Making your first NES emulator

100th_Coin
5 de setembro

Pular navegação

Objective:

- This emulator should run Super Mario Bros. without any issues.

  We're making an NTSC NES, rather than a PAL NES.

Requirements:

- An IDE of your choice for programming.
- Your own copy of Super Mario Bros.

It's just me, you, and 6502.

The Nintendo Entertainment System runs on a 6502 CPU, so a natural starting point would be to become familiar with the language were going to be emulating.

As a fair warning, we might sacrifice accuracy in favor of making the emulator easier to write. This is for your *first* emulator after all. If you are specifically reading my guide because of my work on the TriCNES emulator, I'm going to state right now that we are not going to be making the TriCNES emulator with this guide, and accuracy is not the goal here.

There is no singular *correct* way to make a NES emulator. In this guide, I'm going to be providing example code in C#, though you do not need to follow exactly what I am doing line by line. In fact, even if you are writing your emulator in the same language I am, I encourage you to imagine your own solutions to the problems I explain.

Super Mario Bros. is surprisingly one of the hardest NES games to emulate. Naturally, we're only going to be running a couple test ROMs I whipped up and then Super Mario Bros. until we get this emulator functioning.

Let's set up everything required for some simple 6502 emulation. Open up the IDE of your choice. (I'll be using Visual Studio Winforms.) We'll need a 16-bit value for the Program Counter, and three 8-bit values for the A, X, and Y registers.

```
ushort ProgramCounter;
byte A;
byte X;
byte Y;
```

Let's get a brief overview of those variables we just created.

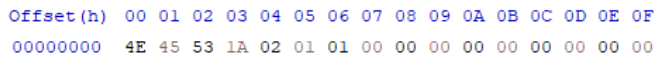
6502 assembly is processed one byte at a time. It's the job of the *Program Counter* to keep track of which byte should be processed next.

The *A Register* is used for math and bitwise operations. The *X and Y Registers* are used for indexing and counting. These three registers can also be used for reading and writing to memory, or to throw in some 6502 lingo, "Loading" and "Storing".

These four variables won't cover everything, but it will actually be enough to start figuring out how our emulator will work.

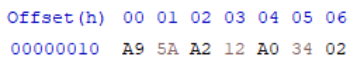
What's in a ROM?

At the bottom of this post, you will find a .zip file containing a series of .nes files. Let's take a look at *1_Example.nes*. We can use a Hex Editor, such as HxD to example the contents of the ROM.



The 16 bytes at the start of the ROM, forming the ines header.

These 16 bytes at the beginning of the ROM form the "ines header". We'll touch more on the header much later, but for now we're interested in the remaining contents of this ROM. 1_Example.nes is a very simple program. It contains a load instruction for the A, X, and Y registers, before halting. But of course, these instructions are also just bytes:



The bytes forming the instructions.

Let's talk about the **LDA** instruction.

This is how the program updates the value of our 8-bit value, A, which we created for our emulator. For instance, the 6052 assembly instruction **LDA #5** would essentially run **A = 5;**

This concept also applies to the LDX and LDY instructions, setting the value of X and Y respectively.

But looking at the ROM, all we can see are various hexadecimal numbers. How does this translate to assembly code? Let's take a look!

The instruction **LDA #5** would be written as **A9 05**.

The instruction **LDA #8** would be written as **A9 08**.

So **A9** is the byte representing the **LDA** instruction, and the byte that follows it represents the value we wish to set the A register. For instance, **A9 5A** would set the A register equal to the hexadecimal value $5A. In 6502 Assembly, the "$" symbol is used to indicate a number is being written in hexadecimal. Let's dissect the instruction "**LDA #$5A**" piece by piece. "LDA" stands for "Load A". The "#" symbol means we're loading A with an "Immediate" value. This is called the addressing mode, which is used to distinguish between wanting to load A with, for instance, the number 5 (**LDA #5**) rather than reading from address 5 in memory (**LDA 5**). Back to the "**LDA #$5A**" example, as explained above the "$" means the following number is in hexadecimal, so the value "**$5A**" is the resulting value for the A register after this instruction is processed.

Let's walk through the first 7 bytes of 1_Example.nes:

"A9 5A" - **LDA #$5A** ; This sets A = $5A

"A2 12" - **LDX #$12** ; This sets X = $12

"A0 34" - **LDY #$34** ; This sets Y = $34

"02" - **HLT** ; This halts the CPU, effectively ending the program.

Our current goal will be to run this program. First things first, we need our emulator to be able to read from our ROM.

byte Read(ushort Address)

The 6502 CPU has a 16-bit address space, which means there are only 65,536 bytes we have access to. If you are unfamiliar with hexadecimal, it might seem tedious now, but I'm going to be using hexadecimal values from here on out. To restate this paragraphs opening sentence, there are $10000 bytes in this address space, ranging from address $0000 to address $FFFF.

 ur "Read()" function is not just for reading from the ROM, but also reading from the console's RAM. Address $0000 to address $07FF is the entirety of the console's RAM. The ROM will be located from address $8000 to $FFFF.

Let's add two arrays of bytes to our emulator to represent the RAM and the ROM.

```
byte[] RAM = new byte[0x800];
byte[] ROM = new byte[0x8000];

0 references
byte Read(ushort Address)
{
```

Keep in mind, not all ROMs are $8000 bytes in size. For now though, let's assume they are.

Reading from RAM is incredibly easy! If the target address is less than 0x800, you can simply return index *Address* of the RAM array.

```
if(Address < 0x800)
{
    return RAM[Address];
}
```

If you are aiming for a very-accurate emulator, check this out! When reading from address $800 through $1FFF, the CPU still reads from RAM, but only using the lower eleven bits for the address. (After all, there are still only $800 bytes worth of RAM.) Therefore, reading from address $0800 is the same thing as reading from address $0000. Likewise, reading from address $1000 and $1800 also read address $0000. These are referred to as "mirrors" of address $0000. Likewise, $1FFF is a mirror of $07FF, and so on. As a bonus optional challenge, modify the Read() function to implement RAM Mirroring!

Now to read from ROM.

```
if(Address >= 0x8000)
{
    return ROM[Address-0x8000];
}
```

Just so we're clear, the ines header of the ROM is not actually part of the program data, (and doesn't even exist on a physical cartridge!) so we're going to need to ignore those bytes when reading from the ROM.

Also keep in mind, the code shown above should be placeholder. We're going to be expanding this Read() function a lot when we implement mapper chips.

. ower On

When you start the NES, the Program Counter (hereafter referred to as "PC") needs to be moved to the start of the program. A programmer making a NES ROM could choose any address for the start of their program, so how does the NES determine where to move the PC? The Reset Vector! The final 6 bytes in a ROM determine where to move the PC in the event of various interrupts (which we will touch on later) including where to move the PC during power on and reset.

```
Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00008000   FF FF FF FF FF FF FF FF FF FF FF FF 00 80 FF FF
```

The reset vector is a pair of bytes pointing to a 16-bit address. These bytes are in "Little Endian" order, which means the "Low Byte" (or the least-significant 8-bits) is read first, with the "High Byte" being read second. So the bytes $00 and $80 would be a pointer to address $8000.

Let's make a function for our emulator called Reset(). Let's load the ROM from a file path, and then remove the header. We won't be using this header for a while, but it's not a bad idea to create an array dedicated to those 16 bytes.

```
public void Reset()
{
    byte[] HeaderedROM = File.ReadAllBytes(filepath);
    Array.Copy(HeaderedROM, 0x10, ROM, 0, 0x8000);
    Array.Copy(HeaderedROM, Header, 0x10);
```

And now we can use the Read() function to determine where to move the PC!

```
    byte PCL = Read(0xFFFC); // PC low byte
    byte PCH = Read(0xFFFD); // PC high byte
    ProgramCounter = (ushort)((PCH * 0x100) + PCL);
}
```

Now let's test it out! Let's set up a breakpoint after the last line of the Reset() function. I'm using Visual Studio Winforms, so I just set up a button on the form to click, which will then run the code setting up the emulator, the filepath, and then running Reset().

```
1 reference
private void button1_Click(object sender, EventArgs e)
{
    Emulator emulator = new Emulator();

    emulator.filepath = @"C:\Users\ Your file path here \1_Example.nes";
    emulator.Reset();
}
```

And then I run the program for my emulator:

```
ProgramCounter = (ushort)((PCH * 0x100) + PCL);
≤ 1ms elapsed   ProgramCounter 0x8000
```

Using Visual Studio's breakpoints, I can see that the "ProgramCounter" variable has a value of 0x8000 after running Reset()

Pular navegação

Running 1_Example.nes

Now that we've powered up the console, the ROM array is populated with the bytes for 1_Example.nes, and the PC is moved to the correct address, it's time to emulate the CPU!

(You might need to scroll a couple hundred feet downwards to the bottom of the post to download the TestRoms.zip file containing the test roms)

It was a while ago, so let's recap. The console will read the byte at the location of the PC. In this case, the byte at address $8000 is **A9**. That byte corresponds to the **LDA Immediate** instruction. Then the following byte will be read, **5A**. That's the value that will be loaded into the A register. So some of these bytes are "Opcodes", which determine what instruction to run, and some of these bytes are "Operands" which determine how or where to run this instruction.

We're going to make the CPU run in a loop infinitely, or until a "Halt instruction" occurs. You could make a while(true) loop, or whatever you would prefer. Here's an example:

```
bool CPU_Halted = false;
1 reference
void Run()
{
    while (!CPU_Halted)
    {
        Emulate_CPU();
    }
}

1 reference
void Emulate_CPU()
{
```

Now, there are 2 ways to proceed. The accurate way (difficult) or the easy way (easy). Since we're aiming to make our *first* emulator here, let's take the easy path.

While the accurate way would run only a single CPU cycle per iteration of the Emulate_CPU() function, we're going to opt to make the Emulate_CPU() function take care of an entire instruction in each iteration.

Step 1: read the opcode of the instruction to emulate.

This is what the PC is all about. We want to read from memory at the location of the PC to determine the opcode. After performing the read, don't forget to increment the PC for the next read!

```
byte opcode = Read(ProgramCounter);
ProgramCounter++;
```


ow we need to make the core of the CPU. It's pretty much just going to be a really big switch statement. Keep in mind, there are 256 possible values for a byte, so there are 256 different opcodes. For now, let's implement **LDA Immediate** (A9), **LDX Immediate** (A2), **LDY Immediate** (A0), and **HLT** (02).

```
switch(opcode)
{
    case 0x02: //HTL
        break;
    case 0xA0: //LDY Immediate
        break;
    case 0xA2: //LDX Immediate
        break;
    case 0xA9: //LDA Immediate
        break;

    default:   // Unknown opcode?
        // I suggest putting a breakpoint here.
        // It can tell you what you need to implement next.
        break;
}
```

The humble beginnings of a thousand-line switch statement.

Let's implement **LDA Immediate** first. Keep in mind, we incremented the PC after reading the opcode, so if we read from the new PC's address, we'll get the operand of our LDA instruction. Here's an incredibly simple implementation:

```
case 0xA9: //LDA Immediate
    A = Read(ProgramCounter);
    ProgramCounter++;
    break;
```

As a heads up, once we start implementing the *Picture Processing Unit* (which won't be for a while) we're going to need to keep track of how long these instructions take to process in

terms of "CPU Cycles".

For instance, if we were going down the "accurate route", we would first run the CPU cycle reading the opcode, then the CPU cycle reading the operand. In total, **LDA Immediate** takes those 2 cycles, and then the next instruction's opcode is read, and so on. This is important, because the PPU runs independently of the CPU, and also three times as fast. When you start asking "How many instructions occur before the end of the frame?" it all boils down to how many PPU cycles have occurred.

Once that kind of timing is important, there are multiple ways to implement it, all with varying degrees of accuracy and effort required to program it.

One of the easiest ways to handle this would be to simply have a value for each opcode representing how many CPU cycles it took. Exhibit A:


```
case 0xA9: //LDA Immediate
    A = Read(ProgramCounter);
    ProgramCounter++;
    cycles = 2;
    break;
```

Pro: Really easy to implement.Con: There will be issues with code requiring precise timing with the PPU.

With this method, after each instruction we can then run **cycles*3** iterations of the Emulate_PPU() function, whenever we get around to programming that.

Keep in mind, this is just one solution of many. Heck, you could do *this*:

```
void Emulate_CPU()
{
    byte opcode = Read(ProgramCounter);
    ProgramCounter++;
    Emulate_PPU();
    Emulate_PPU();
    Emulate_PPU();
    switch (opcode)
    {
        case 0xA9: //LDA Immediate
            A = Read(ProgramCounter);
            ProgramCounter++;
            Emulate_PPU();
            Emulate_PPU();
            Emulate_PPU();
            break;
```

But I wouldn't recommend it, ha!

And hey, if you are interested in the "accurate route" I mentioned earlier, (running each CPU cycle individually) though it might be more difficult to implement, it could look something like this:

```
int cycle = 0;
byte opcode;
1 reference
void Emulate_CPU()
{
    if (cycle == 0)
    {
        opcode = Read(ProgramCounter);
        ProgramCounter++;
        cycle++;
    }
    else
    {
        switch (opcode)
        {
            case 0xA9: //LDA Immediate
                A = Read(ProgramCounter);
                ProgramCounter++;
                cycle = 0; // Ready to read the next opcode!
                break;
```

This looks pretty simple for a 2-cycle-long instruction, but it gets a lot busier for longer instructions.

There's a million ways to go about doing this, but for the purposes of this guide, I'll assume you are going with the easiest way to count cycles. Literally defining an instruction's cycle count with an integer during that instruction's logic, as shown in "Exhibit A".

LDX and LDY are pretty much the same as LDA, but substitute A with X or Y.

For the HLT instruction, I'll just set a variable called CPU_Halted to true, which will be used to exit the infinite loop.

Make sure your Reset() routine ends bu running the function with the infinite loop, set up a breakpoint after the loop, and run the code! If it worked, A should equal $5A, X should equal $12, and Y should equal $34.

The resulting values of the registers after running the ROM.

void Write(ushort Address, byte Value)

Up next, writing to memory!

This will also have a myriad of changes once we start worrying about mappers, but for now, let's just allow writing to RAM. And if you are going above and beyond, include the mirrors of RAM just like you did with Read().

The idea here being, write "Value" to index "Address" of the RAM array.

ROM, or Read-Only-Memory is Read-Only, implying that you cannot write there. Due to this, let's not allow the Write() function to update the ROM array.

Now let's set up some *store* instructions. Unlike the "Immediate" addressed load instructions we already set up, store instructions must point to an address. There are the "Absolute" addressed 16-bit store instructions, and the "Zero Page" addressed 8-bit store instructions.

Opcode $85 is "**STA Zero Page**", and just like the LDA instructions we set up earlier, it has a single operand. For instance, **85 55** would be **STA <$55**, which writes the current value of the A register into address $0055. (The "<" symbol is used to represent a low byte. In this context, implying the high byte will be zero.) **85 00** would be **STA <$00**, writing to address $0000, and so on.

Opcode $8D is "**STA Absolute**", and this instruction has two operands! For example, remember when we wrote the Reset() function and we needed to read two bytes to determine where to

ove the PC? In this case, we're reading two bytes to determine where to write. And just like the case during Reset(), these bytes are in little endian order. So the triplet of bytes, **8D 50 05** would be **STA $0550**. Keep in mind, you can still write **8D 00 00**, using three bytes to write to address $0000 instead of using the zero page version of the instruction, though it just takes more bytes and CPU cycles to do so.

Here's an example of how one might implement these instructions:

```
case 0x85: //STA Zero Page
    byte Temp = Read(ProgramCounter);
    ProgramCounter++;
    Write(Temp, A);
    cycles = 3;
    break;

case 0x8D: //STA Absolute
    byte Temp_Low = Read(ProgramCounter);
    ProgramCounter++;
    byte Temp_High = Read(ProgramCounter);
    ProgramCounter++;
    Write((ushort)(Temp_High*256+ Temp_Low), A);
    cycles = 4;
    break;
```

Likewise, opcode $A5 and $AD are "LDA Zero Page" and "LDA Absolute" respectively. Go ahead and implement both of those opcodes as well, in preparation for running 2_ReadWrite.nes. They work in the same way as STA Zero Page or STA Absolute, but they read from that target address and put the result in the A register.

Running 2_ReadWrite.nes

This ROM requires the implementation of STA Zero Page and Absolute, as well as the LDA counterparts. The end result will be a write to address $0000, $0001, $0002, and $0550, with the values $5A, $5A, $80, and $80.

It also starts at address $8100 instead of $8000, just to confirm that you implemented that part of Reset() correctly.

Before we move on, I also recommend implementing the STX and STY instructions, which store X and Y respectively.

STX Zero Page: Opcode $86

STX Absolute: Opcode $8E

STY Zero Page: Opcode $84

STY Absolute: Opcode $8C

The Status Flags

Before we get carried away implementing dozens of opcodes, let's start implementing the status flags. The simplest way to go about doing this is to create a boolean for the "Carry Flag", "Zero Flag", "Interrupt-Disable Flag", "Overflow Flag", and "Negative Flag".

```
bool flag_Carry;
bool flag_Zero;
bool flag_InterruptDisable;
bool flag_Decimal;
bool flag_Overflow;
bool flag_Negative;
```

Not to be confused with the Six Flags amusement park.

Before we continue, I recommend adding "flag_InterruptDisable = true;" to your Reset() function, as that flag is enabled by default.

We're going to be ignoring the Overflow Flag and Interrupt-Disable flag for a while. (And the Decimal flag can be completely ignored on the NES!) The easiest flag to understand is the "Zero Flag"

If the result of an operation is zero then the Zero Flag is set. So far, the only instructions we have that can affect this is our Load instructions, but we're going to be implementing some basic arithmetic down the road. Basically, if the value read was zero, set the zero flag. Otherwise, clear it.

```
case 0xA9: //LDA Immediate
    A = Read(ProgramCounter);
    flag_Zero = A == 0;
```

Similar to the Zero Flag, the Negative Flag is also set by Load instructions. With an unsigned 8-bit value, we have the numbers 0 to 255, but with a *signed* 8-bit value, we have the numbers -128 to 127. Of course, we still only have 8 bits, so for the purposes of the Negative Flag, anything greater than 127 is considered negative. So just like the Zero Flag, if the value read is greater than 127, set the Negative Flag. Otherwise, Clear it.

```
case 0xA9: //LDA Immediate
    A = Read(ProgramCounter);
    flag_Zero = A == 0;
    flag_Negative = A > 127;
```

Implement these two flags in the load instructions (and not the *store* instructions). Let's see what these flags are for!

Branches

In a higher level programming language, we have "If statements", but in 6502 Assembly, we have Branches.


```
If (Condition)
{
   //A
}
//B
```

```
    LDA <$00
    BNE Skip
    ;A
Skip:
    ;B
```

Comparing an if statement to a 6502 Branch.

On the left, we have an if statement. If a condition is met, it runs the code inside the curly braces, so "A" is executed, and then the code continues to run, flowing into "B". However, if the condition is *not* met, then the code inside the curly braces is skipped, executing "B" without executing "A".

On the right, we have a Branch. "**BNE**" stands for "Branch on Not Equal", which means the branch will be taken if the zero flag is not true. So if **LDA <$00** reads the value $00, setting the zero flag, the branch will not be taken, so "A" is executed, and then the code continues to run, flowing into "B". Likewise, if a non-zero value was read, clearing the zero flag, the branch *is* taken, so the PC is immediately moved to the label "Skip:", executing "B" without executing "A".

Let's see how branches look in byte-form.

For instance, **BNE** is opcode $D0. The instruction has a single operand, indicating a signed 8-bit value for how many bytes to move the PC. For instance **D0 05** would be a BNE instruction moving the PC over 5 additional bytes (after reading the operand and incrementing the PC). Since the operand is a *signed* value, this means you can also branch backwards, creating loops. For instance, consider the bytes **D0 FE**. Read the opcode, $D0, **BNE** (increment PC). Read the operand, **$FE** (increment PC). move the PC back two bytes, and it is once again at the BNE instruction.

With a general understanding of how BNE works, let's look at all the branches:

BPL (opcode $10): Branch on Plus. The branch is taken if the negative flag is cleared.

BMI (opcode $30): Branch on Minus. The branch is taken if the negative flag is set.

BVC (opcode $50): Branch on Overflow Clear. The branch is taken if the overflow flag is cleared.

BVS (opcode $70): Branch on Overflow Set. The branch is taken if the overflow flag is set.

BCC (opcode $90): Branch on Carry Clear. The branch is taken if the carry flag is cleared.

BCS (opcode $B0): Branch on Carry Set. The branch is taken if the carry flag is set.

BNE (opcode $D0): Branch on Not Equal. The branch is taken if the zero flag is cleared.

Pular navegação EQ (opcode $F0): Branch on Equal. The branch is taken if the zero flag is set.

And let's see an example of how one might implement a branch opcode for their emulator:

```
case 0xD0: //BNE
    Temp = Read(ProgramCounter);
    ProgramCounter++;
    if(!flag_Zero)
    {
        int signedVal = Temp;
        if(signedVal > 127)
        {
            signedVal -= 256; // range from -128 to 127.
        }
        ProgramCounter = (ushort)(ProgramCounter+signedVal);
        cycles = 3;
    }
    else
    {
        cycles = 2;
    }
    break;
```

Note: This implementation ignores an edge case where the instruction takes a fourth CPU cycle if the taken branch changes the PC high byte.

In this case, you can see how the branch would move the PC if the zero flag was cleared (taking 3 cycles), or do nothing if the zero flag was set (taking only 2 cycles). For those of you looking to make your emulator extra accurate, can you find a way to verify if this branch would modify the high byte of the program counter, thus requiring one more CPU cycle?

Running 3_Branches.nes

This ROM is a "maze" of branches. Basically, if your **LDA Immediate** instruction sets the Zero and Negative flags correctly, then the PC should branch around a series of **HLT** instructions, before writing $01 to address $0000, and halting on address $8017. In addition to LDA Immediate correctly setting those flags, you need to correctly implement **BPL**, **BMI**, **BNE**, and **BEQ** to run this.

The Stack

The NES has a 256 byte long "First-In Last-Out" Stack, which can be used to store values temporarily. If you've never worked with a stack before, it can be helpful to imagine it as a physical stack of paper, with each sheet holding a single byte.

Suppose you are trying to clean your house, but you need to buy a new broom. So you write down "Clean the house" while getting in your car to buy a broom. But oh no! Your car is low on gas! So you write down "Buy broom", then head to the gas station. You make it to the gas station, but someone steals you wallet! You write down "Buy gas" then chase after the wallet thief. After apprehending the wallet thief, he reveals himself to be the heir of a whimsical leprechaun fortune! Uninterested in that side quest, you take your wallet and look at the stack of notes you took. Now, in this order, you "Buy gas", "Buy broom" and then finally "Clean the house".

The 6502 CPU's Stack works in a similar way, only instead of a series of fun tasks, it writes down bytes, which are then taken off in the reverse of the order in which they were added. These two actions, adding to the stack and removing from it, at referred to as "pushing" and "pulling" respectively. A more modern programming language might call "pulling off the stack" something else, like "popping off the stack", but this is 6502 Assembly, so we say "Pulling".

And last but certainly not least, the key to all of this is the *Stack Pointer*. The Stack Pointer keeps track of how full the stack is. More specifically, if you push a new byte to the stack, the value pushed is placed in memory at address $0100 + the Stack Pointer, and then the stack pointer is decremented (subtracted by 1). Then if you pull off the stack, the Stack Pointer is first incremented (add 1 to it) and then the value at address $0100 + the Stack Pointer is read.

For instance, suppose the stack pointer is currently $80. Running a **PHA** instruction (Push A. Opcode $48) will store the value of the A register at address $0180, and then decrement the Stack Pointer to a value of $7F. Then, **PLA** (Pull A. Opcode $68) would increment the stack pointer back to $80, then read address $0180, putting the result into the A register.

As an additional bit of knowledge, the Stack Pointer is limited to 8 bits, so if it's currently $00, and you run **PHA**, the stack pointer will underflow to $FF. And likewise, **PLA** would then overflow it back to $00.

Oh, and it won't make sense just yet, but when you power on the console, the stack pointer is automatically decremented three times. This can be set up in Reset(), where you can simply initialize the stack pointer to $FD for now. It will make sense many chapters from now why this is the case, but you'll just need to take my word for it right now.

You might want to consider making functions for Push() and Pull():

```
1 reference
void Push(byte Value)
{
    // Store to the stack, and decrement the stack pointer.
    Write((ushort)(0x100 + stackPointer), Value);
    stackPointer--;
}

1 reference
byte Pull()
{
    // Increment the stack pointer, and read from the stack.
    stackPointer++;
    Temp = Read((ushort)(0x100 + stackPointer));
    return Temp;
}
```


And then your **PHA** and **PLA** instructions might look something like this:

```
case 0x48: //PHA
    Push(A);
    cycles = 3;
    break;

case 0x68: //PLA
    A = Pull();
    flag_Zero = A == 0;
    flag_Negative = A >= 0x80;
    cycles = 4;
    break;
```

Don't forget to set up the Zero and Negative flags with the PLA instruction!

Now so far, these two instructions **PHA** and **PLA** seem pretty boring. Like, this whole system of a stack seems unnecessarily over-engineered. Let's take a look at a more important use of the stack. The **JSR** and **RTS** instructions!

JSR and RTS

JSR, or *Jump to Subroutine* (opcode $20), does 2 things. It pushes a return address to the stack, and then moves the PC to a 16-bit address. **RTS**, or *Return from Subroutine* (Opcode $60)pulls the return address off the stack, and moves the PC there.

