Nintendo Entertainment System Hardware Emulation

Daniel Klahn, Sidne Gregory, Israel Bonilla

6.111 Fall 2019

Table of Contents

---

1. Abstract

In a multitude of industries, companies often use legacy hardware. Legacy hardware is any hardware that has been deprecated by its manufacturer but continues in widespread use. While new hardware inevitably presents more capability, it will likely remain unproven until the end of the device's product cycle. Therefore, it is often in the best interest of the end-user to continue using legacy hardware. When legacy systems eventually fail, replacements can be difficult to source, either as a result of rarity, price, or both. Fortunately, the synthesis of deprecated hardware is feasible with the use of an FPGA. The Nintendo Entertainment System (NES) Hardware Emulation project presents an opportunity to employ that functionality.

The NES, manufactured by Nintendo from the mid-1980s to mid-1990s, helped repopularize video games in North America. 20 years later, the hardware is inevitably becoming more scarce, and will only become more scarce with time. Emulators do already exist, but they often fail to capture the true behavior of an NES, as they are adaptations of modern hardware to the older software. With the FPGA, we worked to reproduce the NES hardware as faithfully as possible as an exercise in legacy hardware emulation

2. Motivation

The motivations behind this project were to take on something that sounded crazy, but seemed doable given the time we had and the skills that we had learned in class. Implementing one of the most iconic game systems of all time in just a few weeks was definitely challenging, but it was a project that we were all really excited about. In addition to having a really interesting end product, we felt the project offered a really good learning opportunity by allowing us to implement a very basic processor and very basic graphics unit to gain insight into how modern systems work and also into the history of how hardware has developed and how the constraints of the time (primarily memory) dictated the design decisions that were made when creating the NES.

3. Overall System Architecture

Building an NES emulator requires two main components: the 2A03 (a modified MOS 6502) CPU and the 2C02 PPU (this device is best thought of as a primitive video card). Conjoining the two are two buses. The address bus provides a means for the CPU to interface with all I\O devices and the PPU via memory mapping, and the data bus facilitates the actual data transfer between them. For this project, our implementation was able to construct the PPU and 6502 in separate states, with the PPU rendering frames and the CPU ready to compute the majority of its instruction set.

4. CPU (Israel)

System Overview

At the core of the Nintendo Entertainment System (NES) is the Ricoh 2A03, a modified MOS Systems 6502 processor. From a modern perspective, this processor is rather primitive, with a 2 MHz clock in NTSC markets and no pipelining. However, the 6502 would redefine home computing through the extensive use of a state machine. The 6502 is an 8-bit, little-endian processor with 3 main registers, namely the accumulator, x-index register, and y-index register. Aside from those three, the 6502 has a few support registers for internal use, which are the program counter. the status flag, and the stack pointer. All registers are 8 bit except for the program counter, which holds 16 bits. In normal operation, the 6502 would receive instructions through its 8 data bus pins (in/out) resulting from a request via its 16 address pins (out only). The value from the address pins is either the value in the program counter or an address from an instruction. The 6502 instruction set consists of roughly 150 instructions, with 13 different addressing modes and 56 different possible operations. The addressing modes will be discussed in further detail in sections (b) and (e).

In order to emulate this system, several control signals needed to be generated, resulting in three main modules and six supporting modules for auxiliary control signals. Similar to the actual 6502, one of the most important modules is dedicated to decoding the received instructions from a raw opcode. Obeying von Neumann architecture, the 6502 also had an Arithmetic\Logic Unit (ALU) for computations like add with carry and bitwise or. Lastly, the the main 6502 verilog module encodes a Mealy state machine consisting of 7 states based on clock cycle. Integrating these modules together, the result is a largely cycle-accurate 6502, missing only the indirect addressing mode (which largely went unused by the NES), relative addressing mode, and decimal mode (which was not present in the 2A03 variant used by the NES).

OpCode Decoding

In order to achieve accurate results and timings, the opcodes needed to be decoded into an addressing mode for the controller and an operation for the ALU. This module takes advantage of the encoding scheme employed by most of the 6502 instruction set. Decomposing the byte-sized instruction

into individual bits and labeling them as “aaabbbcc,” there are three main divisions of the instruction set based on the value of “cc.” Within each subset, “aaa” specifies the operation, and “bbb” specifies the addressing mode. For example, the opcode 10101001 ($A9) specifies “LDA immediate,” 6502 Assembly. In this case, cc = 01, which asserts that aaa = 101 calls for a load accumulator operation and bbb = 010 calls for the “immediate” addressing mode (meaning the next byte that the processor receives over its data bus pins is the value to be operated upon by the ALU). Referring to any 6502 instruction set chart, we can confirm that 10101001 in fact refers to LDA in the immediate addressing mode.

In order to implement a decoder, we built a LUT on case statements regarding the bit contents of each opcode. The exceptions to the above encoding scheme are hard-coded in, since there are relatively few. In order to encode the results, the operations are sorted alphabetically by their name in 6502 assembly language and assigned decimal numbers accordingly. The addressing modes are treated the same way. Refer to cpu_constants.sv for the addressing mode and operation values.

Arithmetic and Logic Unit

The 6502 ALU is relatively simple, with the most complicated instructions to implement being the rotate commands and the arithmetic operations. The vast majority of commands simply pass results through our implementation, since they involve transferring information from one register to another, or storing a value in a register. For this purpose, the convention was adopted that the first operand used by the ALU (called operand A) would be sourced strictly from a register, and the second would be sourced specifically by memory input. This design choice did result in a challenge from the bit shifting commands, which have the ability to write directly to memory. By writing to memory, the 6502 contradicts its usual routine of writing results to the accumulator register. We overcame this obstacle by creating a register to store memory data for use by the ALU.

Add with carry and subtract with carry are made more complicated by the carry and overflow flags. The formula for add with carry is as follows: A + M + C → A, meaning that the accumulator is added to the memory data, and the carry flag’s value is then added to that, storing the result in memory. The carry allows the processor to work with larger numbers (albeit slowly) than 8 bit ones. The carry flag is easily set by evaluating the intermediate result in a register and then setting the new carry to be the same value as the last digit of this register. Setting the overflow flag is much more complicated, obeying the following logic: V = (!A[7] & !B[7] & C) | (A[7] & B[7] & !C), where the C flag had already been set by the addition. . Subtract with carry behaves as one would expect: A - M - C → A. The carry is set according to the following logic: C = (A > B), and the overflow flag obeys the same rule as addition.

Registers and Memory

As above, the 6502 has 3 primary registers. The accumulator register is generally where the result of an operation is stored after its completion (the exception being bit shifts and bit rotations, which can directly write to memory depending on addressing mode). The add with carry, subtract with carry, exclusive or, and, and inclusive or always use the accumulator as one operand, the other operand coming from memory or an input value in the case of the immediate addressing mode. The other two main registers are the indexing registers X and Y. Indexing registers are used to refer to a memory location with an offset from the requested location in memory. For instance, in the absolute addressing mode, after the first cycle, which establishes which operation and addressing mode will be used in this instruction cycle, the next two cycles are used to receive a specific address in memory to either read or write data. The absolute,x-index or absolute,y-index addressing modes behave in exactly the same way, except the contents of the X or Y registers are added to the requested address. This addition is implemented with carry, which will add an extra cycle to the instruction time. As an example, if the X register holds the value $FF, and the requested address is $0001, the instruction will take an extra cycle to execute (and the real world 6502 will spend a cycle accessing the wrong address since the carry is added in an extra cycle) and will access location $0100. These three registers are the ones that are most directly controlled by the user.

There are three other registers of note documented within the device. The first of these is a unique 16-bit register. This register is called the program counter. The function of the program counter is to sequentially access points in memory in order to receive the next instruction. For each clock cycle, the program counter increments by one. A few inputs can alter the contents of this register, namely jumps and interrupts. Jump instructions merely assign a new value to the program counter, while interrupts come from specific input pins that necessitate storing the old program counter value, processor status, and accumulator values.

The processor status bits are another important register which are largely controlled by the ALU, although set and clear instructions exist for all of the bits contained. These bits signal specific information about results from ALU computation. These bits are symbolized by NV-BDIZC. The dash is unused. N refers to when the signed two’s complement interpretation of the result is negative. V is the infamous overflow bit, which simply states when the result is out of the range of -128 to 127, which is the range an 8-bit two’s complement number can occupy. B is the break flag, which is tripped by the BRK instruction.

D is the decimal flag, which is unused by the NES and therefore this project. I is the interrupt flag, which is set on non-maskable interrupt (NMI) or interrupt request (IRQ). Z is zero, which occurs when the result is zero. C is the carry flag. Each of these flags are used to control various branching instructions.

The last of these registers is the stack pointer. The 6502 devotes its page of memory to a LIFO stack, meaning all memory locations from $0100 from $01FF represent the 6502 stack. The stack pointer selects where in the stack to write, initializing with the value $FF. It decrements when objects are stored in the stack, and increments when items are retrieved from the stack.

Another special memory location is known as the zeropage. The NES considers the high byte of a memory address to be the page of memory being accessed. Therefore, location $00AB is zeropage, and $0NAB would be Nth page. The zeropage can serve as extra registers for use by the processor, since accessing and writing to the zeropage is a full cycle faster than accessing and writing in the absolute mode, since the processor can assume the high byte is $00 on the zeropage.

In our implementation, there exists a few other registers used to aid in the control of the processor. The first of these is the M register, which was used to hold memory inputs for operation by the ALU in the case of bit shifts targeted at memory. The other two are the ADDR_L and ADDR_H register, which hold on to the address bytes from input data. These are later used to access the desired location for read\write.

Cycle State Machine

In the SystemVerilog implementation of the 6502, we opted to create a 7-state state machine with relatively simple transition logic. The most obvious way to enforce cycle accuracy was to count specific cycles and transition at exactly the right time. Controlling how many cycles should occur until the next instruction receipt ended up being a major hurdle, resulting in lengthy logic statements in each state. Generally, different addressing modes for an operation require a different number of cycles. This is not particularly complicated, since we can simply case on what addressing mode has been specified by the instruction thanks to the decoder module. When doing this, however, we must account for what operation has been specified as well. Bit shifts to memory in the absolute addressing mode take 6 cycles, while most other operations take only 3 in this addressing mode. Indexed addressing modes on the same operation can take a different number of cycles based on if the address calculation requires a carry. In summation, these things together require somewhat convoluted code to capture the function of. This will be elaborated on further in the Insights section.

Support Modules

In order to create a functioning 6502 replica, it was deemed necessary to add a few extra control signals. The first of these extra signals is the PC flag. When reading from a specific memory address, the 6502 needs to send that address out on the address pins. When it finishes this instruction, it needs to return to sending the value in the Program Counter for the next instruction. This was accomplished with the 1-bit PC flag. Another module created to aid in the control of the processor was a controller for the implied addressing mode. The implied addressing mode has a variety of possible outcomes, encapsulating all instructions for which the operand to be used is specified by the operation name. For example, TAX, "transfer A to X," implies that the A register must be the input to this operation. This module cased on several instructions for which the result would take longer than two cycles, taking the requested operation as input and providing three signals as output. These three signals were used in transition logic to decide whether to continue increasing the cycle counter or to go back to cycle zero.

Since the design convention of our ALU module specifies that its first operand is to come from a register (except in those cases where the input must come from memory), a module was needed to select which register would be pulled from. This module cased on operations that specified use of registers other than the A register, since using the A register is most common, thereby reducing the amount of code.

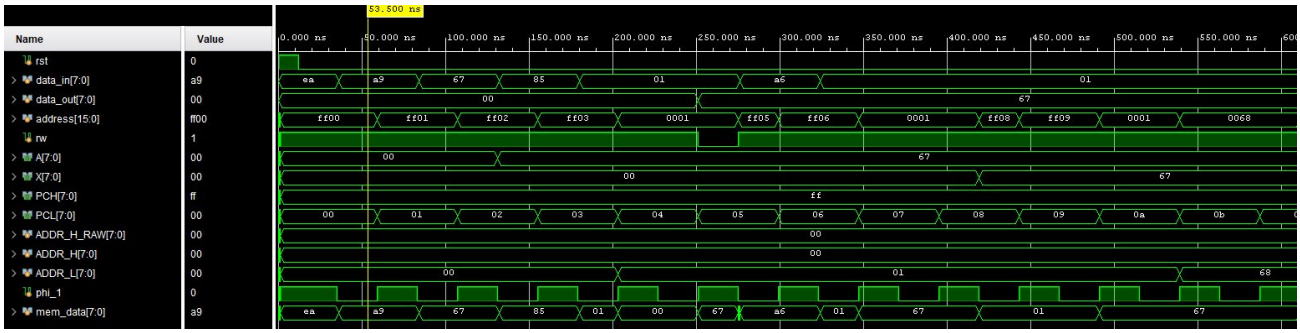
In order to mimic the NES' system clock architecture, a simple module was drawn up to divide an input clock signal by 12. This was accomplished with a simple counter. When the counter reaches the specified divisor, the outgoing signal is inverted and the counter is reset.

The instructions for bit shifts (ASL, LSR, ROL, and ROR), when writing to memory, are particularly slow. The module mem_write was used to hold the cpu from going to cycle 0 until cycle 6, which is when those instructions should terminate.

Lastly, the NES memory map incorporates several mirrored addresses. A module was devised to "demirror" addresses aimed at internal RAM. This was accomplished by recognizing that in each of the 4 mirrored quadrants, the low byte of the address would be the same. Since the 6502 only had 2kB of internal RAM, we could keep within the first quadrant, replacing the first 5 bits of the address high byte with 0s when the raw address was below location $2000. It was further necessary to do this demirroring on the program counter in case the program counter attempted to access internal memory.

Testing and Simulation

In order to prove out our SystemVerilog code, several testbenches were written to verify the behavior of each individual module. Most importantly are the testbenches for the main 6502 module, as verifying behavior of smaller combinational modules like the ALU and opcode decoder were simple and easy to fix in the event of failure. The main module was much more complex, with timing constraints to obey and a lot more signals heading in and out of the module. In order to validate the 6502 behavior, we used the online Easy6502 emulator, which is cycle accurate and reads out the values of each of the registers and several memory locations. Then, in the testbench, machine code was fed in at intervals mimicking possible NES behavior. An instruction would be sent in, along with values specifying either an operand or memory location. Then, if a memory location was called, we checked the requested address to see if the processor was requesting the right location. Lastly, we checked to see if the result each time matched our expectation in the number of cycles required by that instruction. The below waveform presents a nice example of a few addressing modes in action:



First, reset (rst) is deasserted and the clock initializes (note the program counter was initialized at $FF00 to avoid reading from internal memory) . The data_in reads $A9 on the next positive edge, meaning the processor should load the accumulator with an immediate value. The data_in value switches to $67, signifying the value that ought to be stored in the accumulator. Notice that on the negative edge of the same clock cycle, the value $67 propagates to the accumulator. The program counter reads $FF02 at the end of this instruction, matching the cycle timing of the LDA IMMEDIATE instruction. The next input is $85, which refers to STA zeropage (store value of A in the zeropage). The next data point tells us to store at location $0001 since it is a zeropage address. This value is held since the processor needs a cycle to receive this location, and then another cycle to actually write to the location. We see the rw flag go low (write is considered an active low on the 6502). The next instruction is $A6, which is LDX ZEROPAGE.

The data_in line reports $01, telling the processor to access the data from location $0001. Since this location is within the range $2000 to $0000, this is found in internal memory, which we generated as a BRAM with a write width of 8 bits and a depth of 2048. We know the execution of this instruction was successful because the X register then receives the value $67, exactly what was stored at that location from a different register. Other “programs” can be written in this manner. Based on this testbench (and many other iterations of similar programs), successful integration of this module with a working PPU would have favorable results.

5. PPU - Pixel Processing Unit (Daniel and Sidne)

Overview (Sidne and Daniel)

The Pixel Processing Unit (PPU) is the NES equivalent of a graphics card and is responsible for generating the 256x240 video output signal. A full block diagram of our implementation of the PPU can be found in Diagram 1.

Communications between the PPU and CPU mainly occur during the PPU's VBlank period. During this time the CPU can write to the PPU's memory in order to change the background or sprites for the next frame. To change the background there are a few specific things that the CPU can change in memory.

The background that gets sent to the display from the PPU is actually made up of multiple separate components in memory that the PPU must quickly fetch and stitch together in order to output a coherent display. The first component of the background are nametables. These are 32x30 matrices of tile indexes that tell the PPU which pattern table tiles, which will be discussed later, correspond to which location on the screen. The NES has enough memory to internally store up to two nametables which can be overlapped to create interesting background affects or laid out next to each other in different orientations to allow for scrolling.

Once the name table data has been retrieved and a tile index determined that tiles pattern table is retrieved from the memory. While sprites do not have nametables, each sprite does have a tile index stored with it in the OAM which determines which pattern table the sprite tile uses. The pattern tables are two 8x8 bit sections of memory per tile that when overlayed on top of each other for a bit depth of 2 give a piece of the color information for each pixel in the corresponding tile. The function of the pattern tables is explained visually in Figure 1 of the appendices.

The other portion of the pixel color information is determined by its attribute byte. A single attribute byte is shared by an area of 4x4 tiles which is then split into four sections of 2x2 tiles. Each pair of bits in the attribute byte are assigned to one of the 2x2 sections. These two bits determine which 4-color palette the section uses, so each area of 2x2 tiles displayed on the screen has to use the same color palette. This will be explained further in the Color Palette section.

All of these bits and pieces of information must be fetched from memory by the PPU and correctly combined together into display data for every pixel of every tile on every visible scanline in each frame with our frame rate of 60 frames/second.

State Calculator (Daniel)

The PPU outputs 256x240 video at 60 frames per second. In order to do this, it runs a sequence of memory accesses in sync with the video generation. The PPUStateCalculator module serves to keep track of where the PPU is in this progression and to help determine what it should be doing on every cycle.

In this implementation, for every frame that is generated, the PPU runs through 262 vertical scanlines each of which has a duration of 341 PPU clock cycles. In this implementation, scanlines 0 though 239 (inclusive) are visible scanlines in which on screen pixels are generated. During these scanlines, one pixel is generated every clock cycle. Scanline 240 is a postrender scanline in which the PPU idles and does not access memory. After the postrender scanline, the PPU enters the vblank period for 20 scanlines (scanlines 241 through 260, inclusive). During this phase, the PPU asserts its vblank output which would interrupt the CPU in a full implementation. Lastly, the PPU performs one prerender scanline in which no video is generated, but data for the next scanline is prefetched. For more detail on the specific cycle progression of the visual and prerender scanlines, see Figure 5.1 below.

Visible Scanlines

During the visual scanline, the PPU is constantly fetching background tiles just in time for them to be drawn. Each memory access takes two clock cycles, so every eight cycles, the PPU fetches four bytes from VRAM:

- 1 nametable byte (pattern table tile index)
- 1 attribute byte (color data)
- 2 pattern table bytes (from the tile given by the nametable byte)

These four bytes contain the data for the PPU to draw eight pixels on the screen. Therefore, every eight cycles, the PPU obtains enough information to draw eight pixels and can keep up with drawing one pixel every clock cycle. Each visible scanline contains 256 visible pixels which translates to 32 tiles. This

means that during the visible scanline, the PPU repeats this eight-cycle background tile fetch pattern 32 times which takes 256 cycles in total.

