the built-in Python print function. It uses the last parameter as the log level, and it prints if the log level is within the set log level at the start. It uses the loglevels variable to print the correct category of the log, and then prints the intended text.

```
def fetch(self, pc = None):
    if pc == None:
        pc = self.pc # cannot access self as default val i hate you python
        self.log(f"pc = {hex(self.pc)} ins = {hex(self.rambus.memoryReadCPU(self.pc))}", 5)

    opcode = self.rambus.memoryReadCPU(pc)
    params = []
    # determine params array for opcode
    if (opcode in self.__opcodes):
        paramLen = self.__opcodes[opcode][1]
        # params = self.memory[(pc+1):(pc+paramLen)]
        params = self.rambus.memoryReadCPU(pc+1, pc+paramLen) # start from after the instruc

    self.decodeExecute(opcode, params)


def decodeExecute(self, opcode, params):
    # execute the opcode
    if (opcode in self.__opcodes) and (len(params) == self.__opcodes[opcode][1] - 1): # para
        self.__opcodes[opcode][0](params)
        self.pc += self.__opcodes[opcode][1]
        self.cycles += self.__opcodes[opcode][2]
        # TODO: cycle handling
    else:
        self.log(f"Illegal instruction! {hex(opcode)} params: {len(params)}", 2)
        raise Exception("Illegal")
```

Figure 13: Fetch-Decode-Execute Cycle Functions

```
# testcase generator
if cpu.testmode:
    cpu.loglevel = 5
    log = open("testoutput.txt", "w")
    log.write("")
    print("Beginning CPU sanity check. If this takes more than a couple seconds, the CPU isn't working properly!")
```

Figure 14a: File Creation

```
# main loop
cpuCyclesOld = 0
while True:
    if (cpu.pc in breakpoints):
        print(f"Breakpoint {hex(cpu.pc)} hit! A {hex(cpu.a)} X {hex(cpu.x)} Y {hex(cpu.y)} SR {hex(cpu.sr)} SP {hex(cpu.sp)}")
        stepping = True
    if stepping:
        t = input()
        print(f"A {hex(cpu.a)} X {hex(cpu.x)} Y {hex(cpu.y)} SR {hex(cpu.sr)} SP {hex(cpu.sp)}")
        if len(t) > 0:
            stepping = False

    if bus.ppuInterrupt:
        # print(hex(cpu.pc)) # TEMP
        # this needs to be nested
        if ppu.ctrlFlagGet('v'):
            # stepping = True
            # print("NMI enable")
            cpu.interrupt(0xFFFA)
        bus.ppuInterrupt = False

    if cpu.testmode:
        # slow and steady wins the race it seems
        with open("testoutput.txt", "a") as file:
            file.write(f"{cpu.pc:04X} {cpu.a:02X} {cpu.x:02X} {cpu.y:02X} {cpu.sp:02X} {cpu.sr:08b}\n")

    cpu.fetch()

    # ppu is 3x faster than cpu
    for i in range((cpu.cycles - cpuCyclesOld) * 3):
        ppu.fetch()
    cpuCyclesOld = cpu.cycles
```

Figure 14b: Main Emulator Loop

The code shown above in lines 241 - 266 in cpu.py shown in Figure 13 and lines 74 - 105 in main.py shown in Figure 14b shows the CPU's FDE cycle. Before the main emulator loop, the previous CPU cycle count is stored in order to calculate how many PPU cycles to execute after each CPU instruction has been executed. This is because the PPU is 3 times faster than the CPU, and most instructions take more than 1 cycle.

The first thing the emulator checks before executing any CPU code is if the current program counter is in the list of breakpoints. If that is the case, it will notify the user in console with all the registers using an f-string. Then, it will set the stepping variable to true. This makes it so that the loop doesn't execute as fast as possible, but instead waits for the user to make an input before proceeding to the next instruction. This logic is stored right after, where it asks for user input. If the user inputs anything other than a blank string, it will execute normal execution until it hits a breakpoint again by setting the stepping variable back to false.

Then, the CPU will check for an interrupt from the PPU before executing instructions. It first checks If the PPU has requested an interrupt. If so, it checks if the interrupt disable flag is enabled, then jumps execution to the interrupt handler found at 0xFFFA using the interrupt function. More information on this interrupt function can be found later. Regardless of whether the interrupt is rejected or not, the CPU completes the request and disables the PPU interrupt flag.

If the testmode flag is enabled, the CPU will print execution logs to a file. This works similarly to the debugger stepping functionality, and uses ‘with open’ in order to ensure that each line is properly written and the file is closed. Before writing to the file, it is cleared with the code shown in lines 67 - 72. It simply opens the file in write (w) mode and writes a blank string to it.

After all of these preliminary checks, the CPU is finally ready to fetch, decode, and execute the instruction. This is done using the fetch function found in Figure 13 at lines 241 - 254 in the cpu.py file. The function has a default value for the pc parameter to simulate function overloading, since it doesn’t exist in Python. This allows the fetch function to either fetch the next instruction or execute a specific area in memory. If no parameter is passed, it simply uses the current CPU program counter as the pc variable. A debug print is done to tell the programmer where the CPU is executing. However, this print is suppressed with the default log level. It then fetches the opcode by reading the memory at the current program counter, and initializes an empty parameters list. Then, it fills the parameter list. It does this by first checking whether the opcode is inside the valid opcode list defined at the constructor of the CPU object. If it sees a valid opcode, it takes the second item of the opcode’s property list, which is the size of the opcode. Then, it reads only up to the size of the opcode, not including the first byte that defines the opcode itself. This is then filled into the parameter list.

After that, the decodeExecute function from lines 257 - 266 is run. The opcode itself and the parameter list is passed into this function. Before it does anything, it checks whether the opcode is valid or not by checking if the parameter list length matches the opcode’s size. If it doesn’t, it raises an exception and notifies the user. If the pass checks, it executes the opcode. First, it accesses the first entry of the opcode’s property list, which corresponds to the function that handles the opcode’s logic. The code appends (params) to it, which executes the variable as a function, passing the parameters into it. After that, it increments the program counter by the opcode’s length, which is the second entry of the list. Then, it increments the cycle count by the third entry of the list, which corresponds to how long it takes for the opcode to execute.

After the CPU is done executing the opcode, the code goes back to the main emulator loop. Since the CPU has written a new cycles variable, the code can then use the difference as how many cycles the PPU should run. It simply multiplies that number by 3 and runs the PPU in a for loop. To finish the cycle, the new cycle count is now set as the old count and the main loop restarts.

```
def toSign8(self, val):
    sign = (val >> 7) & 1
    if sign == 0:
        return val & 0xFF
    return (((~val) & 0b01111111) + 1) * -1

def srFlagSet(self, flag, enable):
    # TODO: suggestion from friend to use enums instead
    # 7 6 5 4 3 2 1 0
    # N V x B D I Z C
    if flag[0] in self.__srFlags:
        # print(f"set {flag} to {enable}") # TEMP
        mask = 1 << self.__srFlags[flag[0]]
        if enable:
            self.sr |= mask
        else:
            self.sr &= ~mask
    else:
        self.log("Bad status register flag!", 2)

def srFlagGet(self, flag):
    # TODO: suggestion from friend to use enums instead
    # 7 6 5 4 3 2 1 0
    # N V x B D I Z C
    if flag[0] in self.__srFlags:
        return bool((self.sr >> self.__srFlags[flag[0]]) & 1)
    else:
        self.log("Bad status register flag!", 2)
        return 0

def stackPush(self, val):
    val &= 0xFF
    self.rambus.memoryWriteCPU(self.sp + 0x100, val) # stack is on page 1
    self.sp -= 1
    self.sp &= 0xFF

def stackPull(self):
    self.sp += 1
    self.sp &= 0xFF
    return self.rambus.memoryReadCPU(self.sp + 0x100)
    # return retVal
```

Figure 15: Common CPU Function Boilerplate

Since the CPU does a lot of repetitive tasks, it is important that boilerplate code to handle said repetitive tasks is created to reduce repetition in the final product. These 5 functions found between lines 268 - 308 are the most commonly executed code inside the CPU. The first function is the toSign8 function, which converts an unsigned number to a signed 8-bit number. It does this using bitwise operations. First, the sign is extracted by using bit shifts to extract the 8^{th} bit of the number. If the number is positive (sign is 0), then it returns the number but removes all bits after the 8^{th}. Otherwise, it does two's compliment by first flipping all the bits using a bitwise NOT, limiting it to 7 bits, and multiplying it by -1 to make it negative.

