A Modular Approach to 6502 Emulation [*]

M. J. Coppola, N. S. Shelby, L. E. Carleton
Computer Science 330
Siena College
Loudonville, NY, 12211

Abstract

We made a functioning emulator of the RP2A03 CPU that was in the Nintendo Entertainment System. The NES had a modified 6502 that swapped decimal mode for an Audio Processing Unit. In our implementation we included all instructions and addressing modes, the full expanse of memory, and an interface to see registers and sections of memory. This emulator was a resounding success for what it was set out to be and the stage is set to expand it into a full NES emulation.

1 Overview

The RP2A03 is the CPU found in the Nintendo Entertainment System (NES), launched October 18th, 1985 [4, 8]. This processor was based off the MOS Technology 6502, differences being a lack of a decimal mode and the inclusion of the Audio Processing Unit (APU) [4]. Our goal is to create a partial implementation of an emulator for this CPU designed to be easily extended to a simple NES emulator.

The emulator was designed with the NES in mind. That being said, we only emulate the behavior of the 6502, not precisely the actual hardware. This serves us a huge simplification in implementation. Official operations of the 6502 only need to be calculated, and not emulated. We opted in for calculating the output of the operations immediately, and then waiting the number of clock cycles that operation took to execute.

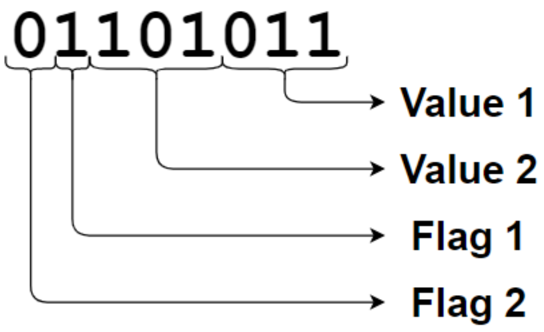
Although our emulator only features the CPU and memory, the modular nature of the implementation allows us to easily extend our software to fully feature emulation of all NES hardware.

[*]This work was completed in partial fulfillment of the final project requirement for Computer Science 330 at Siena College, Fall 2020.

2 Bitwise Operations

Bitwise operations are major aspect of making an efficient emulator. Suppose we have an 8-bit value. We typically express this value as a number, or possibly two hexidecimal digits. One way the 6502 uses 8-bit numbers is by defining parts of the binary expression as flags or smaller width values, shown in figure 1a. In emulation, it's important that we can extract information from this 8-bit number, as well as convert information we already have into information we can store into this number. To achieve this, we use bitwise operations.

To extract information from a binary number, we would first need to isolate that information using a bitmask. We can create a bitmask by creating a new 8-bit number that has 1's placed over the important digits, and 0's on digits that are not. With this mask, we can use a bitwise AND operation (&) and have a result that only includes the digits defined by the bitmask shown in



Value 2 Bitmask: $0011\ 1000_2$

$$\begin{array}{r} 0110\ 1011_2 \\ \underline{\&\ 0011\ 1000_2} \\ 0010\ 1000_2 \end{array}$$

$$0010\ 1000_2 \gg 3_{10} = 0000\ 0101_2 = 5_{10}$$

(a) A single binary value representing multiple values. (b) Using a bitmask to extract values from a binary number.

figure 1b [9].

In order to make this value useful to us, we may have to then use a bitwise shift-left (>>) to push the important digits to the least significant order of the digit. By doing this, the entirety of the resulting 8-bit value numerically represents the value stored in the section of the original 8-bit value [9].

A similar process is used for setting bits as well. We create a bitmask for the bits we want to set and or it with our value. To unset, we can & with the inverse (˜) of our mask. We can toggle/flip bits with a mask that is XOR'd (ˆ) with our value.