On cycle 256, once the PPU is no longer rendering visible pixels and is in its horizontal blanking period, it starts prefetching the image data for the sprites that appear on the next scanline. Although 64 sprites can be held in the PPU's internal Object Attribute Memory (OAM), the NES can only display eight sprites on any given scanline. This is due to the fact that the PPU only has time to prefetch data for eight sprites before it must once again start fetching background data. It is also interesting to note that although sprite tile fetches take the same number of cycles (eight) as background tile fetches, no nametable or attribute table data is fetched because the sprite tile index and color information is held within OAM. Because of this, in this implementation, the PPU does not access VRAM during the first four cycles of each sprite prefetch and only fetches two pattern table bytes.

Once all of the tile data for the sprites on the next scanline has been fetched from VRAM, the PPU prefetches the first two tiles for the next scanline. The reason the PPU fetches these tiles is so that it can support background scrolling. Background scrolling is not implemented in this version of the PPU, but this prefetching was included to maintain the original memory access progression found in the original NES. The background tile prefetches start at cycle 320 and follow the same progression as the previously discussed background tile fetches

Postrender and Vblank Scanlines

During the postrender and vblank scanlines, the PPU idles and does not access VRAM or draw pixels to the screen. The original NES used the vblank period as an opportunity to load new data into the OAM. In order to be able to update all 256 bytes of OAM memory, the CPU had a special mode called Direct Memory Access (DMA) that took some shortcuts when addressing into the PPU so that the entire OAM could be filled within the vblank period. However, because the CPU and PPU are not interfaced, this is not a feature of this implementation.

Prerender Scanline

During the prerender scanline, no pixels are drawn to the screen, so no real-time background tile fetching is necessary. However, pixels will be rendered on the next scanline, so the sprite prefetch and background tile prefetch phases still happen (see Figure 5.1 below).

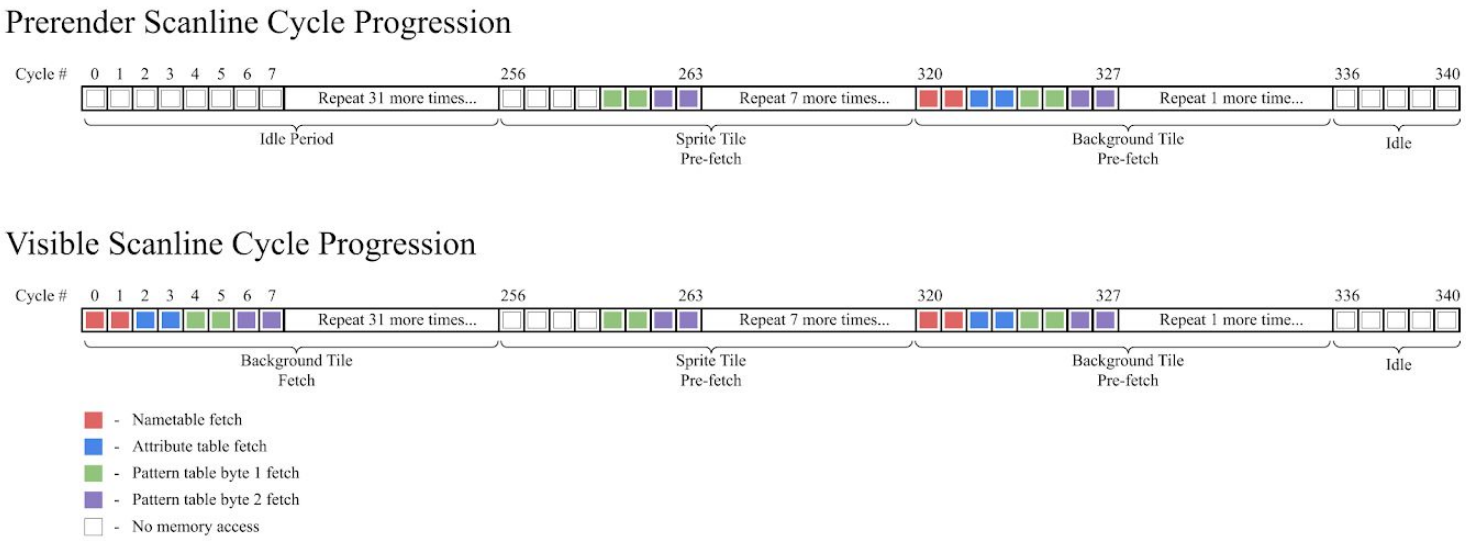


Figure 5.1: Cycle progression of the prerender and visual scanlines

VRAM (Video RAM) (Sidne and Daniel)

VRAM Module (Sidne)

Although unused in our current implementation the VRAM module is an important part of the overall NES system. This module instantiates a BRAM that can be written to from the control signals sent to the PPU from the CPU and then is read by the memory fetcher to give all of the pixel information for our display. Another NES functionality that this module emulates is memory mirroring. The NES uses this because it has a much higher volume of addresses it can write to than actual memory cells. The actual layout of the VRAM and where it mirrors its memory can be seen in Figure 3 in the appendices. This module deals with the top overall VRAM mirror by truncating the original address sent to the module down from 16 bits to 14 bits. It then looks to see if the new address is in a range that is a mirror instead of the actual address of the memory cell. If it is a mirror then it adjusts the address to be the address of the memory cell it should mirror to. While the module itself is fairly simple, when fully implemented it plays a big role in allowing the PPU to get the correct data from memory and in allowing the CPU to write that data to the correct memory cell for the PPU to read. A detailed block diagram of this part of the PPU can be found in Diagram 2.

Object Attribute Memory (Daniel)

The Object Attribute Memory (OAM) is a set of registers internal to the PPU that stores up to 64 sprites that may be drawn in the current frame. Each sprite in OAM contains 4 bytes of data:

Byte 1 - the sprite's y position
Byte 2 - a tile index in the pattern table
Byte 3 - the sprite's attribute byte
    Bits 1, 0: color data
    Bits 2, 3, 4: unused
    Bit 5: background priority
    Bit 6: flip horizontally
    Bit 7: flip vertically
Byte 2 - the sprite's x position

One of the jobs of the OAM module is to determine which of the eight sprites will be drawn on the scanline. Due to the timing constraints on the sprite prefetching, only the first eight sprites in OAM that are on the next scanline will actually be drawn. In this implementation, the OAM does this range evaluation during cycles [0, 63]. Each cycle, the OAM evaluates one sprite and determines if it appears on the next scanline. If it does, it stores it in the secondary OAM if the secondary OAM is not full (see the next section). A detailed block diagram of this part of the PPU can be found in Diagram 2.

Secondary OAM (Daniel)

The secondary OAM stores up to eight sprites chosen by the OAM to be drawn on the next scanline. Our implementation of the secondary OAM is a size eight queue of sprites. The OAM pushes sprites in and the memory fetch unit (next) section pops them out.

We encountered several difficult bugs with the secondary OAM. The first was that originally, when the queue was empty, it still presented the last sprite on its output. This resulted in sprites being rendered on every scanline after they were loaded until the secondary OAM was loaded with something else. A detailed block diagram of this part of the PPU can be found in Diagram 2.

Memory Fetcher (Daniel)

The memory fetch unit interfaces with the VRAM and requests data according to the current cycle state of the PPU (see State Calculator for more details). The memory fetcher requests background tile data for the current scanline, prefetches the sprites based on the contents of the secondary OAM, and prefetches the background tiles for the next scanline.

Once the memory fetcher acquires this data, it presents it to the background and sprite rendering modules. For the background renderer, the memory fetcher also sends a signal that new data is available to make it easier to synchronize the two modules. The sprite renderer does not have this feature and simply assumes that sprite data will be present at the beginning of the scanline.

The memory fetcher is one of the more complicated modules and we encountered quite a few off-by-one bugs during testing as well as errors in the way the memory fetcher was calculating nametable, attribute, and pattern table addresses. This was especially true of the tile prefetches as they commonly requested data for the previous scanline instead of for the next one. However, we were able to find and fix all of these bugs. A detailed block diagram of this part of the PPU can be found in Diagram 2.

Background Renderer (Sidne)

The Background Renderer module is used to generate the background pixel for any given location on the screen. It communicates with the Memory Fetcher module to receive the data needed to determine the pixel that will be displayed and then passes that singular pixel information as well as the h count and v count of that pixel, to be used as coordinates, to the pixel mux module.

The Background Renderer contains four shift registers, 2 16-bit and 2 8-bit as well as two single bit latches. The data this module receives from the Memory Fetcher comes in the form of two pattern table bytes, two bits of attribute data and a new_data wire that pulses when the Memory Fetcher has new background data for this module to load in. When the new_data wire is triggered this module loads the new pattern table bytes into the last 8 positions in their own shift register and separately latches each of the attribute bits. Then each clock cycle the first bit in each shift register is taken and appended together, with an extra 0 to signify that it is a background pixel, to form the 5-bit data sent out by this module to the Pixel Mux. The pattern table shift registers are then loaded with a 0 while the attribute shift registers are fed from the attribute latches. This is because each tile uses the same two attribute bits for the entire tile. These bits are used to signify which palette the tile is using and this is while each tile is limited to one

four-color palette. The h count and v count in are also transferred to h count and v count out each clock cycle to make sure the correct location stays with the correct pixel. A detailed block diagram of this part of the PPU can be found in Diagram 3.

Sprite Renderer (Sidne)

The function of the Sprite Renderer module is to take in information fed to it by the Memory Fetcher once each scanline and output pixel data in each sprite's correct locations to the Pixel Mux. It does this by first obtaining the scanlines sprite data from the Memory Fetcher. To do this it waits until a few clock cycles after the end of the sprite_prefetch phase of the Memory Fetcher and then loads in the sprite data to its shift registers which consist of 16 8-bit shift registers, two for each of the 8 sprites allowed on each scanline, 8 2-bit latches for each sprite's attribute data, and 8 8-bit latches to hold the x location of each sprite on the scanline. It also takes in the current h count and v count. The module then checks each clock cycle if $\textit{the current x location of the sprite} \leq \textit{h count} < \textit{the x location} + 8$ and if it is then it starts shifting the register for that sprite each clock cycle in a similar way as the Background module but instead of outputting that data it loads it into a latch to then be checked for sprite priority.

The Sprite Renderer module is more complex than the Background Renderer because it must also calculate the pixel from the correct sprite to output if two sprites happen to overlap. In this module, this is done combinatorially. When h count is in the x position range for a sprite, the data for that specific pixel at h count on that scan line is loaded into a latch for whichever of the 8 sprites it is. If there is more than one sprite in that location then more than one latch will be filled. These latches are then fed into the combinational block to decide which sprites pixel to output. The way the NES determines which sprite will be output is based on the order they were fed into the secondary OAM. So if sprite 0 is on top of sprite 3 then the pixel for sprite 0 will be output because earlier sprites have priority. However, if the current pixel for sprite 0 is transparent then the pixel for sprite 3 will be output. The combinational portion of this module starts with the latch of sprite 0 and if it is not currently outputting any pixel data or is transparent it will move on to the next sprite.

If no sprites are at the current h count then the module will output a 6-bit 0 otherwise, it will send out the pixel data for the top priority sprite to the Pixel Mux. The reason this module outputs 6 bits of data to the pixel mux while Background only outputs 5 is that the sprites have an extra bit of data to determine the priority over the background which is used by the Pixel Mux. It takes two clock cycles to determine and output the correct pixel data so this module outputs transfers the v count and then adds one to the transferred h count. A detailed block diagram of this part of the PPU can be found in Diagram 3.

Pixel Mux (Sidne)

We now have the chance of two pixels being output for any given location depending on whether or not a sprite is at that location. The job of this module is to take the outputs of the Background Renderer and the Sprite Renderer and decide which set of pixel data should be displayed. This is done by first syncing up the locations of the sprite and background pixel. The background pixel for a certain location will be sent to the Pixel Mux one clock cycle before the sprite pixel. To account for this the Pixel Mux has a latch to hold the incoming background pixel data for one clock cycle until the corresponding sprite pixel is ready to be compared with it. Once the Pixel Mux has both the background and sprite pixel data for a certain location it compares the two of them to see which pixel should be displayed there. If there is no sprite pixel data available then it will output the background pixel, it will also be output if there is a sprite that has a background priority bit of 0 or if the sprite pixel is mapped to one of the transparent palette colors. The sprite pixel will be output if a sprite with background priority of 1 that has a non-transparent color is present or if the background pixel is mapped to a transparent color and a non-transparent sprite pixel with background priority of 0 is present. If both pixels are transparent or only a transparent background is given to the pixel mux it will send out data for the pixel to be the transparent color. A detailed block diagram of this part of the PPU can be found in Diagram 3.

Palette RAM (Daniel)

The original NES was only capable of outputting 55 distinct colors in the onboard color palette. These were mapped to 6-bit color indices shown in Figure 5.2 below. However, even though the NES could select any of the 55 system colors, due to the two-bit color depth used in the pattern tables, any given tile could only use four colors at any given time. The palette RAM stores these sets of colors that are used in the current frame and can be reprogrammed by the CPU.

The palette RAM contains four background palettes and four sprite palettes each containing three, six-bit color IDs and the current transparent color ID. All palettes share the same transparent color ID. The pixel data comes in to the palette RAM as a five-bit number:

Bit 4 - selects the background (1) or sprite palettes (0)
Bits 3, 2 - select which one of the four palettes

Bits 1, 0 - selects one of the four palette colors

The transparent color is mirrored to all palette RAM addresses of the form 5'bXXX00. This module takes in this 5-bit address and maps it to a six-bit PPU color ID stored within the palette RAM. A detailed block diagram of this part of the PPU can be found in Diagram 3.

Figure 5.2

Frame Buffer (Daniel)

The frame buffer is an integral piece of glue logic between the PPU and the VGA module that is used to output video to the monitor. The original NES system was designed to output composite video at a resolution of 256x240 and is not directly compatible with the standard VGA resolutions. Additionally, the PPU and VGA systems work in different clock domains (5.175 MHz and 65 MHz respectively) and synchronizing them would be complicated. To overcome this, two frame buffers are used. While the PPU writes pixel data to one frame buffer, the VGA unit reads pixel data from the other. When the VGA hits its vblank period, the buffers are swapped and the VGA unit reads the fresh frame data while the PPU overwrites the old frame data. To ensure that the two systems map memory addresses to the same screen coordinates, both systems map the pixel at (x, y) to the address 256 * y + x. 256 is the horizontal pixel width of the NES's video output.

However, this system is not perfect. Although the PPU and VGA both work with 60 FPS video, they are not perfectly synchronized. This is evidenced by occasional screen tearing that happens when the PPU has not finished writing the entire frame when the buffers swap. However, due to the fact that game screens don't change very much between adjacent frames and the modules run at close to the same frame rate, this effect is not very noticeable and doesn't affect the user's experience.

In order to reduce the BRAM usage of the system, the frame buffers store the six-bit PPU color IDs rather than the 12-bit VGA colors to which they correspond. This allows the system to use only 128

kB of BRAM to buffer the two frames. A detailed block diagram of this part of the PPU can be found in Diagram 4.

Color Mapper (Daniel)

The color mapper proves the translation from the six-bit PPU color IDs used internally by the NES to 12-bit RGB VGA color values. This combinational block is essentially a look up table that translates the color IDs in real time as they come out of the frame buffer. A detailed block diagram of this part of the PPU can be found in Diagram 4.

VGA (Daniel)

The VGA module used is the same as was used in lab 3. The original intention was to use this module to generate 640x480 video and to scale our NES video output by two times in order to completely fill the screen. However, there were complications with getting the lower resolution video to work and we opted to use the higher resolution mode (1024x768) with scaling to fill a smaller, but still usable, portion of the screen. A detailed block diagram of this part of the PPU can be found in Diagram 4.

Testing and Simulation (Sidne)

While testing and simulating each separate module was fairly straightforward, when we integrated all of the modules and began testing there were many small bugs as well as some bigger ones that we ran into. The first big problem we ran into while trying to test our PPU was a problem with the VGA module and Frame Buffer. This was an immediate problem because without any working display we were unable to test if our other functionalities were working. We ended up having to use 1024x768 resolution instead of the 640x480 that we wanted so we started with a much smaller screen than we wanted. So we would have a larger screen to debug on we ended up scaling our video out, which was one of our stretch goals, before many of our other goals were reached. We also had an issue with our screen scrolling when we wanted a static screen which we ended up fixing by running the Frame Buffer BRAM on a much faster clock than the rest of the PPU to make up for its 2 clock cycle read latency.

Other smaller problems we ran into were that the two prefetched tiles at the beginning of each scanline were shifted up down by one pixel and some small bug in our sprite fetching methods that were causing our sprites to have the lines of their tile drawn on the wrong scanline so they were shifted weirdly, this posed an even bigger problem when we added the ability to move the sprite and it would

look like it was just keeping the same line of tile on the same scanline until it was out of range of the sprite. We were able to fix these two bugs by making small changes in the information used to fetch that data in the Memory Fetcher.

In the end we were able to draw a multi-tiled background and have a correctly drawn sprite that stayed drawn correctly when you moved it around the screen and had the correct priority over the background tiles.

6. Summary

Insights

The biggest challenge in designing the 6502 emulator was in understanding the full extent of its operation. In essence, this project was halfway a research project. Faithful recreation of deprecated hardware requires thorough understanding of its intentional and unintentional function. We made the (right) decision to avoid implementing unofficial opcodes, as no official NES title ever used them. Upon understanding the function and behavior of the 6502, the project becomes much simpler, requiring some ingenuity to be cycle accurate as well as some tedious work to ensure all opcodes are implemented correctly. The other major lesson learned was in implementing the main_6502 module. Simply put, it really ought to be broken up into more modules. The implied addressing mode was controllable via its own module. It likely was worth the effort to build similar modules for the other modules in order to create a cleaner top level module. In hindsight, we ought to have considered a combinational state machine instead, which is closer to what the 6502 actually had. The current implementation does manage cycle accuracy, but there exists potential for contaminating other address locations, since the switching with the PC flag is clocked on both the positive and negative edge of the clock. Ultimately, a second pass at this would be greatly beneficial to the project.

In the same fashion as the CPU one of the biggest challenges of faithfully recreating the NES PPU was fully understanding the way it functioned. We struggled with having enough time for this project because we had to use our first week to read through pages and pages of NES documentation to try and get a firm grasp on how the PPU works. Even when we had created an overall design for the PPU and begun creating our modules we still had to constantly refer back to documentation and change past modules based on specific requirements for the modules we were currently working on. We nearly have a

fully functional PPU, given a week or so more time or having a fuller understanding of how the NES parts specifically worked we could have probably hit full functionality.

Improvements/Next Steps

Moving forward with the CPU, a second pass at the state machine needs to be made, finding a better way to assert results when those results are ready without risking contaminating the memory. The chief worry with the current iteration is the possibility that the rw flag is asserted for too long, resulting in a write to a memory location that should have been read from. The processor is still missing a few parts as well, and, after improving the implementation of the state machine, will need the addition of the relative addressing mode to make branching possible. Once these pieces are assembled properly, then the system will be ready for cycle accurate NES emulation, given a working PPU and data bus connection.

I think that for the PPU we could continue where we left off and have a fully working PPU module fairly quickly. The next step would be going back through the sprite modules (OAM, secondary OAM, Memory Fetcher, and Sprite Renderer) to find where our current bugs with the sprites are located and fix them.