The next two functions are wrappers for the status flag variable. These are to reduce the number of bitwise operations I had to write in the code. Both of these functions use similar logic. First, it gets the bit position of the flag by referencing the __srFlags lookup table. Then, for setting, it shifts that bit to the correct position. When it is enabled, it does a bitwise OR to set it to 1 no matter what. When it is disabled, it does a bitwise AND with a NOTed value of the mask in order to keep all the bits except for the mask. For the read, it looks up the flag's bit position the same way. Then, the register is shifted by the flag's position and all bits after the first bit is removed in order to create a bool to return. If the value is invalid, it returns a zero as a safeguard to not crash the program.

The next two functions are push and pop functions for the stack. The CPU doesn't need to clear values, and instead only moves the pointer up and down. It simply moves the stack pointer downward when a value is pushed, and moves it upward when the value is popped. For pushing, it sets the value at the current stack pointer at page 1 (offset by 0x100) then moves the pointer. For popping, it moves the pointer first and returns the value at the pointer.

```
# addressing boilerplate
# immediate - no need because it's in the params
# accumulator - no need because it's a register
# relative - no need because it's in the params
# zeropage
def getZeroPageAddress(self, param):
    return param[0] & 0xFF # kinda redundant but guess i'll include it here
# zeropage x
def getZeroPageXAddress(self, param):
    return (param[0] + self.x) & 0xFF
# zeropage y
def getZeroPageYAddress(self, param):
    return (param[0] + self.y) & 0xFF
# absolute
def getAbsoluteAddress(self, param):
    return ((param[1] << 8) | param[0]) & 0xFFFF
# absolute x
def getAbsoluteXAddress(self, param):
    return (((param[1] << 8) | param[0]) + self.x) & 0xFFFF # no idea if i should & that or not
# absolute y
def getAbsoluteYAddress(self, param):
    return (((param[1] << 8) | param[0]) + self.y) & 0xFFFF # no idea if i should & that or not
```

Figure 16: Simple Addressing Boilerplate

The NES CPU is very simple, and it doesn't have many addressing modes. The code above from lines 310 - 331 is boilerplate for the simpler addressing modes of the NES. For the immediate, accumulator, and relative addressing modes, no boilerplate is needed because everything needed is inside the parameters of the function, or the function operates on a register directly, which doesn't require any further resolving.

The zeropage addressing mode simply accesses the first page (0x0 - 0xFF) of memory. For the zeropage X and Y modes, that zeropage value is then offset by the value inside either the X or Y register.

The absolute addressing mode works like the zeropage addressing mode. However, it accepts two parameters to address the full 16-bit address space. The first parameter is the offset, and the second parameter is the page. They can be combined into a full 16-bit number to index the memory. This is done by shifting the second parameter 8 bits to the left and combining it with the first. The absolute X and Y modes do the same thing as zeropage X and Y, but with the larger address space instead of being limited to the first page.

```
# indirect
def getIndirectAddress(self, param):
    lobyte = param[0] & 0xFF # bb
    hibyte = param[1] & 0xFF # cc

    # AN INDIRECT JUMP MUST NEVER USE A VECTOR BEGINNING ON THE LAST BYTE OF A PAGE (6502 jmpbug)
    address = (hibyte << 8) | lobyte # ccbb
    if (lobyte == 0xFF):
        self.log("jmpbug acquired!", 5)
        lobyte2 = self.rambus.memoryReadCPU(address) & 0xFF # xx
        hibyte2 = self.rambus.memoryReadCPU((address) & 0xFF00) # wraparound to page start
    else:
        lobyte2 = self.rambus.memoryReadCPU(address) & 0xFF # xx
        hibyte2 = self.rambus.memoryReadCPU((address & 0xFFFF) + 1) # yy

    address2 = (hibyte2 << 8) | lobyte2 # yyxx
    return address2 # set pc to this address for jmp
# indirect x
def getIndirectXAddress(self, param):
    # val = PEEK(PEEK((arg + X) % 256) + PEEK((arg + X + 1) % 256) * 256)
    return (self.rambus.memoryReadCPU((param[0] + self.x) & 0xFF) + self.rambus.memoryReadCPU((param[0] + self.x + 1) & 0xFF) * 0x100) & 0xFFFF
# indirect y
def getIndirectYAddress(self, param):
    # val = PEEK(PEEK(arg) + PEEK((arg + 1) % 256) * 256 + Y)
    return (self.rambus.memoryReadCPU(param[0]) + self.rambus.memoryReadCPU((param[0] + 1) & 0xFF) * 0x100 + self.y) & 0xFFFF
```

Figure 17: Indirect Addressing Boilerplate

The code above from lines 332 - 356 are responsible for the indirect addressing modes. The indirect addressing mode dereferences a pointer found at the parameter. First, the address is resolved by combining the two parameters just like the absolute addressing mode. However, there is a bug where the address will not cross a page boundary. If the address ends in 0xFF, it will read from the first byte and last byte of the same page instead of subsequent pages. Otherwise, the high byte is added by 1 to increment it for a 16-bit read. The address is finally combined using the same method as before and returned.

For indirect X and Y addresses, formulas from the NESDev Wiki were used and ported to my codebase. The formulas can be seen in the comments.

```
# immediate addressing
def __adc69(self, params):
    self.__adc(params[0])

# zeropage
def __adc65(self, params):
    address = self.getZeroPageAddress(params)
    self.__adc(self.rambus.memoryReadCPU(address))

# zeropage x
def __adc75(self, params):
    address = self.getZeroPageXAddress(params)
    self.__adc(self.rambus.memoryReadCPU(address))
```

Figure 18: Addressing Mode Wrappers

Since each instruction has several addressing modes, wrapper functions like the one shown above were used. There are internal functions for each of the instructions that accept a

common value / address format, and all the wrappers resolve the address or value before calling the internal function. Internal functions for the instructions will be shown below. For further information on the addressing modes of each instruction, a comment is placed in the code above every addressing mode wrapper.

```
# LDA: Load Accumulator with Memory
def __lda(self, memory):
    self.log(f"lda {memory}", 5)
    self.a = memory
    # set flags
    self.srFlagSet('z', memory == 0)
    self.srFlagSet('n', bool((memory >> 7) & 1))
```

Figure 19a: LDA Instruction

```
# LDX: Load Index X with Memory
def __ldx(self, memory):
    self.log(f"ldx {memory}", 5)
    self.x = memory
    # set flags
    self.srFlagSet('z', memory == 0)
    self.srFlagSet('n', bool((memory >> 7) & 1))
```

Figure 19b: LDX Instruction

```
# LDY: Load Index Y with Memory
def __ldy(self, memory):
    self.log(f"ldy {memory}", 5)
    self.y = memory
    # set flags
    self.srFlagSet('z', memory == 0)
    self.srFlagSet('n', bool((memory >> 7) & 1))
```

Figure 19c: LDY Instruction

The above code is responsible for the LDA, LDX, and LDY instructions. This instruction reads a value from memory, and stores it in one of the three general-purpose registers. The Z flag is set if the memory is zero, and the N flag is set if the 8^{th} bit is 1 (number is negative when signed).

```
# STA: Store Accumulator in Memory
def __sta(self, address):
    # self.memory[address] = self.a
    self.rambus.memoryWriteCPU(address, self.a)
    self.log(f"sta {self.a}", 5)
```

Figure 20a: STX Instruction

```
# STY: Store Index Y in Memory
def __sty(self, address):
    # self.memory[address] = self.y
    self.rambus.memoryWriteCPU(address, self.y)
    self.log(f"sty {self.y}", 5)
```

Figure 20b: STY Instruction

```
# STA: Store Accumulator in Memory
def __sta(self, address):
    # self.memory[address] = self.a
    self.rambus.memoryWriteCPU(address, self.a)
    self.log(f"sta {self.a}", 5)
```

Figure 20c: STA Instruction

The above code is the functions for the STX, STY, and STA instructions. They work in the exact opposite way as their load counterparts. They store the value from the registers into memory. However, they don’t write to the status register.

```python
# TAX: Transfer Accumulator to Index X
def __tax(self, params):
    self.x = self.a
    # set flags
    self.srFlagSet('z', self.x == 0)
    self.srFlagSet('n', bool((self.x >> 7) & 1))
    self.log(f"tax {self.x}", 5)

# TAY: Transfer Accumulator to Index Y
def __tay(self, params):
    self.y = self.a
    # set flags
    self.srFlagSet('z', self.y == 0)
    self.srFlagSet('n', bool((self.y >> 7) & 1))
    self.log(f"tay {self.y}", 5)
```

```python
# TXA: Transfer Index X to Accumulator
def __txa(self, params):
    self.a = self.x
    # set flags
    self.srFlagSet('z', self.x == 0)
    self.srFlagSet('n', bool((self.x >> 7) & 1))
    self.log(f"txa {self.x}", 5)
```

```python
# TYA: Transfer Index Y to Accumulator
def __tya(self, params):
    self.a = self.y
    # set flags
    self.srFlagSet('z', self.y == 0)
    self.srFlagSet('n', bool((self.y >> 7) & 1))
    self.log(f"tya {self.y}", 5)
```