3 NES Hardware

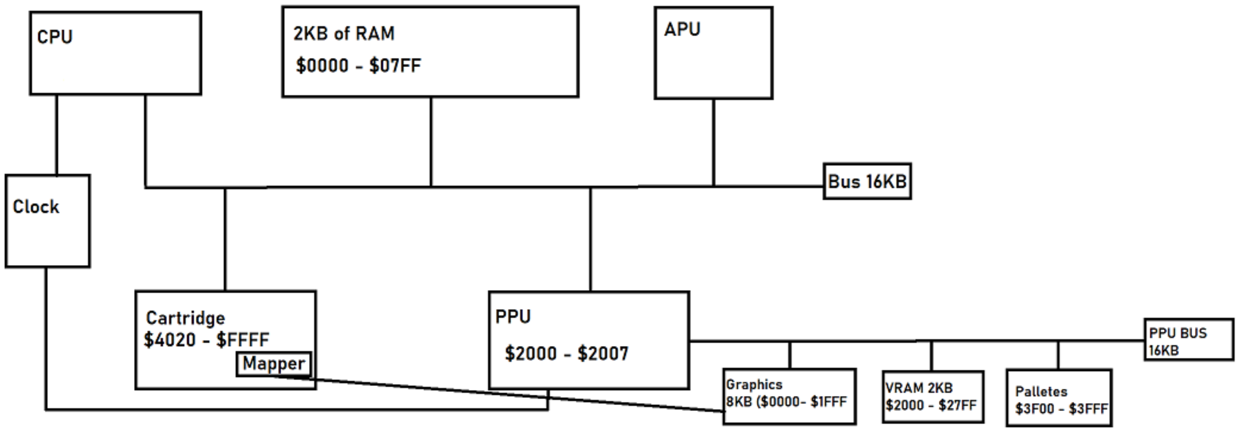


Figure 2: NES hardware bus diagram

The RP20A3 (6502) in the NES communicated to other devices on the machine using a 16-bit addressable hardware bus. The CPU can read and write information to any device connected to the bus [5]. Addresses \$0000 to \$00FF are mapped to the NES's 2KiB of internal memory. Parts of this memory has predefined purposes dictated by the CPU architecture. The zero page is mapped to \$0000-\$00FF and the stack is mapped to address \$0100 and onwards up to \$01FF [5]. In our implementation, we map the entirety of the bus to read/write memory. A full NES implementation would have other devices on the bus.

The PPU is one of the first components we would like to implement into the NES Emulation. The Picture Processing Unit will output the game's visuals to the connected display. It contains 10KiB of its own internal memory used to store the game's visuals. The PPU has its own bus to read and write to its set of memory. The graphics themselves are stored from address \$0000 to \$1FFF and contains the sprites for the game. The VRAM is a 2KiB section of memory addressed from \$2000 to \$27FF. This functions as RAM for the video production, storing large sections of a game, such as a map. The PPU's memory ends with the palettes and the OAM. The palettes being similar to a painter's palette, maintaining the colors being used in a given index. The Object Attribute Memory is dynamic memory that changes according to what is on the screen. The OAM contains the data for the objects being displayed, including the color and position of the sprites [3].

The PPU was still limited in its ability to fit more demanding visuals, but this could be expanded by mappers. There are hundreds of different mappers since these were made by developers and included on the game cartridge, this also means that there was not a limit to the number of mappers in existence. The emulation requires these mappers to function properly, so these would have to be implemented as a game was desired to be emulated. We would have to add them on a case by case basis. That does not mean that mappers were not reused, rather that the list was constantly added to, so certain mappers may extend emulation capabilities more than others. These mappers were stored in the NES game cartridge and function like a page table in memory. This allowed for more functionality in a game, while using less space in memory. The mapper would add to the PPU's memory depending on what needed to be referenced at a given time [7].

The 6502 features 6 registers. The first major register is the 1-byte wide Accumulator (A) register. This register is used with the ALU, and supports using the status register to describe its behavior, such as denoting if its value is overflowing, negative, zero, etc. The X and Y registers are used for different addressing modes. These registers were useful for looping when used with the increment and decrement instructions along side addressing with the value found in them. The program counter (PC) is a 2-byte wide register for tracking the current instruction in memory. The stack pointer register (S) is a 1-byte wide register that holds the manipulatable value that points to the stack portion of memory. The status (P) register hold 8 bytes (6 used) that store the current status flags of the CPU [1].

The 6502 has 3 index addressing modes. Addressing modes define where the CPU operations source operands from. Zero page indexing uses a 1-byte address to find a value found in the hard-coded zero page. We can offset the address to be read using the values in the X and Y register. Absolute addressing uses a 2-byte wide address to find memory outside of the zero page. Similarly, the X and Y registers can be used to offset this address. The 6502 also features indirect addressing, which is it's solution for simulating the function of pointers. The mode reads the value at the address found at the supplied address. Unlike the previous addressing modes, the X and Y offsets server different purposes from each other. The X register is used to offset the supplied address for the mode, while the Y register is used to offset the value found at the supplied address [2].

Other addressing modes are also used. Many instructions operate directly on the accumulator. Others address have immediate values from the program memory. Relative addressing is used by branching instructions to move to a new instruction in program memory from -128 to 128 bytes away from the branch's address.

As instructions execute, the CPU updates the status register to reflect 6 different boolean flags. The carry (C) flag turns on when an addition or subtraction carries or borrows a bit. This flag is also set if a logic shift pushes out a 1. The zero (Z) flag is turned on if an instruction results in a zero. The interrupt disable flag is used to enable and disable maskable CPU interrupts. The overflow flag (V) is set when overflow is detected during an addition or subtraction, and in a few other niche cases. The negative flag (N) is set when an instruction results in a negative value [**?**].