For the project of recreating the NES as a whole, there are many things that we could add on or improve. The biggest step would be finishing the CPU and the PPU, and then connecting them together. Next we would like to focus on implementing our stretch goals and being able to play games from ROMs with the system. Some stretch goals that would be imperative to traditional NES gameplay on this system would be building an interface for the original NES controller, cartridges and also designing and implementing an APU or audio processing unit.

Appendices

Block Diagrams

Diagram 1 - Overall PPU

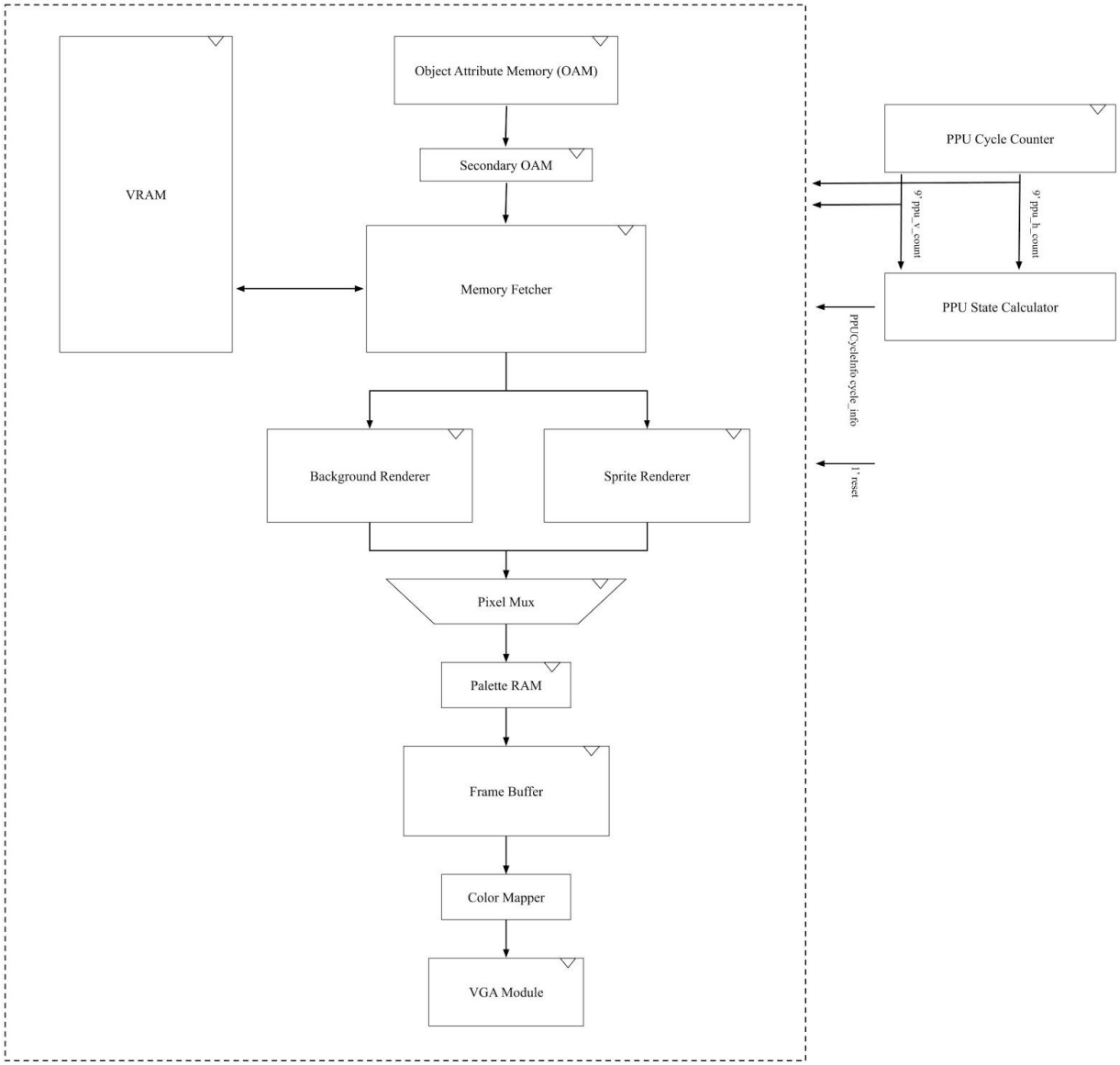


Diagram 2 - PPU Memory

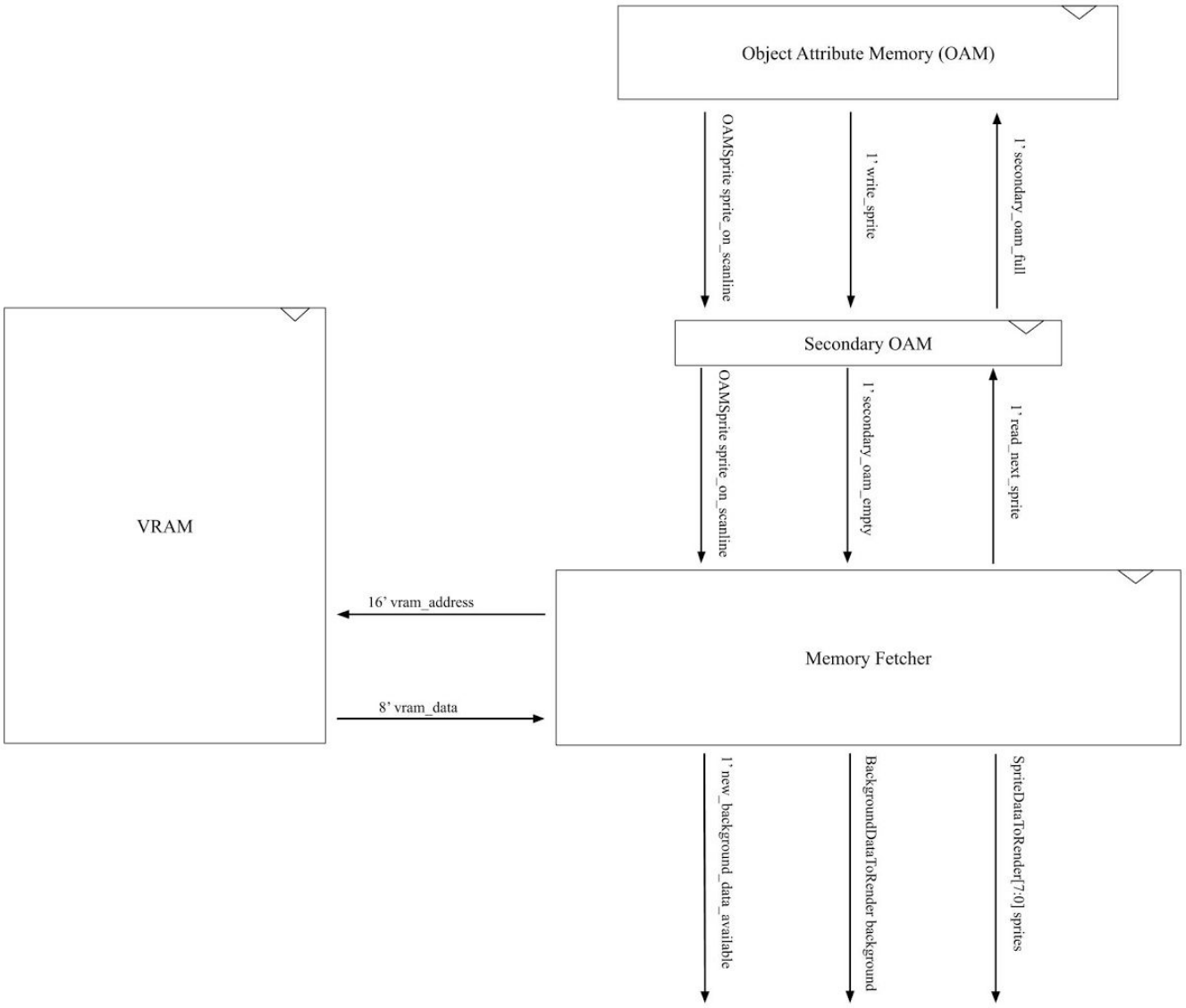


Diagram 3 - PPU Pixel Rendering

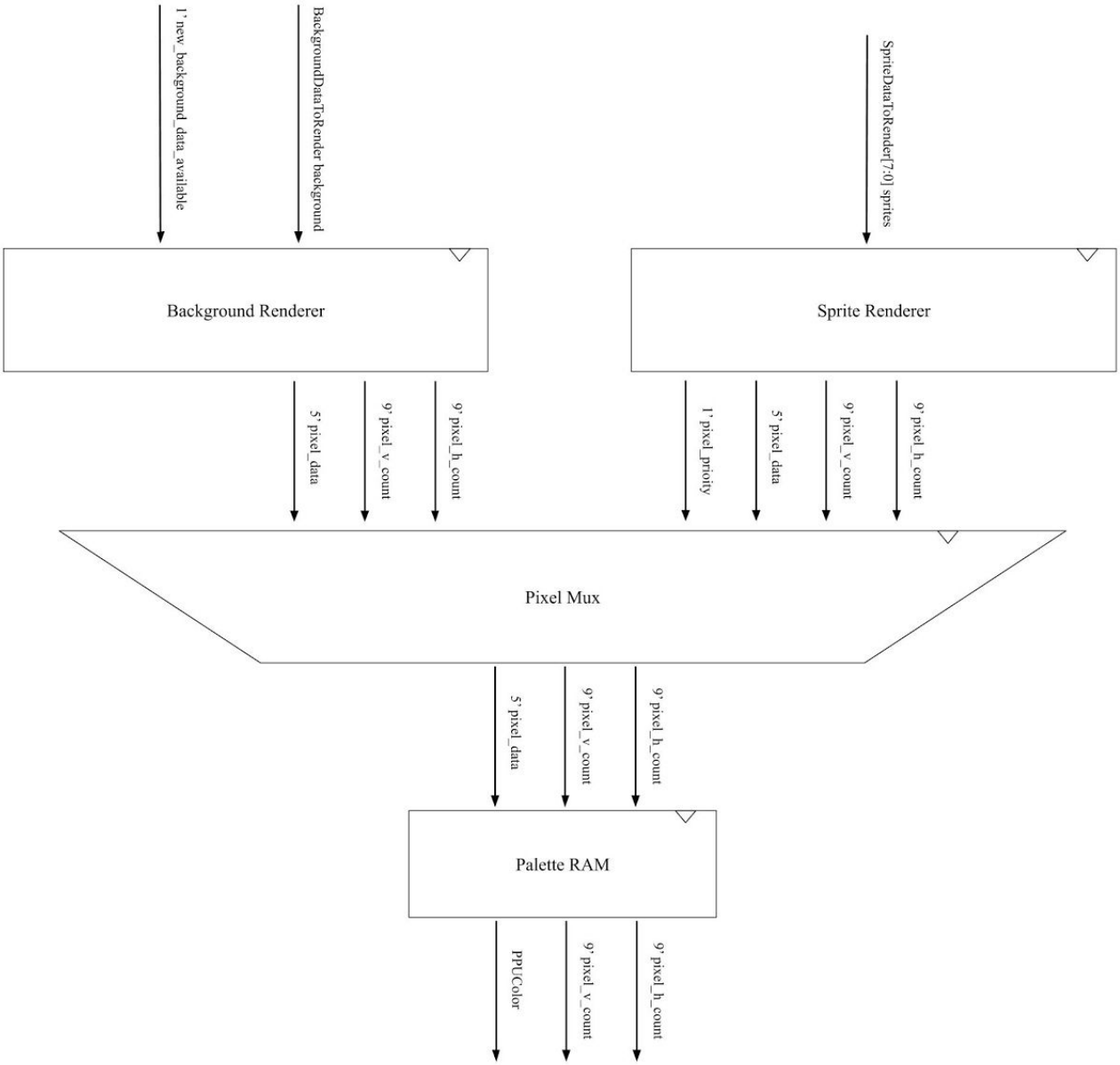


Diagram 4 - PPU Video Output

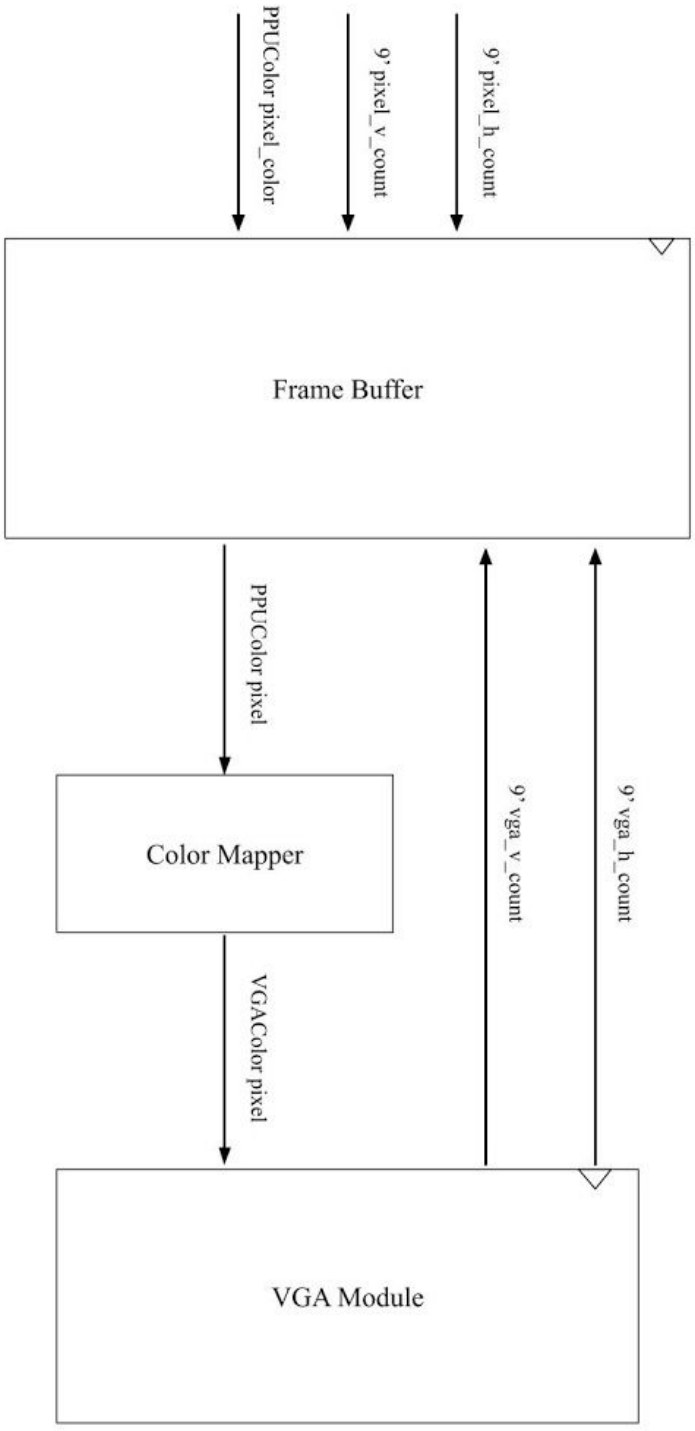
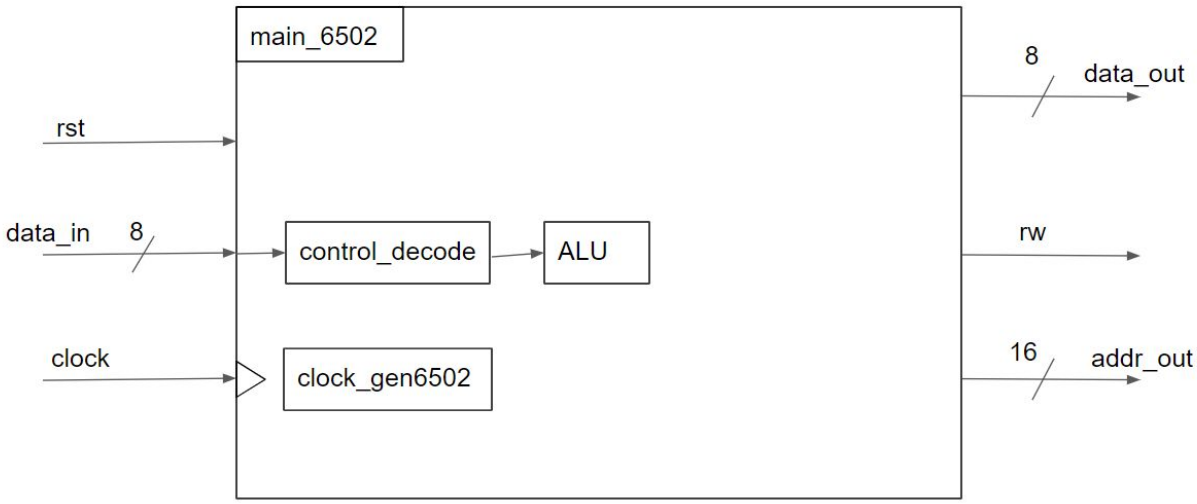


Diagram 5 - main_6502



Figures

Figures taken from “Nintendo Entertainment System Documentation”