So for instance, suppose the PC is at address $8000 and it executes **JSR $9000** (or, written as bytes, **20 00 90**). This instruction will push $80 (the PC high byte), then push $02 (the PC low byte), and then move the PC to address $9000. Notably, the return address is $8002, except since **JSR** is a three-byte instruction, you might assume the return address should be $8003. It's a bit unintuitive, but after the **RTS** instruction moves the PC back to address $8002, the PC is incremented. That way the following CPU would read $8003 as the opcode.

Here's an example of how one might implement these:

```
case 0x20: //JSR
    Temp_Low = Read(ProgramCounter);
    ProgramCounter++;
    Temp_High = Read(ProgramCounter);
    Push((byte)(ProgramCounter/256)); // push PCH
    Push((byte)ProgramCounter);       // push PCL
    ProgramCounter = (ushort)(Temp_High*256 + Temp_Low);
    cycles = 6;
    break;

case 0x60: //RTS
    Temp_Low = Pull();
    Temp_High = Pull();
    ProgramCounter = (ushort)(Temp_High * 256 + Temp_Low);
    ProgramCounter++;
    cycles = 6;
    break;
```

Pular navegação

Note: JSR is actually a very wacky instruction on real hardware, with a slightly different order of operations than what is shown above.

Running 4_TheStack.nes

This ROM requires correct emulation of **PHA**, **PLA**, **JSR**, and **RTS**.

Additionally, the Stack Pointer must be $FD after Reset().

If everything is working perfectly fine, then address $0000 should be set to a value of $01 when the CPU halts.

Now we're reaching the point where we just need to program a whole bunch of instructions. So let's do that.

The official instructions

There are 151 official CPU opcodes, and so far we have implemented 15 of them. (and one unofficial opcode! (Opcode $02, HLT))

Feel free to implement them at your own pace, and in whatever order you please. There are several resources online listing the 151 official opcodes. I recommend either this list on 6502.org, or if you are looking for as much information as possible, this list here. In any case, you can always google "6502 opcodes" to find a list of instructions and their corresponding opcode.

Here's a real easy one to implement. We just implemented JSR, which moves the PC to a 16-bit address while pushing a return address. Check out the JMP instruction, which moves the PC to a 16-bit address *without* pushing a return address. Opcode $4C, taking 3 CPU cycles.

Some of the easier instructions to implement are the "Implied addressing" instructions. All of these instructions have zero operands, and perform a very basic task. The overwhelming majority of them also take exactly 2 CPU cycles. A lot of these also set the Zero and Negative flags, which I'll denote by saying "(sets ZN)".

INX: Increment X. (Opcode $E8). This instruction adds 1 to the X register. (sets ZN)

DEX: Decrement X. (Opcode $CA). This instruction subtracts 1 from the X register. (sets ZN)

INY: Increment Y. (Opcode $C8). This instruction adds 1 to the Y register. (sets ZN)

DEY: Decrement Y. (Opcode $88). This instruction subtracts 1 from the Y register. (sets ZN)

TAX: Transfer A to X. (Opcode $AA). This instruction copies the value of A and overwrites X. (sets ZN)

TXA: Transfer X to A. (Opcode $8A). This instruction copies the value of X and overwrites A. (sets ZN)

TAY: Transfer A to Y. (Opcode $A8). This instruction copies the value of A and overwrites Y. (sets ZN)

TYA: Transfer Y to A. (Opcode $98). This instruction copies the value of Y and overwrites A. (sets ZN)

Pular navegação

TXS: Transfer X to Stack Pointer. (Opcode $9A). This instruction copies the value of X and overwrites the Stack Pointer. (Notably does NOT update ZN)

TSX: Transfer Stack Pointer to X. (Opcode $BA). This instruction copies the value of the Stack Pointer and overwrites X. (sets ZN)

SEC: Set Carry (Opcode $38). This instruction sets the carry flag.

CLC: Clear Carry. (Opcode $18). This instruction clears the carry flag.

CLV: Clear Overflow. (Opcode $B8) This instruction clears the overflow flag.

SEI: Set Interrupt-Disable. (Opcode $78) This instruction sets the interrupt-disable flag.

CLI: Clear Interrupt-Disable. (Opcode $58) This instruction clears the interrupt-disable flag.

SED: Set Decimal. (Opcode $F8) This instruction sets the decimal flag.

CLD: Clear Decimal. (Opcode $D8) This instruction clears the decimal flag.

NOP: No Operation. (Opcode $EA). This instruction does nothing. It still takes 2 CPU cycles though.

PHP and PLP

Push Processor Flags, and Pull Processor Flags.

Let's put a little special attention towards PHP and PLP. Similar to PHA and PLA, these instructions push and pull the stack.

Let's start naming the 8 bits of a byte for some additional clarity in the upcoming section. Typically, the bits are labeled "bit 0" to "bit 7" starting with bit 0 on the far right, and each bit leftward counts up.

So the labels on a binary number would look like %76543210.

With the PHP instruction, here's what gets pushed to the stack:

Bit 0 is the carry flag.

Bit 1 is the zero flag.

Bit 2 is the interrupt-disable flag.

Bit 3 is the decimal flag.

Bit 4 is always set from a PHP instruction.

Bit 5 is always set from a PHP instruction.

Bit 6 is the overflow flag.

bit 7 is the negative flag.

```
case 0x08: //PHP
    Temp = 0;
    Temp += (byte)(flag_Carry ? 1 : 0);
    Temp += (byte)(flag_Zero ? 2 : 0);
    Temp += (byte)(flag_InterruptDisable ? 4 : 0);
    Temp += (byte)(flag_Decimal ? 8 : 0);
    Temp += 0x10;
    Temp += 0x20;
    Temp += (byte)(flag_Overflow ? 0x40 : 0);
    Temp += (byte)(flag_Negative ? 0x80 : 0);
    Push(Temp);
    cycles = 3;
    break;
```

Pular navegação

And likewise, that's what gets pulled off in the PLP instruction, except bits 4 and 5 don't do anything.

```
case 0x28: //PLP
    Temp = Pull();
    flag_Carry = (Temp & 1) != 0;
    flag_Zero = (Temp & 2) != 0;
    flag_InterruptDisable = (Temp & 4) != 0;
    flag_Decimal = (Temp & 8) != 0;
    flag_Overflow = (Temp & 0x40) != 0;
    flag_Negative = (Temp & 0x80) != 0;
    cycles = 3;
    break;
```

Let's start on some of the arithmetic functions.

ASL

Arithmetic Shift Left.

This instruction shifts all the bits in a number to the left, effectively multiplying it by 2. This is also the first instruction mentioned so far that updates the carry flag!

Let's imagine a number in binary, for instance, %10001000.

The bits are numbered from right to left, starting with bit 0, and ending with bit 7. In this case, bit 7 is currently a 1. After shifted, the resulting number is %00010000, but what happens to

the contents of bit 7? It goes into the carry flag! If bit 7 was set, the carry flag is now set. If bit 7 was cleared, the carry flag is now cleared.

Opcode $0A is the "Implied" form of this instruction, which performs ASL on the A register.

Here's what that might look like:


```
case 0x0A: //ASL A
    flag_Carry = A > 127;
    A <<= 1;
    flag_Zero = A == 0;
    flag_Negative = A > 127;
    ProgramCounter++;
    cycles = 2;
    break;
```

Most programming languages support "<<" as the symbol for "Shift this number to the left left by the following number of bits."

Surprise! ASL can affect more than just the A register. For instance, you could run ASL $0550 to shift the value of address $0550 to the left, if you so desire.

Let's also implement the "Zero Page" and "Absolute" forms of this instruction, opcodes $06 and $0E respectively.

Here's an example of ASL Absolute, but truth be told this code is starting to look ugly.

```
case 0x0E: //ASL Absolute
    Temp_Low = Read(ProgramCounter);
    ProgramCounter++;
    Temp_High = Read(ProgramCounter);
    ProgramCounter++;
    Temp = Read((ushort)(Temp_High * 256 + Temp_Low));
    flag_Carry = Temp > 127;
    Temp <<= 1;
    flag_Zero = Temp == 0;
    flag_Negative = Temp > 127;
    Write((ushort)(Temp_High * 256 + Temp_Low), Temp);
    cycles = 6;
    break;
```

Something you might do to clean this up is to have a function dedicated to reading the two bytes for an absolute addressed instruction:

```
1 reference
void ReadOperands_AbsoluteAddressed()
{
    addressBus = Read(ProgramCounter);
    ProgramCounter++;
    addressBus = (ushort)(Read(ProgramCounter)<<8 | addressBus);
    ProgramCounter++;
}
```

Pular navegação

```
case 0x0E: //ASL Absolute
    ReadOperands_AbsoluteAddressed();
    Temp = Read(addressBus);
    flag_Carry = Temp > 127;
    Temp <<= 1;
    flag_Zero = Temp == 0;
    flag_Negative = Temp > 127;
    Write(addressBus,Temp);
    cycles = 6;
    break;
```

Well, this is slightly less terrible...

On top of this, I strongly recommend making individual functions that you can re-use for the various forms of an instruction. This also makes it much easier to debug issues with an instruction in case all of your ASL instructions are doing something slightly wrong:

```
1 reference
void Op_ASL(ushort Address, byte Input)
{
    // Arithmetic shift left.
    flag_Carry = Input >= 0x80;     // If bit 7 was set before the shift
    Input <<= 1;
    Write(Address, Input);          // store the result at the target address
    flag_Negative = Input >= 0x80; // if bit 7 of the result is set
    flag_Zero = Input == 0x00;      // if all bits are cleared
}
```

```
case 0x06: //ASL Zero Page
    ReadOperands_ZeroPageAddressed();
    Op_ASL(addressBus, Read(addressBus));
    cycles = 5;
    break;

case 0x0E: //ASL Absolute
    ReadOperands_AbsoluteAddressed();
    Op_ASL(addressBus, Read(addressBus));
    cycles = 6;
    break;
```

There we go. That's much nicer looking.

It's certainly an improvement.

ROL

Rotate Left.

It's pretty much the same exact thing as ASL, except the previous contents of the carry flag get shifted into bit 0.

So if your old value was %10000001 and the carry flag was set, the new value would be %00000011, and the carry flag would still be set afterwards due to the old value of bit 7.

For example:


```
void Op_ROL(ushort Address, byte Input)
{
    // Rotate Left
    bool Futureflag_Carry = Input >= 0x80;
    Input <<= 1;
    if (flag_Carry)
    {
        Input |= 1; // Put the old carry flag value into bit 0
    }
    Write(Address, Input);          // store the result at the target address
    flag_Carry = Futureflag_Carry; // if bit 7 of the initial value was set
    flag_Negative = Input >= 0x80; // if bit 7 of the result is set
    flag_Zero = Input == 0x00;     // if all bits are cleared
}
```

And then with the way I'm now formatting the instructions, I can pretty much just copy what I had for ASL, swapping out the string "ASL" with "ROL".

```
case 0x26: //ROL Zero Page
    ReadOperands_ZeroPageAddressed();
    Op_ROL(addressBus, Read(addressBus));
    cycles = 5;
    break;

case 0x2E: //ROL Absolute
    ReadOperands_AbsoluteAddressed();
    Op_ROL(addressBus, Read(addressBus));
    cycles = 6;
    break;
```

LSR and ROR

The same thing as ASL and ROL, but this time we're shifting the bits to the right.

LSR A: Opcode $4A

LSR Zero Page: Opcode $46

LSR Absolute: Opcode $4E

ROR A: Opcode $6A

ROR Zero Page: Opcode $66

ROR Absolute: Opcode $6E

Perhaps you are noticing a pattern.

INC and DEC

Increment and Decrement.

Notably, neither of these instructions have an "implied" addressing form to directly increment/decrement the A register.

These are pretty self explanatory, but just to be safe, here's an example for INC.


```
void Op_INC(ushort Address, byte Input)
{
    // Increment
    Input++;                    // increment the value.
    Write(Address, Input);      // store the result at the target address
    flag_Negative = Input >= 0x80; // if bit 7 of the result is set
    flag_Zero = Input == 0x00;     // if all bits are cleared
}
```

ORA

Bitwise OR the A register with memory.

Remember when we implemented **LDA Immediate**? We simply set the A register equal to the operand. Let's start with **ORA Immediate**:

If you are unfamiliar with the bitwise operations, let's focus on some boolean math. 0 representing false, and 1 representing true.

0 or 0 = 0.

0 or 1 = 1.

1 or 0 = 1.

1 or 1 = 1.

Now, imagine performing that operation on all 8 bits of the A register with all 8 bits of the operand for an instruction.

For instance, suppose A = %00001111, (where "%" is a symbol denoting that the following value is in binary), and were running ORA #%11001100.

Starting with the bit on the far right, 1 or 0 = 1.

The next byte: 1 or 0 = 1.

Then 1 or 1 = 1.

1 or 1 = 1.

0 or 0 = 0.

0 or 0 = 0.

0 or 1 = 1.

0 or 1 = 1.

The end result is %11001111, or $CF.

If you've never worked with boolean operations, you might be surprised to know that most programming languages have built-in syntax for these. For C#, the language I'm using for my emulator, I could write "A | B" to represent "A or B".

```
void Op_ORA(byte Input)
{
    // Bitwise OR with A
    A |= Input;
    flag_Negative = A >= 0x80; // if bit 7 of the result is set
    flag_Zero = A == 0x00;     // if all bits are cleared
}
```

Pular navegação

An implementation of ORA using the "|" symbol.

Keep in mind, this instruction cannot be used to immediately modify a value in memory, only to update the A register.

ORA Immediate: Opcode $09.

ORA Zero Page: Opcode $05.

ORA Absolute: Opcode $0D.

AND

Bitwise AND the A register with memory.

Pretty much the same thing as ORA, but with bitwise AND. Most programming languages support this operation with the "&" symbol.

AND Immediate: Opcode $29.

AND Zero Page: Opcode $25.

AND Absolute: Opcode $2D.

EOR

Bitwise Exclusive OR the A register with Memory.

Pretty much the same thing, but XOR. But instead of calling it "XOR" it's called "EOR" for some reason.

Most programming languages support this operation with the "^" symbol. (not to be confused with "to the power of".)

EOR Immediate: Opcode $49.

EOR Zero Page: Opcode $45.

EOR Absolute: Opcode $4D.

ADC and SBC

Add with Carry, and Subtract with Carry.

These instructions are actually slightly unintuitive, so allow me to give a full explanation.

Now, on a typical 6502 CPU this is where the Decimal flag would come in, but we're emulating the NES, so just a reminder, don't bother doing anything with the Decimal flag here.

Let's focus on ADC.

The idea is, A = A + input + carry.

So as an example, if we're running **ADC Immediate**, with the operand being $20, A is $50, and the carry flag is set, the result is $50 + $20 + 1 = $71. At this point, the zero flag is cleared, and the negative flag is set.

But in addition to the zero and negative flags, the ADC instruction also sets the carry and overflow flags! To be more specific, the carry flag is set in the event of an unsigned overflow, and the overflow flag is set in the event of a signed overflow.

Carry Flag: If A + input + carry wraps around past $FF back to $00, set the carry flag.

otherwise, clear it. For instance, $F0 + $20 + 0 = $10. This sets the carry flag.

Overflow Flag: if a positive number + a positive number = a negative number, the overflow flag is set. If a negative number + a negative number = a positive number, the overflow flag is set. In all other cases, the overflow flag is cleared. (And just to be clear, for a signed byte, any number $80 or greater are considered negative.) For instance, $70 + $20 + 0 = $90. This sets the overflow flag.

Since this is a doozy, let's take a look at how one might implement this:

```
void Op_ADC(byte Input)
{
    // Add with Carry
    int IntSum = Input + A + (flag_Carry ? 1 : 0);
    flag_Overflow = (~(A ^ Input) & (A ^ IntSum) & 0x80) != 0;
    flag_Carry = IntSum > 0xFF;
    A = (byte)IntSum;
    flag_Negative = A >= 0x80; // if bit 7 of the result is set
    flag_Zero = A == 0x00;     // if all bits are cleared
}
```

It almost feels like cheating using an integer to perform this math, ha!

And then it got worse. Let's see what **SBC** is doing.

So, **SBC** is quite literally the opposite of **ADC**.

A = A - Input - !carry. (Subtract an extra 1 if the carry flag is not set)

The negative flag and zero flag still act as expected, but now the carry flag is set if the subtraction leaves the result greater than 0 without underflowing.

Likewise, the overflow flag is now set if a positive number minus a negative number = a negative number, or if a negative number minus a positive number = a positive number.

Here's an example:

```
void Op_SBC(byte Input)
{
    // Subtract with Carry
    int IntSum = A - Input - (flag_Carry ? 0 : 1);
    flag_Overflow = ((A ^ Input) & (A ^ IntSum) & 0x80) != 0;
    flag_Carry = IntSum >= 0;
    A = (byte)IntSum;
    flag_Negative = A >= 0x80; // if bit 7 of the result is set
    flag_Zero = A == 0x00;     // if all bits are cleared
}
```

CMP

Compare with Accumulator.

Pular navegação This is a real easy one, and it only affects the status flags!

If A == Input, set the zero flag.

If A - Input would be negative, set the negative flag.

If A >= Input, set the carry flag.

```
void Op_CMP(byte Input)
{
    // Compare A
    flag_Carry = Temp < A;
    flag_Zero = Temp == A;
    flag_Negative = (byte)(A-Temp) > 127;
}
```

This can pretty much be copy/pasted for CPX and CPY, which compare the input with the X and Y registers respectively.

BIT

Bit Test.

This is one of the least commonly used official instructions. Like CMP, it only affects the status flags, but it does so in a fairly odd way.

If A bitwise AND the Input equals zero, set the zero flag.

If bit 7 of the input is set, set the negative flag.

if bit 6 of the input is set, set the overflow flag.

```
void Op_BIT(byte Input)
{
    flag_Zero = (A & Input) == 0;
    flag_Negative = (Input & 0x80) != 0;
    flag_Overflow = (Input & 0x40) != 0;
}
```

Very strange, that one.

BRK and RTI

Break and Return from Interrupt.

Opcode $00 and $40

I've saved these ones for last, since they are amusing. Remember when we implemented Reset(), and we needed to move the PC to a very specific address determined by two bytes all the way at the end of the ROM? Well with BRK we get to do that again! Only this time we're going to use address $FFFE and $FFFF.

Like JSR, the BRK instruction pushes a return address, but unlike how JSR's return address was one byte earlier than one would expect, this return address is actually 1 byte further than one would expect, essentially making BRK 2 bytes wide. And in addition to pushing the return address, BRK also pushes the processor flags, exactly like a PHP instruction.


```
case 0x00: //BRK
    ProgramCounter++;
    Push((byte)(ProgramCounter >> 8));
    Push((byte)ProgramCounter);
    Temp = 0;
    Temp += (byte)(flag_Carry ? 1 : 0);
    Temp += (byte)(flag_Zero ? 2 : 0);
    Temp += (byte)(flag_InterruptDisable ? 4 : 0);
    Temp += (byte)(flag_Decimal ? 8 : 0);
    Temp += 0x10;
    Temp += 0x20;
    Temp += (byte)(flag_Overflow ? 0x40 : 0);
    Temp += (byte)(flag_Negative ? 0x80 : 0);
    Push(Temp);
    byte TempLow = Read(0xFFFE); // PC low byte
    byte TempHigh = Read(0xFFFF); // PC high byte
    ProgramCounter = (ushort)((TempHigh * 0x100) + TempLow);
    cycles = 7;
    break;
```

Then, RTI is the opposite. It pulls of the processor flags, and then the return address. Unlike RTS, RTI does not increment the PC after moving it to the return address.

Make a tracelogger!!!

As the test ROMs get more and more complex, and you add more and more opcodes, the chance for something to go wrong will climb incredibly fast. Let's make the greatest debugging tool possible: The Tracelogger!

The main idea here is to log every single instruction, while providing you with as much information as possible if you ever need to look at it.

Let's take a look at an existing Tracelogger, such as the one found in Bizhawk:

Disasm			Registers						
8000:	4C 04 80	JMP $8004	A:00	X:00	Y:00	SP:FD	P:24	nvTbdIzc	Cy:8
8004:	A2 01	LDX #$01	A:00	X:00	Y:00	SP:FD	P:24	nvTbdIzc	Cy:11
8006:	E8	INX	A:00	X:01	Y:00	SP:FD	P:24	nvTbdIzc	Cy:13
8007:	86 00	STX $00	A:00	X:02	Y:00	SP:FD	P:24	nvTbdIzc	Cy:15
8009:	CA	DEX	A:00	X:02	Y:00	SP:FD	P:24	nvTbdIzc	Cy:18
800A:	8A	TXA	A:00	X:01	Y:00	SP:FD	P:24	nvTbdIzc	Cy:20
800B:	A8	TAY	A:01	X:01	Y:00	SP:FD	P:24	nvTbdIzc	Cy:22
800C:	84 01	STY $01	A:01	X:01	Y:01	SP:FD	P:24	nvTbdIzc	Cy:24
800E:	BA	TSX	A:01	X:01	Y:01	SP:FD	P:24	nvTbdIzc	Cy:27
800F:	86 02	STX $02	A:01	X:FD	Y:01	SP:FD	P:A4	NvTbdIzc	Cy:29
8011:	C8	INY	A:01	X:FD	Y:01	SP:FD	P:A4	NvTbdIzc	Cy:32
8012:	98	TYA	A:01	X:FD	Y:02	SP:FD	P:24	nvTbdIzc	Cy:34
8013:	AA	TAX	A:02	X:FD	Y:02	SP:FD	P:24	nvTbdIzc	Cy:36
8014:	9A	TXS	A:02	X:02	Y:02	SP:FD	P:24	nvTbdIzc	Cy:38
8015:	08	PHP	A:02	X:02	Y:02	SP:02	P:24	nvTbdIzc	Cy:40
8016:	38	SEC	A:02	X:02	Y:02	SP:01	P:34	nvTBdIzc	Cy:43
8017:	F8	SED	A:02	X:02	Y:02	SP:01	P:35	nvTBdIzC	Cy:45
8018:	78	SEI	A:02	X:02	Y:02	SP:01	P:3D	nvTBDIzC	Cy:47
8019:	08	PHP	A:02	X:02	Y:02	SP:01	P:3D	nvTBDIzC	Cy:49
801A:	28	PLP	A:02	X:02	Y:02	SP:00	P:3D	nvTBDIzC	Cy:52
801B:	28	PLP	A:02	X:02	Y:02	SP:01	P:3D	nvTBDIzC	Cy:56
801C:	AD 01 01	LDA $0101	A:02	X:02	Y:02	SP:02	P:34	nvTBdIzc	Cy:60
801F:	85 03	STA $03	A:3D	X:02	Y:02	SP:02	P:34	nvTBdIzc	Cy:64
8021:	AD 02 01	LDA $0102	A:3D	X:02	Y:02	SP:02	P:34	nvTBdIzc	Cy:67
8024:	85 04	STA $04	A:34	X:02	Y:02	SP:02	P:34	nvTBdIzc	Cy:71
8026:	A2 FD	LDX #$FD	A:34	X:02	Y:02	SP:02	P:34	nvTBdIzc	Cy:74
8028:	9A	TXS	A:34	X:FD	Y:02	SP:02	P:B4	NvTBdIzc	Cy:76
8029:	EA	NOP	A:34	X:FD	Y:02	SP:FD	P:B4	NvTBdIzc	Cy:78
802A:	EA	NOP	A:34	X:FD	Y:02	SP:FD	P:B4	NvTBdIzc	Cy:80
802B:	EA	NOP	A:34	X:FD	Y:02	SP:FD	P:B4	NvTBdIzc	Cy:82

The Tracelogger from the Bizhawk emulator.

From left to right, we have the PC, the bytes forming the instruction, the instruction written out as readable assembly code, then a list of the registers as they were *before* the instruction was executed. A, X, Y, Stack Pointer, Processor Flags (written as an 8-bit value), The processor flags again but displayed in their 8-bit configuration, and the total number of CPU cycles since the program began. (NOTE: Bizhawk's tracelogger refers to bit 5 of the processor flags as "The Break Flag" since BRK (and PHP) push that bit as a 1, and bit 6 as "The True Flag" since it's always a 1 when pushed to the stack. We'll eventually get to a situation where "The Break Flag" isn't pushed to the stack, but for now, just know that it's not a real flag, so logging it this way might seem a bit silly.)

Taking a look at the tracelogger in the provided screenshot, you can see how LDX #$01 results in the X register changing from a 00 to a 01 on the following line. Likewise, INX increments X to a 2 on the following line.

In any case, this is a pretty good structure for making your own tracelogger. You don't need a fancy GUI for it like Bizhawk. I'd even suggest just having a text box that gets filled in as you emulate.

Keep in mind, for larger programs (like a full video game) logging every instruction is time consuming and memory-wasting, so I also suggest having a checkbox to enable/disable the logger as you please.

If you are struggling with this upcoming test ROM, a tracelog can be found in the "Tracelogs" folder of the .zip, showing what correct emulation should look like.