Figure 21: Accumulator Transfer Instructions

The above functions for instructions TAX, TAY, TXA, and TYA are responsible for instructions that transfer to and from the accumulator and the other two general purpose registers. Simply put, it either sets A to the value of X or Y, and vice versa. The flags are set in a similar fashion; Z is for a zero result, and N is for a negative result.

```
# TSX: Transfer Stack Pointer to Index X
def __tsx(self, params):
    self.x = self.sp
    # set flags
    self.srFlagSet('z', self.x == 0)
    self.srFlagSet('n', bool((self.x >> 7) & 1))
    self.log(f"tsx {self.x}", 5)
```

Figure 22a: TSX Instruction

```
# TXS: Transfer Index X to Stack Pointer
def __txs(self, params):
    self.sp = self.x
    self.log(f"txs {self.x}", 5)
```

Figure 22b: TXS Instruction

The above functions for instructions TSX and TXS are responsible for direct operations to and from the stack pointer (SP) register. TSX copies the stack pointer to the X register. Flags are set accordingly. TXS does the opposite and does not set flags.

```
# ADC: Add Memory to Accumulator with Carry
def __adc(self, amount):
    self.log(f"adc {amount}", 5)
    orig_a = self.a
    # self.a += amount + int(self.srFlagGet('c'))
    result = self.a + amount + int(self.srFlagGet('c'))

    # set c flag and correct value
    self.srFlagSet('c', result > 255)
    self.a = result & 0xFF

    # set z flag
    self.srFlagSet('z', self.a == 0)

    # set n flag
    self.srFlagSet('n', bool((self.a >> 7) & 1))

    # set v flag
    v = ((orig_a ^ self.a) & (amount ^ self.a) & 0x80) != 0
    self.srFlagSet('v', v)
```

Figure 23a: ADC Instruction

```
# SBC: Subtract Memory from Accumulator with Borrow
def __sbc(self, memory):
    # initial operation
    oldA = self.a
    result = self.a - memory - int(not self.srFlagGet('c'))

    # weird flagset
    self.srFlagSet('c', bool(result >= 0)) # ~(result < $00) but its unsigned here

    # do 8bit wrap to val and do rest of flags
    self.a = result & 0xFF
    self.srFlagSet('v', bool((self.a ^ oldA) & (self.a ^ ~memory) & 0x80)) # (result ^ A) & (result ^ ~memory) & $80
    self.srFlagSet('z', self.a == 0) # result == 0
    self.srFlagSet('n', bool((self.a >> 7) & 1))
    self.log(f"sbc {self.a}", 5)
```

Figure 23b: SBC Instruction

The above functions are responsible for the arithmetic instructions ADC and SBC. Despite being named differently, they perform similar but opposite operations; one being addition and the other being subtraction.

ADC Means “Add Memory to Accumulator with Carry”. This means that the carry flag is taken into account with the addition. The function simply adds the A register, the value in memory, and the carry flag. Then, it sets the carry flag depending on whether or not the final result is above 255. The Z flag is set if the result is zero (after accounting for overflows), the N flag is set if the result is negative, and the V flag is set if the sign of the result and the original value is different. This indicates if there was an overflow. The formula for the V flag was taken from the CPU instructions reference page in the NESDev Wiki.

SBC is the same as ADC, but it subtracts instead of adds. It subtracts the memory, then further subtracts the NOT of the C (carry) flag. This formula was also taken from the NESDev Wiki. The C flag is then set if there was un underflow, and the rest of the flags exhibit the same behavior as ADC. Both ADC and SBC have an 8-bit result, but the internal logic is 16 bits.

```
# INC: Incrament Memory by One
def __inc(self, address):
    self.log(f"inc {address}", 5)
    # self.memory[address] += 1
    # self.memory[address] &= 0xFF
    self.rambus.memoryWriteCPU(address, (self.rambus.memoryReadCPU(address) + 1) & 0xFF)
    # update flags
    self.srFlagSet('z', self.rambus.memoryReadCPU(address) == 0)
    self.srFlagSet('n', bool((self.rambus.memoryReadCPU(address) >> 7) & 1))
```

Figure 24a: Memory Increment Instruction

```
# INX: Increment Index X by One
def __inx(self, params):
    self.log(f"inx", 5)
    self.x += 1
    self.x &= 0xFF
    # update flags
    self.srFlagSet('z', self.x == 0)
    self.srFlagSet('n', bool((self.x >> 7) & 1))

# INY: Increment Index Y by One
def __iny(self, params):
    self.log(f"iny", 5)
    self.y += 1
    self.y &= 0xFF
    # update flags
    self.srFlagSet('z', self.y == 0)
    self.srFlagSet('n', bool((self.y >> 7) & 1))
```

Figure 24b: Register Increment Instructions

The above functions are responsible for the increment instructions INC, INX, and INY that increment a value in memory, the X register, and the Y register respectively. It simply increments the 8-bit number by 1, and could overflow back to 0. The two flags that are set are Z and N, which are set if the result is zero or negative respectively.

```
# DEC: Decrement Memory by One
def __dec(self, address):
    self.log(f"dec {address}", 5)

    # self.memory[address] -= 1
    # self.memory[address] &= 0xFF
    self.rambus.memoryWriteCPU(address, (self.rambus.memoryReadCPU(address) - 1) & 0xFF)

    # update flags
    self.srFlagSet('z', self.rambus.memoryReadCPU(address) == 0)
    self.srFlagSet('n', bool((self.rambus.memoryReadCPU(address) >> 7) & 1))
```

Figure 25a: Memory Decrement Instruction

```
# DEX: Decrement Index X by One
def __dex(self, params):
    self.log(f"dex", 5)
    self.x -= 1
    self.x &= 0xFF
    # update flags
    self.srFlagSet('z', self.x == 0)
    self.srFlagSet('n', bool((self.x >> 7) & 1))

# DEY: Decrement Index Y by One
def __dey(self, params):
    self.log(f"dey", 5)
    self.y -= 1
    self.y &= 0xFF
    # update flags
    self.srFlagSet('z', self.y == 0)
    self.srFlagSet('n', bool((self.y >> 7) & 1))
```

Figure 25b: Register Decrement Instructions

The above code for the DEC, DEX, and DEY instructions behave exactly like the increment instructions. However, they decrement by 1 instead of incrementing by 1.

```
# AND: AND Memory with Accumulator
def __and(self, value):
    self.log(f"and {value}", 5)
    self.a = self.a & value

    # set n and z flag
    self.srFlagSet('n', bool((self.a >> 7) & 1))
    self.srFlagSet('z', self.a == 0)
```

Figure 26a: AND Instruction (Bitwise AND)

```
# ORA: OR Memory with Accumulator
def __ora(self, memory):
    self.log(f"ora {memory}", 5)
    self.a |= memory
    # set cpu flags
    self.srFlagSet('z', self.a == 0)
    self.srFlagSet('n', bool((self.a >> 7) & 1))
```

Figure 26b: ORA Instruction (Bitwise OR)

```
# EOR: Exclusive-OR Memory with Accumulator
def __eor(self, memory):
    self.log(f"eor {memory}", 5)
    self.a ^= memory
    # set flags
    self.srFlagSet('z', self.a == 0)
    self.srFlagSet('n', bool((self.a >> 7) & 1))
```

Figure 26c: EOR Instruction (Bitwise XOR / Exclusive OR)

The above functions are for the AND, ORA, and EOR instructions that do bitwise operations to a value stored in memory. AND does a bitwise AND, ORA does a bitwise OR, and EOR does a bitwise XOR. All of these operate on the memory and the accumulator (A) register. After the operation, the Z and N flags are set for zero and negative values respectively.

```
# BIT: Test Bits in Memory with Accumulator
# zero page
def __bit24(self, params):
    address = self.getZeroPageAddress(params)
    result = self.a & self.rambus.memoryReadCPU(address)
    self.log(f"bit {result}", 5)


    # set bits
    self.srFlagSet('z', result == 0)
    self.srFlagSet('v', bool((self.rambus.memoryReadCPU(address) >> 6) & 1))
    self.srFlagSet('n', bool((self.rambus.memoryReadCPU(address) >> 7) & 1))

def __bit2C(self, params):
    address = self.getAbsoluteAddress(params)
    result = self.a & self.rambus.memoryReadCPU(address)
    self.log(f"bit {result}", 5)

    # set bits
    self.srFlagSet('z', result == 0)
    self.srFlagSet('v', bool((self.rambus.memoryReadCPU(address) >> 6) & 1))
    self.srFlagSet('n', bool((self.rambus.memoryReadCPU(address) >> 7) & 1))
```

Figure 27: BIT Instruction

The above functions are responsible for the BIT instruction. It is a bitwise AND, however it does not write anything to memory or any registers except for the status register. It only has two addressing modes, zeropage and absolute. For this reason, it does not use wrapper / boilerplate functions. It is typically used to check bitmasks in code.