The first byte of an instruction is the opcode. Although only 56 opcodes are documented for the 6502, there are 256 addressable opcodes in the CPU. The undocumented opcodes, known as illegal opcodes, usually contain microcode the assist in the execution of the outward facing legal opcodes. An accurate emulator will simulate the microcode, therefore the legal opcodes utilizing the microcode, as well as executing them on the rare chance a developer uses one. Although a small subset of NES titles utilize these unofficial opcodes, we will not implement them in the interest of time at the cost of 8 titles [6].

4 Implementation

Our implementation begins with the bus. We decided to use a struct that can hold a bitfield of several device memories that the CPU can access. Although our bus is capable of this, we opted in for the bus to use memory in its entire address range as we will not be using any other devices. The bus has only three simple functions for reading, writing and resetting the values in memory. Our bus also can limit the range in which it can read and write data depending on what devices are connected to it. With this limiter in code, we can change the behavior of the read/write depending on the requested address range from the CPU.

To begin the CPU, we created an enumeration for all of the status flags of the CPU's status register. Each flag in the enumeration holds the number 1 bitshifted to the location of the flag in the status register. this allows us an easy way to refer to bitmasks as we set, unset, flip and test for status flags in the cpu.

We then created a structure to hold all of the CPU's information. The first component of the CPU is the registers. We used unsigned chars for each register, except for the program counter, which needs an unsigned short. We added a get flag, and a set flag function to be able to easily test and write to the status register in this CPU structure. In our structure, we include 5 helper values. We keep variables to store the current observed address, the relative address used by branching instructions, the current opcode, how many cycles are left needed to complete the current instruction and the fetched value from the addressing mode.

We also flesh out in put header file functions for our 12 addressing modes and 56 operations. We add an extra opcode named XXX that captures all illegal opcodes. This serves as a no operation (NOP). We then added a clock cycle function that performs 1 clock cycle in the CPU.

We created a structure the represents an instruction. This contains the name, the opcode, the addressing mode and the number of cycles each type of instruction takes. With this structure, we can make a lookup table that associates an opcode, structure and number of cycles each possible instruction takes based off the Rockwell instruction table for the 6502 [10].

Our addressing modes return the number of extra clock cycles the instruction takes, and the opcodes return whether the operation takes an extra clock cycle. Using this information, the clock function can use the lookup table to take an instruction, lookup the opcode in the lookup table, dereference and call the functions found in the instruction structure it has found, and use those function return values to find out how many clock cycles the operation took to execute.

5 Conclusion

Our 6502 emulator was overall a success, featuring all addressing modes and all opcodes. We also created a GUI frontend for viewing the memory, CPU status and current opcode. The design of the emulator is extremely modular and can be very easily extended into a fully functional NES capable of picture, sound and controller input. The next step to continuing this would be to adjust the bus to divide its addresses up into the different devices, and reading ROM data from ines format files and to use a mapper to place the program data onto the system bus. After this, we would need to do research on implementing the PPU, APU and controller input.

References

[1] Cpu registers. `http://wiki.nesdev.com/w/index.php/CPU_registers`, 2015. Accessed: 2020-11-20.

[2] Cpu addressing modes. `http://wiki.nesdev.com/w/index.php/CPU_addressing_modes`, 2018. Accessed: 2020-11-20.

[3] Ppu. `https://wiki.nesdev.com/w/index.php/PPU`, 2018. Accessed: 2020-11-20.

[4] Cpu. `wiki.nesdev.com/w/index.php/CPU`, 2019. Accessed: 2020-11-17.

[5] Cpu memory map. `http://wiki.nesdev.com/w/index.php/CPU_memory_map`, 2019. Accessed: 2020-11-20.

[6] Cpu unofficial opcodes. `https://wiki.nesdev.com/w/index.php/CPU_unofficial_opcodes`, 2020. Accessed: 2020-11-20.

[7] Mapper. `https://wiki.nesdev.com/w/index.php/Mapper`, 2020. Accessed: 2020-11-20.

[8] Nintendo entertainment system. `https://en.wikipedia.org/wiki/Nintendo_Entertainment_System`, 2020. Accessed: 2020-11-17.

[9] O. Lawlor. Bits and bitwise operators. `https://www.cs.uaf.edu/courses/cs301/2014-fall/notes/bits-bitwise/`, 2014. Accessed: 2020-11-17.

[10] Rockwell Automation, Newport Beach, CA. *R650X and R651X Microprocessors (CPU)*, June 1987. Document No. 29000D39.