To get started, my tracelogger will simply use a StringBuilder. In order to log the readable assembly code, we'll need some extra information. For instance, a table of opcode names:

```
String[] OpCodeNames =
{
"BRK","ORA","HLT","SLO","NOP","ORA","ASL","SLO","PHP","ORA","ASL","ANC","NOP","ORA","ASL","SLO",
"BPL","ORA","HLT","SLO","NOP","ORA","ASL","SLO","CLC","ORA","NOP","SLO","NOP","ORA","ASL","SLO",
"JSR","AND","HLT","RLA","BIT","AND","ROL","RLA","PLP","AND","ROL","ANC","BIT","AND","ROL","RLA",
"BMI","AND","HLT","RLA","NOP","AND","ROL","RLA","SEC","AND","NOP","RLA","NOP","AND","ROL","RLA",
"RTI","EOR","HLT","SRE","NOP","EOR","LSR","SRE","PHA","EOR","LSR","ALR","JMP","EOR","LSR","SRE",
"BVC","EOR","HLT","SRE","NOP","EOR","LSR","SRE","CLI","EOR","NOP","SRE","NOP","EOR","LSR","SRE",
"RTS","ADC","HLT","RRA","NOP","ADC","ROR","RRA","PLA","ADC","ROR","ARR","JMP","ADC","ROR","RRA",
"BVS","ADC","HLT","RRA","NOP","ADC","ROR","RRA","SEI","ADC","NOP","RRA","NOP","ADC","ROR","RRA",
"NOP","STA","NOP","SAX","STY","STA","STX","SAX","DEY","NOP","TXA","ANE","STY","STA","STX","SAX",
"BCC","STA","HLT","SHA","STY","STA","STX","SAX","TYA","STA","TXS","SHS","SHY","STA","SHX","SHA",
"LDY","LDA","LDX","LAX","LDY","LDA","LDX","LAX","TAY","LDA","TAX","LXA","LDY","LDA","LDX","LAX",
"BCS","LDA","HLT","LAX","LDY","LDA","LDX","LAX","CLV","LDA","TSX","LAE","LDY","LDA","LDX","LAX",
"CPY","CMP","NOP","DCP","CPY","CMP","DEC","DCP","INY","CMP","DEX","AXS","CPY","CMP","DEC","DCP",
"BNE","CMP","HLT","DCP","NOP","CMP","DEC","DPC","CLD","CMP","NOP","DCP","NOP","CMP","DEC","DCP",
"CPX","SBC","NOP","ISC","CPX","SBC","INC","ISC","INX","SBC","NOP","SBC","CPX","SBC","INC","ISC",
"BEQ","SBC","HLT","ISC","NOP","SBC","INC","ISC","SED","SBC","NOP","ISC","NOP","SBC","INC","ISC",
};
```

Depending on how fancy we want our tracelogger to be, we might also want to know how any operands every opcode has, and the addressing mode of every opcode.

Pular navegação

Here's the code for a pretty simple trace logger:

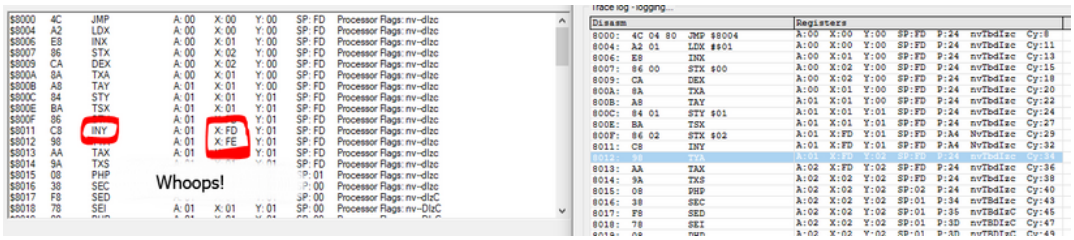
```
void Tracelogger(byte opcode)
{
    if(logging)
    {
        String line = "$" + ProgramCounter.ToString("X4")
            + "\t" + opcode.ToString("X2")
            + "\t" + OpCodeNames[opcode]
            + "\t\tA: " + A.ToString("X2")
            + "\tX: " + X.ToString("X2")
            + "\tY: " + Y.ToString("X2")
            + "\tSP: " + stackPointer.ToString("X2")
            + "\tProcessor Flags: "
            + (flag_Negative ? "N" : "n")
            + (flag_Overflow ? "V" : "v")
            + "--"
            + (flag_Decimal ? "D" : "d")
            + (flag_InterruptDisable ? "I" : "i")
            + (flag_Zero ? "Z" : "z")
            + (flag_Carry ? "C" : "c");

        tracelog.AppendLine(line);
    }
}
```

And here's what it looks like:

```
$8000   4C   JMP   A: 00   X: 00   Y: 00   SP: FD   Processor Flags: nv--dIzc
$8004   A2   LDX   A: 00   X: 00   Y: 00   SP: FD   Processor Flags: nv--dIzc
$8006   E8   INX   A: 00   X: 01   Y: 00   SP: FD   Processor Flags: nv--dIzc
$8007   86   STX   A: 00   X: 02   Y: 00   SP: FD   Processor Flags: nv--dIzc
$8009   CA   DEX   A: 00   X: 02   Y: 00   SP: FD   Processor Flags: nv--dIzc
$800A   8A   TXA   A: 00   X: 01   Y: 00   SP: FD   Processor Flags: nv--dIzc
$800B   A8   TAY   A: 01   X: 01   Y: 00   SP: FD   Processor Flags: nv--dIzc
$800C   84   STY   A: 01   X: 01   Y: 01   SP: FD   Processor Flags: nv--dIzc
$800E   BA   TSX   A: 01   X: 01   Y: 01   SP: FD   Processor Flags: nv--dIzc
$800F   86   STX   A: 01   X: FD   Y: 01   SP: FD   Processor Flags: nv--dIzc
$8011   C8   INY   A: 01   X: FD   Y: 01   SP: FD   Processor Flags: nv--dIzc
$8012   98   TYA   A: 01   X: FE   Y: 01   SP: FD   Processor Flags: nv--dIzc
$8013   AA   TAX   A: 01   X: FE   Y: 01   SP: FD   Processor Flags: nv--dIzc
$8014   9A   TXS   A: 01   X: 01   Y: 01   SP: FD   Processor Flags: nv--dIzc
$8015   08   PHP   A: 01   X: 01   Y: 01   SP: 01   Processor Flags: nv--dIzc
$8016   38   SEC   A: 01   X: 01   Y: 01   SP: 00   Processor Flags: nv--dIzc
$8017   F8   SED   A: 01   X: 01   Y: 01   SP: 00   Processor Flags: nv--dIzC
$8018   78   SEI   A: 01   X: 01   Y: 01   SP: 00   Processor Flags: nv--DIzC
```

It's currently missing the operands of the instructions, but it's a pretty good start!

With your tracelogger finished, you can run a test rom and compare logs with an existing emulator. Here's an example of me finding an issue with my emulator by comparing logs with bizhawk:



INY was accidentally incrementing the X register instead of the Y register. This was easily spotted by comparing logs with Bizhawk.

Trust me when I say this: The more work you put into your tracelogger, the easier it will be to debug issues.

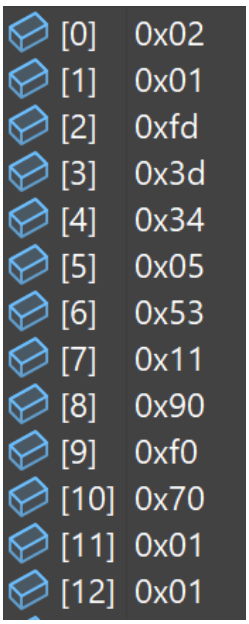
Pular navegação

unning 5_Instructions1.nes

Wow, it has been a while since we ran a test ROM, huh? Lots of instructions to implement for the CPU, and we're probably only halfway there. Anyway, here's a fun test that requires pretty much all the instructions mentioned above.

To assist in debugging, in addition to the tracelogs being in their own folder, the source code to all of these tests can be found in the "source" folder found in the .zip.

If everything works correctly, you should see the following hexadecimal values from address $0000 to address $000C: 02 01 FD 3D 34 05 53 11 90 F0 70 01 01



The contents of RAM as seen from Visual Studio's Debug Display.

And for your convenience, I'll also label what tests each of these addresses are holding the results of:

Address 0 tests INX and STX

Address 1 tests DEX, TXA, TAY, and STY

Address 2 tests for TSX

Address 3 and 4 test TXS, PHP, SEC, SED, and SEI

Address 5 tests ASL and ROL.

Address 6 tests AND, EOR, ORA, ROR, and LSR.

Address 7 and 8 test ADC

Address 9 and 10 test SBC

Address 11 tests CMP, BIT, and BRK

Address 12 tests RTI (assuming BRK worked).

Indexed Addressing

The true use for the X and Y registers. Now we can get arrays in our 6502 assembly code!

The general gist of it is, instead of just saying **LDA $ABCD**, we can say **LDA $ABCD, X**, adding the value of the X register to the address we're performing the instruction on. This is super convenient in loops, as you can iteratively read from or write to incremental addresses in memory.

And if you are using a function like ReadOperands_AbsoluteAddressed(), you can easily make a version for X and Y indexed addressing:

```
void ReadOperands_AbsoluteAddressed_XIndexed()
{
    addressBus = Read(ProgramCounter);
    ProgramCounter++;
    addressBus = (ushort)(Read(ProgramCounter) << 8 | addressBus);
    ProgramCounter++;
    addressBus += X;
}

void ReadOperands_AbsoluteAddressed_YIndexed()
{
    addressBus = Read(ProgramCounter);
    ProgramCounter++;
    addressBus = (ushort)(Read(ProgramCounter) << 8 | addressBus);
    ProgramCounter++;
    addressBus += Y;
}
```

If you are aiming for maximum accuracy, you need to add an extra CPU cycle if the indexing changes the high byte of the target address.

```
case 0xB9: //LDA Absolute, Y
    ReadOperands_AbsoluteAddressed_YIndexed();
    A = Read(addressBus);
    cycles = 4;
    break;

case 0xBD: //LDA Absolute, X
    ReadOperands_AbsoluteAddressed_XIndexed();
    A = Read(addressBus);
    cycles = 4;
    break;
```

Implementing fifty new opcodes has never been so easy.

There's also zero page indexed, and the important thing to note for those is that the address cannot leave the zero page, so **LDA <$FF, X** would wrap around to address $0000 instead of crossing into address $0100.

Indirect Addressing

There is a single instruction using "Indirect addressing" and that's opcode $6C, **JMP indirect**.

Let's just see an example. Assume a triplet of bytes $6C 00 05, or JMP ($0500). This will read a 16-bit value at address $0500 and $0501, and *that* is where the PC will be moved to. So, if address $0500 has the value $5A, and address $0501 has the value $80, the PC will move to address $805A.

Pular navegação

ıdirect Indexed Addressing

Unlike Indirect Addressing, Indirect Indexed Addressing only has one operand, which means the pointer must be on the zero page.

This is remarkably similar to Indirect Addressing, but also after reading the 16-bit value and determining the new location where you want to perform the operation, you then add the Y register to this address. So, **LDA ($05), Y** would read from address $05 and $06 to form a new 16-bit address to read from, and then add Y to that, and read from *that* address.

We're getting a bit off the rails here, so let me share some code:

```
void ReadOperands_IndirectAddressed_YIndexed()
{
    addressBus = Read(ProgramCounter);
    ProgramCounter++;
    byte TempAddress = (byte)addressBus;
    addressBus = Read(TempAddress); //Low byte of new address
    TempAddress++;
    addressBus = (ushort)(Read(TempAddress) << 8 | addressBus); // high byte
    addressBus += Y;
}
```

An example of how one might perform indirect indexed addressing.

Indexed Indirect Addressing

I promise I'm not pulling your leg here, or joking in any capacity. "Indexed Indirect Addressing" is entirely different from "Indirect Indexed Addressing" and I cannot believe this naming convention has survived since the 80's.

Okay, so you know how in "Indirect Indexed Addressing" (the one from the previous section) you add the Y register to the target address *after* reading the 16-bit pointer on the zero page? Okay, well this time, we add the X register to the address being read, before we read the 16-bit pointer. So for instance, LDA ($50, X) where X= $10 would read from address $60 and $61 to read the 16-bit pointer to determine where to read from for the LDA instruction.

```
void ReadOperands_IndirectAddressed_XIndexed()
{
    addressBus = (byte)(Read(ProgramCounter) + X);
    ProgramCounter++;
    byte TempAddress = (byte)addressBus;
    addressBus = Read(TempAddress); //Low byte of new address
    TempAddress++;
    addressBus = (ushort)(Read(TempAddress) << 8 | addressBus); // high byte
}
```

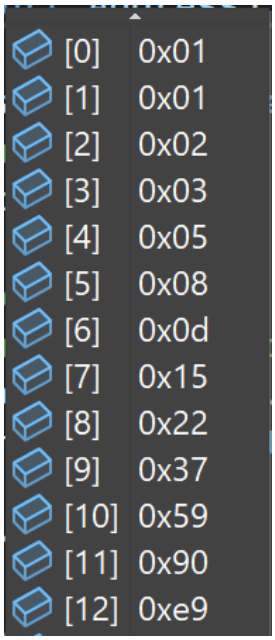
Okay, and that's it. That's all the addressing modes!

Running 6_Instructions2.nes

This is the final test of the CPU before we move on to the Picture Processing Unit.

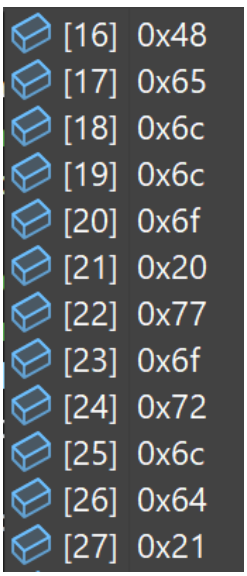
requires Zero Page indexing, CPX, Absolute Indexing, Indirect Indexed Addressing, CPY, ...dexed Indirect Addressing, and finally, Indirect Addressing.

The results should be: From address $0000 to Address $000B, the Fibonacci Sequence:



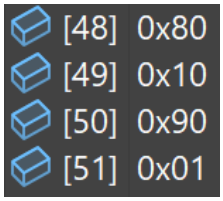
The Fibonacci Sequence, but in Hexadecimal.

From address $0010 to address $001A, the ASCII values for "Hello world!":



From address $0020 to address $002F, just the values $00 to $0F in sequential order. (No picture necessary)

And finally, from address $0030 to address $0033, the values $80, $10, $90, $01.



This should also leave $30 and $00 at address $00FE, but that's not important for the test results.

If you have made it this far, you should now have a working 6502 CPU emulator! Give yourself a pat on the back, and treat yourself to something nice for dinner. Up until now, we've just been looking at text and numbers. It's going to be nice to start getting something visual.

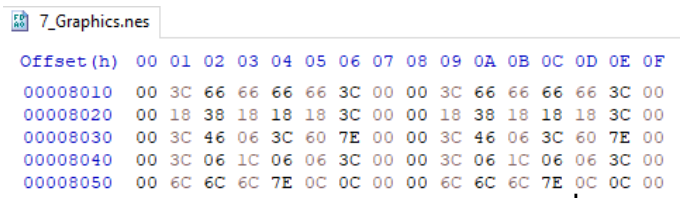
Pular navegação

The Picture Processing Unit.

Hereafter referred to as the PPU, this is essentially the "graphics card" of the NES. While implementing the PPU, we're going to learn about how the graphics are stored in a cartridge, how these graphical characters and color palettes get assigned to portions of the screen, how sprites get rendered, how the screen scrolls, and most importantly, how the CPU can communicate with the PPU.

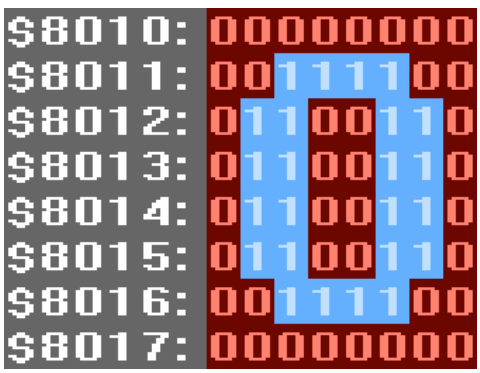
Up until now, every test ROM has been without graphics. Let's take a look at a ROM *with* graphics!

If you open up 7_Graphics.nes in a hex editor and scroll down to offset $8010, you'll see this:



This isn't assembly code (Program Data, or PRG for short), this is the graphics! (Character data, or CHR for short).

Let's look at the first 8 bytes here, written in binary:



The first 8 bytes of the CHR ROM, drawn out 1 bit at a time.

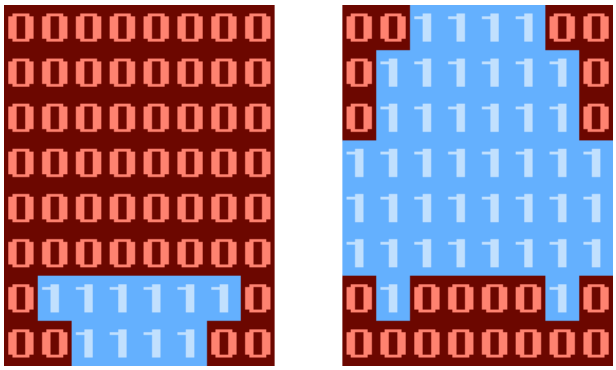
Check it out! You can visually see how this draws the number zero.

You might have noticed that the following 8 bytes, from offset $8018 to $801F are the exact same values, which would also draw the number zero. Every pixel of every character is actually 2-bit, meaning that you can choose one of four colors for each pixel. The first 8 bytes are the "low bit plane", while the second 8 bytes are the "high bit plane". Let's see an example from Super Mario Bros. where both bit planes are different, so we can see what's going on.

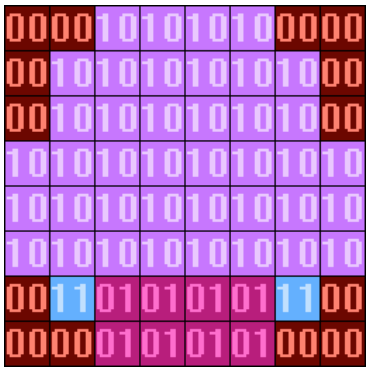
Here are the 16 bytes for a specific character:
00 00 00 00 00 00 7E 3C 3C 7E 7E FF FF FF 42 00

Let's see how these bit planes look:

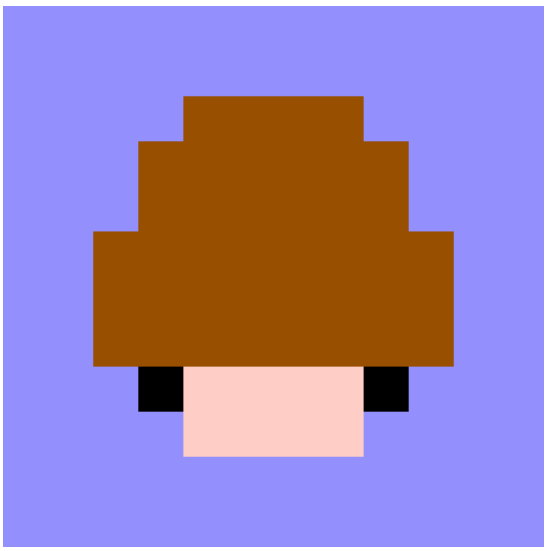
Pular navegação



And when you combine the two bitplanes, you get this:



It's the title screen's cursor!



Let's write a function to visualize a ROM's Character Data!

The Pattern Tables

Our goal here is to draw the "Pattern Tables", which are just a visual form of the entire CHR ROM. We're going to be making two 128x128 pixel boxes. One for the background pattern table, and one for the sprite pattern table.

To begin with, let's copy the PRG ROM into an array, just like we did with the CHR ROM way back when we wrote the Reset() function.

For now, I'm going to be making a brand new function for all of this, completely separate from the functions that ran the CPU.

```
byte[] ROM = File.ReadAllBytes(filepath);
byte[] CHRROM = new byte[0x2000];
Array.Copy(ROM,0x8010, CHRROM,0, 0x2000);
```

Separating the CHR ROM form the full .nes file.

Pular navegação

Visual Studio Winforms has an element called a *PictureBox* that lets me render a Bitmap. This is how I'm going to be visualizing the CHR data.

Let's create a new bitmap with the dimensions 256 by 128.

```
Bitmap bitmap = new Bitmap(256, 128);
```

There's many ways to go about drawing this, but I'm going to set up a series of for loops:

```
for(int table = 0; table < 2; table++)
{
    for (int row = 0; row < 16; row++)
    {
        for(int column = 0; column < 16; column++)
        {
            for(int y = 0; y < 8; y++)
            {
                for(int x = 0; x < 8; x++)
                {
                    //TODO: draw the pixel.
                }
            }
        }
    }
}
```

Let's determine what the low byte and high byte are for the current strip of 8 pixels that we're trying to draw:

```
byte lowByte = CHRROM[y + column*16 + row * 256 + table * 4096];
byte highByte = CHRROM[8 + y + column * 16 + row * 256 + table * 4096];
for (int x = 0; x < 8; x++)
{
    //TODO: draw the pixel.
}
```

And now for each pixel, check the value of the low / high bits:

```
for (int x = 0; x < 8; x++)
{
    int TwoBit = ((lowByte>>(7-x)) & 1) == 1 ? 1 : 0;
    TwoBit += ((highByte >> (7-x)) & 1) == 1 ? 2 : 0;
}
```

Pular navegação

After shifting the bits, bitwise AND with 1, and check if the result is 0 or 1.This starts by checking bit 7, then works towards bit 0.

We now have a 2-bit value representing the color for this pixel. Let's visualize this like so:

00 = Black

01 = Dark Grey

10 = Light Grey

11 = White

```
for (int x = 0; x < 8; x++)
{
    int TwoBit = ((lowByte>>(7-x)) & 1) == 1 ? 1 : 0;
    TwoBit += ((highByte >> (7-x)) & 1) == 1 ? 2 : 0;
    Color color = Color.FromArgb(255, TwoBit* 85, TwoBit * 85, TwoBit * 85);
    bitmap.SetPixel(x + column*8 + table*128, y + row*8, color);
}
```

And then I'll set this bitmap to be the image for the PictureBox, and *viola*!

The pattern tables for 7_Graphics.nes

If your Pattern-Table-Visualizer is fully functional, you should also check out the pattern table for your copy of Super Mario Bros.

I won't show that game's full pattern table here, that way it's more rewarding when you load it into your own code.

Well, back to the main emulator code, let's update our Reset() function to also copy the CHR Data from the ROM.

```
public void Reset()
{
    byte[] HeaderedROM = File.ReadAllBytes(filepath);
    Array.Copy(HeaderedROM, 0x10, ROM, 0, 0x8000);
    Array.Copy(HeaderedROM, Header, 0x10);
    Array.Copy(HeaderedROM, 0x8010, CHRData, 0, 0x2000);
```

The PPU Address Space

While the CPU has a 16-bit address space (Address $0000 to $FFFF), the PPU has a 14-bit address space. (Address $0000 to $3FFF)

The CHR ROM (which we used to visualize the pattern tables) occupy PPU Address $0000 to $1FFF, and the Video RAM, (or VRAM) is arranged in an amusing way, depending on how the cartridge decides to arrange it.

Just like the CPU RAM, VRAM is also composed of $800 bytes. These $800 bytes create two "Nametables". (More on those in a bit) These $800 bytes have several mirrors (just like CPU RAM) and are accessible from PPU Address $2000 through $3EFF. (Though more likely, you're just going to be writing to PPU Address $2000 through $2BFF)

Additionally, there are $20 bytes of "Palette RAM" starting at PPU Address $3F00. The NES background supports 4 color palettes, each with 3 colors plus "transparency" (The backdrop color, shared in all color palettes). Palette RAM determines which colors are assigned to the 4 color palettes. More on the palettes later.

Communicating with the PPU

So let's say you want to write to VRAM in your assembly code. How does one do that, and how do we make our emulator do that?

Just like the CPU, the PPU has registers. Let's create a boolean for the PPU's write-latch register, and a ushort for the ppu's VRAM-Address and Transfer-Address registers.

```
bool WriteLatch;   //PPU's w register
ushort TransferAddress; //PPU's t register
ushort VRAMAddress; //PPU's v register.
```

It's been a long time, but let's update our Write() function! There are special addresses that we can write to in order to communicate with the PPU.

```
void Write(ushort Address, byte Value)
{
    if(Address < 0x2000)
    {
        // Write to RAM
        RAM[Address & 0x7FF] = Value;
    }
    else if(Address < 0x4000)
    {
        // Write to a PPU register
    }
}
```

Write(), including code that supports RAM mirroring, and now a new block for the PPU Registers

If we are writing to any address on the CPU address space from address $2000 to $3FFF, that's going to write to one of the 8 PPU registers. Just like with RAM Mirroring, these 8 PPU registers are mirrored every 8 addresses. We can easily implement this by making a slight change to the address we're writing to:

```
else if(Address < 0x4000)
{
    // Write to a PPU register
    Address &= 0x2007;
}
```

Now if we are writing anywhere from $2000 to $3FFF, we are going to write exclusively from $2000 to $2007.

Let's set up a switch statement for these 8 addresses.

```
// Write to a PPU register
Address &= 0x2007;
switch (Address)
{
    case 0x2000: //PPUCTRL
        break;
    case 0x2001: //PPUMASK
        break;
    case 0x2002: //PPUSTATUS
        break;
    case 0x2003: //OAMADDR
        break;
    case 0x2004: //OAMDATA
        break;
    case 0x2005: //PPUSCROLL
        break;
    case 0x2006: //PPUADDR
        break;
    case 0x2007: //PPUDATA
        break;
}
```

Let's focus on address $2006 and $2007 for now.

Address $2006 and $2007

When you write to address $2006, you modify the PPU's internal VRAM Address. Since the VRAM address is 14-Bit, you need to perform two writes to $2006 to fully set this up. We can use the "WriteLatch" boolean we created to check if this is the first or second time we're writing to $2006.