Figure 1 - PPU Pattern Tables

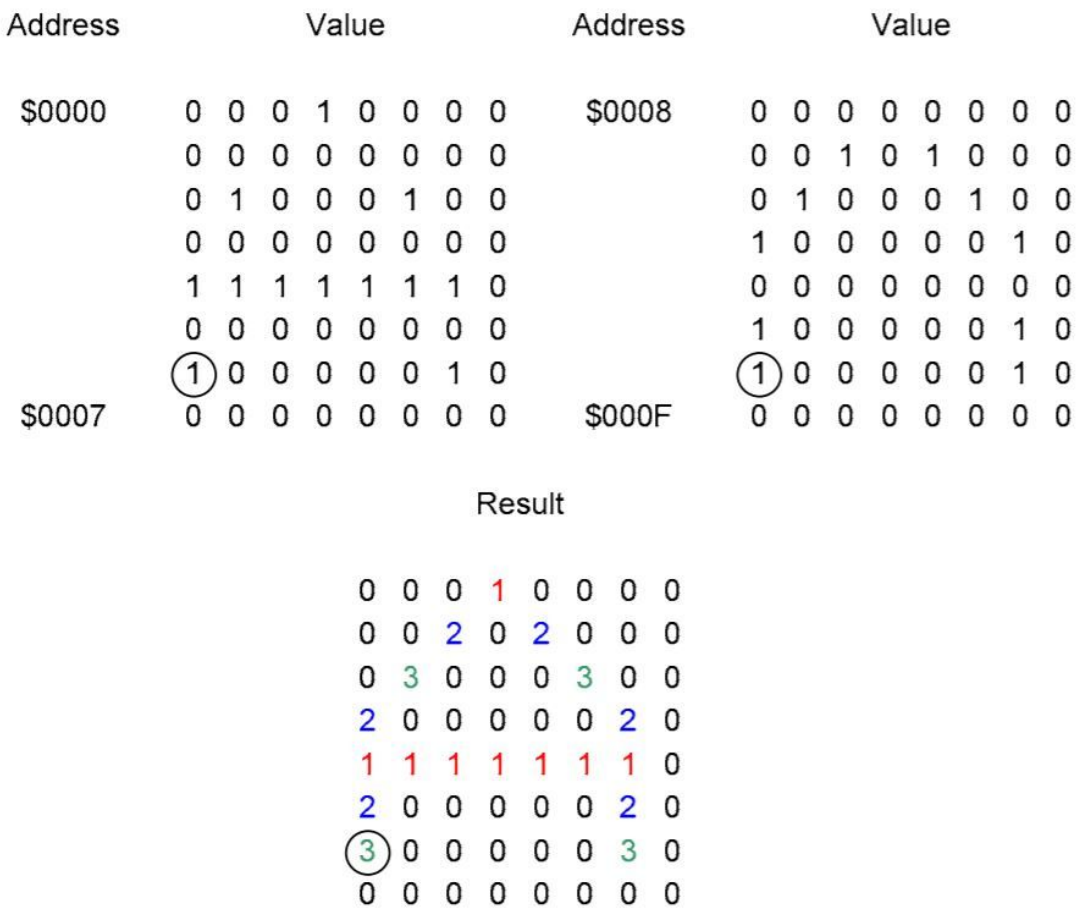
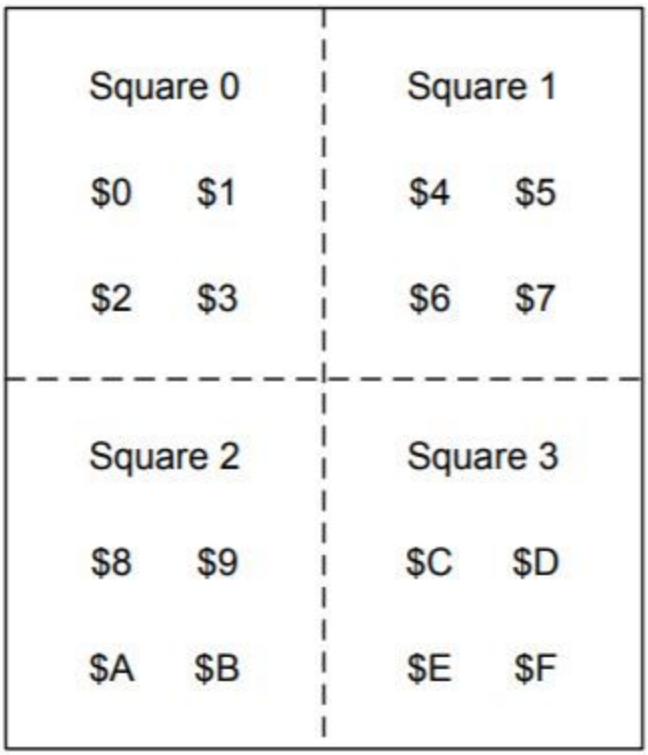


Figure 2 - Attribute Bytes



Each name table has an associated attribute table. Attribute tables hold the upper two bits of the colours for the tiles. Each byte in the attribute table represents a 4x4 group of tiles, so an attribute table is an 8x8 table of these groups. Each 4x4 group is further divided into four 2x2 squares as shown in Figure 2. The 8x8 tiles are numbered $0-$F. The layout of the byte 20 is 33221100 where every two bits specifies the most significant two colour bits for the specified square.

Figure 3 - PPU VRAM

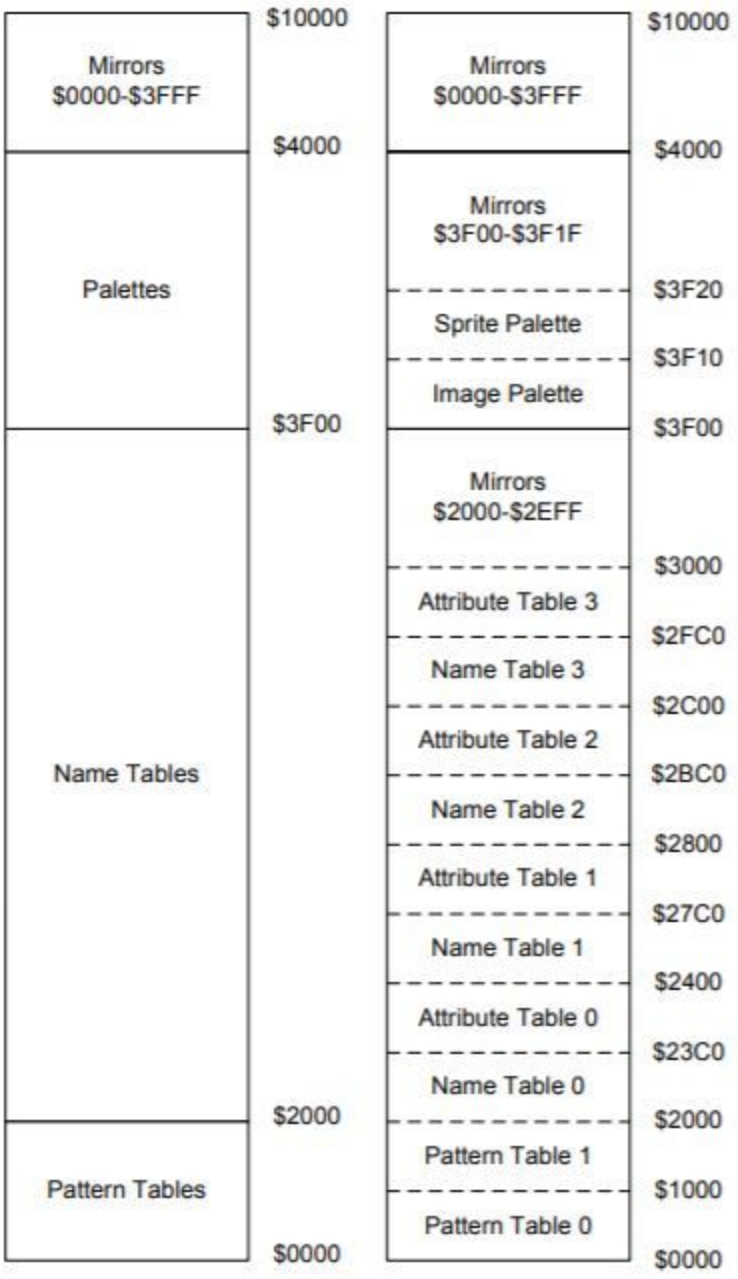


Figure 3-1. PPU memory map.

Verilog

CPU Modules

cpu_constants.sv

```
`timescale 1ns / 1ps
package cpu_constants;

    // operation encoding
    parameter ADC = 1;
    parameter AND1 = 2;
    parameter ASL = 3;
    parameter BCC = 4;
    parameter BCS = 5;
    parameter BEQ = 6;
    parameter BIT1 = 7;
    parameter BMI = 8;
    parameter BNE = 9;
    parameter BPL = 10;
    parameter BRK = 11;
    parameter BVC = 12;
    parameter BVS = 13;
    parameter CLC = 14;
    parameter CLD = 15;
    parameter CLI = 16;
    parameter CLV = 17;
    parameter CMP = 18;
    parameter CPX = 19;
    parameter CPY = 20;
    parameter DEC = 21;
    parameter DEX = 22;
    parameter DEY = 23;
    parameter EOR = 24;
    parameter INC = 25;
    parameter INX = 26;
    parameter INY = 27;
    parameter JMP = 28;
    parameter JSR = 29;
    parameter LDA = 30;
    parameter LDX = 31;
    parameter LDY = 32;
    parameter LSR = 33;
    parameter NOP = 34;
    parameter ORA = 35;
    parameter PHA = 36;
    parameter PHP = 37;
    parameter PLA = 38;
    parameter PLP = 39;
    parameter ROL = 40;
    parameter ROR = 41;
    parameter RTI = 42;
    parameter RTS = 43;
    parameter SBC = 44;
    parameter SEC = 45;
    parameter SED = 46;
    parameter SEI = 47;
    parameter STA = 48;
    parameter STX = 49;
```

```
    parameter STY = 50;
    parameter TAX = 51;
    parameter TAY = 52;
    parameter TSX = 53;
    parameter TXA = 54;
    parameter TXS = 55;
    parameter TYA = 56;

    // addressing modes
    parameter ACCUMULATOR = 1;
    parameter ABSOLUTE = 2;
    parameter ABSOLUTE_XINDEX = 3;
    parameter ABSOLUTE_YINDEX = 4;
    parameter IMMEDIATE = 5;
    parameter IMPLIED = 6;
    parameter INDIRECT = 7;
    parameter INDIRECT_XINDEX = 8;
    parameter INDERECT_YINDEX = 9;
    parameter RELATIVE = 10;
    parameter ZEROPAGE = 11;
    parameter ZEROPAGE_XINDEX = 12;
    parameter ZEROPAGE_YINDEX = 13;

    endpackage
```

alu_6502.sv

```
`timescale 1ns / 1ps
import cpu_constants::*;

module alu_6502(
    input[5:0] op,
    input[7:0] a,
    input[7:0] b,
    input[7:0] curr_status_flag,
    output logic[7:0] result,
    output logic[7:0] status_flag_out
    );

    logic N;
    logic V;
    logic dash;
    logic B;
    logic D;
    logic I;
    logic Z;
    logic C;

    logic[8:0] int_result;

    logic Nn; // generic new flag bits
    logic Vn;
    logic Dn;
    logic In;
    logic Zn;
    logic Cn;

    logic Nv; // for exceptions

    assign Nn = (result[7]); // number always negative when 7th bit high
    assign Zn = (result == 0); // new status flag bit for Z, always true when result is 0

    // FLAG BITS FOLLOW NV-DIZC PATTERN
    always_comb begin
        N = curr_status_flag[7];
```

```
        V = curr_status_flag[6];
       dash = curr_status_flag[5];
        B = curr_status_flag[4];
        D = curr_status_flag[3];
        I = curr_status_flag[2];
        Z = curr_status_flag[1];
        C = curr_status_flag[0];
        // inputs follow convention of a is always from a register, b is always memory or
immediate operand
        case(op)
            ADC: begin
                int_result = a + b + C; // add carry flag
                result = int_result[7:0]; // 8 bit output
                Cn = int_result[8]; // new carry flag
                Vn = (!a[7]&!b[7]&Cn) | (a[7]&b[7]&!Cn); // high if out of signed range
                status_flag_out = {Nn,Vn,dash,B,D,I,Zn,Cn};
            end
            AND1: begin
                result = a & b; // bitwise
                status_flag_out = {Nn,V,dash,B,D,I,Zn,C}; // evaluate result for status flag
            end
            ASL: begin
                Cn = a[7];
                result = a << 1;
                status_flag_out = {Nn, V, dash,B,D,I,Zn,Cn};
            end
            BIT1: begin
                Nv = b[7];
                Vn = b[6];
                result = a & b;
                status_flag_out = {Nv, Vn, dash,B,D,I,Zn,C};
            end
            BRK: begin
                In = 1;
                result = a;
                status_flag_out = {N,V,dash,B,D,In,Z,C};
            end
            CLC: begin
                Cn = 0;
                result = a;
                status_flag_out = {N,V,dash,B,D,I,Z,Cn};
            end
            CLD: begin
                Dn = 0;
                result = a;
                status_flag_out = {N,V,dash,B,Dn,I,Z,C};
            end
            CLI: begin
                In = 0;
                result = a;
                status_flag_out = {N,V,dash,B,D,In,Z,C};
            end
            CLV: begin
                Vn = 0;
                result = a;
                status_flag_out = {N,Vn,dash,B,D,I,Z,C};
            end
            CMP: begin
                Cn = (a > b);
                result = a - b;
                status_flag_out = {Nn,Vn,dash,B,D,I,Zn,Cn};
            end
            CMP: begin
                Cn = (a > b);
                result = a - b;
                status_flag_out = {Nn,Vn,dash,B,D,I,Zn,Cn};
```

```
end
CPX: begin
    Cn = (a > b);
    result = a - b;
    status_flag_out = {Nn,Vn,dash,B,D,I,Zn,Cn};
end
CPY: begin
    Cn = (a > b);
    result = a - b;
    status_flag_out = {Nn,Vn,dash,B,D,I,Zn,Cn};
end
DEC: begin
    result = b - 1;
    status_flag_out = {Nn,V,dash,B,D,I,Zn,C};
end
DEX: begin
    result = a - 1;
    status_flag_out = {Nn,V,dash,B,D,I,Zn,C};
end
DEY: begin
    result = a - 1;
    status_flag_out = {Nn,V,dash,B,D,I,Zn,C};
end
EOR: begin
    result = a ^ b;
    status_flag_out = {Nn,V,dash,B,D,I,Zn,C};
end
INC: begin
    result = b + 1;
    status_flag_out = {Nn,V,dash,B,D,I,Zn,C};
end
INX: begin
    result = a + 1;
    status_flag_out = {Nn, V, dash,B,D,I,Zn,C};
end
INY: begin
    result = a + 1;
    status_flag_out = {Nn, V, dash,B,D,I,Zn,C};
end
LDA: begin
    result = b;
    status_flag_out = {Nn, V, dash,B,D,I,Zn,C};
end
LDX: begin
    result = b;
    status_flag_out = {Nn, V, dash,B,D,I,Zn,C};
end
LDY: begin
    result = b;
    status_flag_out = {Nn, V, dash,B,D,I,Zn,C};
end
LSR: begin
    Cn = a[0];
    result = a >> 1;
    status_flag_out = {Nn,V,dash,B,D,I,Zn,Cn};
end
ORA: begin
    result = a | b;
    status_flag_out = {Nn,V,dash,B,D,I,Zn,Cn};
end
PLA: begin
    result = a;
    status_flag_out = {Nn,V,dash,B,D,I,Zn,Cn};
end
PLP: begin
    result = a;
```

```
                status_flag_out = result;
            end
            ROL: begin
                result = {a[6:0],C};
                Cn = a[7];
                status_flag_out = {Nn,V,dash,B,D,I,Zn,Cn};
            end
            ROR: begin
                result = {C, a[7:1]};
                Cn = a[0];
                status_flag_out = {Nn,V,dash,B,D,I,Zn,Cn};
            end
            SBC: begin
                result = a - b - !C; // intentionally unsigned
                Cn = (a > b);
                Vn = (!a[7]&!b[7]&Cn) | (a[7]&b[7]&!Cn); // same rule as ADC
                status_flag_out = {Nn,Vn,dash,B,D,I,Zn,Cn};
            end
            SEC: begin
                result = a;
                status_flag_out = {N,V,dash,B,D,I,Z,1'b1};
            end
            SED: begin
                result = a;
                status_flag_out = {N,V,dash,B,1'b1,Z,C};
            end
            SEI: begin
                result = a;
                status_flag_out = {N,V,dash,B,1'b1,Z,C};
            end
            default: begin result = a; status_flag_out = {Nn,Vn,dash,B,I,Zn,C}; end
        endcase
    end

endmodule
```

clock_gen6502.sv

```
`timescale 1ns / 1ps

module clock_gen6502(
    input clock,
    input rst,
    output logic phi_1,
    output logic phi_2
    );

    parameter DIVISOR = 11; // NES divides clock signal by 12
    logic[3:0] clk_ct; // ct holder

    assign phi_2 = !phi_1; // phi_2 exactly out of phase with phi_1 always

    always_ff@(posedge clock) begin
        if(rst) begin
            clk_ct <= 4'b0; // initialize values
            phi_1 <= 1;
        end else begin
            phi_1 <= (clk_ct == DIVISOR) ? !phi_1 : phi_1;
            clk_ct <= (clk_ct == DIVISOR) ? 0 : clk_ct + 1;
        end

    end


endmodule
```

control_decode.sv

```
`timescale 1ns / 1ps
import cpu_constants::*;

module control_decode(
    input[7:0] opcode,
    output logic[3:0] addr_mode,
    output logic[5:0] op
    );

    always_comb begin
    // some opcodes must be called out by name, they break pattern
        if(opcode == 8'b0) begin op = BRK; addr_mode = IMPLIED; end
        else if(opcode == 8'h20) begin op = JSR; addr_mode = ABSOLUTE; end
        else if(opcode == 8'h40) begin op = RTI; addr_mode = IMPLIED; end
        else if(opcode == 8'h60) begin op = RTS; addr_mode = IMPLIED; end
        else if(opcode == 8'h08) begin op = PHP; addr_mode = IMPLIED; end
        else if(opcode == 8'h28) begin op = PLP; addr_mode = IMPLIED; end
        else if(opcode == 8'h48) begin op = PHA; addr_mode = IMPLIED; end
        else if(opcode == 8'h68) begin op = PLA; addr_mode = IMPLIED; end
        else if(opcode == 8'h88) begin op = DEY; addr_mode = IMPLIED; end
        else if(opcode == 8'hA8) begin op = TAY; addr_mode = IMPLIED; end
        else if(opcode == 8'hC8) begin op = INY; addr_mode = IMPLIED; end
        else if(opcode == 8'hE8) begin op = INX; addr_mode = IMPLIED; end
        else if(opcode == 8'h18) begin op = CLC; addr_mode = IMPLIED; end
        else if(opcode == 8'h38) begin op = SEC; addr_mode = IMPLIED; end
        else if(opcode == 8'h58) begin op = CLI; addr_mode = IMPLIED; end
        else if(opcode == 8'h78) begin op = SEI; addr_mode = IMPLIED; end
        else if(opcode == 8'h98) begin op = TYA; addr_mode = IMPLIED; end
        else if(opcode == 8'hB8) begin op = CLV; addr_mode = IMPLIED; end
        else if(opcode == 8'hD8) begin op = CLD; addr_mode = IMPLIED; end
        else if(opcode == 8'hF8) begin op = SED; addr_mode = IMPLIED; end
        else if(opcode == 8'h8A) begin op = TXA; addr_mode = IMPLIED; end
        else if(opcode == 8'h9A) begin op = TXS; addr_mode = IMPLIED; end
        else if(opcode == 8'hAA) begin op = TAX; addr_mode = IMPLIED; end
        else if(opcode == 8'hBA) begin op = TSX; addr_mode = IMPLIED; end
        else if(opcode == 8'hCA) begin op = DEX; addr_mode = IMPLIED; end
        else if(opcode == 8'hEA) begin op = NOP; addr_mode = IMPLIED; end
        else if(opcode == 8'h6C) begin op = JMP; addr_mode = INDIRECT; end // special
exception to pattern

        // following opcodes obey xxy10000 pattern, xx specifying branch flag
        // y specifies what the flag is compared to
        else if(opcode[4:0] == 5'b10000) begin
            case(opcode[7:6])
                2'b00: op = opcode[5] ? BMI : BPL; // check against negative flag
                2'b01: op = opcode[5] ? BVS : BVC; // check against overflow flag
                2'b10: op = opcode[5] ? BCS : BCC; // check against carry flag
                2'b11: op = opcode[5] ? BEQ : BNE; // check against zero flag
                default: op = NOP;
            endcase
            addr_mode = RELATIVE; // all branch instructions are relative mode addressed
        end

        // following opcodes obey the aaabbbcc pattern, aaa specifying operation,
        // bbb specifying addressing mode, cc selecting different maps
        else if(opcode[1:0] == 2'b01) begin // cc = 01 opcodes
            case(opcode[7:5])
                3'b000: op = ORA;
                3'b001: op = AND1;
                3'b010: op = EOR;
                3'b011: op = ADC;
                3'b100: op = STA;
```

```
                3'b101: op = LDA;
                3'b110: op = CMP;
                3'b111: op = SBC;
                default: op = NOP;
            endcase
            case(opcode[4:2])
                3'b000: addr_mode = ZEROPAGE_XINDEX;
                3'b001: addr_mode = ZEROPAGE;
                3'b010: addr_mode = IMMEDIATE;
                3'b011: addr_mode = ABSOLUTE;
                3'b100: addr_mode = ZEROPAGE_YINDEX;
                3'b101: addr_mode = ZEROPAGE_XINDEX;
                3'b110: addr_mode = ABSOLUTE_YINDEX;
                3'b111: addr_mode = ABSOLUTE_XINDEX;
                default: addr_mode = IMPLIED;
            endcase
        end
        else if(opcode[1:0] == 2'b10) begin // cc = 10 opcodes
            case(opcode[7:5])
                3'b000: op = ASL;
                3'b001: op = ROL;
                3'b010: op = LSR;
                3'b011: op = ROR;
                3'b100: op = STX;
                3'b101: op = LDX;
                3'b110: op = DEC;
                3'b111: op = INC;
                default: op = NOP;
            endcase
            case(opcode[4:2])
                3'b000: addr_mode = IMMEDIATE;
                3'b001: addr_mode = ZEROPAGE;
                3'b010: addr_mode = ACCUMULATOR;
                3'b011: addr_mode = ABSOLUTE;
                3'b101: addr_mode = (op == STX || op == LDX) ? ZEROPAGE_YINDEX :
ZEROPAGE_XINDEX;
                3'b111: addr_mode = (op == LDX) ? ABSOLUTE_YINDEX: ABSOLUTE_XINDEX;
                default: addr_mode = IMPLIED;
            endcase
        end else if(opcode[1:0] == 2'b00) begin // cc = 00 opcodes
            case(opcode[7:5])
                3'b001: op = BIT1;
                3'b010: op = JMP;
                3'b100: op = STY;
                3'b101: op = LDY;
                3'b110: op = CPY;
                3'b111: op = CPX;
                default: op = NOP;
            endcase
            case(opcode[4:2])
                3'b000: addr_mode = IMMEDIATE;
                3'b001: addr_mode = ZEROPAGE;
                3'b011: addr_mode = ABSOLUTE;
                3'b101: addr_mode = ZEROPAGE_XINDEX;
                3'b111: addr_mode = ABSOLUTE_XINDEX;
                default: addr_mode = IMPLIED;
            endcase
        end
    end

endmodule
```

