6502 Emulator Progress Report

By: Nicklaus Shelby, Michael Coppola, and Lauren Carleton

We have done some research on a 6502 and its functions. As a byproduct of this we have also extended our knowledge of binary transformations and operations. Our group has not reached a point of programming, but we expect to be in the coming days. This is a summary of our current research.

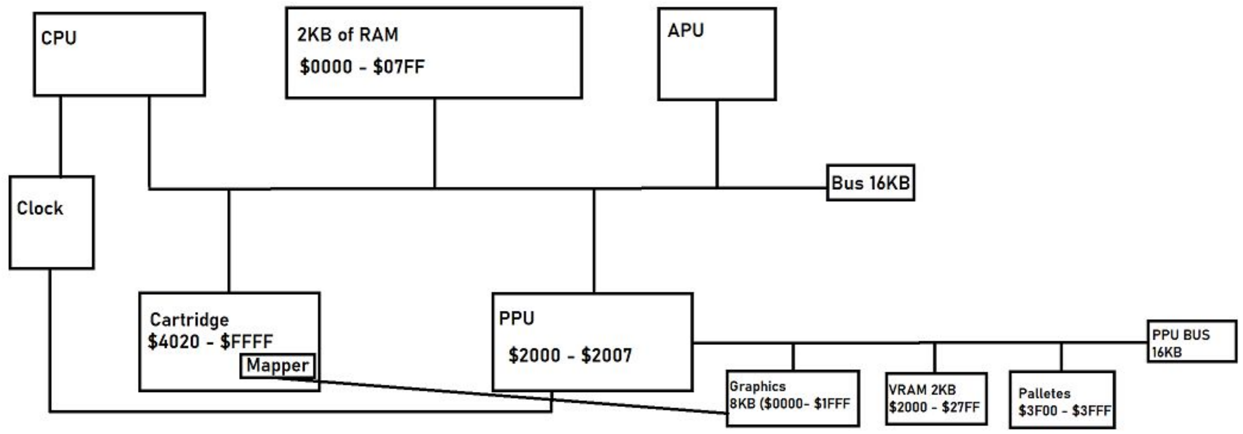
We have found techniques that will allow us to manipulate and retrieve certain bits in a larger number. A bitwise *and* (&) can be used to find what bits are true/on (1) in a certain group. This occurs by taking a binary number, e.g. $0001\ 1000_2$, and using a bitwise and on that and the bytes being checked, e.g. $1001\ 0110_2$, which would result in $0001\ 0000_2$. This gave us only the bits that were true in the area we were "searching for" - the fourth and fifth bits. This technique applies to more than the bitwise *and*. It can also be used with bitwise *or*, *not*, and *xor* operations. The mask - in this example it was the $0001\ 1000_2$ - allows us to directly interact with only certain bits out of a number or instruction.

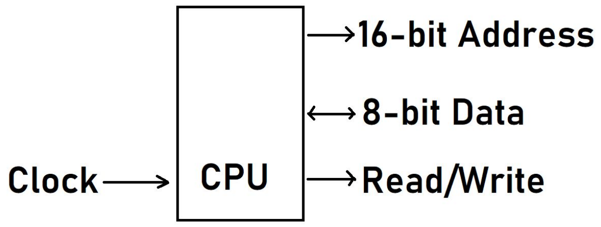
We also researched unions and bitfields in C. This functionality - in the context of bits and bytes as instructions in C - allows us to create a data type, e.g. instruction, which might be a 3 byte field. This instruction will also contain variables that are a sort of inner field, broken into parts of the instruction, like opcode and address. The colon allows us to define the width of the memory in bits for said inner fields.

```
union 3byteInstruction{
    struct {
        unsigned int opcode : 8;
        unsigned int address : 16;
    };
    unsigned int instruction;
};
```

Leading into the portion of the project that is directly based on the 6502 and its hardware, we have done research on the actual hardware and its communication with other components. The Clock consistently ticks and causes a cycle. While this is happening the CPU is carrying out instructions and will send or receive data from the bus. This 16-bit addressable bus will transfer data between the different components in the system. To start our system will only consist of the memory, the 6502, and the bus. As this project stems from, and hopes to lead into, a functioning NES (Nintendo Entertainment System) emulator, we would also like to implement a Picture

Processing Unit, an Audio Processing Unit, mappers, and obviously a cartridge element to read game data from. Mappers are the more interesting portion of this. These act like page tables in the system in that they include more than what can fit in RAM, so when it is needed it will physically swap what is in RAM with what is on the cartridge and now be able to use that data. See the two diagrams below for the hardware components.





The 6502 uses only 6 core registers. These are: A, X, Y, stkp, pc, and status. These are the Accumulator, 2 working registers, stack pointer, the program counter, and the status register, respectively. As we researched this we also found that the instructions are not all the same length in text nor are they all the same length in time. Some of these instructions take more clock ticks to execute and others take more input. There are 56 legal instructions, but these can vary in input length by a matter of bytes. Example: For LDA (LDA load accumulator) there is LDA $41 - a 2-byte instruction vs. LDA $0105 - a 3-byte instruction. This means that all instructions must be emulated with proper functionality, as well as addressing and cycles.

These registers do nicely fit into a 16x16 table, with some empty spots - these empty spots being illegal opcodes. These illegal opcodes do still do something with the CPU, but they are not necessary or intended for use and we will not be using them for our emulator. Now that we have our understanding of the architecture and the CPU's instructions, we just need to emulate it. This is broken down into 5 steps. First: Read the first byte at the program counter. Second: Find the addressing mode for the opcode and how many cycles are needed. Third: Read the first three bytes as instruction input. Fourth: Execute the instruction. Fifth: Wait the proper number of cycles and then repeat this process until the program is complete.