The second write to $2006 will update both VRAMAddress and TransferAddress.

```
case 0x2006: //PPUADDR
    if(!WriteLatch)
    {
        // first write sets the high byte.
        TempVRAMAddress = (ushort)((Value & 0x3F) << 8);
        // The actual VRAMAddress isn't changed until the 2nd write.
    }
    else
    {
        // second write sets the low byte.
        VRAMAddress = (ushort)(TempVRAMAddress | Value);
        TransferAddress = VRAMAddress;
    }
    WriteLatch = !WriteLatch;
    break;
```

With the VRAM address all set up, we can then write to $2007. This is how we're going to be writing tile data to the screen's background, or modify the color palettes.

```
case 0x2007: //PPUDATA
    // What happens next depends on where the VRAMAddress is.
    if (VRAMAddress < 0x2000)
    {
        // Write to pattern table. (If the cartridge supports it.)
    }
    else if (VRAMAddress < 0x3F00)
    {
        // Write to the Nametables.
    }
    else
    {
        // Write to Palette RAM
    }
    break;
```

Pular navegação

:t's start by verifying if the cartridge supports writing to the pattern tables! (CHR-RAM)

This will be the first instance where we need to refer to the ROM's ines header! Remember those 16 bytes that we've been ignoring? Let's take a look at them.

Offset 5 of the ROM's header determines how large the CHR Data is for the cartridge. If this value is non-zero, then the cartridge has CHR-ROM, which is read-only. And of course, if it is read-only, then nothing happens when writing here.

```
if (VRAMAddress < 0x2000)
{
    // Write to pattern table. (If the cartridge supports it.)
    if (Header[5] == 0)
    {
        CHRData[VRAMAddress] = Value;
    }
    //else, nothing happens.
}
```

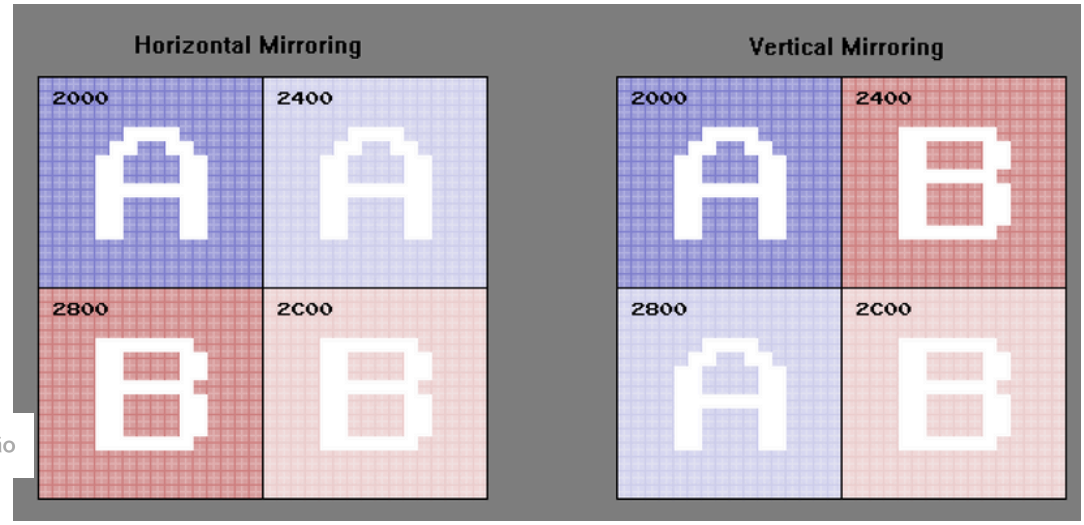
When writing to the nametables, there is either "horizontal mirroring" or "vertical mirroring" which is also determined by the header. This time, we need to check bit 0 of offset 6.

```
else if (VRAMAddress < 0x3F00)
{
    // Write to the Nametables.
    if ((Header[6] & 1) == 0)
    {
        // "horizontal mirroring"
    }
    else
    {
        // "vertical mirroring"
    }
}
```

As a reminder, the console's VRAM only has $800 bytes. Writing from PPU address $2000 to $23FF will write to the first $400 bytes of VRAM, but things get a bit complicated for the next

$400 bytes.

On their page about Nametable Mirroring, the NesDev wiki has a wonderful graphic for this:



Imagine "Box A" as the first $400 bytes of VRAM, and "Box B" as the second $400 bytes. Depending on the mirroring arrangement, writing from PPU address $2400 to $27FF could be writing to "Box A" or "Box B". Likewise, writing from PPU address $2800 to $2BFF could also write to either "box" depending on the mirroring arrangement.

```
else if (VRAMAddress < 0x3F00)
{
    // Write to the Nametables.
    if ((Header[6] & 1) == 0)
    {
        // "horizontal mirroring"
        VRAM[(VRAMAddress & 0x3FF) | (VRAMAddress & 0x800) >> 1] = Value;
    }
    else
    {
        // "vertical mirroring"
        VRAM[VRAMAddress & 0x7FF] = Value;
    }
}
```

Now to write to palette RAM. Let's make Palette RAM a 32-byte long array.

The first 16 bytes are for the background color palettes, and the following 16 bytes are for the sprite background palettes. The background contains 4 color palettes, each 4 bytes long (totaling 16 bytes), however, index $00 of Palette RAM is *The Backdrop Color*. The Backdrop Color is re-used in every color palette, even though you could store a different value at Palette RAM address $4, $8, or $C. The backdrop color also applies to the color palettes of sprites.

Though index 0 of each palette can be a different color, index 0 of each background palette is a mirror of index 0 of the corresponding sprite palette. So address $00 and address $10 of palette RAM are mirrors. Address $14 is a mirror of address $04, address $18 is a mirror of address $08, and address $1C is a mirror of address $0C.

```
else
{
    // Write to Palette RAM
    if ((VRAMAddress & 3) == 0)
    {
        PaletteRAM[VRAMAddress & 0x0F] = Value;
    }
    else
    {
        PaletteRAM[VRAMAddress & 0x1F] = Value;
    }
}
```


And finally, after the write to $2007, the VRAM Address is automatically incremented! We haven't implemented it yet, but the PPU has a setting to make this increment by 32 instead of by 1. I just created a boolean for that, even though we're not yet setting up said boolean.

```
VRAMAddress += (ushort)(PPU_VRAMInc32Mode ? 32 : 1);
VRAMAddress &= 0x3FFF;
break;
```

Don't forget the VRAM Address is limited to 14 bits!

And now we bring it all together.

With writing to address $2006 and $2007 implemented, we have the ability to set up a nametable and draw it in a similar way to how we showcased the pattern tables!

VRAM is composed of $800 bytes, with each $400 bytes being it's own nametable. A nametable is composed of two parts. $3C0 bytes of "tile data", and the $40 byte "Attribute Table". Let's talk about "Tile data" first.

There's a PPU register you can write to in order to choose which pattern table to use for the backgrounds. We'll implement that register in a moment, but first, let's just assume we're using pattern table 0 for the background. Let's get another look at it:

Pular navegação

You can see how it is divided into 256 "characters". We have the values 0 to 9, A to Z, some punctuation marks, some crumbling brickwork, and then a massive waste of space featuring a goofy doodle of a scientist saying "Nice" and a cone shaped confetti cannon.

Suppose we're making a game with this pattern table. (Well, mostly using the top 5 rows of characters...), If you want to print text on screen, how would you go about doing that?

It's pretty easy to figure out a character's index, as the lower nybble (the lower 4 bytes. In Hexadecimal, it's the right-most digit) is simply what column the character is in, while the upper nybble is what row the character is is. For instance, the "?" symbol is in row 3, column 7, so if we want to put a "?" on the nametable, we would write $37 to address $2007. (Presumably after you set up the VRAM Address to point to the nametable.)

Let's write "Hello world!" to the nametable, starting at PPU Address $2000. First, we need to set the VRAM address to $2000, so we'd run:

LDA #$20

STA $2006

LDA #0

STA $2006

Now we want to write an "H" to address $2007. In this pattern table, "H" is row 1, column 1, so we'll use $11.

LDA #$11

STA $2007

Since STA $2007 will automatically increment "VRAMAddress" to $2001, we can skip the process of writing to $2006 again, and immediately write the next character, "E", which is simply $0E.

LDA #$0E

STA $2007

And so on. (I'm not actually writing it all out, but that's pretty much what 7_Graphics.nes is doing.)

And after running this code, the nametable would look like "HELLO WORLD!" and then 948 copies of "0".

Let's draw that!

Running 7_Graphics.nes

The idea here being: We run the ROM, the CPU halts, and afterwards we draw the current state of the nametables.

When drawing a nametable, it's going to be 32 tiles by 30 tiles, each tile being an 8 by 8 pixel character.

Pular navegação

For this specific test, I'm only going to draw the first $3C0 bytes of VRAM. And again, this specifically is being drawn after the CPU halts:

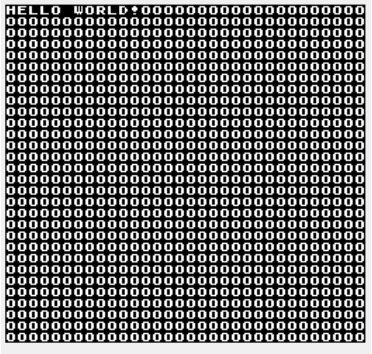
```
Bitmap nametable = new Bitmap(32 * 8, 30 * 8);
for (int row = 0; row < 30; row++)
{
    for (int column = 0; column < 32; column++)
    {

    }
}
```

Since we already have code for drawing the 8x8 pixel characters for the pattern table, we can re-purpose that and just change a few values.

```
Bitmap nametable = new Bitmap(32 * 8, 30 * 8);
for (int row = 0; row < 30; row++)
{
    for (int column = 0; column < 32; column++)
    {
        for (int y = 0; y < 8; y++)
        {
            byte lowByte = CHRROM[VRAM[column + row * 32] * 16 + y];
            byte highByte = CHRROM[VRAM[column + row * 32] * 16 + 8 + y];
            for (int x = 0; x < 8; x++)
            {
                int TwoBit = ((lowByte >> (7 - x)) & 1) == 1 ? 1 : 0;
                TwoBit += ((highByte >> (7 - x)) & 1) == 1 ? 2 : 0;
                Color color = Color.FromArgb(255, TwoBit * 85, TwoBit * 85, TwoBit * 85);
                nametable.SetPixel(x + column * 8, y + row * 8, color);
            }
        }
    }
}
pictureBox1.Image = nametable;
```

Highlighting the modified lines in order to draw a character based on the values in VRAM.

And if your emulator does everything correctly, you should be able to render this:



Pretty cool, huh? Now we're starting to get somewhere.

Reading from VRAM

Pular navegação

the previous section, we implemented writing to 2 of the 8 PPU registers. As it turns out, …ost PPU registers are "write-only", so we're not even going to worry about what happens when you read from them. It's not accurate, but let's just return 0 in those cases for now. Let's add support for reading from address $2007.

```
byte Read(ushort Address)
{
    if (Address < 0x2000)
    {
        // Reading from RAM
        return RAM[Address & 0x7FF];
    }
    else if (Address < 0x4000)
    {
        // Reading a PPU register
        Address &= 0x2007;
        switch (Address)
        {
            case 0x2007:
                return 0;
            default:
                return 0;
        }
    }
    else if (Address >= 0x8000)
    {
        // Reading from ROM
        return ROM[Address - 0x8000];
    }

    return 0;
}
```

Here's how my Read() routine is looking while I start on these changes.

When you read from address $2007, you do not immediately get the value from VRAM. Instead, there is a 1-byte buffer that you read from, which then gets populated with the value form VRAM.

For instance, suppose you wish to read from PPU address $2000. First you would set the VRAM Address to $2000, and read from $2007. This reads 1 byte of garbage from the PPU Read Buffer, while setting up the new contents of the PPU Read Buffer with the value from PPU address $2000. This action *also* increments the VRAM Address just like writing to address $2007.

Just like writing to $2007, you could be reading from the pattern tables, the name tables, or palette RAM, so you need to make sure each case is reading from the correct place.

```
case 0x2007:
    // What happens next depends on where the VRAMAddress is.
    byte temp = PPUReadBuffer;

    if (VRAMAddress < 0x2000)
    {
        // Read from the pattern table.
        PPUReadBuffer = CHRData[VRAMAddress];
        //else, nothing happens.
    }
    else if (VRAMAddress < 0x3F00)
    {
        // Read from the Nametables.
        if ((Header[6] & 1) == 0)
        {
            // "horizontal mirroring"
            PPUReadBuffer = VRAM[(VRAMAddress & 0x3FF) | (VRAMAddress & 0x800) >> 1];
        }
        else
        {
            // "vertical mirroring"
            PPUReadBuffer = VRAM[VRAMAddress & 0x7FF];
        }
    }
    else
    {
        // Read Palette RAM
        if((VRAMAddress & 3) == 0)
        {
            temp = PaletteRAM[VRAMAddress & 0x0F];
        }
        else
        {
            temp = PaletteRAM[VRAMAddress & 0x1F];
        }
    }
    VRAMAddress += (ushort)(ppuVRAMInc32Mode ? 32 : 1);
    VRAMAddress &= 0x3FFF;
    return temp;
```

Additionally, the PPU Read Buffer is *not* used when reading from Palette RAM (when the VRAM address is greater than or equal to $3F00). When reading from Palette RAM, you just return the value from VRAM at that address.

PPUSTATUS

We're only half-implementing this one for now. This PPU register is super important for actual NES games, as it can inform the CPU of three specific conditions of the PPU:

Is the PPU in VBlank? (bit 7)

Has a "Sprite Zero Hit" occurred? (bit 6)

Has a "Sprite Overflow" occurred? (bit 5)

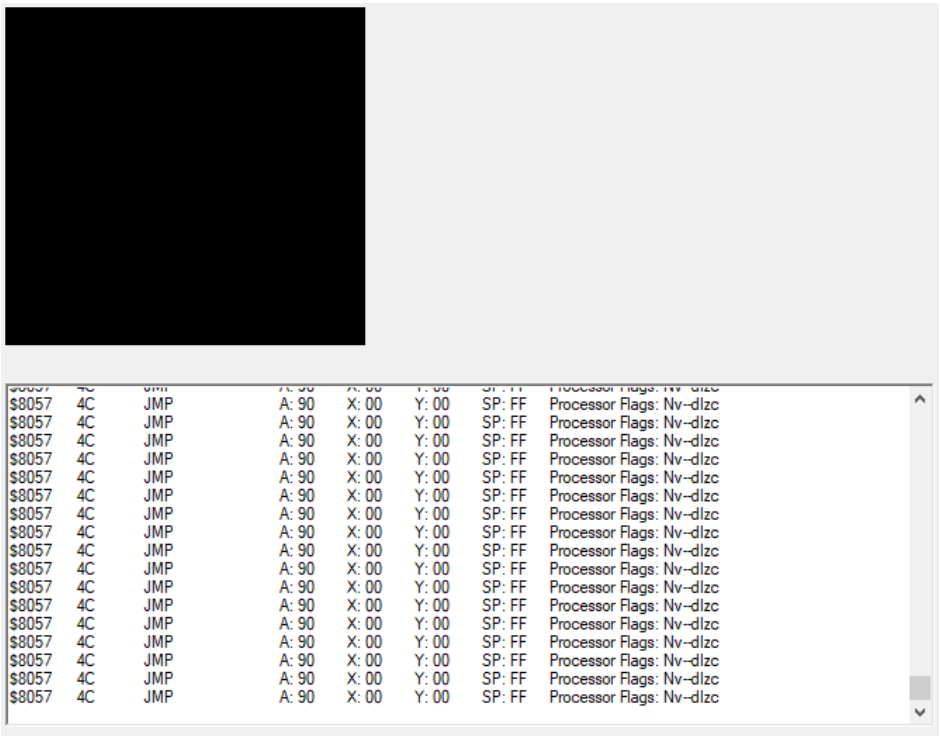
All of these specific events need to be implemented later, but for now, let's just say this address always return $80. Due to a niche edge case of the PPU where it should be incapable of writing to PPU registers for the first frame or so, almost every NES game will begin by reading from address $2002 in a loop until VBlank occurs twice. If you have tried running any real NES games in your emulator at some point before reading this, you might have noticed it getting stuck trying to read from $2002 over and over again, and that's why. By pretending it's always VBlank (as a placeholder until we implement this properly), we can circumvent this "infinite loop" issue.

```
case 0x2002:
    return 0x80;
```

we *will* be back to properly implement PPUSTATUS later, but for now, let's just see what happens if we run Super Mario Bros. in our emulator!

(By the way, I obviously cannot provide a copy of that ROM, even for research purposes such as this. You'll have to figure that part out on your own.)

If your emulator works in the same way mine currently does, you might find the nametable completely empty, and the game is stuck in an infinite loop, endlessly running JMP $8057.



As you might have guessed, our emulator is missing something, and that thing would be the "Non Maskable Interrupt".

It is now time we start implementing the PPU clock!

void Emulate_PPU()

At the end of Emulate_CPU(), let's add a small while loop that runs one iteration per cpu cycle of the instruction we just emulated. This is why we've been writing down how many CPU cycles each instruction takes.

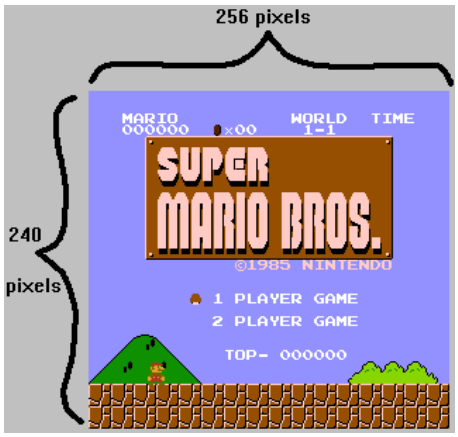
Since the PPU clock is 3 times as fast as the CPU clock, let's run 3 instances of Emulate_PPU() per CPU cycle.

```
    // the end of Emulate_CPU(), after the giant switch(opcode)
    while(cycles > 0)
    {
        cycles--;
        Emulate_PPU();
        Emulate_PPU();
        Emulate_PPU();
    }
}

3 references
void Emulate_PPU()
{

}
```

Now, those of you looking to make your emulator as accurate as possible will eventually notice a fault in the way we're clocking the PPU. As it is set up now, all the CPU cycles occur, and then all the PPU cycles occur. On the actual console both clocks run simultaneously, 3 PPU clocks per CPU clock. Since we're just aiming for our emulator to be functional, rather than very-accurate, we're not going to concern ourselves with the accurate behavior, and this will do just fine.

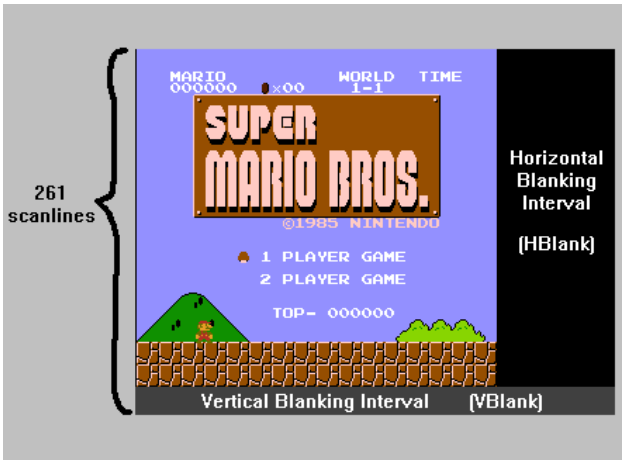
So, what exactly occurs inside Emulate_PPU()?

Let's take a look at the screen of the NES. It's 256 pixels wide and 240 scanlines tall.



However, that's just the visible part of the screen. Each scanline has both the visible portion, and the horizontal-blanking-interval. (Where on an old CRT television, the scanning beam which draws the sreen would need to be turned off and moved back to the left of the screen) Every PPU cycle can draw 1 pixel (outside of blanking) so a scanline + blanking takes 341 PPU cycles.

In addition to the HBlank on each visible scanline, there are also 20 "Vertical Blanking Interval" scanlines:



And we're going to be emulating this just like an old CRT screen. One pixel at a time, left to right, top to bottom.

Let's begin.

I'm creating two ints, essentially tracking the X and Y position of the hypothetical CRT Scanning beam.

```
int ppuDot;     // the X position of the scanning beam.
int ppuScanline;// the Y position of the scanning beam.
```

Since we're running Emulate_PPU() 1 PPU cycle at a time, we're not going to create for loops iterating over the entire screen. Instead, let's start by incrementing ppuDot, and handle it wrapping around to the next scanline. Likewise, wrapping the scanline around back to scanline 0 after 261 lines.

```
void Emulate_PPU()
{
    ppuDot++;
    if(ppuDot > 341)
    {
        ppuDot = 0;
        ppuScanline++;
        if(ppuScanline > 261)
        {
            ppuScanline = 0;
        }
    }
}
```

Let's make a boolean that tracks if the PPU is in VBlank.

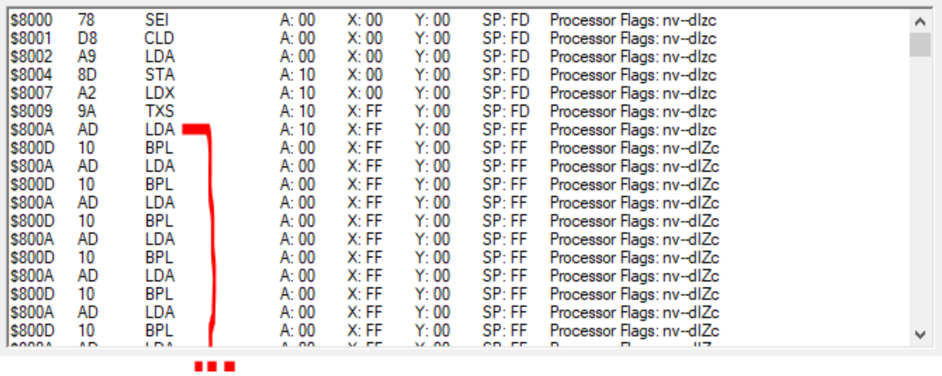
```
void Emulate_PPU()
{
    if(ppuDot == 1 && ppuScanline == 241)
    {
        ppuVBlank = true;
    }
    else if (ppuDot == 1 && ppuScanline == 261)
    {
        ppuVBlank = false;
    }
```


'ith this all set up, we can begin updating what happens when we read from PPUSTATUS ddress $2002).

```
case 0x2002:
    byte ppustatus = 0;
    ppustatus |= (byte)(ppuVBlank ? 0x80 : 0);
    ppuVBlank = false;
    return ppustatus;
```

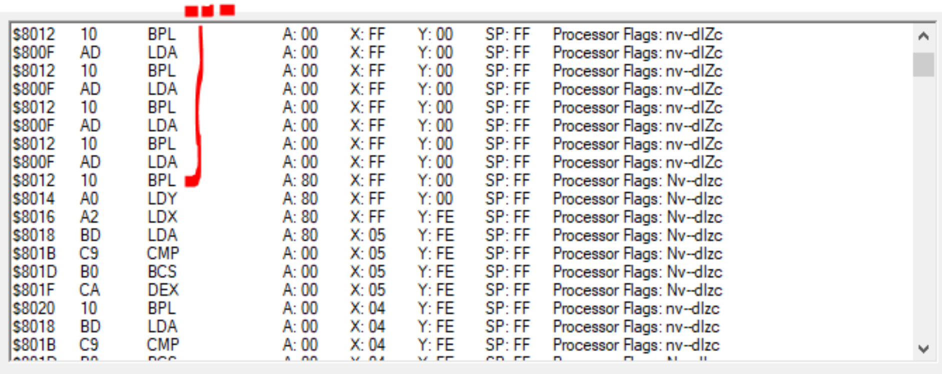
Keep in mind, the "ppuVBlank" variable is cleared when reading from $2002. This means, if you were to read from it twice during VBlank, even if VBlank has not yet ended, PPUSTATUS will read as if VBlank *did* end.

With this change implemented, our tracelogger now shows us looping while we wait for VBlank after running Super Mario Bros.