After the bitwise AND, it sets some flags. The Z flag is set if the result is 0, and bits 6 and 7 of the result are put into the V and N flags respectively.

```
# ASL: Shift Left One Bit (Memory or Accumulator)
def __asl(self, address):
    old = self.rambus.memoryReadCPU(address)
    # self.memory[address] = self.memory[address] << 1
    self.rambus.memoryWriteCPU(address, (self.rambus.memoryReadCPU(address) << 1) & 0xFF) # can overflow
    # set status flags
    self.srFlagSet('c', bool((old >> 7) & 1))
    self.srFlagSet('n', bool((self.rambus.memoryReadCPU(address) >> 7) & 1))
    self.srFlagSet('z', self.rambus.memoryReadCPU(address) == 0)
    pass
```

Figure 28a: ASL Instruction (Shift Left)

```
# LSR: Shift One Bit Right (Memory or Accumulator)
def __lsr(self, address):
    self.log(f"lsr {address}", 5)
    old = self.rambus.memoryReadCPU(address)
    # self.memory[address] = self.memory[address] >> 1
    self.rambus.memoryWriteCPU(address, (self.rambus.memoryReadCPU(address) >> 1) & 0xFF)
    # set status flags
    self.srFlagSet('c', bool(old & 1))
    self.srFlagSet('n', False)
    self.srFlagSet('z', self.rambus.memoryReadCPU(address) == 0)
```

Figure 28b: LSR Instruction (Shift Right)

The two functions above are responsible for the ASL and LSR instructions, which does a logical shift left or right with the carry flag. This means that the carry flag is placed as an extra bit to the left for ASL, and to the right for LSR. For both instructions, the Z flag is set to whether or not the result is zero (not including the carry flag).

For ASL (shift left), the N flag is set to whether the result is negative or not. This is done by extracting the sign, or the 8th bit. The carry (C) flag is set to the 8th bit of the old value before shifting, as it is meant to be shifted left into C.

For LSR, the N flag will always be false, as shifting right fills the 8th bit with a zero. However, the carry (C) flag is set to the first instead of last bit, as the first bit is what's shifted into the carry flag.

```
# ROL: Rotate One Bit Left (Memory or Accumulator)
def __rol(self, address):
    # Move each of the bits in either A or M one place to the left. Bit 0 is filled with the current value of the carry flag whilst the old bit 7 becomes the
    self.log(f"rol {address}", 5)
    old = self.rambus.memoryReadCPU(address)
    # self.memory[address] = self.memory[address] << 1 # Move each of the bits in either A or M one place to the left
    # self.memory[address] |= int(self.srFlagGet('c')) # Bit 0 is filled with the current value of the carry flag
    self.rambus.memoryWriteCPU(address, (self.rambus.memoryReadCPU(address) << 1) & 0xFF) # Move each of the bits in either A or M one place to the left
    self.rambus.memoryWriteCPU(address, self.rambus.memoryReadCPU(address) | int(self.srFlagGet('c'))) # Bit 0 is filled with the current value of the carry
    # set all da flags
    self.srFlagSet('c', bool((old >> 7) & 1)) # the old bit 7 becomes the new carry flag value
    self.srFlagSet('z', self.rambus.memoryReadCPU(address) == 0)
    self.srFlagSet('n', bool((self.rambus.memoryReadCPU(address) >> 7) & 1))
```

Figure 29a: ROL Instruction (Rotate Left)

```
# ROL: Rotate Right Bit Left (Memory or Accumulator)
def __ror(self, address):
    # Move each of the bits in either A or M one place to the right. Bit 7 is filled with the current value of the carry flag whilst the old bit 0 becomes the new
    self.log(f"rol {address}", 5)
    old = self.rambus.memoryReadCPU(address)
    # self.memory[address] = self.memory[address] >> 1 # Move each of the bits in either A or M one place to the right
    # self.memory[address] |= ((int(self.srFlagGet('c'))) << 7) # Bit 7 is filled with the current value of the carry flag
    self.rambus.memoryWriteCPU(address, (self.rambus.memoryReadCPU(address) >> 1) & 0xFF) # Move each of the bits in either A or M one place to the right
    self.rambus.memoryWriteCPU(address, (self.rambus.memoryReadCPU(address) | ((int(self.srFlagGet('c'))) << 7)) & 0xFF) # Bit 7 is filled with the current value

    # set all da flags
    self.srFlagSet('c', bool(old & 1)) # the old bit 0 becomes the new carry flag value
    self.srFlagSet('z', self.rambus.memoryReadCPU(address) == 0)
    self.srFlagSet('n', bool((self.rambus.memoryReadCPU(address) >> 7) & 1))
```

Figure 29b: ROR Instruction (Rotate Right)

The following functions are responsible for the ROL and ROR instructions. They simply rotate a value left or right. The logic for these instructions was taken from both the Obelisk 6502 Guide, and the updated 6502 instruction reference in the NESDev wiki. First, the old, pre-shifted value is saved for later. Then, the value is shifted to the left or right according to the instruction. Then, the contents of the carry flag are filled into bit 0 for ROL, and bit 7 for ROR. Then, the carry flag is filled with the old (pre-shifted) value of bit 7 for ROL, and bit 0 for ROR.

```
# CMP: Compare Memory with Accumulator
def __cmp(self, memory, register = None):
    self.log(f"cmp {memory}", 5) # TODO: fix instruction name print for cpx and cpy
    # for other cmps
    if (register == None):
        register = self.a

    result = (register - memory) & 0xFF # no idea if this should wraparound or not

    # set flags
    self.srFlagSet('c', register >= memory)
    self.srFlagSet('z', register == memory)
    self.srFlagSet('n', bool((result >> 7) & 1))
```

Figure 30: CMP Instruction

The following function is the internal function responsible for the CMP, CPX, and CPY instructions. They all do the same thing, but operate on different registers. These instructions are compare instructions that are typically used before branches, just like an if statement in a modern programming language. CMP operates on A, CPX operates on X, and CPY operates on Y. The code uses a default value for the register parameter to simulate function overloading, and defaults the register to the A register if nothing is passed into it.

The formula for this is taken from the NESDev Wiki. The instruction works by subtracting the value in the register with the value in memory. It does not write back to the register or memory, but instead writes to the CPU's status register, just like BIT. C is set if the value in the register is higher than the value in memory, Z is set if the result is zero (which occurs when both register and memory holds the same value), and N is set according to the 8th bit of the result as usual.

```
# branch wrapper
def __branch(self, size):
    self.pc += self.toSign8(size)
    self.pc &= 0xFFFF
    self.log(f"branching {self.toSign8(size)} to {hex(self.pc)}", 5)
```

Figure 31: Branch Wrapper Function

Since there are many branch or jump instructions in the CPU, it is wise to create a wrapper for it. This function found in lines 509 - 513 handles branching. It simply converts the branch distance (named size in the function) to a signed 8-bit number, and adds it to the current program counter. It then limits the result to 16-bit to prevent out-of-bounds errors.

```
# BCC: Branch on Carry Clear
def __bcc(self, params):
    self.log(f"bcc {params[0]}", 5)
    if self.srFlagGet('c') == False:
        self.__branch(params[0]) # function size will be added to pc after exec

# BCS: Branch on Carry Set
def __bcs(self, params):
    self.log(f"bcs {params[0]}", 5)
    if self.srFlagGet('c'):
        self.__branch(params[0]) # function size will be added to pc after exec

# BEQ: Branch on Result Zero
def __beq(self, params):
    self.log(f"beq {params[0]}", 5)
    if self.srFlagGet('z'):
        self.__branch(params[0]) # function size will be added to pc after exec
```

```
# BMI: Branch on Result Minus
def __bmi(self, params):
    self.log(f"bmi {params[0]}", 5)
    if self.srFlagGet('n'):
        self.__branch(params[0]) # function size will be added to pc after exec

# BNE: Branch on Result Not Zero
def __bne(self, params):
    self.log(f"bne {params[0]}", 5)
    if self.srFlagGet('z') == False:
        self.__branch(params[0]) # function size will be added to pc after exec

# BPL: Branch on Result Zero
def __bpl(self, params):
    self.log(f"bpl {params[0]}", 5)
    if self.srFlagGet('n') == False:
        self.__branch(params[0]) # function size will be added to pc after exec
```

```
# BVC: Branch on Overflow Clear
def __bvc(self, params):
    self.log(f"bvc {params[0]}", 5)
    if self.srFlagGet('v') == False:
        self.__branch(params[0]) # function size will be added to pc after exec

# BVS: Branch on Overflow Set
def __bvs(self, params):
    self.log(f"bvs {params[0]}", 5)
    if self.srFlagGet('v'):
        self.__branch(params[0]) # function size will be added to pc after exec
```