cpu_demo.sv

```
module cpu_demo(
    input[15:0] sw,
    input btnc,
    input btnr,
    input clk_100mhz,
    output logic [15:0] led,
    output ca, cb ,cc ,cd, ce, cf, cg,
    output[7:0] an
    );
    logic[7:0] A;
    logic[7:0] M;
    logic[7:0] X;
    logic[7:0] Y;
    logic[7:0] PCL;
    logic[7:0] PCH;
    logic[15:0] address;
    logic[7:0] data_out;
    logic phi_2;

    logic clean_btnr;
    logic[6:0] segments;
    assign {cg, cf, ce, cd, cc, cb, ca} = segments[6:0];
    logic[32:0] data_in;

    debounce
btnr_cleaner(.clk_in(clk_100mhz),.rst_in(btnc),.bouncey_in(btnr),.clean_out(clean_btnr));
    display_7seg(.clk_in(clk_100mhz),.data_in(data_in),.seg_out(segments),.strobe_out(an));
    main_6502
demo_6502(.rst(btnc),.clock(clean_btnr),.data_in(sw[7:0]),.data_out(data_out),.address(address
),

.rw(led[0]),.phi_2(phi_2),.A(A),.M(M),.X(X),.Y(Y),.PCL(PCL),.PCH(PCH));

    always_comb begin
        case(sw[15:14])
            2'b00: data_in = {Y,X,M,A}; // display register values
            2'b01: data_in = {address, PCH, PCL}; // display program counter and output
            2'b10: data_in = {24'b0, data_out}; // display 6502 output
            default: data_in = {Y,X,M,A};
        endcase
    end

endmodule
```

implied_handler.sv

```
`timescale 1ns / 1ps
// use this module to control implied addressing mode

module implied_handler(
    input[5:0] op,
    output logic brk,
    output logic pushing,
    output logic pulling
    );

    always_comb begin
        case(op)
            BRK: begin brk = 1; pushing = 0; pulling = 0; end // detect brk command
            PHA: begin brk = 0; pushing = 1; pulling = 0; end // detect push to stack
```

```
            PHP: begin brk = 0; pushing = 1; pulling = 0; end
            PLA: begin brk = 0; pushing = 0; pulling = 1; end
            PLP: begin brk = 0; pushing = 0; pulling = 1; end
            default: begin brk = 0; pushing = 0; pulling = 0; end // all other are 2 byte
instructions handled by ALU

        endcase
    end
endmodule
```

mem_write.sv

```
`timescale 1ns / 1ps


module mem_write(
    input[5:0] op,
    input[3:0] addr_mode,
    output logic long_wea
    );

    always_comb begin
        case(op)
            ASL: long_wea = !(addr_mode == ACCUMULATOR);
            DEC: long_wea = 1;
            INC: long_wea = 1;
            LSR: long_wea = !(addr_mode == ACCUMULATOR);
            ROL: long_wea = !(addr_mode == ACCUMULATOR);
            ROR: long_wea = !(addr_mode == ACCUMULATOR);
            default: long_wea = 0; // writing memory is specific to a few operations

        endcase
    end
endmodule
```

memory_controller.sv

```
`timescale 1ns / 1ps
module memory_controller(
    input[7:0] outer_data,
    input[7:0] inner_data,
    input[7:0] ADDR_H_RAW,
    input[7:0] ADDR_L,
    input[7:0] PCL,
    input[7:0] PCH_RAW,
    input pc_flag,
    output logic[7:0] mem_dat_out,
    output logic[15:0] address
    );

    logic[7:0] ADDR_H; // need to demirror both PC and ADDR accesses
    logic[7:0] PCH_OUT;
    // only need to mirror for addresses below $2000 and above $7FF
    always_comb begin
        ADDR_H = (ADDR_H_RAW < 8'h20) ? {5'b0, ADDR_H_RAW[2:0]} : ADDR_H_RAW;
        PCH_OUT = (PCH_RAW < 8'h20) ? {5'b0, PCH_RAW[2:0]} : PCH_RAW;
        address = pc_flag ? {PCH_OUT,PCL} : {ADDR_H,ADDR_L};
        mem_dat_out = (address[15:8] < 8'h20) ? inner_data : outer_data;
    end
endmodule
```

reg_sel.sv

```
`timescale 1ns / 1ps
import cpu_constants::*;


module reg_sel(
    input[5:0] op,
    input[3:0] addr_mode,
    input[7:0] x,
    input[7:0] y,
    input[7:0] a,
    input[7:0] m,
    input[7:0] s,
    output logic[7:0] reg_out
    );

    // if not one of the cases listed, the register to use is a if at all
    // special cases ASL, LSR, ROL, ROR can use memory as input "register"
    always_comb begin
        case(op)
            ASL: reg_out = (addr_mode == ACCUMULATOR) ? a : m;
            CPX: reg_out = x;
            CPY: reg_out = y;
            DEX: reg_out = x;
            DEY: reg_out = y;
            INX: reg_out = x;
            INY: reg_out = y;
            JSR: reg_out = m;
            LSR: reg_out = (addr_mode == ACCUMULATOR) ? a : m;
            ROL: reg_out = (addr_mode == ACCUMULATOR) ? a : m;
            ROR: reg_out = (addr_mode == ACCUMULATOR) ? a : m;
            STX: reg_out = x;
            STY: reg_out = y;
            TSX: reg_out = s;
            TXA: reg_out = x;
            TXS: reg_out = x;
            TYA: reg_out = y;
            default: reg_out = a;
        endcase
    end
endmodule
```

main_6502.sv

```
`timescale 1ns / 1ps
import cpu_constants::*;

module main_6502(
    input rdy,
    input irq_low,
    input nmi_low,
    input rst,
    input so_low,
    input clock,
    input[7:0] data_in,
    output logic[7:0] data_out,
    output logic[15:0] address,
    output logic sync,
    output logic rw,
    output logic phi_2,
    output logic[7:0] A,
    output logic[7:0] M,
```

```
    output logic[7:0] X,
    output logic[7:0] Y,
    output logic[7:0] PCL,
    output logic[7:0] PCH // for demo purposes, added registers and program counter as outputs
    );

    // logic[7:0] A; // accumulator register
    // logic[7:0] M; // memory read placeholder
    // logic[7:0] X; // index pointer
    // logic[7:0] Y; // index pointer
    logic[7:0] P; // status flag bits  N V - B D I Z C  *note D unused here
    logic[7:0] S; // stack pointer
    //logic[7:0] PCL; // program counter bytes
    //logic[7:0] PCH;
    logic[7:0] ADDR_H_RAW;// raw addresses to be demirrored
    logic[7:0] ADDR_L; // address line bytes
    logic[7:0] ADDR_H;
    logic[8:0] addr_calc; // allow for carry
    logic[7:0] PCL_HOLD; // pc holder for jsr instruction to push
    logic[7:0] PCH_HOLD; // pc holder for jsr instruction
    logic[7:0] PCL_JUMP; // pc holder for jsr instruction to jump to
    logic[7:0] PCH_JUMP;

    logic pc_flag; // high when address bus should recieve value in program counter, low when
should recieve from addr_bus

    logic phi_1;

    logic[3:0] addr_mode;
    logic[5:0] op;
    logic[5:0] working_op;
    logic[3:0] working_addr;
    logic[2:0] cycle;
    logic[7:0] result; // ALU output
    logic[7:0] flag_out;

    logic[7:0] operand_a; // ALU operands, selected by reg_sel instance input_sel
    logic[7:0] operand_b;

    logic internal_wea; // internal write enable
    logic internal_ena; // internal ram enable, decide when to use BRAM
    logic ready_flag; // assert when ALU result is ready
    logic[10:0] internal_addr; // internal address space
    logic[7:0] internal_out; // internal memory out
    logic[7:0] mem_data; // either input from data line or from internal RAM on MEMORY READ
ONLY

    logic long_wea; // flag for enabling cpu to write to memory based on recieved instruction
    logic shifting; // flag for when shifts take place, only important when memory is the
target
    logic STORE; // flag for STA, STX, and STY

    logic brk; // variables for control signals of implied addressing mode
    logic pulling;
    logic pushing;

    assign STORE = ((working_op == STA) | (working_op == STX) | (working_op == STY)); //
control signal for when memory will be written to from registers
    assign shifting = ((working_op == ASL) | (working_op == LSR) | (working_op == ROL) |
(working_op == ROR));
    assign internal_addr = address[10:0];
    assign internal_wea = (!rw & internal_ena);

    // instantiate helper modules
    clock_gen6502 clock_gen_inst(.clock(clock),.rst(rst),.phi_1(phi_1),.phi_2(phi_2));
    control_decode instr(.opcode(data_in),.addr_mode(addr_mode),.op(op));
```

```
    alu_6502
alu_inst(.op(working_op),.a(operand_a),.b(operand_b),.curr_status_flag(P),.result(result),.sta
tus_flag_out(flag_out));
    reg_sel
input_sel(.op(working_op),.addr_mode(working_addr),.x(X),.y(Y),.a(A),.m(M),.s(S),.reg_out(oper
and_a));
    mem_write cpu_inst(.op(working_op),.addr_mode(working_addr),.long_wea(long_wea));
    memory_controller
controller_inst(.outer_data(data_in),.inner_data(internal_out),.ADDR_H_RAW(ADDR_H_RAW),.ADDR_L
(ADDR_L),.PCL(PCL),.PCH_RAW(PCH),.pc_flag(pc_flag),.mem_dat_out(mem_data),.address(address));
    internal_6502 ram_6502(.clka(clock), .ena(1),
.wea(internal_wea),.addra(internal_addr),.dina(result),.douta(internal_out));
    implied_handler
implied_mode(.op(working_op),.brk(brk),.pushing(pushing),.pulling(pulling));

    always_ff@(posedge phi_1) begin
        if(rst) begin
            // initialize all registers
            //sync <= 0;
            data_out <= 0;
            X <= 8'b0;
            Y <= 8'b0;
            S <= 8'hFF;
            PCH <= 8'hFF;
            PCL <= 8'h00;
            ADDR_H_RAW <= 8'b0;
            ADDR_H <= 8'b0;
            ADDR_L <= 8'b0;
            rw <= 1'b1;
            cycle <= 0;
            A <= 8'b0;
            P <= 8'b00110000; // dash and B start as high for status flag bits
            M <= 8'b0;
            pc_flag <= 1'b1; // high if using the pc counter as the address access
            ready_flag <= 1'b0;
            internal_wea <= 1'b0;

        end else begin
            case(cycle)
                3'd0: begin
                    ready_flag <= 1'b0;
                    rw <= 1'b1; // write active low / read first cycle
                    working_op <= op; // use registers to hold op and addr_mode steady steady
when data_in changes
                    working_addr <= addr_mode;
                    cycle <= cycle + 1; // increment cycle count
                end

                3'd1: begin
                    // next step depends on addressing mode
                    case(working_addr)
                        ACCUMULATOR: begin ready_flag <= 1; cycle <= 0; end // two byte
instructions terminate on this cycle
                        ABSOLUTE: begin ADDR_L <= mem_data; cycle <= cycle + 1; end // receive
first byte of memory read/write location
                        ABSOLUTE_XINDEX: begin ADDR_L <= mem_data + X; addr_calc <= mem_data +
X; cycle <= cycle + 1; end
                        ABSOLUTE_YINDEX: begin ADDR_L <= mem_data + Y; addr_calc <= mem_data +
Y; cycle <= cycle + 1; end
                        IMMEDIATE: begin operand_b <= data_in; ready_flag <= 1; cycle <= 0;
end // receive byte for operation, result ready on negedge
                        IMPLIED: begin
                            if(brk) begin
                                cycle <= cycle + 1;
                            end else if(pushing | pulling) begin
                                ADDR_H_RAW <= 8'h01;
```

```
                            ADDR_L <= S;
                        end else begin
                            cycle <= 0; // ALU will handle if not pushing, pulling, or brk
                            ready_flag <= 1;
                        end
                    end
                    ZEROPAGE: begin ADDR_H_RAW <= 0; ADDR_L <= mem_data; pc_flag <= 0;
cycle <= cycle + 1; end
                    ZEROPAGE_XINDEX: begin ADDR_H_RAW <= 0; ADDR_L <= mem_data; pc_flag <=
0; cycle <= cycle + 1; end // don't increment address yet for sake of cycle accuracy
                endcase
            end
            3'd2: begin
                case(working_addr)
                    ABSOLUTE: begin
                        ADDR_H_RAW <= mem_data;
                        if(working_op == JMP) begin
                            PCL <= ADDR_L;
                            PCH <= ADDR_H_RAW;
                            cycle <= 0;
                        end else if(working_op == JSR) begin
                            PCL_JUMP <= ADDR_L;
                            PCH_JUMP <= mem_data;
                            PCL_HOLD <= PCL;
                            PCH_HOLD <= PCH;
                            cycle <= cycle + 1;
                        end else begin
                            cycle <= cycle + 1;
                            pc_flag <= 1'b0;
                        end
                    end // assign high byte
                    ABSOLUTE_XINDEX: begin
                        ADDR_H_RAW <= mem_data;
                        pc_flag <= addr_calc[8]; // can use this address if there's no
carry
                        cycle <= cycle + 1;
                        end
                    ABSOLUTE_YINDEX: begin
                        ADDR_H_RAW <= mem_data;
                        pc_flag <= addr_calc[8]; // can use this address if there's no
carry
                        cycle <= cycle + 1;
                        end
                    ZEROPAGE: begin
                        if(shifting) begin // need to feed data_in to the M register so
ALU     performs operation on right data
                            M <= mem_data;
                        end else begin
                            operand_b <= mem_data; // otherwise feed data_in as operand b
                        end
                        if(long_wea) begin // if shifting or decrementing memory continue
                            cycle <= cycle + 1;
                        end else begin
                            rw <= !STORE; // rw on STA, STX, STY
                            ready_flag <= 1;
                            cycle <= 0; // if reading, data is ready
                        end
                    end

                    ZEROPAGE_XINDEX: begin ADDR_L <= ADDR_L + X; cycle <= cycle + 1; end
//increment

                endcase
            end

            3'd3: begin
```

```
                    case(working_addr)
                        ABSOLUTE: begin
                            if(shifting) begin // need to feed data from memory to the M
register so ALU performs operation on right data
                                M <= mem_data;
                            end else begin
                                operand_b <= mem_data; // otherwise feed data from memory as
operand b
                            end
                            if(long_wea) begin // if writing operation, continue
                                cycle <= cycle + 1;
                            end else if(working_op != JSR) begin // jsr special exception
                                rw <= !STORE;
                                ready_flag <= 1;
                                cycle <= 0; // if reading, data is ready
                            end else begin
                                ADDR_H_RAW <= 8'h01;
                                ADDR_L <= S;
                                M <= PCL_HOLD;
                                rw <= 0;
                                pc_flag <= 0;
                                ready_flag <= 1;
                                cycle <= cycle + 1;
                                S <= S - 1;
                            end
                        end
                        ABSOLUTE_XINDEX: begin
                            ADDR_H_RAW <= ADDR_H_RAW + addr_calc[8]; // add carry if there,
more accurate to 6502 this way
                            if(!long_wea) begin
                                operand_b <= mem_data;
                                rw <= (!STORE & !addr_calc[8]);
                                ready_flag <= !pc_flag; // if pc_flag low ready to calculate
                                cycle <= (pc_flag) ? cycle + 1 : 0; // continue if pc_flag
high
                                pc_flag <= 0;
                            end else begin
                                cycle <= cycle + 1;
                            end
                        end
                        ABSOLUTE_YINDEX: begin
                            ADDR_H_RAW <= ADDR_H_RAW + addr_calc[8]; // add carry if there,
more accurate to 6502 this way
                            if(!long_wea) begin
                                operand_b <= mem_data;
                                rw <= (!STORE & !addr_calc[8]);
                                ready_flag <= !pc_flag; // if pc_flag low ready to calculate
                                cycle <= (pc_flag) ? cycle + 1 : 0; // continue if pc_flag
high
                                pc_flag <= 0;
                            end else begin
                                cycle <= cycle + 1;
                            end
                        end
                        ZEROPAGE: cycle <= cycle + 1;
                        ZEROPAGE_XINDEX: begin
                            if(shifting) begin
                                M <= mem_data;
                            end else begin
                                operand_b <= mem_data;
                            end
                            if(long_wea) begin
                                cycle <= cycle + 1;
                            end else begin
                                rw <= !STORE;
                                ready_flag <= 1;
```

```
                            cycle <= 0;
                        end
                    end
                endcase
            end

            3'd4: begin
                case(working_addr)
                    ABSOLUTE: begin
                        cycle <= cycle + 1;
                        if(working_op == JSR) begin
                            M <= PCH_HOLD;
                            ADDR_H_RAW <= 8'h01;
                            ADDR_L <= S;
                            rw <= 0;
                            ready_flag <= 1;
                            pc_flag = 0;
                            cycle <= cycle + 1;
                            S <= S - 1;

                        end
                    end
                    ABSOLUTE_XINDEX: begin // if at this point, ready to calc just like
absolute mode
                        if(!long_wea) begin
                            rw <= !STORE;
                            operand_b <= mem_data;
                            ready_flag <= 1;
                            cycle <= 0;
                        end else begin
                            cycle <= cycle + 1; // if writing, this addressing mode takes
7 cycles to complete
                        end
                    end
                    ABSOLUTE_YINDEX: begin // if at this point, ready to calc just like
absolute mode
                        if(!long_wea) begin
                            rw <= !STORE;
                            operand_b <= mem_data;
                            ready_flag <= 1;
                            cycle <= 0;
                        end else begin
                            cycle <= cycle + 1; // if writing, this addressing mode takes
7 cycles to complete
                        end
                    end
                    ZEROPAGE: begin ready_flag <= 1; rw <= 0; cycle <= 0; end
                    ZEROPAGE_XINDEX: cycle <= cycle + 1; // preserving cycle accuracy
                endcase
            end
            3'd5: begin
                case(working_addr)
                    ABSOLUTE: begin
                        if(working_op != JSR) begin
                            ready_flag <= 1; rw <= 0; cycle <= 0;
                        end else begin
                            PCH <= PCH_JUMP;
                            PCL <= PCL_JUMP;
                            cycle <= 0;
                        end
                    end
                    ABSOLUTE_XINDEX: cycle <= cycle + 1;
                    ABSOLUTE_YINDEX: cycle <= cycle + 1;
                    ZEROPAGE_XINDEX: begin ready_flag <= 1; rw <= 0; cycle <= 0; end
                endcase
            end
```

```
                3'd6: begin
                    case(working_addr)
                        ABSOLUTE_XINDEX: begin M <= mem_data; ready_flag <= 1; rw <= 0; cycle
<=0 ;end
                        ABSOLUTE_YINDEX: begin M <= mem_data; ready_flag <= 1; rw <= 0; cycle
<=0 ;end
                    endcase
                end
                default: begin cycle <= 3'd0; PCH <= 8'b0; PCL <= 8'b0; end
            endcase
            PCH <= (PCL == 8'hFF) ? PCH + 1 : PCH; // increment program counter with carry
            PCL <= PCL + 1; // always increment program counter
        end
    end

    always_ff@(negedge phi_1) begin
        if(ready_flag) begin
            if(!long_wea) begin
                case(working_op)
                    LDX: X <= result;
                    LDY: Y <= result;
                    JSR: A <= A;
                    default: A <= result;
                endcase
            end
            P <= (working_op == JSR) ? P : flag_out;
            ready_flag <= 1'b0;
            pc_flag <= 1'b1;
            rw <= 1;
        end
    end

    always_comb begin
        if(rst) begin
            data_out = 0;
            internal_out = 0;
        end
        if(ready_flag & (long_wea | STORE)) begin
            data_out = result;
            internal_out = result;
        end

    end

endmodule
```