$8000	78	SEI	A: 00	X: 00	Y: 00	SP: FD	Processor Flags: nv--dlzc
$8001	D8	CLD	A: 00	X: 00	Y: 00	SP: FD	Processor Flags: nv--dlzc
$8002	A9	LDA	A: 00	X: 00	Y: 00	SP: FD	Processor Flags: nv--dlzc
$8004	8D	STA	A: 10	X: 00	Y: 00	SP: FD	Processor Flags: nv--dlzc
$8007	A2	LDX	A: 10	X: 00	Y: 00	SP: FD	Processor Flags: nv--dlzc
$8009	9A	TXS	A: 10	X: FF	Y: 00	SP: FD	Processor Flags: nv--dlzc
$800A	AD	LDA	A: 10	X: FF	Y: 00	SP: FF	Processor Flags: nv--dlzc
$800D	10	BPL	A: 00	X: FF	Y: 00	SP: FF	Processor Flags: nv--dlZc
$800A	AD	LDA	A: 00	X: FF	Y: 00	SP: FF	Processor Flags: nv--dlZc
$800D	10	BPL	A: 00	X: FF	Y: 00	SP: FF	Processor Flags: nv--dlZc
$800A	AD	LDA	A: 00	X: FF	Y: 00	SP: FF	Processor Flags: nv--dlZc
$800D	10	BPL	A: 00	X: FF	Y: 00	SP: FF	Processor Flags: nv--dlZc
$800A	AD	LDA	A: 00	X: FF	Y: 00	SP: FF	Processor Flags: nv--dlZc
$800D	10	BPL	A: 00	X: FF	Y: 00	SP: FF	Processor Flags: nv--dlZc
$800A	AD	LDA	A: 00	X: FF	Y: 00	SP: FF	Processor Flags: nv--dlZc
$800D	10	BPL	A: 00	X: FF	Y: 00	SP: FF	Processor Flags: nv--dlZc
$800A	AD	LDA	A: 00	X: FF	Y: 00	SP: FF	Processor Flags: nv--dlZc
$800D	10	BPL	A: 00	X: FF	Y: 00	SP: FF	Processor Flags: nv--dlZc

Though this doesn't loop forever, since VBlank will eventually begin, which will set the Negative flag, so the branch will no longer be taken:



$8012	10	BPL	A: 00	X: FF	Y: 00	SP: FF	Processor Flags: nv-dIZc
$800F	AD	LDA	A: 00	X: FF	Y: 00	SP: FF	Processor Flags: nv-dIZc
$8012	10	BPL	A: 00	X: FF	Y: 00	SP: FF	Processor Flags: nv-dIZc
$800F	AD	LDA	A: 00	X: FF	Y: 00	SP: FF	Processor Flags: nv-dIZc
$8012	10	BPL	A: 00	X: FF	Y: 00	SP: FF	Processor Flags: nv-dIZc
$800F	AD	LDA	A: 00	X: FF	Y: 00	SP: FF	Processor Flags: nv-dIZc
$8012	10	BPL	A: 00	X: FF	Y: 00	SP: FF	Processor Flags: nv-dIZc
$800F	AD	LDA	A: 00	X: FF	Y: 00	SP: FF	Processor Flags: nv-dIZc
$8012	10	BPL	A: 80	X: FF	Y: 00	SP: FF	Processor Flags: Nv-dIzc
$8014	A0	LDY	A: 80	X: FF	Y: 00	SP: FF	Processor Flags: Nv-dIzc
$8016	A2	LDX	A: 80	X: FF	Y: FE	SP: FF	Processor Flags: Nv-dIzc
$8018	BD	LDA	A: 80	X: 05	Y: FE	SP: FF	Processor Flags: Nv-dIzc
$801B	C9	CMP	A: 00	X: 05	Y: FE	SP: FF	Processor Flags: Nv-dIzc
$801D	B0	BCS	A: 00	X: 05	Y: FE	SP: FF	Processor Flags: Nv-dIzc
$801F	CA	DEX	A: 00	X: 05	Y: FE	SP: FF	Processor Flags: Nv-dIzc
$8020	10	BPL	A: 00	X: 04	Y: FE	SP: FF	Processor Flags: nv-dIzc
$8018	BD	LDA	A: 00	X: 04	Y: FE	SP: FF	Processor Flags: nv-dIzc
$801B	C9	CMP	A: 00	X: 04	Y: FE	SP: FF	Processor Flags: nv-dIzc

Real quick while we're here, let's set up another PPU Register inside our write() function.

PPUMASK

 ddress $2001.

This register is pretty much used to tell the PPU to forcefully stop rendering. This will be useful, since writing to PPUDATA ($2007) on a visible scanline while rendering is enabled causes issues. Let's create 4 bools. We won't be using these bools for a while, but it will be nice to have them all set up.

```
case 0x2001: //PPUMASK
    ppuMask_8pxMaskBG        = (Value & 2)    != 0;
    ppuMask_8pxMaskSprites = (Value & 4)    != 0;
    ppuMask_RenderBG        = (Value & 8)    != 0;
    ppuMask_RenderSprites  = (Value & 0x10) != 0;
    break;
```

The ppuMask_Render bools will be used to enable or disable rendering of the background/sprites. The ppuMask_8pxMask bools are used to mask away the background or sprites on the 8 leftmost pixels of the screen, opting instead to just draw the backdrop color there. We'll implement those when we actually get to rendering.

Let's add one more important PPU Register to our write() function.

PPUCTRL

When you write to address $2000, you are setting up 7 flags of the PPU. (from bit 0 to bit 7:)

Bit 0 and Bit 1: A 2-bit value to determine which nametable the screen should start drawing from.

Bit 2: The VRAM Increment mode that affects how the increment should occur when reading or writing from address $2007. We already implemented this boolean's affect on the VRAM address, but now we get to set this boolean's state!

Bit 3: Which pattern table should be used for the sprites?

Bit 4: Which pattern table should be used for the backgrounds?

Bit 5: Are sprites being drawn as 8x8 characters, or 8x16 characters?

Bit 6: we can ignore this one.

Bit 7: is the NMI enabled?

We're going to create these variables and set them up here, though a lot of them won't be used for a while.

```
case 0x2000: //PPUCTRL
    ppuNametableSelect    = Value & 3;
    ppuVRAMInc32Mode      = (Value & 4)    != 0;
    ppuSpritePatternTable = (Value & 8)    != 0;
    ppuBGPatternTable     = (Value & 0x10) != 0;
    ppuUse8x16Sprites     = (Value & 0x20) != 0;
    ppuEnableNMI          = (Value & 0x80) != 0;
    break;
```


We can now hook up ppuBGPatternTable to our placeholder Nametable renderer like so:

```
for (int y = 0; y < 8; y++)
{
    int useSecondPatternTable = ppuBGPatternTable ? 4096 : 0;
    byte lowByte = CHRROM[VRAM[column + row * 32] * 16 + y + useSecondPatternTable];
    byte highByte = CHRROM[VRAM[column + row * 32] * 16 + 8 + y + useSecondPatternTable];
    for (int x = 0; x < 8; x++)
    {
        int TwoBit = ((lowByte >> (7 - x)) & 1) == 1 ? 1 : 0;
        TwoBit += ((highByte >> (7 - x)) & 1) == 1 ? 2 : 0;
        Color color = Color.FromArgb(255, TwoBit * 85, TwoBit * 85, TwoBit * 85);
        nametable.SetPixel(x + column * 8, y + row * 8, color);
    }
}
```

Now let's focus on ppuEnableNMI.

The Non Maskable Interrupt

The NMI is a CPU interrupt driven by the PPU. Basically, if the PPU says "It's time for an NMI." then the CPU stops whatever it is currently running, and the PC is moved to a specific address, just like what happens when executing a BRK instruction, except instead of using a 16-bit pointer from address $FFFE, the NMI uses a 16-bit pointer from address $FFFA.

So, what on the PPU side triggers the NMI?

The NMI occurs when the "NMI Level Detector" changes from a high voltage to a low voltage. (Which in our emulator, will simply be a boolean going from false to true.)

This "Level detector" boolean will only be true if ppuEnableNMI is true, AND ppuVBlank is true. Then, if the "Level Detector" boolean is now true, when just a moment ago it was not, it is time to run an NMI. (In other words, we're not going to infinitely continue running NMI's over and over again until these conditions stop being met. Rather, only a single NMI occurs.)

```
bool NMILevelDetector;
bool DoNMI;
1 reference
void Emulate_CPU()
{
    bool PreviousNMILevelDetector = NMILevelDetector;
    NMILevelDetector = ppuEnableNMI && ppuVBlank;
    if(!PreviousNMILevelDetector && NMILevelDetector)
    {
        DoNMI = true;
    }
```


`ow if we *are* running an NMI, let's force the opcode to be the opcode for the NMI. Since the gic for the NMI is pretty much the same thing as BRK, I like to re-use the BRK instruction for the NMI, but if you would prefer to make the NMI have it's own opcode, you can pick something like $100.

```
byte opcode;
if (!DoNMI)
{
    // If we're not running an NMI:
    opcode = Read(ProgramCounter);
    Tracelogger(opcode);
    ProgramCounter++;
}
else
{
    // If we're running an NMI, force opcode $00
    opcode = 0x00;
}
int cycles = 0;
switch(opcode)
{
```

If you are using something like $100 as the opcode for NMI, make sure the ` opcode` variable is larger than a byte.

Now we change opcode $00 to also allow for the NMI behavior. The key differences are:

The program counter is not incremented,

Bit 5 of the processor flags is *not* set from an NMI,

And we read from $FFFA and $FFFB instead of $FFFE and $FFFF to determine where to move the PC.

```
switch(opcode)
{
    case 0x00: //BRK and NMI
        if (!DoNMI)
        {
            ProgramCounter++;
        }
        Push((byte)(ProgramCounter >> 8));
        Push((byte)ProgramCounter);
        Temp = 0;
        Temp += (byte)(flag_Carry ? 1 : 0);
        Temp += (byte)(flag_Zero ? 2 : 0);
        Temp += (byte)(flag_InterruptDisable ? 4 : 0);
        Temp += (byte)(flag_Decimal ? 8 : 0);
        Temp += (byte)(DoNMI ? 0 : 0x10);
        Temp += 0x20;
        Temp += (byte)(flag_Overflow ? 0x40 : 0);
        Temp += (byte)(flag_Negative ? 0x80 : 0);
        Push(Temp);
        byte TempLow = Read((ushort)(DoNMI ? 0xFFFA : 0xFFFE)); // PC low byte
        byte TempHigh = Read((ushort)(DoNMI ? 0xFFFB : 0xFFFF)); // PC high byte
        ProgramCounter = (ushort)((TempHigh * 0x100) + TempLow);
        DoNMI = false;
        cycles = 7;
        break;
```


You probably also want to make sure the NMI doesn't happen next instruction too. I added ` DoNMI = false;` at the end of this instruction.

Now if we try running Super Mario Bros. we'll get stuck in another interesting loop.

loop:

LDA $2002

AND #$40

BEQ loop

This loop is waiting for a "Sprite Zero Hit", but we are definitely not ready to implement that, so let's modify PPUSTATUS to always return with bit 6 set.

While we're here modifying PPUSTATUS, it's also worth noting that reading from this address also resets the WriteLatch used for writes to $2006.

```
case 0x2002:
    byte ppustatus = 0;
    ppustatus |= (byte)(ppuVBlank ? 0x80 : 0);
    ppustatus |= 0x40;
    ppuVBlank = false;
    WriteLatch = false;
    return ppustatus;
```

At this point, if we run Super Mario Bros. we're in for a real treat! Since this game isn't going to encounter a HLT instruction, I'll forcefully halt the emulator after 500,000 instructions.

Assuming all your instructions are implemented correctly, and your PPU Read Buffer works correctly too, you should be able to see this with our placeholder Nametable viewer:

Keep in mind, there are several things that could have gone wrong undetected up to this point. If you aren't seeing the title screen exactly as shown, the issue could be flags improperly being set on a single instruction, or maybe you forgot to set up the values inside the CHRData array during Reset(). If you are looking at an incorrect screen, let me simply say

Pular navegação

odspeed", and I'm wishing you the best of luck debugging this.

The color palettes

Though we're still working with our placeholder "Nametable viewer" instead of the actual screen, it couldn't hurt to finally implement color palettes.

As explained earlier, Palette RAM contains 16 bytes for the background, and 16 bytes for the sprites. These 16 bytes are split into four groups of four bytes. Here's an example of the 16 background bytes:

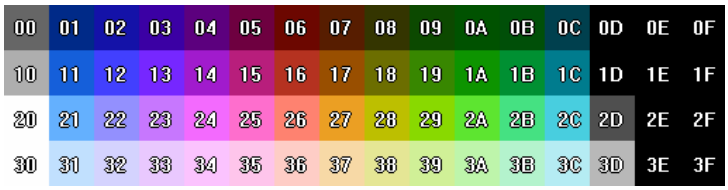
[22 29 1A 0F] [0F 36 17 0F] [0F 30 21 0F] [0F 27 17 0F]

Something to note: index 0 of each palette will always be drawn with the same color as index 0 of palette 0. It was mentioned way earlier, but it's relevant, so I'm mentioning it again. This is called "The Backdrop color".

But how do these bytes correspond to colors? Well, the actual process used by the hardware is way too complex to implement for our first NES emulator, so we're going to approximate it with a LookUp Table of RGB values.

And what a marvelous color palette it is!

Here it is again, but with each color labeled with it's index:



You might have noticed there are two instances of white, and 10 instances of black. That's just how it is.

Anyway, there's no "official" RGB values for all of these (due to the aforementioned very complex process the real hardware uses to encode and decode these colors) but for your own convenience, here is the list of RGB values that I will be using (each triplet of bytes corresponding to the R, G, and B component), and the for loop to turn these bytes into the System.Drawing.Color format:

```
byte[] Pal = {
0x65, 0x65, 0x65, 0x00, 0x2A, 0x84, 0x15, 0x13, 0xA2, 0x3A, 0x01, 0x9E, 0x59, 0x00, 0x7A,
0x6A, 0x00, 0x3E, 0x68, 0x08, 0x00, 0x53, 0x1D, 0x00, 0x32, 0x34, 0x00, 0x0D, 0x46, 0x00,
0x00, 0x4F, 0x00, 0x00, 0x4C, 0x09, 0x00, 0x3F, 0x4B, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00,
0xAE, 0xAE, 0xAE, 0x17, 0x5F, 0xD6, 0x43, 0x41, 0xFF, 0x75, 0x29, 0xFA, 0x9E, 0x1D, 0xCA,
0xB4, 0x20, 0x7B, 0xB1, 0x33, 0x22, 0x96, 0x4E, 0x00, 0x6A, 0x6C, 0x00, 0x39, 0x84, 0x00,
0x0F, 0x90, 0x00, 0x00, 0x8D, 0x33, 0x00, 0x7B, 0x8C, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
Pular navegação ‹00, 0x00, 0x00,
ᴗxFE, 0xFE, 0xFE, 0x66, 0xAF, 0xFF, 0x93, 0x90, 0xFF, 0xC5, 0x78, 0xFF, 0xEE, 0x6C, 0xFF,
0xFF, 0x6F, 0xCA, 0xFF, 0x82, 0x71, 0xE6, 0x9E, 0x25, 0xBA, 0xBC, 0x00, 0x88, 0xD5, 0x01,
0x5E, 0xE1, 0x32, 0x47, 0xDD, 0x82, 0x4A, 0xCB, 0xDC, 0x4E, 0x4E, 0x4E, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00,
0xFE, 0xFE, 0xFE, 0xC0, 0xDE, 0xFF, 0xD2, 0xD1, 0xFF, 0xE7, 0xC7, 0xFF, 0xF8, 0xC2, 0xFF,
0xFF, 0xC3, 0xE9, 0xFF, 0xCB, 0xC4, 0xF5, 0xD7, 0xA5, 0xE2, 0xE3, 0x94, 0xCE, 0xED, 0x96,
0xBC, 0xF2, 0xAA, 0xB3, 0xF1, 0xCB, 0xB4, 0xE9, 0xF0, 0xB6, 0xB6, 0xB6, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00};

Color[] Palette = new Color[64];
int i = 0;
for (int j = 0; j < 64; j++)
{
Palette[j] = Color.FromArgb(Pal[i++], Pal[i++], Pal[i++]);
}
```

Attribute Tables

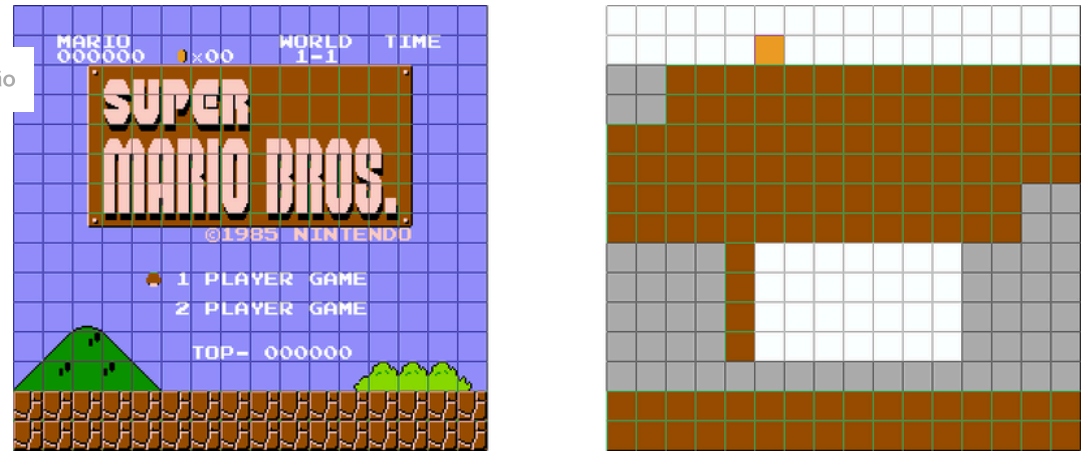
That header mostly exists to separate this text from the code at the end of the previous section. Before we actually talk about the attribute tables, let's draw our nametable with palette 0.

Currently, we're just multiplying the color palette index by 85, which gives us the values 0, 85, 190, and 255. Instead of that, we're going to use the color palette index as the index into the color palette.

```
for (int x = 0; x < 8; x++)
{
    int TwoBit = ((lowByte >> (7 - x)) & 1) == 1 ? 1 : 0;
    TwoBit += ((highByte >> (7 - x)) & 1) == 1 ? 2 : 0;
    Color color = Palette[PaletteRAM[TwoBit]];
    nametable.SetPixel(x + column * 8, y + row * 8, color);
}
```

Now that we're rendering this in color, let's figure out how the PPU determines which tiles use which color palette!

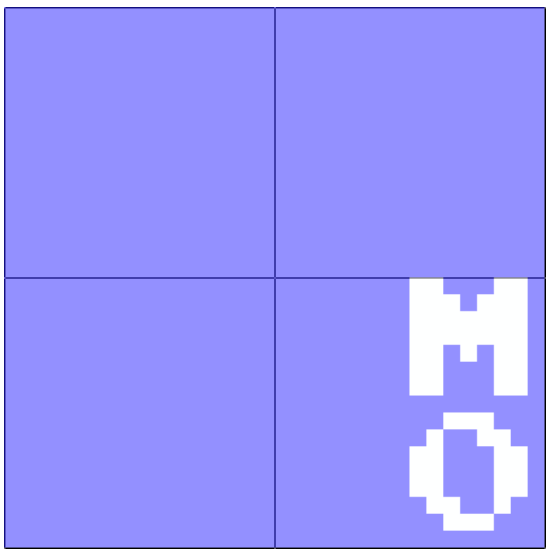
Pular navegação



The title screen of Super Mario Bros. split up into 16x16 pixel boxes. On the right, each box is colored to showcase the palette used.

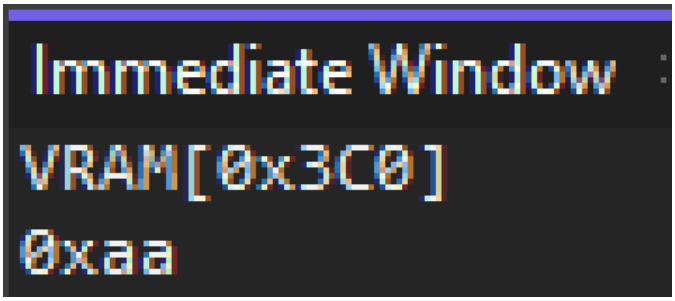
As you can see, each 16x16 pixel box is assigned a color palette! This is where the Attribute Table comes in. The attribute table contains $40 bytes, and each byte assigns a color palette to four 16x16 pixel boxes (in a 2x2 arrangement).

For now, let's just look at the upper left corner of the screen. It's a pretty good place to start since all 4 of the 16x16 boxes use the same palette:



This 32x32 pixel box (containing four 16x16 pixel boxes) will get its color palette info from index $0 of the attribute table. (The attribute table is the final $40 bytes of the $400 byte long nametable. So index $0 of the attribute table would be VRAM Address $23C0, or index $3C0 into our VRAM Array.)

Let's see what's currently at that address:



So this byte has the value $AA. Written as binary, that's %10101010.

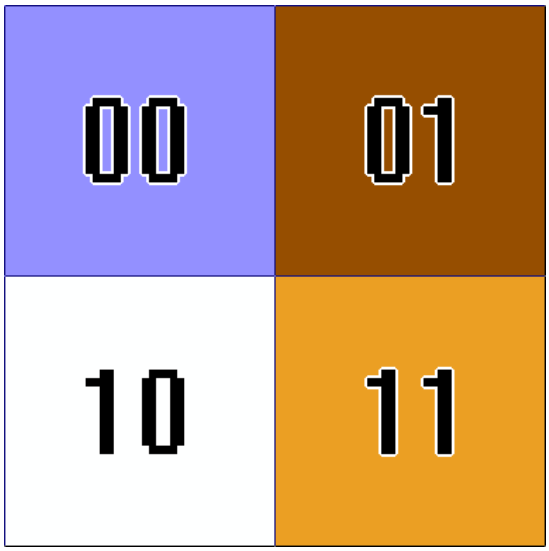
Let's break this binary number into groups of 2-bit values:

10, 10, 10, and 10.

%10 is binary for (decimal) 2, meaning every 16x16 box in this 32x32 pixel region will use color palette 2.

Let's look at a spicier example. Suppose the values were all different: %11 10 01 00

Here's how these 16x16 pixel boxes would be assigned palettes:



So, for a byte with the values %wwxxyyzz, the "zz" pair represents the top left, "yy" represents the top right, "xx" represents the bottom left, and "ww" represents the bottom right.

Using some bitwise math, we can determine what byte to use, what quadrant a given tile is in, take this byte and separate the pair of bits we're looking for, then use that to determine which color palette to use:

```
for (int column = 0; column < 32; column++)
{
    byte attributeOffset = (byte)((column >> 2) + (row>>2) * 8);
    byte Attributes = VRAM[0x3C0 + attributeOffset];
    byte Quadrant = (byte)(((column >> 1) & 1) + ((row >> 1) & 1) * 2);
    byte Pair = (byte)((Attributes >> (Quadrant * 2)) & 3);

    for (int y = 0; y < 8; y++)
    {
```

```
for (int x = 0; x < 8; x++)
{
    int TwoBit = ((lowByte >> (7 - x)) & 1) == 1 ? 1 : 0;
    TwoBit += ((highByte >> (7 - x)) & 1) == 1 ? 2 : 0;
    Color color = Palette[PaletteRAM[TwoBit + Pair * 4]];
    nametable.SetPixel(x + column * 8, y + row * 8, color);
}
```

And just like that, our nametable viewer is in full color!

Uh... oh my.

This is where the backdrop color comes in. Essentially, if we would use index 0 of any color palette, we'll always use color 0 of palette 0.

```
for (int x = 0; x < 8; x++)
{
    int TwoBit = ((lowByte >> (7 - x)) & 1) == 1 ? 1 : 0;
    TwoBit += ((highByte >> (7 - x)) & 1) == 1 ? 2 : 0;
    Color color;
    if (TwoBit == 0)
    {
        color = Palette[PaletteRAM[0]];
    }
    else
    {
        color = Palette[PaletteRAM[TwoBit + Pair * 4]];
    }
    nametable.SetPixel(x + column * 8, y + row * 8, color);
}
```

And just like that, the nametable renderer is looking good!

Just so we're clear, the code we've been using is *not* how we're actually going to render the screen. This exercise was a means to explain how the data is stored in VRAM. So, why not start making the real screen?

The Screen.

So here's the plan:

Inside Emulate_PPU(), we're currently tracking the "dot" and "scanline" as a form of X and Y coordinates. It's now time to write the internal logic of the PPU, such that it can read the CHR data from the cartridge and read the nametable data from VRAM, take this data, and decide how to color each pixel.

Let's try and understand the logic here. There is an 8-step process happening on visible scanlines (and also the final scanline of VBlank, called the pre-render line), where each PPU

cle takes care of a single step, before looping back to the first step again. This process prepares a 1x8 pixel strip of tile information which will be used during the following 8 ppu cycles. This process will:

1: Read a byte from the nametable. This will be the 8x8 pixel character that we're going to draw next.

2: This value read is stored in an internal PPU register.

3: Read a byte from the attribute table. This will determine the color palettes for the upcoming 1x8 pixel strip.

4: Determine which pair of bits to use from the attribute byte.

5: Read from the CHR data of the cartridge. This grabs the "Low Bit Plane" of the character we want to draw.

6: This value read is stored in an internal PPU register for the low bit plane.

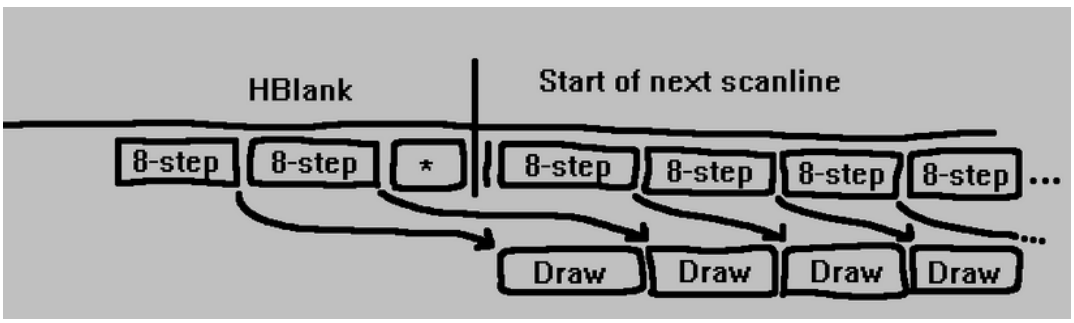
7: Read from the CHR data of the cartridge again. This grabs the "High Bit Plane" of the character we want to draw.

8: This value read is stored in an internal PPU register for the high bit plane. Increment the PPU's VRAM Address.