Figure 32: Branch Instructions

The 8 functions shown above are all the conditional branch instructions found in the CPU. They branch if a certain condition is met inside one of the CPU status register flags. Here is what each instruction does:

- BCS: Branches if C is 1
- BCC: Branches if C is 0
- BEQ: Branches if Z is 1
- BNE: Branches if Z is 0
- BMI: Branches if N is 1
- BPL: Branches if N is 0
- BVS: Branches if V is 1
- BVC: Branches if V is 0

```
# JMP: Jump to New Location
def __jmp(self, address):
    self.log(f"jmp {address}", 5)
    self.pc = address - 3 # function size will be added to pc after exec
```

Figure 33: JMP Instruction

The function shown above is responsible for the JMP instruction. It simply changes the program counter to the pointed address. However, the code subtracts it by 3, since the opcodes for this instruction are 3 bytes long. The FDE cycle logic will add those 3 bytes later. This differs from branch instructions, which don't need to compensate for this.

```python
# JSR: Jump to New Location Saving Return Address
def __jsr(self, params):
    address = self.getAbsoluteAddress(params)
    # this is pretty much a call, pushes return address to top of stack for later
    ret = self.pc+2
    self.log(f"jsr {hex(address)}; ret to {hex(ret)}", 5)
    hibyte = (ret >> 8) & 0xFF
    lobyte = ret & 0xFF
    self.stackPush(hibyte)
    self.stackPush(lobyte)
    self.pc = address - 3 # function size will be added to pc after exec
```

Figure 34a: JSR Instruction

```python
# RTS: Return from Subroutine
def __rts(self, params):
    lobyte = self.stackPull()
    hibyte = self.stackPull()
    self.pc = (((lobyte & 0xFF) | (hibyte << 8))) & 0xFFFF # limit to 16 bit address space
    self.log(f"rts {hex(self.pc)}", 5)
```

Figure 34b: RTS Instruction

The two functions shown above are responsible for the JSR and RTS instructions, which is how the 6502 handles functions or methods. The JSR instruction works like the JMP instruction. However, it saves the current program counter (incremented by 2 to account for the opcode size) to the stack, pushing the high byte first before the low byte. Then, it executes the jump.

The RTS instruction does the opposite. It means return from subroutine, which does the same thing as returning from a function in a modern programming language. It pulls the return address from the stack (previously pushed by JSR) and sets the program counter to the return address after reconstructing it back to a 16-bit number.

```python
# PHA: Push Accumulator on Stack
def __pha(self, params):
    self.log(f"pha {self.a}", 5)
    self.stackPush(self.a)

# PHP: Push Processor Status on Stack
def __php(self, params):
    self.log(f"php {self.sr}", 5)
    res = self.sr | 0b00110000 # modified before push
    self.stackPush(res)

# PLA: Pull Accumulator from Stack
def __pla(self, params):
    self.a = self.stackPull()
    self.log(f"pla {self.a}", 5)

    # set cpu flags
    self.srFlagSet('z', self.a == 0)
    self.srFlagSet('n', bool((self.a >> 7) & 1))

# PLP: Pull Processor Status from Stack
def __plp(self, params):
    # TODO: interrupt disable delayed 1 instruction
    old = (self.sr & 0b00110000) # save only 2 old bits
    self.sr = (self.stackPull() & 0b11001111) | old # set the status register
    self.log(f"plp {self.sr}", 5)
```

Figure 35: Stack Instructions

The 4 functions above are responsible for the stack instructions PHA, PHP, PLA, and PLP. PHA and PLA simply pushes and pulls the accumulator (A) register into the stack. Z and N flags are set accordingly for PLA according to whether or not the value is zero or negative.

PHP pushes the status register to the stack. However, a bitwise OR with a mask is done to permanently set two bits to 1. When pulling this value with PLP, these two bits are also ignored. This is done by saving the old bits first as a mask with a bitwise AND, then doing another bitwise AND with the opposite mask on the pulled value in order to set them to zero. Then, it is ORed with the old mask in order to restore the proper value to the zeroed bits. Finally, this value is moved into the status register.

```
# CLC: Clear Carry Flag
def __clc(self, params):
    self.log(f"clc", 5)
    self.srFlagSet('c', False)

# CLD: Clear Decimal Mode
def __cld(self, params):
    self.log(f"cld", 5)
    self.srFlagSet('d', False)

# CLI: Clear Interrupt Disable Bit
def __cli(self, params):
    self.log(f"cli", 5)
    self.srFlagSet('i', False)

# CLV: Clear Overflow Flag
def __clv(self, params):
    self.log(f"clv", 5)
    self.srFlagSet('v', False)
```

Figure 36a: Clear Flag Instructions

```
# SEC: Set Carry Flag
def __sec(self, params):
    self.log("sec", 5)
    self.srFlagSet('c', True)

# SED: Set Decimal Flag
def __sed(self, params):
    self.log("sed", 5)
    self.srFlagSet('d', True)

# SEI: Set Interrupt Disable Status
def __sei(self, params):
    self.log("sei", 5)
    self.srFlagSet('i', True) # TODO: this is delayed by 1 instruction
```

Figure 36b: Set Flag Instructions

The 7 functions shown above are responsible for the flag setting and clearing instructions CLC, CLD, CLI, CLV, SEC, SED, and SEI. The naming convention is simple. CL is clear, and SE is set. The last character is the flag that's being set to 1 or cleared to 0.

```
def interrupt(self, handler, isBrk = False):
    # push ret ptr to stack
    hibyte = (self.pc >> 8) & 0xFF
    lobyte = self.pc & 0xFF

    # brk sets the B flag to true
    if isBrk:
        flagPush = self.sr | 0b00010000
    else:
        flagPush = self.sr & 0b11101111

    self.stackPush(hibyte)
    self.stackPush(lobyte)
    self.stackPush(flagPush)

    # set i flag according to nesdev
    self.srFlagSet('i', 1)

    # set pc to interrupt handler
    lo = handler & 0xFF
    hi = (handler >> 8) & 0xFF
    self.pc = self.getIndirectAddress([lo, hi])
```

Figure 37: Interrupt Handler Function

The code shown above is the function to help the CPU jump to an interrupt handler. It uses a default value for the isBrk parameter to simulate function overloading. It also allows for a configurable interrupt handler, since there are different kinds of interrupts with different handlers. It works similar to the JSR instruction. First, pushes the current program counter to stack. Then, it pushes the current status register to stack. However, the B flag is set or cleared depending on whether or not the interrupt is invoked by the BRK function. Then, it set the interrupt disable flag to true, and sets the program counter to the interrupt handler. The interrupt handler address is obtained using the same method as the entrypoint using the indirect addressing logic, however it's typically in a different address from the ROM's entrypoint.

```
# BRK: Force Break
def __brk(self, params):
    self.interrupt(0xFFFE, True)
```

Figure 38: BRK Instruction

The above function is responsible for the BRK instruction. All it does is call for an interrupt with indirect handler address 0xFFFE. The isBrk flag is passed to push the correct B flag into the stack, as mentioned in the interrupt function section.

```
# RTI: Return from Interrupt
def __rti(self, params):
    old = (self.sr & 0b00110000) # save only 2 old bits

    flags = (self.stackPull()) & 0b11001111 # 2 flags are ignored
    lobyte = self.stackPull()
    hibyte = self.stackPull()

    # set the stuff
    self.sr = flags | old
    # self.pc = (lobyte & 0xFF) | ((hibyte << 8) & 0xFF)
    self.pc = (((lobyte & 0xFF) | (hibyte << 8)) - 1) & 0xFFFF # limit to 16 bit address space
    self.log(f"rti {self.pc}", 5)
```

Figure 39: RTI Instruction

The above function is responsible for the RTI instruction. It returns from the interrupt handler. The interrupt handler works like a function called using JSR and RTS, however interrupt handlers use RTI to return. Additionally, it also sets the status register flags, ignoring two flags using opposite bitwise ANDs for the masks, and using a bitwise OR to set it. This logic is similar to the PLP instruction.

```
# NOP: No Operation
def __nop(self, params):
    self.log("nop", 5)
```

Figure 40: NOP Instruction

The above function is responsible for the NOP instruction. It does nothing.