alu_tb.sv

```
`timescale 1ns / 1ps
Import cpu_constants::*;
module alu_tb;

    // inputs
    logic[6:0] op;
    logic[7:0] a;
    logic[7:0] b;
    logic[7:0] curr_status_flag;

    //outputs
    logic[7:0] result;
    logic[7:0] status_flag_out;
```

```
    alu_6502
uut(.op(op),.a(a),.b(b),.curr_status_flag(curr_status_flag),.result(result),.status_flag_out(s
tatus_flag_out));

    initial begin
        #10;
        a = 8'hFF;
        b = 8'h01;
        op = ADC;
        curr_status_flag = 8'b00110000;
        #10;
        $display("ADC $FF + $01 with carry flag not set: %2h",result);
        $display("Status Flag: %8b",status_flag_out);
        #10;
        op = SBC;
        a = 8'h01;
        b = 8'h01;
        curr_status_flag = 8'b00110000;
        #10;
        $display("SBC $01 - $01 with carry flag not set: %2h",result);
        $display("Status Flag: %8b",status_flag_out);
        #10;

        op = AND1;
        a = 8'h81;
        b = 8'h83;
        #10;
        $display("AND 81 & 83: %2h",result);
        $display("Status Flag: %8b",status_flag_out);

        op = ASL;
        a = 8'h83;
        b = 8'h83;
        #10;
        $display("ASL: %2h",result);
        $display("Status Flag: %8b",status_flag_out);


        op = ORA;
        a = 8'h83;
        b = 8'h84;
        #10;
        $display("ORA: %2h",result);
        $display("Status Flag: %8b",status_flag_out);

        op = ROL;
        a = 8'h83;
        curr_status_flag = 8'b00110000;
        #10;
        $display("ROL, CARRY CLEAR: %2h",result);
        $display("Status Flag: %8b",status_flag_out);

        op = ROL;
        a = 8'h83;
        curr_status_flag = 8'b00110001;
        #10;
        $display("ROL, CARRY SET: %2h",result);
        $display("Status Flag: %8b",status_flag_out);

        op = ROR;
        a = 8'h83;
        curr_status_flag = 8'b00110000;
        #10;
        $display("ROR, CARRY CLEAR: %2h",result);
        $display("Status Flag: %8b",status_flag_out);
```

```
        op = ROR;
        a = 8'h83;
        curr_status_flag = 8'b00110001;
        #10;
        $display("ROR, CARRY SET: %2h",result);
        $display("Status Flag: %8b",status_flag_out);

        op = BIT1;
        a = 8'h24;
        b = 8'h83;
        curr_status_flag = 8'b00110000;
        #10;
        $display("BIT: %2h",result);
        $display("Status Flag: %8b",status_flag_out);

    end

endmodule
```

clk_gen_tb.sv

```
`timescale 1ns / 1ps

module clk_gen_tb;

    // inputs
    logic clock;
    logic rst;

    //outputs
    logic phi_1;
    logic phi_2;

    clock_gen6502 uut(.clock(clock),.rst(rst),.phi_1(phi_1),.phi_2(phi_2));
    always begin
        #2;
        clock = !clock;
    end

    initial begin
        clock = 0;
        rst = 0;
        #10;
        rst = 1;
        #10;
        rst = 0;
        #100;
    end

endmodule
```

main_6502_tb.sv

```
`timescale 1ns / 1ps
import cpu_constants::*;

module main_6502_tb;

    // inputs
    logic irq_low;
    logic nmi_low;
    logic rst;
    logic so_low;
```

```
    logic clock;
    logic[7:0] data_in;

    // outputs
    logic[7:0] data_out;
    logic[15:0] address;
    logic sync;
    logic rw;
    logic phi_2;

    parameter ROLA = 8'h2A;

    main_6502
uut(.irq_low(irq_low),.nmi_low(nmi_low),.rst(rst),.so_low(so_low),.clock(clock),.data_in(data_
in),
                  .data_out(data_out),.address(address),.sync(sync),.rw(rw),.phi_2(phi2));

    always begin
        #1;
        clock = !clock;
    end

    initial begin
        clock = 0;
        irq_low = 0;
        nmi_low = 0;
        rst = 1;
        so_low = 0;
        data_in = 8'hEA;
        #12;
        rst = 0;
        #24;
        data_in = 8'hA9;
        #48;
        data_in = 8'h67;
        #48;
        data_in = 8'h85;
        #48;
        data_in = 8'h01;
        #48;
        data_in = 8'h01;
        #48;
        data_in = 8'hA6;
        #48;
        data_in = 8'h01;
        #200;

    end


endmodule
```

PPU Modules

Top.sv

```
import PPU_Types::*;
```

```
module top(
        input clk_100mhz,
        input[15:0] sw,
        input btnc, btnu, btnl, btnr, btnd,
        output[3:0] vga_r,
        output[3:0] vga_b,
        output[3:0] vga_g,
        output vga_hs,
        output vga_vs,
        output[15:0] led
    );

    parameter NUM_PIXELS_HORIZONTAL = 256;
    parameter NUM_PIXELS_VERTICAL = 240;

    parameter SCALE_FACTOR = 2;
    parameter SCALED_HORIZONTAL_BOUND = NUM_PIXELS_HORIZONTAL * SCALE_FACTOR;
    parameter SCALED_VERTICAL_BOUND = NUM_PIXELS_VERTICAL * SCALE_FACTOR;

    logic reset;
    assign reset = btnc;

    logic up, down, left, right;
    assign up = btnu;
    assign down = btnd;
    assign left = btnl;
    assign right = btnr;


    logic vga_clock_wire;
    logic ppu_clock_wire;

    top_clock(
            .clk_in(clk_100mhz),
            .reset(reset),

            .vga_clock_65mhz(vga_clock_wire),
            .ppu_clock_5mhz(ppu_clock_wire)
    );

    /*
        VGA module
     */
     logic[9:0] vga_h_count;
     logic[9:0] vga_v_count;
     logic vga_vsync_out, vga_hsync_out;
     logic vga_blank_out;
     vga_1024 vga_module(
        .vclock_in(vga_clock_wire),
        .hcount_out(vga_h_count),
        .vcount_out(vga_v_count),
        .vsync_out(vga_vsync_out),
        .hsync_out(vga_hsync_out),
        .blank_out(vga_blank_out)
     );

     /*
        PPU module
    */

//    logic[8:0] ppu_v_count;
//    logic[8:0] ppu_h_count;
//    PPUCounter ppu_counter(
//        .clock_in(ppu_clock_wire),
//        .reset_in(reset),
```

```
//        .ppu_v_count_out(ppu_v_count),
//        .ppu_h_count_out(ppu_h_count)
//    );


    logic [8:0] pixel_h_count;
    logic [8:0] pixel_v_count;
    PPUColor pco;
    PPU ppu(
        .clock_in(ppu_clock_wire),
        .reset_in(reset),

        // Interface with buttons for sprite control
        .up(up),
        .down(down),
        .left(left),
        .right(right),

        // Interface with FrameBuffer
        .pixel_color_out(pco),
        .pixel_h_count_out(pixel_h_count),
        .pixel_v_count_out(pixel_v_count)
    );

    logic[8:0] pixel_h_count_fix;
    assign pixel_h_count_fix = pixel_h_count < NUM_PIXELS_HORIZONTAL ? pixel_h_count: 0;
    logic[8:0] pixel_v_count_fix;
    assign pixel_v_count_fix = pixel_v_count < NUM_PIXELS_VERTICAL ? pixel_v_count: 0;


    /*
        Frame Buffer
    */
    logic frame_number;
    PPUColor frame_buffer_out;

    FrameBuffer frames(
        .reset_in(reset),

        .frame_num_in(frame_number),

        // Output pixel interface
        .vga_clock_in(vga_clock_wire),
        .vga_v_count_in(vga_v_count[9:1]), // Cut off the last bit so that it scales the image
by 2x
        .vga_h_count_in(vga_h_count[9:1]),
        .pixel_color_out(frame_buffer_out),

        // Input pixel interface
        .ppu_clock_in(ppu_clock_wire),
        .ppu_v_count_in(pixel_v_count_fix),
        .ppu_h_count_in(pixel_h_count_fix),
        .pixel_color_in(pco)
    );

    /*
        PPU Color Mapper Module
    */

    VGAColor vco;

    PPUColorMapper color_mapper(
        .ppu_color_in(frame_buffer_out),

        .vga_color_out(vco)
```

```
    );

    //This may need to be changed
    always_ff @(negedge vga_vsync_out) begin
        if(reset) begin
            frame_number <= 0;

        end else begin
            frame_number <= ~frame_number;

        end
    end

    assign led = sw;

    logic [3:0] vga_out_r;
    logic [3:0] vga_out_g;
    logic [3:0] vga_out_b;
    logic[3:0] vga_r_temp;
    logic[3:0] vga_g_temp;
    logic[3:0] vga_b_temp;

    logic vga_display;
    assign vga_display = (vga_h_count <= SCALED_HORIZONTAL_BOUND && vga_v_count <=
SCALED_VERTICAL_BOUND);


    assign vga_out_r = vco.r;
    assign vga_out_g = vco.g;
    assign vga_out_b = vco.b;

    assign vga_r_temp = vga_display ? vga_out_r: 0;
    assign vga_g_temp = vga_display ? vga_out_g: 0;
    assign vga_b_temp = vga_display ? vga_out_b: 0;

    assign vga_hs = ~vga_hsync_out;
    assign vga_vs = ~vga_vsync_out;

    assign vga_r = ~vga_blank_out ? vga_r_temp: 0;
    assign vga_g = ~vga_blank_out ? vga_g_temp: 0;
    assign vga_b = ~vga_blank_out ? vga_b_temp: 0;

endmodule
```

PPU.sv

```
import PPU_Types::*;

module PPU(
        input logic clock_in,
        input logic reset_in,
        input logic up,down,right,left,

        // Interface with FrameBuffer
        output PPUColor pixel_color_out,
        output logic[8:0] pixel_h_count_out,
        output logic[8:0] pixel_v_count_out
    );

    // Interface with PPUCounter
    logic[8:0] ppu_v_count_in;
    logic[8:0] ppu_h_count_in;

    // Interface with PPUStateCalculator
    PPUCycleInfo cycle_info_in;

    // Interface with VRAM
    logic[15:0] vram_address_out;
    logic[7:0] vram_data_in;

    // Interface with the OAMSprite queue
    OAMSprite sprite_in;
    logic secondary_oam_empty_in;
    logic secondary_oam_pop_out;

    // Interface with renderer module
    logic new_background_data_available_out;
    BackgroundDataToRender background_data_to_render_out;
    SpriteDataToRender[7:0] sprites_to_render_out;


    //SecondaryOAM I/O
    logic secondary_oam_push_in;
    OAMSprite secondary_oam_sprite_in;
    logic secondary_oam_full_out;

    //Interface with background
    logic [4:0] background_pixel_data;
    logic background_drawing;

    //Interface with sprite_pixels
    logic [5:0] sprite_pixel_data;
    logic [8:0] sprite_hcount;
    logic [8:0] sprite_vcount;
    logic sprite_drawing;

    //Interface with pixel_mux
    logic [4:0] palette_info;
    logic [8:0] pixel_mux_h_count_out;
    logic [8:0] pixel_mux_v_count_out;

    // Palette RAM interface variables
    PPUColor palette_ram_pixel_color_out;
    logic[8:0] palette_ram_pixel_h_count_out;
    logic[8:0] palette_ram_pixel_v_count_out;
```

```
PPUCounter ppu_counter(
    .clock_in(clock_in),
    .reset_in(reset_in),

    .ppu_v_count_out(ppu_v_count_in),
    .ppu_h_count_out(ppu_h_count_in)
);


PPUStateCalculator ppu_state_calculator(
    .ppu_v_count_in(ppu_v_count_in),
    .ppu_h_count_in(ppu_h_count_in),

    .cycle_info_out(cycle_info_in)
);



OAM oam(
    .clock_in(clock_in),
    .reset_in(reset_in),

    //Interface with buttons for sprite control
    .up(up),
    .down(down),
    .left(left),
    .right(right),

    // Interface with PPU Cycle Counter
    .cycle_info_in(cycle_info_in),
    .ppu_v_count_in(ppu_v_count_in),
    .ppu_h_count_in(ppu_h_count_in),

    // Interface with Secondary OAM
    .queue_full_in(secondary_oam_full_out),
    .queue_write_out(secondary_oam_push_in),
    .queue_sprite_out(secondary_oam_sprite_in)
);


SecondaryOAM secondary_oam(
    .clock_in(clock_in),
    .reset_in(reset_in),

    // Interface with OAM
    .push_in(secondary_oam_push_in),
    .sprite_in(secondary_oam_sprite_in),
    .full_out(secondary_oam_full_out),

    //Interface with PPUMemoryFetcher
    .pop_in(secondary_oam_pop_out),
    .sprite_out(sprite_in),
    .empty_out(secondary_oam_empty_in)
);


VRAMSpoofer vram_spoofer(
    .clock_in(clock_in),
    .reset_in(reset_in),

    .addr_in(vram_address_out),
    .data_out(vram_data_in)
);


PPUMemoryFetcher memory_fetcher(
```

```
    .clock_in(clock_in),
    .reset_in(reset_in),

    // Interface with PPUCounter
    .ppu_v_count_in(ppu_v_count_in),
    .ppu_h_count_in(ppu_h_count_in),

    // Interface with PPUStateCalculator
    .cycle_info_in(cycle_info_in),

    // Interface with VRAM
    .vram_address_out(vram_address_out),
    .vram_data_in(vram_data_in),

    // Interface with the SecondaryOAM
    .sprite_in(sprite_in),
    .secondary_oam_empty_in(secondary_oam_empty_in),
    .secondary_oam_pop_out(secondary_oam_pop_out),

    // Interface with renderer module
    .new_background_data_available_out(new_background_data_available_out),
    .background_data_to_render_out(background_data_to_render_out),
    .sprites_to_render_out(sprites_to_render_out)
);

logic [8:0] background_h_count;
logic [8:0] background_v_count;
background background(
    .clock(clock_in),
    .reset(reset_in),
    .background_data_to_render_in(background_data_to_render_out),
    .new_data(new_background_data_available_out),
    .hcount_in(ppu_h_count_in),
    .vcount_in(ppu_v_count_in),
    .ppu_cycle_info_in(cycle_info_in),

    //Interface with pixel_mux module
    .background_pixel_data(background_pixel_data),
    .hcount_out(background_h_count),
    .vcount_out(background_v_count),
    .drawing(background_drawing)
);

sprite_pixels sprites(
    .clock(clock_in),
    .reset(reset_in),
    .hcount_in(ppu_h_count_in),
    .vcount_in(ppu_v_count_in),
    .sprites_data_in(sprites_to_render_out),
    .cycle_info_in(cycle_info_in),

    //interface with pixel_mux module
    .sprite_pixel_data(sprite_pixel_data),
    .hcount_out(sprite_hcount),
    .vcount_out(sprite_vcount),
    .drawing(sprite_drawing)
);

pixel_mux pixel_renderer(
    .clock(clock_in),
    .reset(reset_in),
    .sprite_hcount(sprite_hcount),
    .sprite_vcount(sprite_vcount),
    .background_pixel_data(background_pixel_data),
    .sprite_pixel_data(sprite_pixel_data),
    .background_drawing(background_drawing),
```

```
        .sprite_drawing(sprite_drawing),

        //interface with color mapper
        .pixel_to_render(palette_info),

        //interface with palette ram
        .hcount_out(pixel_mux_h_count_out),
        .vcount_out(pixel_mux_v_count_out),
        .drawing()
    );


    PaletteRAM palette_ram(
        .clock_in(clock_in),
        .reset_in(reset_in),

        // Interface with the pixel_mux module
        .palette_bank_select_in(palette_info[4]),
        .palette_select_in(palette_info[3:2]),
        .color_select_in(palette_info[1:0]),
//        .palette_bank_select_in(background_pixel_data[4]),
//        .palette_select_in(background_pixel_data[3:2]),
//        .color_select_in(background_pixel_data[1:0]),

        // The position on the display of the pixel being input
        .pixel_h_count_in(pixel_mux_h_count_out),
        .pixel_v_count_in(pixel_mux_v_count_out),
//        .pixel_v_count_in(background_v_count),
//        .pixel_h_count_in(background_h_count),

        // Actual module outputs here
        .pixel_color_out(pixel_color_out),
        .pixel_h_count_out(pixel_h_count_out),
        .pixel_v_count_out(pixel_v_count_out)
    );

endmodule
```