With all this information prepared, once step 1 runs again next ppu cycle, a series of 16-bit "shift registers" are prepared with the Low Byte pattern data, the High Byte pattern data, the Low bit Attribute data, and the High bit Attribute data.

A "Shift Register" is simply a series of bits which get shifted when "clocked". For instance, these shift registers will be "clocked" just before every step of the 8-step process occurs.

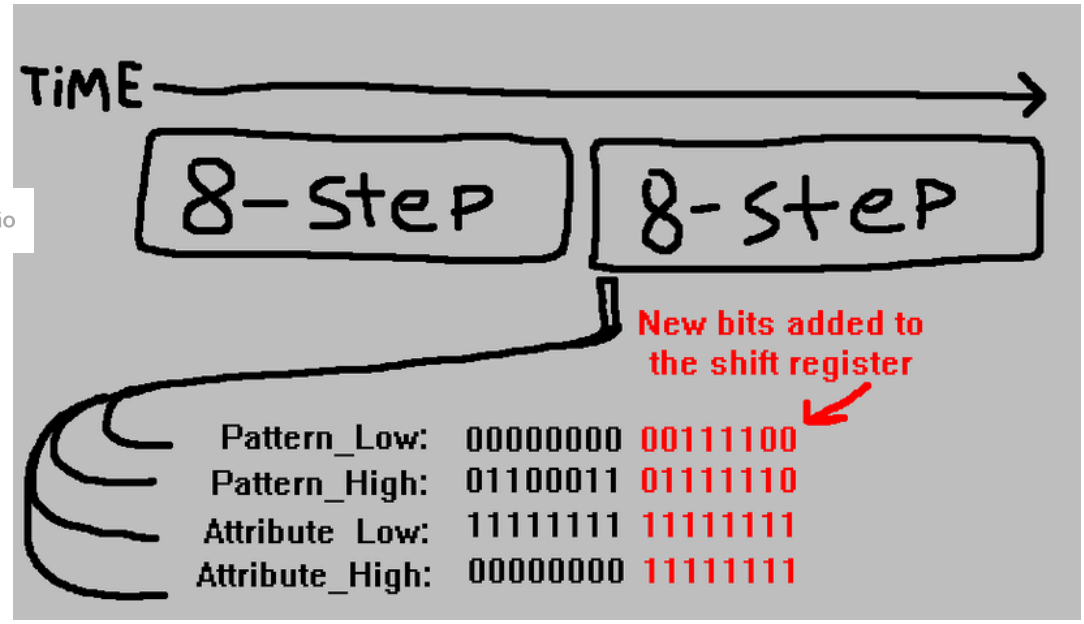
Let's look at a timeline of these events: (For a more detailed timeline, check out this table from the nesdev wiki)



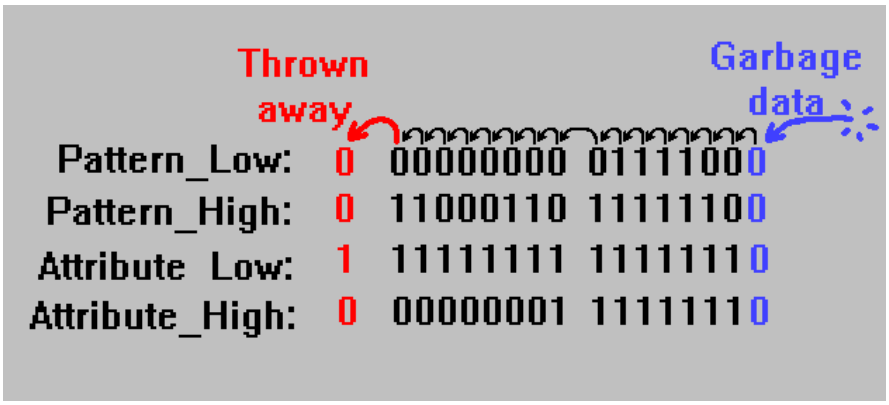
At the very end of the horizontal blank, this 8 step process occurs twice, followed by 4 ppu cycles of unused nametable reading. We probably don't need to implement the unused part. Then, after a single PPU cycle, the 8-step process will begin running until dot 257.

While the 8-step process is occurring, the shift registers are shifting. This essentially creates a 16-ppu cycle delay between step 1 filling the shift register with data, and that data being drawn on screen in the form of a colored pixel.

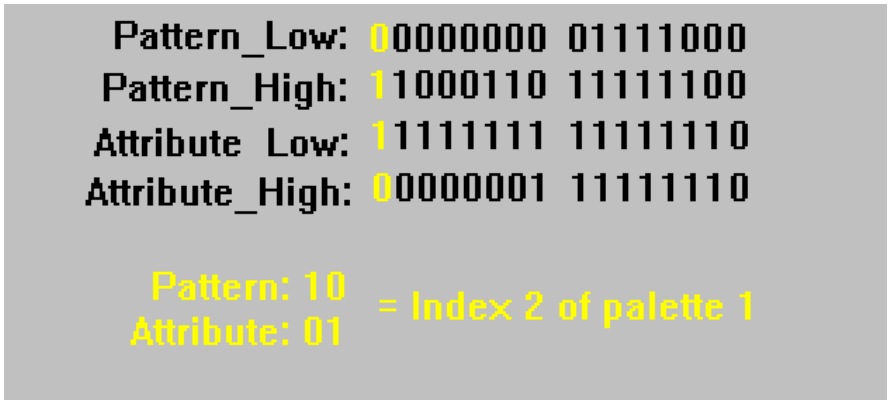
Here's a visual of the shift register being filled in:



And then the bits get shifted. Bit 15 is discarded, and bit 0 is fed in with garbage data (these garbage bytes will be overwritten during step 1 of the next 8-step process.)



And now, to determine what color to make the pixel at the current (dot, scanline) coordinates, we take the leftmost bit of the shift registers, where the attributes determine what color palette to use, and the pattern determines the index into the color palette:



And that's how the background is drawn!

Let's start implementing this 8-step process!

Pular navegação probably doesn't matter all that much where inside Emulate_PPU() you put this, but I'm adding it towards the beginning of the function. before I increment ppuDot.

Let's make in if statement determining if we are on a visible scanline (or the pre-render line)

```
void Emulate_PPU()
{
    if(ppuDot == 1 && ppuScanline == 241)
    {
        ppuVBlank = true;
    }
    else if (ppuDot == 1 && ppuScanline == 261)
    {
        ppuVBlank = false;
    }

    if ((ppuScanline < 240 || ppuScanline == 261))
    {

    }
```

Now let's check if this is one of the dots in which the 8-step process occurs.

```
if ((ppuScanline < 240 || ppuScanline == 261))
{
    // if this is a visible scanline, or the pre-render line.
    if ((ppuDot > 0 && ppuDot <= 256) || (ppuDot > 320 && ppuDot <= 336))
    {
        // if this is a visible pixel, or preparing the start of next scanline

    }
}
```

This 8-step process only occurs if rendering is enabled. Surprisingly, even if *only* sprites are being rendered, this process still occurs.

```
if ((ppuDot > 0 && ppuDot <= 256) || (ppuDot > 320 && ppuDot <= 336))
{
    // if this is a visible pixel, or preparing the start of next scanline
    if(ppuMask_RenderBG || ppuMask_RenderSprites)
    {
        // if rendering is enabled

    }
}
```

If the background is being rendered, let's shift all the shift registers to the left. (you'll also need to create these shift registers. They can just be ushorts. (16-bit))

```
if(ppuMask_RenderBG || ppuMask_RenderSprites)
{
    // if rendering is enabled
    if (ppuMask_RenderBG) // if rendering the backgound, update the shift registers for the background.
    {
        ppuShiftRegister_patternL = (ushort)(ppuShiftRegister_patternL << 1); // shift 1 bit to the left.
        ppuShiftRegister_patternH = (ushort)(ppuShiftRegister_patternH << 1); // shift 1 bit to the left.
        ppuShiftRegister_attributeL = (ushort)(ppuShiftRegister_attributeL << 1); // shift 1 bit to the left.
        ppuShiftRegister_attributeH = (ushort)(ppuShiftRegister_attributeH << 1); // shift 1 bit to the left.
    }
}
```


Now let's make the switch statement to determine which step of the 8-step process should run this PPU cycle.

```
byte cycleTick; // what part of the 8-step process to run this cycle
cycleTick = (byte)((ppuDot - 1) & 7);
switch(cycleTick)
{
    case 0:
        break;
    case 1:
        break;
    case 2:
        break;
    case 3:
        break;
    case 4:
        break;
    case 5:
        break;
    case 6:
        break;
    case 7:
        break;
}
```

The first step of case 0 will be to set up these shift registers. We'll need to make variables for the low bit plane, high bit plane, and attributes that will be determined during the 8-step process.

```
case 0:
    ppuShiftRegister_patternL = (ushort)((ppuShiftRegister_patternL & 0xFF00) | ppu8Step_patternLowBitPlane);
    ppuShiftRegister_patternH = (ushort)((ppuShiftRegister_patternH & 0xFF00) | ppu8Step_patternHighBitPlane);
    ppuShiftRegister_attributeL = (ushort)((ppuShiftRegister_attributeL & 0xFF00) | ((ppu8Step_attribute & 1) == 1 ? 0xFF : 0));
    ppuShiftRegister_attributeH = (ushort)((ppuShiftRegister_attributeH & 0xFF00) | ((ppu8Step_attribute & 2) == 2 ? 0xFF : 0));
    break;
```

Don't worry about the fact that the first time case 0 runs, these values will be uninitialized. This is why the 8-step process briefly occurs in H-Blank before the visible parts of the screen. Whatever garbage values are put in the shift registers the first time this runs will be gone before the screen starts being drawn.

We're still updating case 0 here, but the next step involves reading from the PPU address space. I recommend cutting/pasting the logic we have inside read() for PPUDATA into it's own function. I'll call it ReadPPU()

This is also super important, but make sure the read buffer is only changed inside Read() and NOT ReadPPU().


```
case 0x2007:
    byte temp = PPUReadBuffer;
    if(VRAMAddress > 0x3F00)
    {
        temp = ReadPPU(VRAMAddress);
    }
    else
    {
        PPUReadBuffer = ReadPPU(VRAMAddress);
    }
    VRAMAddress += (ushort)(ppuVRAMInc32Mode ? 32 : 1);
    VRAMAddress &= 0x3FFF;
    return temp;
```

```
byte ReadPPU(ushort Address)
{
    if (Address < 0x2000)
    {
        // Read from the pattern table.
        return CHRData[Address];
        //else, nothing happens.
    }
    else if (Address < 0x3F00)
    {
        // Read from the Nametables.
        if ((Header[6] & 1) == 0)
        {
            // "horizontal mirroring"
            return VRAM[(Address & 0x3FF) | (Address & 0x800) >> 1];
        }
        else
        {
            // "vertical mirroring"
            return VRAM[Address & 0x7FF];
        }
    }
    else
    {
        // Read Palette RAM
        if ((Address & 3) == 0)
        {
            return PaletteRAM[Address & 0x0F];
        }
        else
        {
            return PaletteRAM[Address & 0x1F];
        }
    }
}
```

This allows us to re-use the ReadPPU function during this 8-step process's logic.

```
case 0:
    ppuShiftRegister_patternL = (ushort)((ppuShiftRegister_patternL & 0xFF00) | ppu8Step_patternLowBitPlane);
    ppuShiftRegister_patternH = (ushort)((ppuShiftRegister_patternH & 0xFF00) | ppu8Step_patternHighBitPlane);
    ppuShiftRegister_attributeL = (ushort)((ppuShiftRegister_attributeL & 0xFF00) | ((ppu8Step_attribute & 1) == 1 ? 0xFF : 0));
    ppuShiftRegister_attributeH = (ushort)((ppuShiftRegister_attributeH & 0xFF00) | ((ppu8Step_attribute & 2) == 2 ? 0xFF : 0));
    ppuAddressBus = (ushort)(0x2000 + (VRAMAddress & 0x0FFF));
    ppu8Step_temp = ReadPPU(ppuAddressBus);
    break;
```

For case 1, we're going to transfer this temp value to a new varaible that will hold this byte for future use.

```
case 1:
    ppu8Step_NextCharacter = ppu8Step_temp;
    break;
```

Case 2 will perform some bitiwse math on the current VRAMAddress to determine what byte of the attribute table to read.

```
case 2:
    ppuAddressBus = (ushort)(0x23C0 | (VRAMAddress & 0x0C00) | ((VRAMAddress >> 4) & 0x38) | ((VRAMAddress >> 2) & 0x07));
    ppu8Step_temp = ReadPPU(ppuAddressBus);
    break;
```

case 3 will transfer the temp value to ppu8Step_attribute, then perform some bitwise math on it to keep just the 2 bits we're looking for with the current tile we want to draw.


```
case 3:
    ppu8Step_attribute = ppu8Step_temp;
    // 1 byte of attribute data covers 4 tiles. determine which tile this is for.
    if ((VRAMAddress & 3) >= 2) // If this is on the right tile
    {
        ppu8Step_attribute = (byte)(ppu8Step_attribute >> 2);
    }
    if ((((VRAMAddress & 0b0000001111100000) >> 5) & 3) >= 2) // If this is on the bottom tile
    {
        ppu8Step_attribute = (byte)(ppu8Step_attribute >> 4);
    }
    ppu8Step_attribute = (byte)(ppu8Step_attribute & 3);
    break;
```

case 4 will read from the pattern table. We need to know what character to read (what was written in case 1) and what pattern table to read from (a value that we implemented when writing to address $2000, PPUCTRL)

We're reading the low bit plane of the character. The $y000 digit of the VRAM address will determine which row of the character to read. You can think of it as reading from CHR ROM at NextCharacter*16 + y.

```
case 4:
    ppuAddressBus = (ushort)(((VRAMAddress & 0b0111000000000000) >> 12) | ppu8Step_NextCharacter * 16 | (ppuBGPatternTable ? 0x1000 : 0));
    ppu8Step_temp = ReadPPU(ppuAddressBus);
    break;
```

case 5 transfers the temp value into the low bti plane, then adds 8 to the PPU address bus.

```
case 5:
    ppu8Step_patternLowBitPlane = ppu8Step_temp;
    ppuAddressBus += 8;
    break;
```

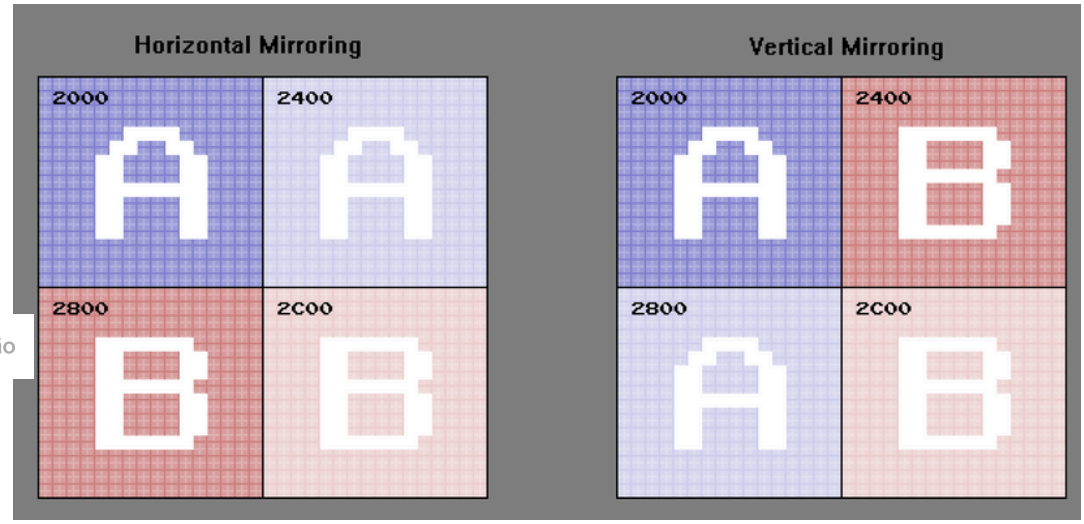
case 6 and 7 simply read from thew new Address Bus value, then transfer the temp value into thie high bit plane:

```
case 6:
    ppu8Step_temp = ReadPPU(ppuAddressBus);
    break;
case 7:
    ppu8Step_patternHighBitPlane = ppu8Step_temp;
    break;
```

We're not quite done with case 7 yet, as we need to increment the VRAM Address in preparation for the next 8-step process.

This will require a slight detour into implementing some of the screen scrolling functionality.

Since each character is 8 pixels wide, every 8 PPU cycles requires the VRAM address to be incremented so we can read the next character. However, this process of incrementing the VRAM address *allows crossing nametables*. Remember how we needed to implement horizontal and vertical mirroring for the nametables, resulting in 2 blocks for the first nametable and two blocks for the second nametable? Let me pull up that graphic again...



Pular navegação

The VRAM address being incremented during the 8-step process needs to be able to cross from the $2000 box into the $2400 box, and vice versa. This will make more sense once we implement scrolling the screen, but here's how I suggest implementing this "increment".

```
case 7:
    ppu8Step_patternHighBitPlane = ppu8Step_temp;
    if ((VRAMAddress & 0x001F) == 31)
    {
        VRAMAddress &= 0xFFE0; // resetting the scroll
        VRAMAddress ^= 0x0400; // crossing into next nametable
    }
    else
    {
        VRAMAddress++; // increment
    }
    break;
```

The VRAM Address needs to be automatically updated at the end of every scanline to prepare for drawing the next scanline. Somewhere after this 8-step process in our Emulate_PPU() routine (but before we run ppuDot++;), let's check for some specific dots in which we'll increment the Y scroll, reset the X scroll, and reset the Y scroll.

```
// If this is a visible scanline and rendering sprites / background is enabled:
if (ppuDot == 256) //The Y scroll is incremented on dot 256.
{
    PPU_IncrementScrollY();
}
else if (ppuDot == 257) //The X scroll is reset on dot 257.
{
    PPU_ResetXScroll();
}
if (ppuDot >= 280 && ppuDot <= 304 && ppuScanline == 261) //numbers from the nesdev wiki
{
    PPU_ResetYScroll(); //The Y scroll is reset on every dot from 280 through 304 on the pre-render scanline.
}
```

PPU_IncrementScrollY()

Since each character is 8 scanlines tall, we need to know how many scanlines deep we are into drawing this specific character. Since the 8-step process only uses the lower 12 bits of VRAMAddress, we can track how many scanlines into a character we've drawn using the upper 4 bits.

So every scanline, increment VRAMAddress by $1000. If we would increment from $7xxx to $8xxx, instead, reset to $0xxx, and then we need to perform some math to determine the new value of VRAM address.

There are 0x20 tiles per scanline, so we'll need to add $20 to the VRAM address, but what would happen if we are crossing from one nametable to another? We're going to need to prevent the attribute tables from being drawn as tile data as well, so if we're moving from row 29 of the nametable into row 30 (the attribute table) instead, we'll need to skip 2 lines and go directly into the next nametable.

 › work this out, let's pull the Y position from the VRAM Address. That would be a 5 bit value from bit 5 to bit 9 of VRAMAddress.

```
int y = (VRAMAddress & 0x03E0) >> 5;
```

Now we can chck what row we're on.

If we're on row 29, we're incrementing Y into a new nametable! At which point, we'll set Y back to 0, and flip bit 11 of the VRAMAddress.

```
if (y == 29)
{
    y = 0; // reset the Y value and also flip some other bit in the 'v' register
    VRAMAddress ^= 0x0800;
}
```

Otherwise, simply increment y, and make sure it's limited to a 6 bit value.

```
else
{
    y++; // increment the Y value
    y &= 0x1F;
}
```

At this point, we just need to shove y back into the VRAM Address, and we've completed our IncrementScrollY() function:

```
void PPU_IncrementScrollY()
{
    if ((VRAMAddress & 0x7000) != 0x7000)
    {
        VRAMAddress += 0x1000;
    }
    else
    {
        VRAMAddress &= 0x0FFF;
        int y = (VRAMAddress & 0x03E0) >> 5;
        if (y == 29)
        {
            y = 0; // reset the Y value and also flip some other bit in the 'v' register
            VRAMAddress ^= 0x0800;
        }
        else
        {
            y++; // increment the Y value
            y &= 0x1F;
        }
        VRAMAddress = (ushort)((VRAMAddress & 0xFC1F) | (y << 5));
    }
```


PPU_ResetXScroll()

This one is remarkably simple. Remember how writing to $2006 also prepares the PPU's ` t` register, or "Transfer Address"? Here's where that comes into play!

We need to copy bits 0 to 4, and bit 10 from the transfer address and paste it into bits 0 to 4 and bit 10 of the VRAM address!

```
void PPU_ResetXScroll()
{
    VRAMAddress = (ushort)((VRAMAddress & 0b0111101111100000) | (TransferAddress & 0b0000010000011111));
}
```

And that's it!

PPU_ResetYScroll()

Just like the Reset X scroll function, this is a single line where we copy values from the Transfer Address.

```
void PPU_ResetYScroll()
{
    VRAMAddress = (ushort)((VRAMAddress & 0b0000010000011111) | (TransferAddress & 0b0111101111100000));
}
```

And that's the whole function!

PPUSCROLL

One final part of this PPU scroll detour will be setting up the PPU register at CPU Address $2005.

Remember how writing to $2006 took two writes to set up the 14 bit address? Writing to $2005 also takes two writes. One to set up the fine y position, and one to set up the fine x position.

This will also re-use the WriteLatch variable we used with $2006.

Just like the ResetScroll functions we just made, we're going to be poking very specific bits, but this time we're just changing the Transfer Address.

We're also going to need to set up the ppu's x register, which I called ppuScrollFineX.

```
case 0x2005: //PPUSCROLL
    if (!WriteLatch)
    {
        ppuScrollFineX = (byte)(Value & 7);
        TempVRAMAddress = (ushort)((TempVRAMAddress & 0b0111111111100000) | (Value >> 3));
    }
    else
    {
        TransferAddress = (ushort)((TempVRAMAddress & 0b0000110000011111) | (((Value & 0xF8) << 2) | ((Value & 7) << 12)));
    }
    WriteLatch = !WriteLatch;

    break;
```

Okay, let's summarize all this scroll stuff.

So, by writing to $2006 and $2005, we set up the TransferAddress. This value will be transferred to VRAMAddress before a visible scanline. The VRAMAddress is used to read from the nametables to determine which character to draw on screen as well as what corresponding attribute table byte needs to be read in order to know what color palette to use. After the low and high bit planes are read from the CHR ROM, the VRAM address is incremented, ready to read the tile to the right of the one that was just read.

At the end of every scanline, the X position of the VRAMAddress is reset by copying values from the TransferAddress, then the VRAMAddress is incremented by $1000. This makes the following scanline read from the same tiles as before, but from the next 1x8 strip of bits one row downwards.

Then for the next frame, at the end of the pre-render line, the Y position of the VRAMAddress is reset by copying the remaining bits from the TransferAddress.

Let's draw something on screen!

And now we're ready to draw the screen.

The screen is drawn on dots 1 to 256 of scanline 0 to 240:

Keep in mind, these lines of code will run even if rendering is disabled!

```
if (ppuScanline < 241 && ppuDot > 0 && ppuDot <= 256)
{

}
```

We're going to need to calculate the high and low parts of the color palette. Let's start by setting these to a value of 0, which will be the backdrop color.

```
if (ppuScanline < 241 && ppuDot > 0 && ppuDot <= 256)
{
    byte PalHi = 0; // which color palette to use?
    byte PalLow = 0; // index into a color palette
}
```

If we aren't rendering anything, then we'll simply draw the backdrop color on screen.

However, if we *are* rendering the background, we need to determine the palette stuff. This is why we set up PPUMASK earlier. On that note, PPUMASK also had that "8 pixel mask" that would leave the 8 leftmost pixels of the screen using the backdrop color. This is where we'll implement that:

```
if (ppuScanline < 241 && ppuDot > 0 && ppuDot <= 256)
{
    byte PalHi = 0; // which color palette to use?
    byte PalLow = 0; // index into a color palette
    if (ppuMask_RenderBG && (ppuDot > 8 || ppuMask_8pxMaskBG))
    {

    }
}
```

So, if we are rendering this pixel, let's pull from our shift registers!

Instead of simply making the leftmost bit of the shift register the bit we read here, we're going to take the bit that is "fineXScroll" bits to the right of the leftmost bit. That will allow the screen scroll smoothly, so the leftmost pixel of the screen doesn't need to be aligned with the ftmost pixel of a tile.


```
if (ppuMask_RenderBG && (ppuDot > 8 || ppuMask_8pxMaskBG))
{
    byte col0 = (byte)(((ppuShiftRegister_patternL >> (15 - ppuScrollFineX))) & 1);
    byte col1 = (byte)(((ppuShiftRegister_patternH >> (15 - ppuScrollFineX))) & 1);
    PalLow = (byte)((col1 << 1) | col0);

    byte pal0 = (byte)(((ppuShiftRegister_attributeL) >> (15 - ppuScrollFineX)) & 1);
    byte pal1 = (byte)(((ppuShiftRegister_attributeH) >> (15 - ppuScrollFineX)) & 1);
    PalHi = (byte)((pal1 << 1) | pal0);
}
```

Don't forget to make sure index 0 of all color palettes are mirrors of the backdrop color!