That concludes all the CPU instructions. However, the emulator also has a test mode for the CPU in order to sanity check it. It does this by running a test ROM and comparing it to a known good test. The emulator doesn't implement illegal instructions, so a single illegal instruction is wrongly implemented to end the test.

```python
def __tst(self, params):
    if self.testmode:
        # print("Test complete. Please compare the output with the expected output at testcase.txt.")
        with open("testcase.txt", "r") as tc:
            expected = tc.readlines()
        with open("testoutput.txt", "r") as to:
            result = to.readlines()
        # it should exit here if anything is bad
        try:
            for i in range(len(expected)):
                if expected[i] != result[i]:
                    self.__endtst(False, i + 1)
        except Exception as e:
            print(e)
            self.__endtst(False, len(result) + 1)

        # if nothing is bad then it goes here
        self.__endtst(True)

def __endtst(self, success = False, line = None):
    if success:
        print("Test succeeded. CPU is working properly.")
    else:
        print(f"Test failed at line {line}. Please check testoutput.txt and compare it with testcase.txt.")
    sys.exit(0)
```

Figure 41: CPU Sanity Check Functions

The code above is the handler for the end of the test. It is mapped to one of the illegal instructions. After the test finishes, it opens both the output and the known good test. It reads every line and compares them. If an exception occurs (likely out-of-bounds), it prints a fail with the last line of the output. Otherwise, it prints a fail if one line doesn't match and tells the user where it failed. This is done using a for loop, where the i variable is which line is currently being read. The print adds 1 to the line because the counter starts at 0.

Solution Design - NES PPU/Graphics Implementation

```
class dmppu:
    def __init__(self, rambus, loglevel = 3):
        self.window = dmslopywindow()

        self.rambus = rambus
        self.rambus.readHooks.append(self.ram_read)
        self.rambus.writeHooks.append(self.ram_write)
        self.secondWrite = False

        self.oam = [0] * 0x256
        self.patternTable = []

        # init state
        self.rambus.cpumem[0x2000] = 0 # ctrl
        self.rambus.cpumem[0x2001] = 0 # mask
        self.rambus.cpumem[0x2002] = 0 # status
        self.rambus.cpumem[0x2003] = 0 # oamaddr
        self.rambus.cpumem[0x2004] = 0 # oamdata
        self.rambus.cpumem[0x2005] = 0 # scroll (internal 2 byte)

        self.cycles = 0
        self.lastFrame = time.time()

        # read from PPU memory?
        # 2 byte internal register for addr
        self.intlAddr = 0
        self.__intlDataBuf = 0
        self.rambus.cpumem[0x2006] = 0 # addr
        self.rambus.cpumem[0x2007] = 0 # data

        # skibidi dma
        self.rambus.cpumem[0x4014] = 0 # oamdma

        # readregisters
        self.ctrl = self.rambus.cpumem[0x2000] # done
        self.mask = self.rambus.cpumem[0x2001] # done
        self.status = self.rambus.cpumem[0x2002] # done
        self.oamaddr = self.rambus.cpumem[0x2003] # done
        self.oamdata = self.rambus.cpumem[0x2004] # done
        self.scrollx = self.rambus.cpumem[0x2005] # done
        self.scrolly = self.rambus.cpumem[0x2005] # done
        self.addr = self.rambus.cpumem[0x2006] # done
        self.data = self.rambus.cpumem[0x2007] # done
        self.oamdma = self.rambus.cpumem[0x4014] # done

        # constants
        self.__ctrlFlags = {
            'n0': 0,
            'n1': 1,
            'i': 2, # ppudata address incrament
            's': 3, # sprite pattern table addr
            'b': 4, # bg pattern table addr
            'h': 5, # sprite size
            'p': 6, # ppu master/slave
            'v': 7 # vblank NMI (INTERRUPT!!!)
        }
        self.__maskFlags = {
            'g': 0,
            'm': 1,
            'M': 2,
            'b': 3,
            's': 4,
            'R': 5,
            'G': 6,
            'B': 7
        }
        self.__statusFlags = {
            'o': 5,
            's': 6,
            'v': 7
        }
```

Figure 42: PPU Constructor

The code shown above found in lines 41 - 111 of ppu.py is the constructor for the PPU. It contains mostly the same things as the CPU constructor, initializing the registers. However, the PPU communicates to the CPU using memory-mapped I/O, where it exposes its registers. There are a lot of registers in the PPU. The registers are as follows:

- Ctrl
- Mask
- Status
- OamAddr
- OamData
- ScrollX
- ScrollY
- Addr
- Data
- OamData

The purposes of these registers will be further explained in this section. There are also bitfields similar to the CPU's status registers. However, the PPU has three registers that work as bitfields. In the start, the PPU creates read and write hooks in order to intercept the CPU's reads and writes for memory-mapped I/O. The code also initializes a Pygame window, which could be in the window.py file below:

```
import pygame
class dmslopywindow():
    def __init__(self):
        pygame.init()
        self.screen = pygame.display.set_mode((640, 600), pygame.RESIZABLE)
        pygame.display.set_caption("slopylator main")

        self.framebuffer = pygame.Surface((256, 240))
```

Figure 43: Window Constructor

In order to make the window resizable, a framebuffer is created that will then stretch to the window. More on this later.

```
def ctrlFlagGet(self, flag):
    # return an int for this
    if flag == 'n':
        return self.ctrl & 0b00000011
    if flag[0] in self.__ctrlFlags:
        return bool((self.ctrl >> self.__ctrlFlags[flag[0]]) & 1)

def ctrlFlagSet(self, flag, enable):
    if flag == 'n':
        self.ctrlFlagSet('n0', bool(enable & 1))
        self.ctrlFlagSet('n1', bool(enable & 2))
        return
    if flag[0] in self.__ctrlFlags:
        mask = 1 << self.__ctrlFlags[flag[0]]
        if enable:
            self.ctrl |= mask
        else:
            self.ctrl &= ~mask
```

Figure 44a: Ctrl Flag Wrapper

```
def maskFlagGet(self, flag):
    if flag[0] in self.__maskFlags:
        return bool((self.mask >> self.__maskFlags[flag[0]]) & 1)

def maskFlagSet(self, flag, enable):
    if flag[0] in self.__maskFlags:
        mask = 1 << self.__maskFlags[flag[0]]
        if enable:
            self.mask |= mask
        else:
            self.mask &= ~mask

def statusFlagGet(self, flag):
    if flag[0] in self.__statusFlags:
        return bool((self.status >> self.__statusFlags[flag[0]]) & 1)

def statusFlagSet(self, flag, enable):
    if flag[0] in self.__statusFlags:
        mask = 1 << self.__statusFlags[flag[0]]
        if enable:
            self.status |= mask
        else:
            self.status &= ~mask
```

Figure 44b: Mask and Status Flag Wrappers

The code shown above found between lines 113 - 154 is responsible for the three status registers in the PPU. It works just like the CPU's one.

```
def ram_read(self, address):
    if address == 0x2000:
        return self.ctrl
    if address == 0x2001:
        return self.mask
    if address == 0x2002:
        status = self.status

        # reset vblank on read
        self.statusFlagSet('v', False)

        #read resets write pair for $2005/$2006
        self.secondWrite = False

        return status
    if address == 0x2003:
        return self.oamaddr
    if address == 0x2004:
        return self.oam[self.oamaddr]
    if address == 0x2005:
        return self.scroll
    if address == 0x2006:
        return self.addr
    if address == 0x2007:
        # print(f"vram read {hex(self.intlAddr)}")

        # directly read from vram for palette data
        if (self.intlAddr >= 0x3F00) and (self.intlAddr <= 0x3FFF):
            return_value = self.rambus.memoryReadPPU(self.intlAddr)
            # buffer the data "underneath"
            self.data = self.rambus.memoryReadPPU(self.intlAddr - 0x1000)
        else:
            return_value = self.data
            self.data = self.rambus.memoryReadPPU(self.intlAddr)
        self.intlAddr += (int(self.ctrlFlagGet('i')) * 31 + 1)
        return return_value
```