PPUCounter.sv

```
import PPU_Types::*;

/*
 Module Notes:

    This module is combinational and doesn't need a clock or reset

    Inputs:
        clock_in - the 5.369 MHz PPU clock
        reset_in - when high, resets the module to vcount = 0, hcount = 0 on the rising edge
                   of the next clock cycle
    Outputs:
        vcount_out - the current scanline being drawn (ranges from 0 to 261, inclusive)
        hcount_out - the current horizontal cycle number (ranges from 0 to 340, inclusive)

*/
module PPUCounter(
        input logic clock_in,
        input logic reset_in,

        output logic[8:0] ppu_v_count_out,
        output logic[8:0] ppu_h_count_out
    );

    always_ff @(posedge clock_in) begin

        if(reset_in) begin
            ppu_v_count_out <= 0;
            ppu_h_count_out <= 0;

        end else begin
            if(ppu_h_count_out == NUM_CYCLES_PER_SCANLINE - 1) begin
                ppu_h_count_out <= 0; // Reset to zero

                if(ppu_v_count_out == NUM_SCANLINES - 1) begin // When vcount overflows, reset
to zero
                    ppu_v_count_out <= 0;

                end else begin
                    ppu_v_count_out <= ppu_v_count_out + 1;

                end

            end else begin // Increment hcount_out
                ppu_h_count_out <= ppu_h_count_out + 1;

            end

        end
    end
endmodule
```

PPUStateCalculator.sv

```
import PPU_Types::*;
/*
 Module Notes:

    This module is combinational and doesn't need a clock or reset

    Inputs:
        ppu_v_count_in - this is the number (from 0 to 261, inclusive) of the scanline that
the PPU is currently drawing:

            Scanlines [0,239] = visible/drawn
            Scanline 240 = postrender
            Scanlines [241,260] = vblank
            Scanline 261 = prerender


        ppu_h_count_in - this is the scanline cycle number (from 0 to 340, inclusive) of the
pixel that the PPU is currently drawing.



    Outputs:
        vstate_out - the current vertical state of the PPU
        hstate_out - the current horizontal state of the PPU
        tileFetchstate_out - the current tile fetch state of the PPU
        memoryFetchstate_out - the current memoryFetchstate_out of the PPU

*/
module PPUStateCalculator(
        input logic[8:0] ppu_v_count_in,
        input logic[8:0] ppu_h_count_in,

        output PPUCycleInfo cycle_info_out
    );

    VerticalState vstate;
    HorizontalState hstate;
    TileFetchState tile_fetch_state;
    MemoryFetchState memory_fetch_state;

    assign cycle_info_out = {
        vstate: vstate,
        hstate: hstate,
        tile_fetch_state: tile_fetch_state,
        memory_fetch_state: memory_fetch_state
    };

    // the tile state can be determined from the 2nd and 3rd lowest bits of (ppu_h_count_in)
    logic[1:0] tile_id;
    assign tile_id = (ppu_h_count_in) >> 1;

    // The possible tile_id values and their meanings
    parameter NAMETABLE_FETCH_ID = 0;
    parameter ATTRIBUTE_FETCH_ID = 1;
    parameter PATTERN_FETCH_1_ID = 2;
    parameter PATTERN_FETCH_2_ID = 3;


    always_comb begin

        if (ppu_v_count_in <= LAST_VISIBLE_SCANLINE) begin // In visual phase
            vstate = VISIBLE;
```

```
            if(ppu_h_count_in <= VISIBLE_SCANLINE_LAST_BACKGROUND_PREFETCH_CYCLE) begin
                // The sprite and background prefetchs go through the same tile data fetch
progression, so they're combined here

                // Determine hstate_out
                if(ppu_h_count_in <= VISIBLE_SCANLINE_LAST_BACKGROUND_DRAW_CYCLE) begin
                    hstate = BACKGROUND_DRAW;

                end else if(ppu_h_count_in <= VISIBLE_SCANLINE_LAST_SPRITE_PREFETCH_CYCLE)
begin
                    hstate = SPRITE_PREFETCH;

                end else if(ppu_h_count_in <= VISIBLE_SCANLINE_LAST_BACKGROUND_PREFETCH_CYCLE)
begin
                    hstate = BACKGROUND_PREFETCH;

                end else begin
                    hstate = HORIZONTAL_IDLE;

                end

                // Determine tile_fetch_state_out
                case(tile_id)
                    NAMETABLE_FETCH_ID: tile_fetch_state = NAMETABLE_FETCH;
                    ATTRIBUTE_FETCH_ID: tile_fetch_state = ATTRIBUTE_FETCH;
                    PATTERN_FETCH_1_ID: tile_fetch_state = PATTERN_FETCH_1;
                    PATTERN_FETCH_2_ID: tile_fetch_state = PATTERN_FETCH_2;
                endcase

                memory_fetch_state = ppu_h_count_in[0] == 1 ? VRAM_RECIEVE: VRAM_REQUEST;

            end else if(ppu_h_count_in <= VISIBLE_SCANLINE_LAST_IDLE_CYCLE) begin
                hstate = HORIZONTAL_IDLE;
                tile_fetch_state = TILE_FETCH_IDLE;
                memory_fetch_state = MEMORY_FETCH_IDLE;

            end

        end else if (ppu_v_count_in <= LAST_POSTRENDER_SCANLINE) begin // In postrender phase
            vstate = POSTRENDER;
            hstate = HORIZONTAL_IDLE;
            tile_fetch_state = TILE_FETCH_IDLE;
            memory_fetch_state = MEMORY_FETCH_IDLE;

        end else if (ppu_v_count_in <= LAST_VBLANK_SCANLINE) begin // In vblank stage
            vstate = VBLANK;
            hstate = HORIZONTAL_IDLE;
            tile_fetch_state = TILE_FETCH_IDLE;
            memory_fetch_state = MEMORY_FETCH_IDLE;

        end else if(ppu_v_count_in <= LAST_PRERENDER_SCANLINE) begin // In prerender phase
            vstate = PRERENDER;

            if(ppu_h_count_in <= VISIBLE_SCANLINE_LAST_BACKGROUND_DRAW_CYCLE) begin
                hstate = HORIZONTAL_IDLE;
                tile_fetch_state = TILE_FETCH_IDLE;
                memory_fetch_state = MEMORY_FETCH_IDLE;

            end else if(ppu_h_count_in <= VISIBLE_SCANLINE_LAST_BACKGROUND_PREFETCH_CYCLE)
begin
                // The sprite and background prefetchs go through the same tile data fetch
progression, so they're combined here

                // Determine hstate_out
                if(ppu_h_count_in <= VISIBLE_SCANLINE_LAST_SPRITE_PREFETCH_CYCLE) begin
```

```
                    hstate = SPRITE_PREFETCH;
                end else if(ppu_h_count_in <= VISIBLE_SCANLINE_LAST_BACKGROUND_PREFETCH_CYCLE)
begin
                    hstate = BACKGROUND_PREFETCH;
                end

                // Determine tile_fetch_state_out
                case(tile_id)
                    NAMETABLE_FETCH_ID: tile_fetch_state = NAMETABLE_FETCH;
                    ATTRIBUTE_FETCH_ID: tile_fetch_state = ATTRIBUTE_FETCH;
                    PATTERN_FETCH_1_ID: tile_fetch_state = PATTERN_FETCH_1;
                    PATTERN_FETCH_2_ID: tile_fetch_state = PATTERN_FETCH_2;
                endcase

                memory_fetch_state = ppu_h_count_in[0] == 1 ? VRAM_RECIEVE: VRAM_REQUEST;

            end else if(ppu_h_count_in <= VISIBLE_SCANLINE_LAST_IDLE_CYCLE) begin
                hstate = HORIZONTAL_IDLE;
                tile_fetch_state = TILE_FETCH_IDLE;
                memory_fetch_state = MEMORY_FETCH_IDLE;

            end

        end

    end

endmodule
```

OAM.sv

```
import PPU_Types::*;

module OAM(
        input logic clock_in,
        input logic reset_in,

        // Interface with buttons/sprite controller
        input logic up,down,left,right,

        // Interface with PPU Cycle Counter
        input PPUCycleInfo cycle_info_in,
        input logic[8:0] ppu_v_count_in,
        input logic[8:0] ppu_h_count_in,

        // Interface with Secondary OAM
        input logic queue_full_in,
        output logic queue_write_out,
        output OAMSprite queue_sprite_out
    );

    // Internal state
    OAMSprite[63:0] all_sprites;
    logic[5:0] sprite_to_evaluate;



    // Determine if the evaluation should run this cycle
    logic[8:0] NEXT_SCANLINE;
    assign NEXT_SCANLINE = (ppu_v_count_in == 261) ? 0: ppu_v_count_in + 1;

    logic V_COUNT_IN_RANGE;
    assign V_COUNT_IN_RANGE = NEXT_SCANLINE < LAST_VISIBLE_SCANLINE;

    logic H_COUNT_IN_RANGE;
    assign H_COUNT_IN_RANGE = (0 <= ppu_h_count_in) && (ppu_h_count_in < 64);

    logic DO_EVALUATION;
    assign DO_EVALUATION = V_COUNT_IN_RANGE && H_COUNT_IN_RANGE;



    // Determine if the sprite will be drawn on the next scanline
    OAMSprite evaluated_sprite;
    assign evaluated_sprite = all_sprites[sprite_to_evaluate];

    logic sprite_is_on_next_scanline;
    assign sprite_is_on_next_scanline = (evaluated_sprite.y_position <= NEXT_SCANLINE) &&
(NEXT_SCANLINE < evaluated_sprite.y_position + 8);



    always_ff @(posedge clock_in) begin
        if(reset_in) begin
            // Reset internal state
            all_sprites[0] <= {
                x_position: 8'd1,
                y_position: 8'd230,
                tile_index: 8'd0,

                // Attribute byte
                color_data: 2'd1,
                background_priority: 1'd0,
```

```
    horizontal_flip: 1'd0,
    vertical_flip: 1'd0
};

all_sprites[1] <= {
    x_position: 8'd1,
    y_position: 8'd222,
    tile_index: 8'd0,

    // Attribute byte
    color_data: 2'd1,
    background_priority: 1'd0,
    horizontal_flip: 1'd0,
    vertical_flip: 1'd0
};

all_sprites[2] <= {
    x_position: 8'd16,
    y_position: 8'd230,
    tile_index: 8'd1,

    // Attribute byte
    color_data: 2'd1,
    background_priority: 1'd1,
    horizontal_flip: 1'd0,
    vertical_flip: 1'd0
};

all_sprites[3] <= {
    x_position: 8'd24,
    y_position: 8'd222,
    tile_index: 8'd1,

    // Attribute byte
    color_data: 2'd1,
    background_priority: 1'd1,
    horizontal_flip: 1'd0,
    vertical_flip: 1'd0
};


all_sprites[4] <= {
    x_position: 8'd16,
    y_position: 8'd222,
    tile_index: 8'd1,

    // Attribute byte
    color_data: 2'd1,
    background_priority: 1'd1,
    horizontal_flip: 1'd0,
    vertical_flip: 1'd0
};

all_sprites[5] <= {
    x_position: 8'd24,
    y_position: 8'd230,
    tile_index: 8'd1,

    // Attribute byte
    color_data: 2'd1,
    background_priority: 1'd1,
    horizontal_flip: 1'd0,
    vertical_flip: 1'd0
};

all_sprites[6] <= {
```

```
                x_position: 8'd254,
                y_position: 8'd230,
                tile_index: 8'd0,

                // Attribute byte
                color_data: 2'd0,
                background_priority: 1'd0,
                horizontal_flip: 1'd0,
                vertical_flip: 1'd0
            };


            all_sprites[63:7] <= 0;
            sprite_to_evaluate <= 0;

            //Reset outputs
            queue_write_out <= 0;
            queue_sprite_out <= 0;

        end else begin
            //moving the sprites at a rate of one pixel per frame
            if(ppu_v_count_in == 247 && ppu_h_count_in == 260) begin
                if (up && all_sprites[1].y_position > 8) begin
                    all_sprites[5].y_position <= all_sprites[5].y_position - 1;
                    all_sprites[6].y_position <= all_sprites[6].y_position - 1;
                    all_sprites[2].y_position <= all_sprites[2].y_position - 1;
                    all_sprites[3].y_position <= all_sprites[3].y_position - 1;
                    all_sprites[4].y_position <= all_sprites[4].y_position - 1;
                    all_sprites[0].y_position <= all_sprites[0].y_position - 1;
                    all_sprites[1].y_position <= all_sprites[1].y_position - 1;
                end else if (down && all_sprites[4].y_position <= 232) begin
                    all_sprites[2].y_position <= all_sprites[2].y_position + 1;
                    all_sprites[3].y_position <= all_sprites[3].y_position + 1;
                    all_sprites[4].y_position <= all_sprites[4].y_position + 1;
                    all_sprites[5].y_position <= all_sprites[5].y_position + 1;
                    all_sprites[6].y_position <= all_sprites[6].y_position + 1;
                    all_sprites[0].y_position <= all_sprites[0].y_position + 1;
                    all_sprites[1].y_position <= all_sprites[1].y_position + 1;
                end

                if (left ) begin
                    all_sprites[4].x_position <= all_sprites[4].x_position - 1;
                    all_sprites[5].x_position <= all_sprites[5].x_position - 1;
                    all_sprites[2].x_position <= all_sprites[2].x_position - 1;
                    all_sprites[3].x_position <= all_sprites[3].x_position - 1;
                end else if (right ) begin
                    all_sprites[5].x_position <= all_sprites[5].x_position + 1;
                    all_sprites[4].x_position <= all_sprites[4].x_position + 1;
                    all_sprites[2].x_position <= all_sprites[2].x_position + 1;
                    all_sprites[3].x_position <= all_sprites[3].x_position + 1;

                end
            end

            if(DO_EVALUATION) begin

                queue_sprite_out <= evaluated_sprite;
                queue_write_out <= sprite_is_on_next_scanline && !queue_full_in;

                sprite_to_evaluate <= sprite_to_evaluate + 1;

            end else begin
                sprite_to_evaluate <= 0; // Reset this pointer so that the next evaluation
starts correctly
                queue_write_out <= 0;
```

```
            end
        end
    end

endmodule
```

SecondaryOAM.sv

```
import PPU_Types::*;

module SecondaryOAM(
        input logic clock_in,
        input logic reset_in,

        // Interface with OAM
        input logic push_in,
        input OAMSprite sprite_in,
        output logic full_out,

        //Interface with PPUMemoryFetcher
        input logic pop_in,
        input logic clear_in,
        output OAMSprite sprite_out,
        output logic empty_out
    );

    /*
        The size of this queue is 8 because it can store 8 elements before it is full

        The queue is empty when write_pointer == read_pointer

        The queue is full when (write_pointer + 1) % (size + 1) == read_pointer

        Data cannot be inserted into a full queue until the cycle after data is popped.
        Data cannot be removed from an empty queue until the cycle after data is pushed

        Pushing and popping cannot be done at the same time. Pushing takes precedence.
    */

    parameter SIZE = 8;

    logic[3:0] write_pointer; // Array index where the next sprite will be written
    logic[3:0] read_pointer; // Array index of the sprite being currently presented on
sprite_out

    OAMSprite[8:0] sprite_queue; // This array has SIZE + 1 elements so that the queue can
differentiate between being full and empty

    always_ff @(posedge clock_in) begin
        if(reset_in || clear_in) begin
            // Reset internal state
            write_pointer <= 0;
            read_pointer <= 0;
            sprite_queue <= 0;

            // Reset outputs
            full_out <= 0;
            sprite_out <= 0;
            empty_out <= 1; // The queue is empty when it is reset

        end else begin

            if(!full_out && push_in) begin // If queue is not full, we can push new data

                logic[3:0] is_full_check_pointer;
                is_full_check_pointer = (write_pointer + 2) % (SIZE + 1);

                full_out <= is_full_check_pointer == read_pointer;
                empty_out <= 0; // When data is pushed, the queue can no longer be empty
```

```
                sprite_queue[write_pointer] <= sprite_in;

                write_pointer <= (write_pointer + 1) % (SIZE + 1); // Increment write pointer
and overflow if it goes past SIZE

            end else if(!empty_out && pop_in) begin // If queue is not empty, we can pop data

                logic[3:0] next_read_pointer;
                next_read_pointer = (read_pointer + 1) % (SIZE + 1);

                empty_out <= next_read_pointer == write_pointer;
                full_out <= 0; // Whenever data is popped, the queue can no longer be full

                sprite_out <= sprite_queue[read_pointer];
                sprite_queue[read_pointer] <= 0;

                read_pointer <= next_read_pointer; // Increment read pointer and overflow if
it goes past SIZE

            end
        end
    end

endmodule
```

VRAM.sv

```
`timescale 1ns / 1ps


module VRAM( input clock,
             input BRAM_clock,
             input reset,
             input write,
             input [15:0] int_address,
             input [7:0] data_in,
             output logic [7:0] data_out
    );

    logic [13:0] address14;
    logic [7:0] bram_datain;
    logic[7:0] bram_dataout;
    logic [13:0] new_address;
    logic bram_write;

    ppuBRAM BRAM(.clka(BRAM_clock), .ena(1), .wea(bram_write), .addra(new_address),
.dina(bram_datain), .douta(data_out));

    always_ff @(posedge clock) begin
        bram_write <= write;
        bram_datain <= data_in;
        // For transparent tile color mirroring in palettes
        if (int_address[13:0] == 14'h3F10 ||int_address[13:0] == 14'h3F30 || int_address[13:0]
==14'h3F50 || int_address[13:0] ==14'h3F70 || int_address[13:0] ==14'h3F90 ||
int_address[13:0] ==14'h3FB0 || int_address[13:0] ==14'h3FD0 || int_address[13:0] ==14'h3FF0)
begin
            new_address <= 14'h3F00;
        end else if (int_address[13:0] == 14'h3F14 ||int_address[13:0] == 14'h3F34 ||
int_address[13:0] ==14'h3F54 || int_address[13:0] ==14'h3F74 || int_address[13:0] ==14'h3F94
|| int_address[13:0] ==14'h3FB4 || int_address[13:0] ==14'h3FD4 || int_address[13:0]
==14'h3FF4) begin
            new_address <= 14'h3F04;
        end else if (int_address[13:0] == 14'h3F18 ||int_address[13:0] == 14'h3F38 ||
int_address[13:0] ==14'h3F58 || int_address[13:0] ==14'h3F78 || int_address[13:0] ==14'h3F98
|| int_address[13:0] ==14'h3FB8 || int_address[13:0] ==14'h3FD8 || int_address[13:0]
==14'h3FF8) begin
            new_address <= 14'h3F08;
        end else if (int_address[13:0] == 14'h3F1C ||int_address[13:0] == 14'h3F3C ||
int_address[13:0] ==14'h3F5C || int_address[13:0] ==14'h3F7C || int_address[13:0] ==14'h3F9C
|| int_address[13:0] ==14'h3FBC || int_address[13:0] ==14'h3FDC || int_address[13:0]
==14'h3FFC) begin
            new_address <= 14'h3F0C;
        end
        //for nametable mirrors
        else if (int_address[13:0]>=14'h3000 && int_address[13:0]<14'h3F00) begin
            new_address <= int_address[13:0] - 14'h1000;
        end
        //for palette mirrors
        else if (int_address[13:0]>=14'h3F20 && int_address[13:0]<14'h3F40) begin
            new_address <= int_address[13:0] - 14'h0020;
        end
        else if (int_address[13:0]>=14'h3F40 && int_address[13:0]<14'h3F60)begin
            new_address <= int_address[13:0] - 14'h0040;
        end
        else if (int_address[13:0]>=14'h3F60 && int_address[13:0]<14'h3F80)begin
            new_address <= int_address[13:0] - 14'h0060;
        end
        else if (int_address[13:0]>=14'h3F80 && int_address[13:0]<14'h3FA0)begin
            new_address <= int_address[13:0] - 14'h0080;
```

```
        end
        else if (int_address[13:0]>=14'h3FA0 && int_address[13:0]<14'h3FC0)begin
            new_address <= int_address[13:0] - 14'h00A0;
        end
        else if (int_address[13:0]>=14'h3FC0 && int_address[13:0]<14'h3FE0)begin
            new_address <= int_address[13:0] - 14'h00C0;
        end
        else if (int_address[13:0]>=14'h3FE0 && int_address[13:0]<=14'h3FFF) begin
            new_address <= int_address[13:0] - 14'h00E0;
        end else begin
            new_address <= int_address[13:0];
        end
    end

endmodule
```