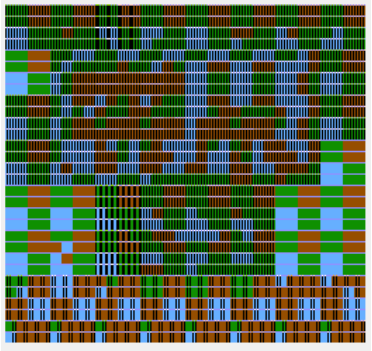
```
PalHi = (byte)((pal1 << 1) | pal0);

if (PalLow == 0 && PalHi != 0) // color 0 of all palettes are mirrors of color 0 of palette 0
{
    PalHi = 0;
}
```

And now we can simply set the color at coordinates (ppuDot-1, ppuScanline) of a bitmap equal to the color assigned to index "PalLow" of palette "PalHi".

And now, let's run the game and display the screen after the emulator halts. (again, there won't be a HLT isntruction, so I'm just waiting for 250,000 CPU instructions to complete.)

Beautiful isn't it?

By the way, I mentioned it earlier, but in case you forgot to make the change at the time, make sure our new ReadPPU() function doesn't update the buffer, and that we're handling that inside Read() still, otherwise your screen might look like this: (HA!)



Not ideal.

Just like the CPU, there are countless ways something could go wrong. A single mistyped character in that 8-step process, or the scroll resetting functions, or maybe the 8-step process is happening at the wrong time or when rendering should be disabled.... anything could break the delicate balance required to properly draw the screen. I know we haven't used any test ROMs in a while, and there has been a lot of work between setting up the nametable viewer and setting up the screen. If your screen isn't looking correct, I wish you the best of luck in debugging it.

Frame Advance

If you want to actaully watch the screen while playing the game (I imagine most people would) we're going to need to update the screen in our application as the emulator is running. Once this is set up, we can remove the function to abort after 250,000 CPU instruction too!

In the code for my winforms application, I'll create a loop to run my emulator and update the screen.

```
while(true)
{
    emulator.Run();
    pictureBox1.Image = emulator.Screen;
    pictureBox1.Update();
}
```

And I'll modify Run() to exit whenever it's time to draw a new frame:

Pular navegação

```
bool DrawNewFrame = false;
1 reference
public void Run()
{
    while (!CPU_Halted)
    {
        Emulate_CPU();
        if(DrawNewFrame)
        {
            DrawNewFrame = false;
            break;
        }
    }
}
```

And finally, at the moment VBlank begins, I'll set "DrawNewFrame" to true:

```
void Emulate_PPU()
{
    if(ppuDot == 1 && ppuScanline == 241)
    {
        ppuVBlank = true;
        DrawNewFrame = true;
    }
```

If everything is set up correctly, you should be able to watch the coin in the HUD blink!

Sprite Evaluation

Sprite evaluation is the process of taking values in "Object Attribute Memory" and translating that into the sprites that get drawn on screen. The natural starting point would be to set up Object Attribute Memory.

```
byte[] OAM = new byte[0x100];
```

The way the NES writes to "OAM" is actually incredibly complicated, so we're going to cut a bunch of corners and simplify.

When you write to CPU address $4014, 256 bytes will be copied from memory and pasted into OAM. The 256 bytes that get copied depend on the value written to $4014. For isntance, if you write a value of $02, the bytes from $0200 to $02FF get copied. If you write a value of $07, then $0700 to $07FF get copied.

```
if(Address == 0x4014)
{
    // OAM DMA! (but we're cutting corners.)
    // Instead of actually performing the DMA,
    // let's just copy/paste everything from here.
    for(int i = 0; i < 256; i++)
    {
        OAM[i] = Read((ushort)((Value << 8) + i));
    }
}
```

This is hardly accurate, but it will still successfully set up OAM, which is good enough for your first emulator!

Pular navegação kay, so what exactly *is* OAM, and how does it translate to sprites on screen?

OAM is composed of 64 "objects" composed of 4 bytes. Each Object has: (in this order)

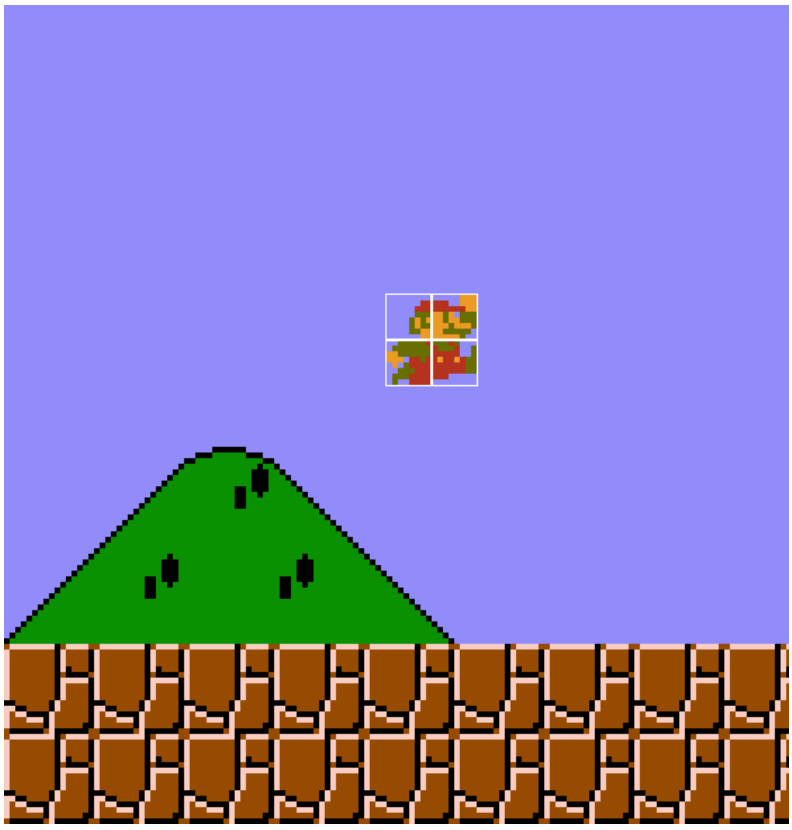
A Y position.

A Pattern.

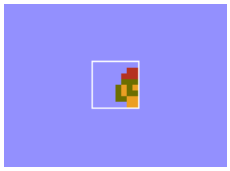
Attribute data. (should the object be horizontally or vertically flipped? In front or behind the background? What palette should this object use?)

An X position.

Let's see an example.



In this image, Mario is composed of 4 objects. Let's focus on just one of these objects:



This object has 4 bytes, as mentioned above. They are: $91, $32, $00, and $44

- This means that this object is at Y position $91 (the top of the screen is $00)
- This object is using index $32 of the pattern table.
- This attribute byte is set up like this:
  - bits 0 and 1 determine the color palette. (in this case, palette 0)
  - Bits 2, 3, and 4 don't do anything.
  - bit 5 determines if the object should be rendered over the background pixels, or under the background pixels. (a value of 0 means this is over the background)
  - bit 6 determines if this object should be horizontally flipped. (0 means not flipped)
  - bit 7 determines if this should be vertically flipped. (0 means not flipped)
- And this object is at X position $44.

Pular navegação

You might have noticed that there's not really any way to toggle if an object should be drawn or not, suggesting that all 64 objects will attempt to be drawn. Since it's pretty clear that this game doesn't draw 64 objects on every frame, you might begin wondering how to avoid having 64 objects on screen.

It's easy! Since the screen is only 240 pixels tall, you can simply hide objects off-screen at Y coordinates greater than or equal to 240. Super Mario Bros. for instance, hides objects at a Y coordinate of $F8 (or decimal, 248).

So that's the OAM format, but how does the PPU actually take this data and draw the characters?

On any given visible scanline with rendering enabled, the process works like this:

The sprites we evaluate on scanline n will be drawn on scanline n+1.

There can be up to 8 sprites per scanline, so let's create an array that can hold $20 bytes.

```
byte[] SecondaryOAM = new byte[0x20];
```

Step 1: Clear secondary OAM such that every value is $FF.

Step 2: initialize the "OAM Address" with the value $00. This will be the index into OAM which we read from.

Step 3: The first byte in OAM is a Y position, so let's read this byte.

Step 4: We need to check if this Y position would result in an object being drawn on the current scanline. If the object is *not* on this scanline, add 4 to that OAM address, so we can read from the Y position of the *next* object.

Step 5: If this object *is* in range of this scanline, we need to copy the 4 bytes in OAM for this object into SecondaryOAM. Then move on to the next object's Y position and repeat steps 4 and 5. This process occurs until secondaryOAM is full, or until the OAM address wraps around back to address $00.

Step 6: With the contents of SecondaryOAM all figured out, we need to prepare a series of shift registers with another 8-step process. Except unlike the background, where the shift registers were immediately used, these shift registers are clocked on the *following* scanline.

Let's start working this out. I'm going to put this right before the 8-step process for the background. Since it's a bit of a doozy, I'm giving this its own function just for organization.

```
void SpriteEvaluation()
{

}
```


:ep 1 occurs from ppu dot 1 through 64, and this clears secondaryOAM:

```
void SpriteEvaluation()
{
    if(ppuDot > 0 && ppuDot <= 64)
    {
        if((ppuDot & 1) == 1)
        {
            // odd ppu cycles load the value $FF
            ppuSpriteEvalTemp = 0xFF;
        }
        else
        {
            // even ppu cycles store the value in secondaryOAM
            SecondaryOAM[ppuDot >> 1] = ppuSpriteEvalTemp;
        }
    }
}
```

We're going to need to keep track of how many bytes we've placed in secondaryOAM, so you might even implement step 1 like this:

```
if (ppuDot == 0)
{
    ppuSecondaryOAMAddress = 0;
    ppuSecondaryOAMFull = false;
}
else if (ppuDot > 0 && ppuDot <= 64)
{
    if((ppuDot & 1) == 1)
    {
        // odd ppu cycles load the value $FF
        ppuSpriteEvalTemp = 0xFF;
    }
    else
    {
        // even ppu cycles store the value in secondaryOAM
        SecondaryOAM[ppuSecondaryOAMAddress] = ppuSpriteEvalTemp;
        ppuSecondaryOAMAddress++;
        ppuSecondaryOAMAddress &= 0x1F; // keep this limited from $00 to $1F
    }
}
```

Step 2 occurs between dots 65 and 256.

Let's start by reading from OAM on odd cycles:

```
else if (ppuDot > 64 && ppuDot <= 256)
{
    if ((ppuDot & 1) == 1)
    {
        // odd ppu cycles load the value from OAM
        ppuSpriteEvalTemp = OAM[ppuOAMAddress];
    }
    else
    {
        // even ppu cycles store the value in secondaryOAM

    }
}
```


And we evaluate it on even cycles. As long as secondaryOAM isn't full, we're always going to write the value we just read on the odd cycle into the current secondaryOAM index.

Make a "Sprite Evaluation Tick" variable. This will determine if we are evaluating the Y position, or reading the pattern / attributes / X position.

To determine if an object is on the current scanline, let's subtract the Y position from the current scanline. If the result is between 0 and 8 (or 16 if the PPU is using 8x16 pixel sprites) then it's on this scanline!

```
if ((ppuDot & 1) == 1)
{
    // odd ppu cycles load the value from OAM
    ppuSpriteEvalTemp = OAM[ppuOAMAddress];
}
else
{
    // even ppu cycles store the value in secondaryOAM
    if (!ppuSecondaryOAMFull) // If secondary OAM is not yet full,
    {
        // as long as secondaryOAM isn't full, this write *always* occurs, regardless of evaluation.
        SecondaryOAM[ppuSecondaryOAMAddress] = ppuSpriteEvalTemp; // store this value at the secondary oam address.
    }
    if(ppuSpriteEvalTick == 0)
    {
        // reading index 0 of an object's set of four bytes.
        if (ppuScanline - ppuSpriteEvalTemp >= 0 && ppuScanline - ppuSpriteEvalTemp < (ppuUse8x16Sprites ? 16 : 8))
        {
            // This object *is* on this scanline!
        }
    }
}
```

Here's where we're going to implement the remaining bits of PPUSTATUS (reading from CPU address $2002)

```
bool ppuStatusOverflow;
bool ppuStatusSprZeroHit;
```

We're going to keep track of the secondaryOAM Address, and when we fill in address $1F, we toggle "SecondaryOAMFull". And if we just evaluated another object on this scanline, but secondaryOAM is already full, we're going to set the ppuStatusOverflow flag.

```
// This object *is* on this scanline!
if (!ppuSecondaryOAMFull)
{

}
else
{
    ppuStatusOverflow = true;
}
```

Pular navegação

Sprite Zero" is the first sprite to be evaluated on a scanline. (index 0 of OAM)

The PPU also has a flag called a "Sprite Zero Hit", which occurs if a non-backdrop pixel of a sprite overlaps a non-backdrop pixel of the background. **We are NOT setting the sprite zero hti flag here during sprite evaluation**, but we *do* need to toggle a flag letting the PPU know that index 0 of secondary OAM is sprite zero. When we draw all the sprites on screen, we can check if we're drawing sprite zero, at which point we will be able to set the sprite zero hit flag. But for now, let's just make a flag telling us that Sprite Zero will be drawn on the next scanline.

```
// This object *is* on this scanline!
if (!ppuSecondaryOAMFull)
{
    ppuSecondaryOAMAddress++; // increment this for the next write to secondary OAM
    ppuOAMAddress++; // increment this for next read of Object Attribute Memory.
    if (ppuDot == 66)
    {
        // Rather than verifying that this is OAM index 0,
        // the ppu sets this flag if we found an object on this scanline
        // during ppu dot 66, which would be the ppu cycle evaluating index 0.
        ppuScanlineContainsSpriteZero = true;
    }
}
else
{
    ppuStatusOverflow = true;
}
ppuSpriteEvalTick++;
```

Increment the "Sprite Evaluation Tick" variable, indicating that we need to read the next byte from this object.

```
else
{
    ppuStatusOverflow = true;
}
ppuSpriteEvalTick++;
```

And finally, if this object was *not* on this scanline, we can increment the OAM address by 4, ready to read the Y position of the next object.

```
// reading index 0 of an object's set of four bytes.
if (ppuScanline - ppuSpriteEvalTemp >= 0 && ppuScanline - ppuSpriteEvalTemp < (ppuUse8x16Sprites ? 16 : 8))
{
    // This object *is* on this scanline!
    if (!ppuSecondaryOAMFull)
    {
        ppuSecondaryOAMAddress++; // increment this for the next write to secondary OAM
        ppuOAMAddress++; // increment this for next read of Object Attribute Memory.
        if (ppuDot == 66)
        {
            // Rather than verifying that this is OAM index 0,
            // the ppu sets this flag if we found an object on this scanline
            // during ppu dot 66, which would be the ppu cycle evaluating index 0.
            ppuScanlineContainsSpriteZero = true;
        }
    }
    else
    {
        ppuStatusOverflow = true;
    }
    ppuSpriteEvalTick++;
}
else
{
    ppuOAMAddress += 4;
```


So that was tick 0, but what about ticks 1, 2, and 3? Well, we really only need to check if the Y position is on this scanline. The rest of the bytes can simply get written to secondary OAM, and we just need to increment the OAM addresses.

This is also the place to check if secondary OAM was just filled:

```
else // if ppuSpriteEvalTick != 0:
{
    // reading index 1, 2, or 3 of an object's OAM data.
    // we're not going to be making any checks for if things are on this scanline,
    // so we can just simply increment the oam addresses.
    ppuSecondaryOAMAddress++; // increment this for the next write to secondary OAM
    ppuOAMAddress++; // increment this for next read of Object Attribute Memory.
    if(ppuSecondaryOAMAddress == 0x20)
    {
        ppuSecondaryOAMFull = true;
    }
    ppuSpriteEvalTick++;
    ppuSpriteEvalTick &= 3; // wrap around to tick 0 after tick 3.
}
```

Finally, let's check if the OAM address wraps around to 0, at which point we stop evaluating sprites until dot 257:

```
    ppuSpriteEvalTick++;
    ppuSpriteEvalTick &= 3; // wrap around to tick 0 after tick 3.
}
if (ppuOAMAddress == 0)
{
    // if we overflow the OAM address, we want to stop running the sprite evaluation checks until dot 257.
    ppuSpriteEvaluationOAMOverflowed = true;
}
```

We can now wrap this entire section in a ` if (!ppuSpriteEvaluationOAMOverflowed)` statement.

```
else if (ppuDot > 64 && ppuDot <= 256)
{
    if ((ppuDot & 1) == 1)
    {
        // odd ppu cycles load the value from OAM
        ppuSpriteEvalTemp = OAM[ppuOAMAddress];
    }
    else
    {
        if (!ppuSpriteEvaluationOAMOverflowed)
        {
            // even ppu cycles store the value in secondaryOAM
            if (!ppuSecondaryOAMFull) // If secondary OAM is not yet full,
            {
                // as long as secondaryOAM isn't full, this write *always* occurs, regardless of evaluation.
                SecondaryOAM[ppuSecondaryOAMAddress] = ppuSpriteEvalTemp; // store this value at the secondary oam address.
            }
            if (ppuSpriteEvalTick == 0)
            {
                // reading index 0 of an object's set of four bytes.
                if (ppuScanline - ppuSpriteEvalTemp >= 0 && ppuScanline - ppuSpriteEvalTemp < (ppuUse8x16Sprites ? 16 : 8))
                {
                    // This object *is* on this scanline!
                    if (!ppuSecondaryOAMFull)
                    {
                        ppuSecondaryOAMAddress++; // increment this for the next write to secondary OAM
                        ppuOAMAddress++; // increment this for next read of Object Attribute Memory.
                        if (ppuDot == 66)
                        {
                            // Rather than verifying that this is OAM index 0,
                            // the ppu sets this flag if we found an object on this scanline
                            // during ppu dot 66, which would be the ppu cycle evaluating index 0.
                            ppuScanlineContainsSpriteZero = true;
                        }
                    }
                    else
                    {
                        ppuStatusOverflow = true;
                    }
                    ppuSpriteEvalTick++;
                }
                else
                {
                    ppuOAMAddress += 4;
                }
            }
            else // if ppuSpriteEvalTick != 0:
            {
                // reading index 1, 2, or 3 of an object's OAM data.
                // we're not going to be making any checks for if things are on this scanline,
                // so we can just simply increment the oam addresses.
                ppuSecondaryOAMAddress++; // increment this for the next write to secondary OAM
                ppuOAMAddress++; // increment this for next read of Object Attribute Memory.
                if (ppuSecondaryOAMAddress == 0x20)
                {
                    ppuSecondaryOAMFull = true;
                }
                ppuSpriteEvalTick++;
                ppuSpriteEvalTick &= 3; // wrap around to tick 0 after tick 3.
            }
            if (ppuOAMAddress == 0)
            {
                // if we overflow the OAM address, we want to stop running the sprite evaluation checks until dot 257.
                ppuSpriteEvaluationOAMOverflowed = true;
            }
        }
    }
}
```

Up next, ppu dot 257 to dot 320:

```
else if(ppuDot > 256 && ppuDot <= 320)
{

}
```

Let's make a variable to track how many objects are in secondaryOAM. On dot 257, let's initialize the new size variable, and then reset the secondary OAM address and Sprite Evaluation Tick.

```
else if(ppuDot > 256 && ppuDot <= 320)
{
    ppuOAMAddress = 0; // this is set to $00 during every one of these cycles.
    if (ppuDot == 257)
    {
        ppuSecondaryOAMSize = ppuSecondaryOAMAddress;
        ppuSecondaryOAMAddress = 0;
        ppuSpriteEvalTick = 0;
    }
}
```

Now we need to set up the 8-step process of preparing Sprite Evaluation's shift registers.

```
else if (ppuDot > 256 && ppuDot <= 320)
{
    ppuOAMAddress = 0; // this is set to $00 during every one of these cycles.
    if (ppuDot == 257)
    {
        ppuSecondaryOAMSize = ppuSecondaryOAMAddress;
        ppuSecondaryOAMAddress = 0;
        ppuSpriteEvalTick = 0;
    }

    switch (ppuSpriteEvalTick)
    {
        case 0:
            break;
        case 1:
            break;
        case 2:
            break;
        case 3:
            break;
        case 4:
            break;
        case 5:
            break;
        case 6:
            break;
        case 7:
            break;
    }
    ppuSpriteEvalTick++;
    ppuSpriteEvalTick &= 7; // and reset at 8
}
```

Let's create the shift registers. Unlike the background, which has 4 total shift registers (a high/low byte register for the pattern and attributes) the sprites have two shift registers per sprite. And there can be up to 8 sprites.

```
byte[] ppu_SpriteShiftRegisterL = new byte[8];
byte[] ppu_SpriteShiftRegisterH = new byte[8];

byte[] ppu_SpriteAttribute = new byte[8];
byte[] ppu_SpritePattern = new byte[8];
byte[] ppu_SpriteXposition = new byte[8];
byte[] ppu_SpriteYposition = new byte[8];
```

Setting up this switch statement is actually pretty easy. Case 0 just sets up an array full of the Y position data:

```
case 0:
    // set this object's Y position in the array
    ppu_SpriteYposition[ppuSecondaryOAMAddress / 4] = SecondaryOAM[ppuSecondaryOAMAddress];
    ppuSecondaryOAMAddress++;
    break;
```

Case 1 does pretty much the exact same thing, but setting up an array of pettern data:

```
case 1:
    // set this object's pattern in the array
    ppu_SpritePattern[ppuSecondaryOAMAddress / 4] = SecondaryOAM[ppuSecondaryOAMAddress];
    ppuSecondaryOAMAddress++;
    break;
```

Case 2 sets up an array of attributes:

```
case 2:
    // set this object's attributes in the array
    ppu_SpriteAttribute[ppuSecondaryOAMAddress / 4] = SecondaryOAM[ppuSecondaryOAMAddress];
    ppuSecondaryOAMAddress++;
    break;
```

Case 3 (as you would expect) sets up the X position array. **But this time do NOT increment the secondaryOAM address!!!**

```
case 3:
    // set this object's X position in the array
    ppu_SpriteXposition[ppuSecondaryOAMAddress / 4] = SecondaryOAM[ppuSecondaryOAMAddress];
    break;
```

Case 4 is all about finding the address in the CHR ROM for the pattern data of the sprite. This is actually trickier than it sounds, as we need to know what row of this object we're trying to draw based on it's y position relative to this scanline, as well as consider the vertical flipping tribute. This is even more complicated if we're drawing objects in 8x16 mode.

This one takes up a good amount of space, so I'm making a function dedicated to finding this address in the CHR ROM.

Let's calculate the address for 8x8 pixel objects!

Start at either $0000 or $1000, based on the value of "ppuSpritePatternTable" which is one of the bools that get set by writing to CPU address $2000.

Add the Pattern * 16, since each character in the CHR ROM takes 16 bytes.

Add the number of scanlines from the top of the object, since each row is 1 byte further. If the attributes are set up to flip the Y position, make this number 7 - the number of scanlines from the top of the object.

```
if (!ppuUse8x16Sprites) //8x8 sprites
{
    // The address is $0000 or $1000 depending on the pattern table.
    // plus the pattern value from OAM * 16
    // plus the number of scanlines from the top of the object.
    // if the attributes are set to flip Y, it's 7 - the number of scanlines from the top of the object.
    if (((ppu_SpriteAttribute[SecondaryOAMSlot] >> 7) & 1) == 0) // Attributes are not set up to flip Y
    {
        return (ushort)((ppuSpritePatternTable ? 0x1000 : 0) + (ppu_SpritePattern[SecondaryOAMSlot] << 4) + (ppuScanline - ppu_SpriteYposition[SecondaryOAMSlot]));
    }
    else  // Attributes are set up to flip Y
    {
        return (ushort)((ppuSpritePatternTable ? 0x1000 : 0) + (ppu_SpritePattern[SecondaryOAMSlot] << 4) + ((7 - (ppuScanline - ppu_SpriteYposition[SecondaryOAMSlot])) & 7));
    }
}
else //8x16 sprites
```

8x16 pixel sprites don't show up in Super Mario Bros. but I'm going to implement them anyway.

In 8x16 mode, instead of using PPU_PatternSelect_Sprites to determine which pattern table to fetch data from, these sprites instead use bit 0 of the object's pattern information from OAM. So drawing a sprite with pattern $00 would draw index $00 and $01 of pattern table 0, while pattern $01 would draw index $00 and $01 of pattern table 1.

Quite frankly, 8x16 bytes are a mess, and I'm greatful I've already implemented them in a previous emulator I made so I can just copy/paste the code for that, ha!

```
else //8x16 sprites
{
    // In 8x16 mode, instead of using PPU_PatternSelect_Sprites to determine which pattern table to fetch data from...
    // these sprites instead use bit 0 of the object's pattern information from OAM.

    // The address is $0000 or $1000 depending on the pattern table.
    // plus (the pattern value from OAM, clearing bit 0) * 16
    // plus the number of scanlines from the top of the object.
    // if the attributes are set to flip Y, it's 7 - the number of scanlines from the top of the object.

    // if we're drawing the bottom half of the sprite, add 16.
    if (((ppu_SpriteAttribute[SecondaryOAMSlot] >> 7) & 1) == 0) // Attributes are not set up to flip Y
    {
        if (ppuScanline - ppu_SpriteYposition[SecondaryOAMSlot] < 8)
        {
            return (ushort)((((ppu_SpritePattern[SecondaryOAMSlot] & 1) == 1) ? 0x1000 : 0) | ((ppu_SpritePattern[SecondaryOAMSlot] & 0xFE) << 4) + (ppuScanline - ppu_SpriteYposition[SecondaryOAMSlot]));
        }
        else
        {
            return (ushort)((((ppu_SpritePattern[SecondaryOAMSlot] & 1) == 1) ? 0x1000 : 0) | (((ppu_SpritePattern[SecondaryOAMSlot] & 0xFE) << 4) + 16) + ((ppuScanline - ppu_SpriteYposition[SecondaryOAMSlot]) & 7));
        }
    }
    else // Attributes are set up to flip Y
    {
        if (ppuScanline - ppu_SpriteYposition[SecondaryOAMSlot] < 8)
        {
            return (ushort)((((ppu_SpritePattern[SecondaryOAMSlot] & 1) == 1) ? 0x1000 : 0) | (((ppu_SpritePattern[SecondaryOAMSlot] & 0xFE) << 4) + 16) - ((ppuScanline - ppu_SpriteYposition[SecondaryOAMSlot]) & 7) + 7);
        }
        else
        {
            return (ushort)((((ppu_SpritePattern[SecondaryOAMSlot] & 1) == 1) ? 0x1000 : 0) | (((ppu_SpritePattern[SecondaryOAMSlot] & 0xFE) << 4) + 7) - ((ppuScanline - ppu_SpriteYposition[SecondaryOAMSlot]) & 7));
        }
    }
}
```

case 5 in that switch statement we were making is all about setting up the low bit plane. If we calculated this on the pre-render scanline, let's just make sure the value written to the shift register is 0. Also, let's check the attribute bytes to see if we need to flip this byte around.