Figure 45: PPU RAM Read Interception

The code shown above from lines 156 - 191 is the read hook for the PPU's memory mapped I/O read operations. A couple addresses are handled here. For 0x2000 - 0x2006, it simply returns the corresponding PPU register. For 0x2002, it resets the VBLANK flag, which is used for some game logic regarding rendering and timing. For 0x2007, it reads data from the address pointed by internal address register for the VRAM data inside the PPU. This includes pallete data which has special handling by reading from the VRAM directly. However, it otherwise returns the value of the data register and sets it to the value inside the internal address register.

```python
def ram_write(self, address, value):
    if address == 0x2000:
        self.ctrl = value
        return True
    if address == 0x2001:
        self.mask = value
        return True
    if address == 0x2002:
        self.status = value
        return True
    if address == 0x2003:
        self.oamaddr = value
        return True
    if address == 0x2004:
        # print("oam write")
        self.oam[self.oamaddr] = value
        self.oamaddr = (self.oamaddr + 1) & 0xFF
        return True
    if address == 0x2005:
        if self.secondWrite:
            self.scrolly = value
        else:
            self.scrollx = value
        self.secondWrite = not self.secondWrite
        return True
    if address == 0x2006:
        self.addr = value

        if self.secondWrite:
            self.intlAddr = (self.intlAddr & 0xFF00) + value
        else:
            self.intlAddr = (self.intlAddr & 0x00FF) + (value << 8)

        self.secondWrite = not self.secondWrite
        return True
    if address == 0x2007:
        self.rambus.memoryWritePPU(self.intlAddr, value)
        self.intlAddr += (int(self.ctrlFlagGet('i')) * 31 + 1)
        return True
    if address == 0x4014:
        # TODO: 513-514 cycle delay
        page = value * 0x100
        self.oam[0x00:0xFF] = self.rambus.memoryReadCPU(page, page + 0xFF)
        # print(f"oam dma {hex(page)}")
        return True
```

Figure 46: PPU RAM Write Interception

The code shown above is the opposite of the read. It is a write hook for the PPU MMIO. 0x2000 - 0x2003 are handled normally by just writing the register value verbatim. However, several registers write to an internal 16-bit register with only an 8-bit value. This is why on some addresses like 0x2005 and 0x2006, the first write writes to the top 8 bits, while the second writes to the lower 8 bits. This applies to the scroll register, which is internally one register, but it's been split into 2 in the emulator code in order to simplify scrolling behavior. Keep in mind that they share the same internal up/down flip-flop. Writes to 0x2004 increment the OAM address, as the CPU could write to it multiple lines to write into the OAM buffer, which is sprite memory. 0x2007 writes behave just like the reads but in reverse, using the i flag to determine the proper address direction (going up or down). 0x4014 simply points to a page in main memory that could be used to DMA into the OAM. The emulator simply does a copy.

```python
def buildPatternTable(self):
    address = 0
    patternTable = []

    for i in range(2): # iterate through both pattern tables
        tiles = []
        for j in range(256): # iterate through all tiles in table
            tile = []
            for k in range(8): # 8*2 byte pairs per tile
                lobyte = self.rambus.memoryReadPPU(address)
                hibyte = self.rambus.memoryReadPPU(address + 8)
                address += 1

                row = []
                for x in range(8): # for each pixel in the row
                    # decrement, take msb first
                    bit0 = (lobyte >> (7 - x)) & 1
                    bit1 = (hibyte >> (7 - x)) & 1
                    color_index = (bit1 << 1) | bit0
                    #print(color_index)
                    row.append(color_index)
                tile.append(row)
            address += 8 # already parsed the other 8 hibytes, skip
            tiles.append(tile)

        patternTable.append(tiles)
    self.patternTable = patternTable
```

Figure 47: Pattern Table Builder Function

The function shown above in lines 261 - 287 builds the pattern tables for the PPU. Pattern tables store tile information as a 4-bit color bitmap. The code works by using nested loops. The PPU has two pattern tables, where each pattern table contains 256 tiles and each of those pair 8x2 bytes. It increments an address in the pattern table this way, and processes it per row. It extracts color data by taking the color information bit from the high and low bytes, since the high byte stores all the high bits of the color and the low byte stores all the low bits of the color. After this is done, it appends a row, then constructs tiles out of the rows. After all of that, it finally finishes the pattern table and stores it in the PPU.

```python
def renderSprites(self):
    pattern_table = self.patternTable[int(self.ctrlFlagGet('s'))]

    # oam fits 64 sprites (256 bytes / 4 bytes per sprite)
    for i in range(64):
        base = i * 4

        # sprite object
        # back in MY DAY we did OOP manually in assembly
        sprite_y = self.oam[base + 0]
        tile_index = self.oam[base + 1]
        attr = self.oam[base + 2]
        sprite_x = self.oam[base + 3]
        palette_id = attr & 0b00000011
        flip_x = bool(attr & 0b01000000) # bit six
        flip_y = bool(attr & 0b10000000) # bit SEVENNNNNNN

        # sprite to render
        tile = pattern_table[tile_index]

        # reverse order if flip
        if (flip_y):
            tile = tile[::-1]

        screen_y = sprite_y
        for row in tile:
            screen_x = sprite_x

            # reverse order if flip
            if (flip_x):
                row = row[::-1]

            for pixel in row:
                if pixel != 0:
                    color = spriteColors[palette_id][pixel]
                    self.window.framebuffer.set_at((screen_x, screen_y), color)
                screen_x += 1

            screen_y += 1
```

Figure 48a: Sprite Rendering Code

```python
spriteColors = [
    [
        None,
        (255, 0, 0),
        (200, 0, 0),
        (150, 0, 0),
    ],

    [
        None,
        (0, 255, 0),
        (0, 200, 0),
        (0, 150, 0),
    ],

    [
        None,
        (0, 0, 255),
        (0, 0, 200),
        (0, 0, 150),
    ],

    [
        None,
        (255, 255, 0),
        (200, 200, 0),
        (150, 150, 0),
    ],
]
```

Figure 48b: Sprite Color Palette

The code shown above from lines 337 - 375 is the function responsible for rendering sprites on screen. The sprites are stored in the OAM (Object Attribute Memory), and the OAM fits 64 sprites of 4 bytes each. First, it selects a pattern table based on the s flag. Then, it iterates through every entry in the OAM. It first gets the base address by multiplying the index with the struct's size, which is 4 bytes. The first byte is the Y coordinate, the second byte is what

texture to use for the sprite (defined in the OAM), the third byte is a bitfield for the sprite attributes, and the last byte is the X coordinate.

The palette ID and flip parameters are extracted from the sprite attribute using bitwise operations using bits 7, 6, 1, and 0, corresponding to the Y flip, X flip, and palette ID respectively. Then, the raw bitmap is extracted from the pattern table.

For X and Y flips, string slices are done on the column and row of the texture respectively. It then goes row-by-row and left-to-right and sets the pixel using the Pygame set_at function with a hardcoded color palette shown in Figure 48b. the coordinates are incremented but they start at the top left defined by the sprite's properties found in the OAM.

```
def renderBackground(self):
    # put these here for performance (dont loop this 30*32 times)
    base_nt = self.ctrl & 0b11
    pattern = self.patternTable[int(self.ctrlFlagGet('b'))]

    for y in range(30):
        for x in range(32):

            # apply scroll vars to renderer
            fixed_x = x * 8 + self.scrollx
            fixed_y = y * 8 + self.scrolly

            # find nametable to render
            nt_x = (fixed_x // 256) & 1
            nt_y = (fixed_y // 240) & 1

            nametable = base_nt ^ (nt_x | (nt_y << 1))
            base = 0x2000 + nametable * 0x400

            # fix tile coords
            tile_x = (fixed_x // 8) & 31
            tile_y = (fixed_y // 8) % 30

            # old logic
            # tile index from nametable
            address = base + tile_y * 32 + tile_x # base + y offset + x offset (like a 2d array access)
            tile_index = self.rambus.memoryReadPPU(address)

            # select pattern table
            tile = pattern[tile_index]

            # iterate per pixel for drawing
            # 8*8 pixel tiles
            screen_y = y * 8
            for row in tile:
                screen_x = x * 8

                for pixel in row:
                    # check if pixel is inside the screen
                    if 0 <= screen_x < 256 and 0 <= screen_y < 240:
                        color = bgColors[pixel] # faster to hardcode 4 colors than indexing attribute table

                        self.window.framebuffer.set_at((screen_x, screen_y), color)
                    screen_x += 1

                screen_y += 1
```

Figure 49: Background Rendering Code

The function shown above in lines 289 - 334 is responsible for rendering the background. It works a lot like the sprite rendering, but it contains extra logic for scrolling and using nametables instead of the OAM. The screen is 32 by 30 tiles wide, which is what the code uses for the nested loop. It renders it top to bottom, which is why it iterates through x all the way in each y iteration. First, it takes the base nametable and pattern table which is part of the ctrl flags in the PPU. Then, multiplies the scrolling registers by 8 in order to fully scroll the

whole screen. It uses a floor division with how far the screen is scrolled in order to determine which nametable to render, where it then offsets from the base nametable address of 0x2000 in order to render the actual proper nametable. This is done by getting the nametable index, and offsetting by 0x400 per nametable. It then accesses the nametable like a 2D array using the formula stated in the code to get the tile it needs to render. After all of that, it is able to render the tile similar to how the sprite renderer worked. However, an additional check is done so out-of-bounds pixels aren't rendered because doing so is unnecessary.

```
# frame render logic
def renderFrame(self):
    for event in pygame.event.get():
        # print(event)
        pass

    self.window.framebuffer.fill(0x400000) # blood red for the blood sweat and tears going into ts

    # RENDER YOUR GAME HERE
    self.renderBackground()
    self.renderSprites()

    scaled = pygame.transform.scale(
        self.window.framebuffer, self.window.screen.get_size()
    )

    self.window.screen.blit(scaled, (0, 0))

    # flip() the display to put your work on screen
    pygame.display.flip()
    self.lastFrame = time.time() # this line HAS to be last
```