Pular navegação

```
se 5:
    ppuSpriteEvalTemp = ReadPPU(ppuAddressBus);
    if (ppuScanline == 261)
    {
        ppuSpriteEvalTemp = 0; // clear this if this is the pre-render line
    }
    if (((ppu_SpriteAttribute[ppuSecondaryOAMAddress / 4] >> 6) & 1) == 1) // Attributes are set up to flip X
    {
        // real nice way to change the oder of bits from 76543210 to 01234567:
        ppuSpriteEvalTemp = (byte)(((ppuSpriteEvalTemp & 0xF0) >> 4) | ((ppuSpriteEvalTemp & 0xF) << 4));
        ppuSpriteEvalTemp = (byte)(((ppuSpriteEvalTemp & 0xCC) >> 2) | ((ppuSpriteEvalTemp & 0x33) << 2));
        ppuSpriteEvalTemp = (byte)(((ppuSpriteEvalTemp & 0xAA) >> 1) | ((ppuSpriteEvalTemp & 0x55) << 1));
    }
    ppu_SpriteShiftRegisterL[ppuSecondaryOAMAddress / 4] = ppuSpriteEvalTemp;
    break;
```

case 6 is the easiest to implement, as we're just adding 8 to the ppu address bus so we can read the high bit plane next cycle.

```
case 6:
    ppuAddressBus += 8;
    break;
```

case 7 is pretty much the same as case 5, but we write to the high bit plane, and finally increment the secondary OAM address in preparation for tick 0 next cycle.

```
case 7:
    ppuSpriteEvalTemp = ReadPPU(ppuAddressBus);
    if (ppuScanline == 261)
    {
        ppuSpriteEvalTemp = 0; // clear this if this is the pre-render line
    }
    if (((ppu_SpriteAttribute[ppuSecondaryOAMAddress / 4] >> 6) & 1) == 1) // Attributes are set up to flip X
    {
        // real nice way to change the oder of bits from 76543210 to 01234567:
        ppuSpriteEvalTemp = (byte)(((ppuSpriteEvalTemp & 0xF0) >> 4) | ((ppuSpriteEvalTemp & 0xF) << 4));
        ppuSpriteEvalTemp = (byte)(((ppuSpriteEvalTemp & 0xCC) >> 2) | ((ppuSpriteEvalTemp & 0x33) << 2));
        ppuSpriteEvalTemp = (byte)(((ppuSpriteEvalTemp & 0xAA) >> 1) | ((ppuSpriteEvalTemp & 0x55) << 1));
    }
    ppu_SpriteShiftRegisterH[ppuSecondaryOAMAddress / 4] = ppuSpriteEvalTemp;
    ppuSecondaryOAMAddress++;
    break;
```

Phew! And that's the end of sprite evaluation!

Drawing sprites on screen

Now that we have the shift registers and everything all set up, let's start drawing these sprites!

First off, recall that on every ppu cycle, we shift the shift registers for the background. We're going to need to determine if the x position of the ppuDot is in range of a sprite we're trying to

draw. Rather, the easier way to compute this is to simply decrement the "X position" of every sprite on each ppu cycle, then if the "X position" equals zero for a dot, we know it's time to draw that sprite. At which point, we begin shifting the low and high shift registers for the sprite.

Here I'm writing this right after the shift registers for the background:

```
// if rendering is enabled
if (ppuMask_RenderBG) // if rendering the backgound, update the shift registers for the background.
{
    ppuShiftRegister_patternL = (ushort)(ppuShiftRegister_patternL << 1); // shift 1 bit to the left.
    ppuShiftRegister_patternH = (ushort)(ppuShiftRegister_patternH << 1); // shift 1 bit to the left.
    ppuShiftRegister_attributeL = (ushort)(ppuShiftRegister_attributeL << 1); // shift 1 bit to the left.
    ppuShiftRegister_attributeH = (ushort)(ppuShiftRegister_attributeH << 1); // shift 1 bit to the left.
}
if(ppuMask_RenderBG || ppuMask_RenderSprites) // if rendering at all, let's decrement the X position of the objects.
{
    if (ppuDot > 1 && ppuDot <= 256) // don't decrement until dot 1
    {
        for(int i = 0;i < 8; i++)
        {
            if (ppu_SpriteXposition[i] > 0)
            {
                ppu_SpriteXposition[i]--; // decrement the X position of all objects in secondary OAM. When this is zero, the ppu can draw it.
            }
            else
            {
                ppu_SpriteShiftRegisterL[i] = (byte)(ppu_SpriteShiftRegisterL[i] << 1); // shift 1 bit to the left.
                ppu_SpriteShiftRegisterH[i] = (byte)(ppu_SpriteShiftRegisterH[i] << 1); // shift 1 bit to the left.
            }
        }
    }
}
```


It's been a while since we wrote this routine, so I'm sharing an image of it again. Here is how we chose a color for a pixel based on the background information:

```
if (ppuScanline < 241 && ppuDot > 0 && ppuDot <= 256)
{
    byte PalHi = 0; // which color palette to use?
    byte PalLow = 0; // index into a color palette
    if (ppuMask_RenderBG && (ppuDot > 8 || ppuMask_8pxMaskBG))
    {
        byte col0 = (byte)(((ppuShiftRegister_patternL >> (15 - ppuScrollFineX))) & 1);
        byte col1 = (byte)(((ppuShiftRegister_patternH >> (15 - ppuScrollFineX))) & 1);
        PalLow = (byte)((col1 << 1) | col0);

        byte pal0 = (byte)(((ppuShiftRegister_attributeL) >> (15 - ppuScrollFineX)) & 1);
        byte pal1 = (byte)(((ppuShiftRegister_attributeH) >> (15 - ppuScrollFineX)) & 1);
        PalHi = (byte)((pal1 << 1) | pal0);

        if (PalLow == 0 && PalHi != 0) // color 0 of all palettes are mirrors of color 0 of palette 0
        {
            PalHi = 0;
        }
    }
    Screen.SetPixel(ppuDot - 1, ppuScanline, Palette[PaletteRAM[PalHi * 4 + PalLow]]);
}
```

Let's modify this to include the sprite information!

This is going to start off in pretty much the same way as the background:

```
if (ppuScanline < 241 && ppuDot > 0 && ppuDot <= 256)
{
    byte PalHi = 0; // which color palette to use?
    byte PalLow = 0; // index into a color palette
    if (ppuMask_RenderBG && (ppuDot > 8 || ppuMask_8pxMaskBG))
    {
        byte col0 = (byte)(((ppuShiftRegister_patternL >> (15 - ppuScrollFineX))) & 1);
        byte col1 = (byte)(((ppuShiftRegister_patternH >> (15 - ppuScrollFineX))) & 1);
        PalLow = (byte)((col1 << 1) | col0);

        byte pal0 = (byte)(((ppuShiftRegister_attributeL) >> (15 - ppuScrollFineX)) & 1);
        byte pal1 = (byte)(((ppuShiftRegister_attributeH) >> (15 - ppuScrollFineX)) & 1);
        PalHi = (byte)((pal1 << 1) | pal0);

        if (PalLow == 0 && PalHi != 0) // color 0 of all palettes are mirrors of color 0 of palette 0
        {
            PalHi = 0;
        }
    }
    byte SpritePalHi = 0; // which color palette to use?
    byte SpritePalLow = 0; // index into a color palette
    bool SpritePriority = false; // is the sprite in front or behind the BG?
    if (ppuMask_RenderSprites && (ppuDot > 8 || ppuMask_8pxMaskSprites))
    {

    }
    Screen.SetPixel(ppuDot - 1, ppuScanline, Palette[PaletteRAM[PalHi * 4 + PalLow]]);
}
```

Pular navegação

There can be up to 8 objects in a scanline, so let's make a loop to check all 8 objects.

```
byte SpritePalHi = 0; // which color palette to use?
byte SpritePalLow = 0; // index into a color palette
bool SpritePriority = false; // is the sprite in front or behind the BG?
if (ppuMask_RenderSprites && (ppuDot > 8 || ppuMask_8pxMaskSprites))
{
    for(int i = 0; i < 8; i++)
    {

    }
}
```

Let's start by checking if the object's X position == 0, and then reading from the shift registers and setting up the palette high and low values, as well as the "priority" attribute.

```
for (int i = 0; i < 8; i++)
{
    if (ppu_SpriteXposition[i] == 0 && i < (ppuSecondaryOAMSize / 4)) // if the sprite X position == 0 (the X position is decremented each ppu cycle)
    {
        bool SpixelL = ((ppu_SpriteShiftRegisterL[i]) & 0x80) != 0; // take the bit from the shift register for the pattern low bit plane
        bool SpixelH = ((ppu_SpriteShiftRegisterH[i]) & 0x80) != 0; // take the bit from the shift register for the pattern high bit plane
        SpritePalLow = 0;
        if (SpixelL) { SpritePalLow = 1; }
        if (SpixelH) { SpritePalLow |= 2; }

        SpritePalHi = (byte)((ppu_SpriteAttribute[i] & 0x03) | 0x04); // read the palette from secondary OAM attributes.
        SpritePriority = ((ppu_SpriteAttribute[i] >> 5) & 1) == 0;    // read the priority from secondary OAM attributes.
    }
}
```

We're not done writing this loop.

If we find an object is *not* on this dot, continue the loop. Otherwise, (if we *did* find an object) only exit the loop if the low byte of the color is *not* zero. (otherwise it would be the backdrop color)

```
for (int i = 0; i < 8; i++)
{
    if (ppu_SpriteXposition[i] == 0 && i < (ppuSecondaryOAMSize / 4))
    {
        bool SpixelL = ((ppu_SpriteShiftRegisterL[i]) & 0x80) != 0;
        bool SpixelH = ((ppu_SpriteShiftRegisterH[i]) & 0x80) != 0;
        SpritePalLow = 0;
        if (SpixelL) { SpritePalLow = 1; }
        if (SpixelH) { SpritePalLow |= 2; }

        SpritePalHi = (byte)((ppu_SpriteAttribute[i] & 0x03) | 0x04);
        SpritePriority = ((ppu_SpriteAttribute[i] >> 5) & 1) == 0;
    }
    else
    {
        continue;
    }
    if(SpritePalLow != 0)
    {
        break;
    }
}
```


Now that we have just chosen a color for a sprite, we can check for sprite zero hits! Remember, a "Sprite Zero Hit" occurs if a non-backdrop color of "sprite zero" overlaps a non-backdrop color of the background. In this context, "Sprite Zero" is the currently drawn sprite if it was index 0 of the for loop, and we detected "sprite zero" on this scanline during the sprite evaluation.

Additionally, a Sprite Zero hit can only occur if rendering is enabled for both the background and sprites, and this ppuDot is before dot 256.

```
if(SpritePalLow != 0)
{
    if (i == 0 && ppuScanlineContainsSpriteZero && SpritePalLow != 0 && PalLow != 0 && ppuMask_RenderBG && ppuDot < 256)
    {
        ppuStatusSprZeroHit = true;
    }
    break;
}
```

At this point, we can finally finish the code for reading from PPUSTATUS:

```
case 0x2002:
    byte ppustatus = 0;
    ppustatus |= (byte)(ppuVBlank           ? 0x80 : 0);
    ppustatus |= (byte)(ppuStatusSprZeroHit ? 0x40 : 0);
    ppustatus |= (byte)(ppuStatusOverflow   ? 0x20 : 0);

    ppuVBlank = false;
    WriteLatch = false;
    return ppustatus;
```

These two values also get cleared when VBlank ends:

```
void Emulate_PPU()
{
    if(ppuDot == 1 && ppuScanline == 241)
    {
        ppuVBlank = true;
        DrawNewFrame = true;
    }
    else if (ppuDot == 1 && ppuScanline == 261)
    {
        ppuVBlank = false;
        ppuStatusOverflow = false;
        ppuStatusSprZeroHit = false;
    }
```

Pular navegação

Back to the rendering process!

All that's left is to determine if a pixel should be drawn with the background or with a sprite, and that's the job of "SpritePriority":

```
if((SpritePriority && SpritePalLow != 0) || PalLow == 0)
{
    PalLow = SpritePalLow;
    PalHi = SpritePalHi;
    if(PalLow == 0)
    {
        PalHi = 0;
    }
}
Screen.SetPixel(ppuDot - 1, ppuScanline, Palette[PaletteRAM[PalHi * 4 + PalLow]]);
```

And just like that, we've completed sprite rendering!

Check it out! Mario is standing on the left of the screen. What a champ.

I know i've already said this twice before in this writeup, but it's been a while since we've actually been able to run our code and see results. If your sprites are not appearing, or are visually broken in some way, I wish you the best of luck debugging.

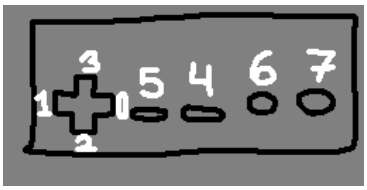
If they aren't appearing at all, double check exactly when your sprite evaluation code is running. It needs to occur on every ppu cycle of a visible scanline, not just the visible dots. Place breakpoints and see if there is ever a scanline where "ppuSecondaryOAMSize" is greater than 0. If not, see if there's every a situation where an object's Y position is considered in-range of the scanline. If not, double check that the contents of OAM are being set up when writing to $4014.

There's certainly a lot that could go wrong (as usual), but working backwards until you find where the malfunction occurs is a pretty good way to debug.

Reading the controllers

We're pretty close to being able to actually play a game with our emulator! With the exception of the audio chip, the only thing we're really missing is the ability to take in user input.

We've had a lot of fun lately with Shift Registers, and that trend will continue. The controllers have an 8-bit shift register inside them, which will hold state of the 8 buttons.



In this diagram, you can see how the bits in the shift register correspond to the buttons. From bit 0 to bit 7, we have:

DPad Right,

DPad Left,

DPad Down,

DPad Up,

Start,

Select,

B,

A.

Before we can read tha data from the controller, we need to fill in this shift register, which occurs when writing the value 1 to CPU address $4016.

I'm writing this near where we set up the write to $4014:

```
else if(Address == 0x4016)
{
    // Set up the Shift Registers for both controllers!
    Controller1ShiftRegister = 0;
    Controller2ShiftRegister = 0;
}
```

Obviously we don't actually want to write a value of zero to the shift registers. We need to take in the user input and fill in the shift register accordingly.

There are many ways to poll for user input while writing your application, and it will likely vary from language to language.

With Visual Studio Winforms, if the program is stuck in an infinite loop, then System.Windows.Input won't be able to update the buttons we're pressing. One solution is to make the emulator run on it's own thread. That's the solution I went with. Now when I press the button that starts my emulator, it actually starts a separate thread to ru nthe emulator on:

```
private void button1_Click(object sender, EventArgs e)
{
    EmuClock = new Thread(ClockEmulator);
    EmuClock.SetApartmentState(ApartmentState.STA);
    EmuClock.IsBackground = true;
    EmuClock.Start();
}
```

The biggest cahnge with makign this run in it's won thread will be an issue with the picture box we're using to render the screen. Both this thread and Visual Studio Winforms are going to try accessing the picturebox at the same time, so we need to create a LockObject, and then run some extra checks to amke sure we're able to update the screen with the new frame:

```
void ClockEmulator()
{
    Emulator emulator = new Emulator();
    emulator.Reset();
    LockObject = pictureBox1;
    while (true)
    {

        lock (LockObject)
        {
            emulator.Run();
            if (pictureBox1.InvokeRequired)
            {
                pictureBox1.Invoke(new MethodInvoker(
                delegate ()
                {
                    pictureBox1.Image = emulator.Screen;
                    pictureBox1.Update();
                }));
            }
            else
            {
                pictureBox1.Image = emulator.Screen;
                pictureBox1.Update();
            }
        }
    }
}
```

This is *far* from a perfect implementation, but here's how I'll be polling for user input, and binding keys to specific buttons.

```
byte controller1 = 0;
if (Keyboard.IsKeyDown(Key.X))          { controller1 |= 0x80; }
if (Keyboard.IsKeyDown(Key.Z))          { controller1 |= 0x40; }
if (Keyboard.IsKeyDown(Key.RightShift)) { controller1 |= 0x20; }
if (Keyboard.IsKeyDown(Key.Enter))      { controller1 |= 0x10; }
if (Keyboard.IsKeyDown(Key.Up))         { controller1 |= 0x08; }
if (Keyboard.IsKeyDown(Key.Down))       { controller1 |= 0x04; }
if (Keyboard.IsKeyDown(Key.Left))       { controller1 |= 0x02; }
if (Keyboard.IsKeyDown(Key.Right))      { controller1 |= 0x01; }
emulator.controller1 = controller1;
emulator.Run();
```

And this "controller1" value will be passed into the shift register when writing a 1 to $4016.

Technically, the controller is continuously re-filling the shift registers while a value of 1 is written to $4016, all the way until you write a value of 0 to $4016. For this emulator, we're not going to worry about that, and we're just going to simply fill the shift register once at the moment we write a 1 to $4016.

Time to read from the controller ports! Controller 1 is readable through $4016, and controlelr 2 is readable through $4017. (But keep in mind, writing to $4016 "strobes" both controllers.)

When reading from a controller port, you only read one bit from the shift register at a time:

```
if(Address == 0x4016)
{
    byte controllerBit = (byte)((Controller1ShiftRegister & 0x80) >> 7);
    Controller1ShiftRegister <<= 1;
    return controllerBit;
}
else if (Address == 0x4017)
{
    byte controllerBit = (byte)((Controller2ShiftRegister & 0x80) >> 7);
    Controller2ShiftRegister <<= 1;
    return controllerBit;
}
```

And that should be everything required for controller reading!

And if everything works as expected, pressing the start button should work, and you can play Super Mario Bros!

Where do we go from here?

There are many branching pathes we could take.

We could implement that audio chip, and start playing some audio.

So far, we're only emulating "NROM" games. Games without fancy "mapper chips".

If you wanted to try emulating Super Mario Bros. 3, for isntance, you would need to implement support for the MMC3 Mapper, which you can read about here.

If you are interested in going down the rabbit hole of accuracy, there are a myriad of test cartrdiges that you can find here.

I strongly suggest running tests for the CPU, such as nestest.nes (and reading the accompanying README to see what the error codes mean) This ROM only has $4000 bytes of PRG ROM, so you'll need to make changes to your Reset() function to account for ROMs with smaller sizes.

```
byte[] HeaderedROM = File.ReadAllBytes(filepath);
Array.Copy(HeaderedROM, Header, 0x10);
byte size = Header[4];
Array.Copy(HeaderedROM, 0x10, ROM, 0, 0x4000 * size);
if (Header[5] != 0)
{
    Array.Copy(HeaderedROM, 0x4000 * size + 0x10, CHRData, 0, 0x2000);
}
```

If the PRG ROM only has $4000 bytes, we need address $C000 to $FFFF to be mirrors of the addresses from $8000 to $BFFF.

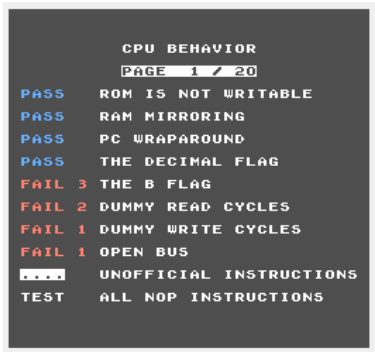
I'll achieve this by taking the address AND the length of the PRG ROM - 1.

```
else if (Address >= 0x8000)
{
    // Reading from ROM
    return ROM[(Address - 0x8000) & ((Header[4] * 0x4000))-1];
}
```

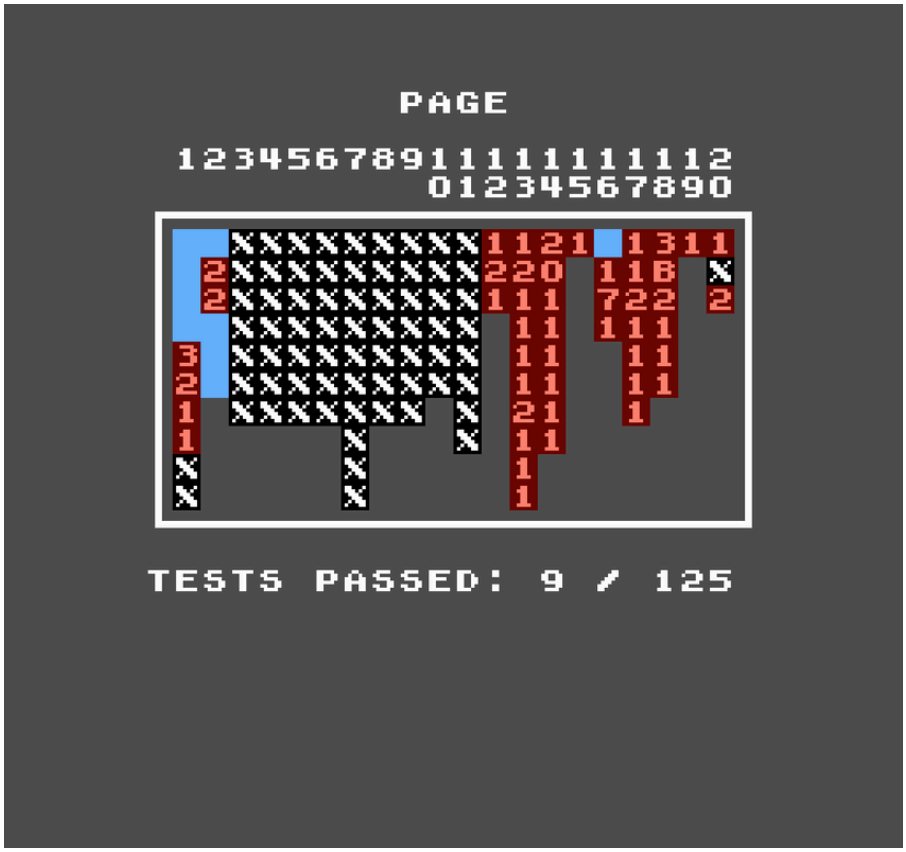
Of course, if you are unable to run that test ROM, you might be interested in this suite of tests which check the behavior of every opcode, grouped by addressing mode.

Very few feelings can match the rush you feel from seeing that simple 6 letter word, "Passed" after running a test ROM.

Depending on just how ambitious you are, you might even consider giving my Accuract Test ROM, "AccuracyCoin" a try. This ROM contains a list of over 100 tests for you to work on. And though the current state of the emulator leaves much to be desired...



It might not be perfect, but you gotta start somewhere.



9 / 125 tests might sound like a shockingly bad score, but most of these tests are for the unofficial opcodes and some wild edge cases.

Since my hobbies include making and watching Tool-Assisted-Speedruns, one of my first goals with my emulators that I have made is to perfect the PPU frame timing, making sure every instruction has the correct number of CPU cycles, and that every the NMI happens at the exact right time. With the frame timing in perfect sync with an actual console, you should be able to replay a TAS in your emulator by replacing the code that polls for user input with code that feeds in the next input from a TAS file.

Conclusion

Well, that's it. You've (hopefully) made a functional NES emulator! It's no easy feat, so I seriously congratulate those who have made it this far. Just to recap, we've learned how most of the 6502 instructions work, (so we're all basically experts at 6502 by this point), we've learned how the CPU communicates with the PPU by writing to the special PPU registers, we've learned how the PPU organizes its own RAM, how it reads this RAM and CHR ROM to draw the background, we've learned about Object Attribute Memory, and how sprites are evaluated and drawn on screen, and how the console reads from the controller ports!

Anexos

Baixar tudo

TestRoms.zip

Na coleção

NES Emulation
1 post

5 comentários

Participe da conversa...

Dominic Willis · 2s

Really great job with this guide, I'm really enjoying working through it so far! I'm a bit confused about the variables in the PPUMASK: it looks like you're assigning a variable within a variable assignment (i.e. ppuMask_8pxMaskBG = ppuVRAMInc32Mode = (Value & 2) != 0). Is the "=" after ppuVRAMInc32Mode supposed to be a different operator?

1

100th_Coin AUTOR · 2s

Whoops! It's supposed to look like:
ppuMask_8pxMaskBG = (Value & 2) != 0;
ppuMask_8pxMaskSprites = (Value & 4) != 0;
ppuMask_RenderBG = (Value & 8) != 0;
ppuMask_RenderSprites = (Value & 0x10) != 0;
I'll fix that in the post.

1

Minimizar respostas

freyacantsleep · 3s

What kind of project do I create? Do I create a project from a template, or do I just create a file?

2

butteredbreadbox · 3s

same question here

Carregar respostas