Figure 50: Frame Rendering Code

The function shown in lines 239 - 259 above presents all the graphics to the screen. Most of it is basic Pygame code. First, it fills the screen with red. Then, it executes the aforementioned background and sprite render logic. Obviously, the background is rendered first before the sprites are layered on top of it. Then, it takes the framebuffer and scales it to the main window's size. Finally, the frame is finished and the screen is rendered. The current time of the computer at the end of the frame render is stored in order to calculate FPS.

```python
def fetch(self):
    # frame timing stuff
    cycle = self.cycles % 89342

    # handling vblank with NMI
    if (cycle) == 82181:
        self.statusFlagSet('v', True)
        # self.ctrlFlagSet('v', True)
        self.rambus.ppuInterrupt = True
    elif (cycle) == 0:
        # self.statusFlagSet('v', False)

        # rendering new frames
        delta = time.time() - self.lastFrame
        # dont sleep if fps is less than 60 LOL
        try:
            time.sleep(0.0167 - delta)
        except:
            pass
        self.renderFrame()
        # print(f"frame rendered {1/delta} fps")
        # print(self.oam)

    self.cycles += 1
```

Figure 51: PPU Fetch Function

The function shown above from lines 378 - 401 works as the PPU's cycle. Everything that happens in a PPU cycle happens in this code. First, it calculates which cycle within the frame the PPU is in, as the PPU does 89342 cycles per frame. Then, it sets the vblank flag and does a PPU non-maskable interrupt (NMI) if the PPU is currently in the vertical blank period (the time between frames). This executes an interrupt handler in the CPU that could be used for many things such as preparing for the next frame. When the PPU is on the first cycle of the frame, the time module in Python is used to determine how much time has elapsed between the last frame and the current frame. If it is less than 16ms (for 60fps), it sleeps the remaining time. If the value is negative, the code throws an exception that gets caught by the except block and does nothing to limit FPS. Then, it renders the next frame and increments the cycle count of the PPU.

Solution Design - NES Gamepad Implementation

The gamepad or controller is yet another device that's connected to the system bus in the NES. It is handled entirely by the pad.py file. Shown below is the constructor for the controller object:

```python
import pygame
class dmjoypad:
    def __init__(self, bus):
        bus.readHooks.append(self.ram_read)
        bus.writeHooks.append(self.ram_write)

        self.state = 0
        self.shift_register = 0
        self.strobe = 0

    def update_state(self):
        pressed = pygame.key.get_pressed()
```

Figure 52: Controller Constructor

Just like the PPU, the controller also uses read and write hooks that it appends into the list of hooks for the CPU's R/W in the bus object. It initializes the button states, a shift register (which ended up being unused in the code), and a strobe.

```python
def update_state(self):
    pressed = pygame.key.get_pressed()

    state = 0
    if pressed[pygame.K_z]: # a
        state |= 1 << 0
    if pressed[pygame.K_x]: # b
        state |= 1 << 1
    if pressed[pygame.K_c]: # select
        state |= 1 << 2
    if pressed[pygame.K_RETURN]: # start
        state |= 1 << 3
    if pressed[pygame.K_UP]:
        state |= 1 << 4
    if pressed[pygame.K_DOWN]:
        state |= 1 << 5
    if pressed[pygame.K_LEFT]:
        state |= 1 << 6
    if pressed[pygame.K_RIGHT]:
        state |= 1 << 7
    self.state = state
```

Figure 53: Button State Update

The function shown above found in lines 11 - 31 updates the state of the controller. It works by writing to the state bitmap according to which keys are pressed. It gets the current pressed keys on the keyboard using Pygame, and uses bitwise ORs in order to set the corresponding bit of the controller button. Bits 0 and 1 are for A and B, 2 and 3 and for select and start, and the remaining bits are for the D-pad.

```python
def ram_read(self, address):
    if address == 0x4016:
        self.update_state()

        bit = (self.state >> self.bit) & 1
        self.bit += 1

        # https://github.com/jameskmurphy/nes
        return (bit & 0b00011111) + (0x40 & 0b11100000)
    if address == 0x4017:
        return 0

def ram_write(self, address, value):
    if address == 0x4016:
        if value & 1:
            self.bit = 0
        return True
    if address == 0x4017:
        return True
```

Figure 54: Controller R/W Hooks

The code shown above found in lines 33 - 51 are the aforementioned hooks that are attached to the bus. The NES controller is simple, and only two addresses need to be emulated. If a read to 0x4016 is done, it returns one bit of the controller's state. This means that 8 reads to this address needs to be done to read the entire controller, as it only returns the property of one button per read. The return format is taken from the GitHub repo listed in the comment.

For writes, writing a 1 to the controller read address "strobes" or resets the controller to start reading from the first bit.

Reflection - What Have I Learned?

Making a NES emulator was certainly a very difficult feat for a junior programmer like me. When the final project was first announced, I didn't know what to make. However, I thought a NES emulator would be a very interesting project. I thought the deadline for the code was after the new year, but it was actually before finals. So, I had a rush in the final week where all I did was work on the emulator. It was really stressful, but I honestly enter a sort of zen state whenever I work on the emulator. Having reverse engineered lots of programs before, I am no stranger to assembly and how a CPU works. However, I have never written anything close to an interpreter, much less an emulator. It was really interesting seeing the hacks I had to do in Python in order to minimize code duplication, as opposed to having direct access to memory in a language like C++. I wanted to know how performant an emulator that I wrote myself in a slow language would be. When I first did the performance test without graphics

and the emulator outputted 20 FPS, I was laughing really hard with my friend who happened to be watching at the time. I learned a lot about code optimization, as removing certain things such as strings ended up making the program a lot faster. I also learned a lot about computer architecture, and how to make an interpreter. Fixing the bugs in the project was especially annoying. At first, I insisted on just testing on commercial games. However, after being pressed for time, I decided to just go the old-fashioned way and compare the trace of my program to a known good trace of a test ROM. My CPU worked after that. The PPU was hard to implement because I have never done anything rendering-related. I asked for help from my friend who had used Pygame before, and he showed me how to render pixel-per-pixel to the screen. Then, it turns out that making it resizable was as simple as creating a framebuffer that could stretch to the window. After this project, I want to make my own programming language and interpreter. With the knowledge from this project, it should be relatively easy. Overall, I really enjoyed doing this project, and I would do another similar project in a heartbeat.

References

NES Documentation:

- https://www.nesdev.org/NESDoc.pdf
- https://www.masswerk.at/6502/6502_instruction_set.html
- https://www.nesdev.org/wiki/Instruction_reference
- https://www.nesdev.org/wiki/CPU_addressing_modes
- https://forums.nesdev.org/viewtopic.php?t=6331
- https://www.nesdev.org/wiki/Stack
- https://www.nesdev.org/obelisk-6502-guide/reference.html#ROL
- https://bugzmanov.github.io/nes_ebook/chapter_4.html
- http://nickmass.com/images/nestest.nes
- https://forums.nesdev.org/viewtopic.php?t=25315
- https://www.nesdev.org/wiki/Status_flags
- http://www.6502.org/tutorials/6502opcodes.html#JMP
- https://bugzmanov.github.io/nes_ebook/chapter_6.html
- https://www.nesdev.org/wiki/PPU
- https://bugzmanov.github.io/nes_ebook/chapter_6_1.html#mirroring
- https://www.nesdev.org/wiki/Mirroring
- https://www.nesdev.org/wiki/PPU_OAM
- https://www.nesdev.org/wiki/NMI

- https://bugzmanov.github.io/nes_ebook/chapter_6_2.html
- https://www.nesdev.org/wiki/PPU_rendering#Line-by-line_timing
- https://github.com/jameskmurphy/nes/blob/main/nes/cycore/ppu.pyx
- https://www.nesdev.org/wiki/Standard_controller
- https://austinmorlan.com/posts/nes_rendering_overview/
- https://www.nesdev.org/wiki/PPU_nametables
- https://www.nesdev.org/wiki/PPU_pattern_tables
- https://www.nesdev.org/wiki/PPU_scrolling

Pygame Documentation:

- https://www.pygame.org/docs/ref/pygame.html
- https://www.pygame.org/docs/ref/display.html
- https://www.pygame.org/docs/ref/surface.html
- https://www.pygame.org/docs/ref/transform.html
- https://www.pygame.org/docs/ref/key.html

Zybooks Completion