VRAMSpoofer.sv

```
/*
    Module notes:
        This module pretends to be VRAM and returns tile data, nametable data, and attribute
table data
        as if it were VRAM. However, regardless of the tile address, it will always return the
same pattern
        table bytes.
        Addresses of the form 15'b000X_XXXX_XXXX_0abc will map to pattern_table_1[abc]
        Addresses of the form 15'b000X_XXXX_XXXX_1abc will map to pattern_table_2[abc]
        Note: the nes has 2 pattern tables, but this mirrors them into one

        Regardless of the attribute table address, it will always return the attribute byte
        Addresses in the range [0x23C0, 02400) will return the attribute byte

        All other addresses will return 0xFF
*/
module VRAMSpoofer(
        input logic clock_in,
        input logic reset_in,

        input logic[15:0] addr_in,
        output logic[7:0] data_out
    );

    parameter NAMETABLE_BYTE_1 = 8'h00;
    parameter NAMETABLE_BYTE_2 = 8'h05;
    parameter NAMETABLE_BYTE_3 = 8'h06;
    parameter NAMETABLE_LOWER_BOUND = 16'h2000;
    parameter NAMETABLE_CHANGE = 16'h2200;
    parameter NAMETABLE_UPPER_BOUND = 16'h23C0;

    parameter ATTRIBUTE_BYTE = 8'b00_00_00_00;
    parameter ATTRIBUTE_LOWER_BOUND = 16'h23C0;
    parameter ATTRIBUTE_UPPER_BOUND = 16'h2400;

    parameter DEFAULT_BYTE = 8'hFF;

    typedef struct packed {
        logic[7:0][7:0] pattern_bytes_1;
        logic[7:0][7:0] pattern_bytes_2;
    } Tile;

    parameter NUM_TILES_LOADED = 7;
    parameter MAX_VALID_PATTERN_TABLE_ADDRESS = NUM_TILES_LOADED * 16 - 1;

    //This need to change if you are adding or removing tiles
    Tile[7:0] pattern_table;

    assign pattern_table[0] = {
        pattern_bytes_1: {
            8'b00000000,
            8'b00000000,
            8'b00000000,
            8'b00000000,
            8'b00000000,
            8'b00000000,
            8'b00000000,
            8'b00000000
        },

        pattern_bytes_2: {
            8'b00000000,
```

```
        8'b00000000,
        8'b00000000,
        8'b00000000,
        8'b00000000,
        8'b00000000,
        8'b00000000,
        8'b00000000
    }
};

assign pattern_table[1] = {
    pattern_bytes_1: {
        8'b00000011,
        8'b00000000,
        8'b00011110,
        8'b00000100,
        8'b00000010,
        8'b00000000,
        8'b00001111,
        8'b00000001
    },

    pattern_bytes_2: {
        8'b00000000,
        8'b00001111,
        8'b00000001,
        8'b00111011,
        8'b00011101,
        8'b00000011,
        8'b00001111,
        8'b00000001
    }
};

assign pattern_table[2] = {

    pattern_bytes_1: {
        8'b11110000,
        8'b00000000,
        8'b00011100,
        8'b11001000,
        8'b01001000,
        8'b11110000,
        8'b11110000,
        8'b11100000
    },

    pattern_bytes_2: {
        8'b00000000,
        8'b11100000,
        8'b11100000,
        8'b00110000,
        8'b10110000,
        8'b00000000,
        8'b11110000,
        8'b11100000
    }
};

assign pattern_table[3] = {

    pattern_bytes_1: {
        8'b00000111,
        8'b00000110,
        8'b00001111,
        8'b00001111,
```

```
        8'b00001111,
        8'b00000111,
        8'b00000111,
        8'b00001111
    },

    pattern_bytes_2: {
        8'b00000001,
        8'b00000111,
        8'b00001111,
        8'b00001111,
        8'b00001111,
        8'b00000111,
        8'b00000000,
        8'b00000000
    }
};

assign pattern_table[4] = {
    pattern_bytes_1: {
        8'b01111000,
        8'b00111000,
        8'b01111000,
        8'b10011000,
        8'b00011000,
        8'b11110000,
        8'b11110000,
        8'b11110000
    },

    pattern_bytes_2: {
        8'b00000000,
        8'b00000000,
        8'b01111000,
        8'b11111000,
        8'b11101000,
        8'b10000000,
        8'b11000000,
        8'b10000000
    }
};

    assign pattern_table[5] = {
    pattern_bytes_1: {
        8'b1111_1111,
        8'b1000_0001,
        8'b1011_1101,
        8'b1010_0101,
        8'b1010_0101,
        8'b1011_1101,
        8'b1000_0001,
        8'b1111_1111
    },

    pattern_bytes_2: {
        8'b0000_0000,
        8'b0111_1110,
        8'b0111_1110,
        8'b0111_1110,
        8'b0111_1110,
        8'b0111_1110,
        8'b0111_1110,
        8'b0000_0000
    }
};
```

```
        assign pattern_table[6] = {
        pattern_bytes_1: {
            8'b1111_1111,
            8'b1111_1111,
            8'b1111_1111,
            8'b1111_1111,
            8'b1111_1111,
            8'b1111_1111,
            8'b1111_1111,
            8'b1111_1111
        },

        pattern_bytes_2: {
            8'b0000_0000,
            8'b0111_1110,
            8'b0111_1110,
            8'b0111_1110,
            8'b0111_1110,
            8'b0111_1110,
            8'b0111_1110,
            8'b0000_0000
        }
    };



    always_ff @(negedge clock_in) begin
        if(reset_in) begin
            data_out <= 0;
        end else begin
            if(addr_in < NAMETABLE_LOWER_BOUND) begin

                if(addr_in <= MAX_VALID_PATTERN_TABLE_ADDRESS) begin
                    data_out <= addr_in[3] ?
pattern_table[addr_in[11:4]].pattern_bytes_2[addr_in[2:0]] :
pattern_table[addr_in[11:4]].pattern_bytes_1[addr_in[2:0]];

                end else begin
                    data_out <= 0;

                end

            end else if(NAMETABLE_LOWER_BOUND <= addr_in && addr_in < NAMETABLE_CHANGE) begin
                data_out <= NAMETABLE_BYTE_1;

            end else if (NAMETABLE_CHANGE<= addr_in && addr_in < NAMETABLE_UPPER_BOUND &&
addr_in%2) begin
                data_out <= NAMETABLE_BYTE_2;

            end else if (NAMETABLE_CHANGE<= addr_in && addr_in < NAMETABLE_UPPER_BOUND &&
~addr_in%2) begin
                data_out <= NAMETABLE_BYTE_3;

            end else if(ATTRIBUTE_LOWER_BOUND <= addr_in && addr_in < ATTRIBUTE_UPPER_BOUND)
begin
                data_out <= ATTRIBUTE_BYTE;

            end else begin
                data_out <= DEFAULT_BYTE;

            end
        end
    end

endmodule
```

PPUMemoryFetcher.sv

```
import PPU_Types::*;

module PPUMemoryFetcher(
        input logic clock_in,
        input logic reset_in,

        // Interface with PPUCounter
        input logic[8:0] ppu_v_count_in,
        input logic[8:0] ppu_h_count_in,

        // Interface with PPUStateCalculator
        input PPUCycleInfo cycle_info_in,

        // Interface with VRAM
        output logic[15:0] vram_address_out, // The address being requested from VRAM
        input logic[7:0] vram_data_in, // The data returned from VRAM. Is latched at the end
of each VRAM_RECIEVE cycle

        // Interface with the SecondaryOAM
        input OAMSprite sprite_in, // This is the sprite that will be fetched at the next
8-cycle sprite fetch phase
        input logic secondary_oam_empty_in, // This is set when the OAMSprite queue is empty
and not presenting a valid sprite
        output logic secondary_oam_pop_out, // This is asserted for one clock cycle once
sprite_in has been latched within this module. Signals that is is safe to present the next
sprite to fetch from memory on sprite_in
        output logic secondary_oam_clear_out,

        // Interface with renderer module
        output logic new_background_data_available_out, //This line is asserted for one cycle
when new backgound data is available on background_data_to_render_out
        output BackgroundDataToRender background_data_to_render_out,
        output SpriteDataToRender[7:0] sprites_to_render_out
    );


    // Internal state
    OAMSprite current_sprite_to_fetch; // Internal copy of the current sprite to fetch
    logic[2:0] sprite_write_pointer; // Contains the index into spritePixelsToRender in which
the next loaded sprite will be stored

    logic[7:0] background_tile_index; // This value is the tile index that is loaded from the
nametable
    logic[7:0] background_attribute_byte;
    logic[1:0] background_attribute_bits; // These are the bits selected from the attribute
byte that will be sent to the background module

    logic[7:0] tile_byte_1; // Byte 1 from the pattern table for the tile being fetched
    logic[7:0] tile_byte_2; // Byte 2 from the pattern table for the tile being fetched

    // Parse the cycle horizontal state
    logic fetching_background;
    assign fetching_background = (cycle_info_in.hstate == BACKGROUND_PREFETCH) ||
(cycle_info_in.hstate == BACKGROUND_DRAW);

    //v_count has to be incremented by one if we are prefetching for the next line
    logic [8:0] prefetch_v_count;
    //the condition was added in to try and fix top scanline of prefetched tiles, may need to
change condition to greater than
    assign prefetch_v_count = ppu_v_count_in + 1 > 240 ? 0 : ppu_v_count_in + 1;

    logic[15:0] nametable_addr;
    // Translate vcount and hcount into a nametable address.
    // Note: only supporting one nametable from addresses 0x2000-0x23BF
```

```
    assign nametable_addr[15:10] = 6'b001000;
    assign nametable_addr[9:5] = cycle_info_in.hstate == BACKGROUND_PREFETCH ?
prefetch_v_count[7:3] : ppu_v_count_in[7:3];
    assign nametable_addr[4:0] = cycle_info_in.hstate == BACKGROUND_PREFETCH ? {3'b0,
ppu_h_count_in[3]}: (ppu_h_count_in[7:3] + 2);

    // Hardcode attribute address to be in the right memory range
    logic [15:0] attribute_addr;
    assign attribute_addr = 16'h23C0;


    always_ff @(posedge clock_in) begin
        if(reset_in) begin
            // Reset all outputs
            vram_address_out <= 0;
            secondary_oam_pop_out <= 0;
            background_data_to_render_out <= 0;
            sprites_to_render_out <= 0;
            new_background_data_available_out <= 0;
            secondary_oam_clear_out <= 0;

            // Reset internal state
            current_sprite_to_fetch <= 0;
            sprite_write_pointer <= 0;
            background_tile_index <= 0;
            background_attribute_byte <= 0;
            background_attribute_bits <= 0;
            tile_byte_1 <= 0;
            tile_byte_2 <= 0;


        end else begin
            // If fetching background tiles
            if(fetching_background) begin

                // Determine the address to present on vram_address_out
                case(cycle_info_in.tile_fetch_state)
                    NAMETABLE_FETCH: begin
                        vram_address_out <= nametable_addr;

                        // Store response from VRAM in the background tile index register
                        if(cycle_info_in.memory_fetch_state == VRAM_RECIEVE) begin
                            background_tile_index <= vram_data_in;
                        end
                    end


                    ATTRIBUTE_FETCH: begin
                        vram_address_out <= attribute_addr; // TODO: calculate actual address
here

                        // Store response from VRAM in the background attribute byte register
                        if(cycle_info_in.memory_fetch_state == VRAM_RECIEVE) begin
                            background_attribute_byte <= vram_data_in;
                        end
                    end


                    PATTERN_FETCH_1: begin
                        /*
                            The address of tile byte 1 takes the form of:
                            MSB
                                4'b0000 - indicating pattern table 1
                                8 bits - the tile index
                                1'b0 - indicating that it is the first of two bytes defining
the tile row
```

```
                                ppu_v_count_in[2:0] - the last three bits of ppu_v_count_in
tell which row of the tile to read
                                LSB
                            */
                            logic[15:0] tile_byte_1_addr;
                            tile_byte_1_addr[15:12] = 0;
                            tile_byte_1_addr[11:4] = background_tile_index;
                            tile_byte_1_addr[3] = 0;
                            tile_byte_1_addr[2:0] = cycle_info_in.hstate == BACKGROUND_PREFETCH ?
prefetch_v_count[2:0]:ppu_v_count_in[2:0];
                            vram_address_out <= tile_byte_1_addr;

                            // Store response from VRAM in the background tile byte 1 register
                            if(cycle_info_in.memory_fetch_state == VRAM_RECIEVE) begin
                                tile_byte_1 <= vram_data_in;
                            end

                            // Determine attribute bits
                            if(cycle_info_in.hstate == BACKGROUND_PREFETCH) begin
                                // Depending on the scanline, we'll either use the square 0 or
square 2 attribute bits
                                background_attribute_bits <= prefetch_v_count[4] ?
background_attribute_byte[5:4] : background_attribute_byte[1:0];

                            end else begin
                                if(ppu_h_count_in[4] && ppu_v_count_in[4]) begin // Square 2
                                    background_attribute_bits <= background_attribute_byte[5:4];

                                end else if(ppu_h_count_in[4] && ~ppu_v_count_in[4]) begin //
Square 0
                                    background_attribute_bits <= background_attribute_byte[1:0];

                                end else if(~ppu_h_count_in[4] && ppu_v_count_in[4]) begin //
Square 3
                                    background_attribute_bits <= background_attribute_byte[7:6];

                                end else begin // Square 1
                                    background_attribute_bits <= background_attribute_byte[3:2];

                                end

                            end

                        end


                        PATTERN_FETCH_2: begin
                            /*
                                The address of tile byte 2 takes the form of:
                                MSB
                                    4'b0000 - indicating pattern table 1
                                    8 bits - the tile index
                                    1'b1 - indicating that it is the second of two bytes defining
the tile row
                                    ppu_v_count_in[2:0] - the last three bits of ppu_v_count_in
tell which row of the tile to read
                                LSB
                            */
                            logic[15:0] tile_byte_2_addr;
                            tile_byte_2_addr[15:12] = 0;
                            tile_byte_2_addr[11:4] = background_tile_index;
                            tile_byte_2_addr[3] = 1;
                            tile_byte_2_addr[2:0] = cycle_info_in.hstate == BACKGROUND_PREFETCH ?
prefetch_v_count[2:0]:ppu_v_count_in[2:0];
                            vram_address_out <= tile_byte_2_addr;
```

```
                        // Store response from VRAM in the background tile byte 2 register
                        if(cycle_info_in.memory_fetch_state == VRAM_RECIEVE) begin
                            tile_byte_2 <= vram_data_in; // TODO don't really need this here,
but useful for debugging

                            // Write new background data to output
                            background_data_to_render_out <= {
                                tile_byte_1: tile_byte_1,
                                tile_byte_2: vram_data_in,
                                attributes: background_attribute_bits
                            };

                            new_background_data_available_out <= 1; // Indicate that new
background data is available
                        end
                    end


                    TILE_FETCH_IDLE: begin
                        // Should never be in this state
                    end

                endcase
            end else if(cycle_info_in.hstate == SPRITE_PREFETCH) begin

                case(cycle_info_in.tile_fetch_state)

                    NAMETABLE_FETCH: begin
                        vram_address_out <= 0; // Nothing is fetched during this cycle for
sprites
                        current_sprite_to_fetch <= sprite_in; // Latch the current sprite to
fetch
                    end


                    ATTRIBUTE_FETCH: begin
                        vram_address_out <= 0; // Nothing is fetched during this cycle for
sprites

                        if(cycle_info_in.memory_fetch_state == VRAM_RECIEVE) begin
                            secondary_oam_pop_out <= 1; // Ask the sprite queue to present the
next sprite
                        end

                    end


                    PATTERN_FETCH_1: begin
                        /*
                            The address of tile byte 1 takes the form of:
                            MSB
                                4'b0000 - indicating pattern table 1
                                8 bits - the tile index
                                1'b0 - indicating that it is the first of two bytes defining
the tile row
                                ppu_v_count_in[2:0] - the last three bits of ppu_v_count_in
tell which row of the tile to read
                            LSB
                        */
                        logic[15:0] tile_byte_1_addr;
                        tile_byte_1_addr[15:12] = 0;
                        tile_byte_1_addr[11:4] = sprite_in.tile_index;
                        tile_byte_1_addr[3] = 0;
                        tile_byte_1_addr[2:0] = prefetch_v_count[2:0]-sprite_in.y_position;
                        vram_address_out <= tile_byte_1_addr;
```

```
                        // Store response from VRAM in the background tile byte 1 register
                        if(cycle_info_in.memory_fetch_state == VRAM_RECIEVE) begin
                            tile_byte_1 <= vram_data_in;
                        end
                    end


                    PATTERN_FETCH_2: begin
                        /*
                            The address of tile byte 2 takes the form of:
                            MSB
                                4'b0000 - indicating pattern table 1
                                8 bits - the tile index
                                1'b1 - indicating that it is the second of two bytes defining
the tile row
                                ppu_v_count_in[2:0] - the last three bits of ppu_v_count_in
tell which row of the tile to read
                            LSB
                        */
                        logic[15:0] tile_byte_2_addr;
                        tile_byte_2_addr = {4'b0, sprite_in.tile_index, 1'b1,
ppu_v_count_in[2:0]};
                        tile_byte_2_addr[15:12] = 0;
                        tile_byte_2_addr[11:4] = sprite_in.tile_index;
                        tile_byte_2_addr[3] = 1;
                        tile_byte_2_addr[2:0] = prefetch_v_count-sprite_in.y_position;
                        vram_address_out <= tile_byte_2_addr;

                        // Store response from VRAM in the background tile byte 2 register
                        if(cycle_info_in.memory_fetch_state == VRAM_RECIEVE) begin
                            tile_byte_2 <= vram_data_in; // TODO don't really need this here,
but useful for debugging

                            if(current_sprite_to_fetch == 0) begin
                                sprites_to_render_out[sprite_write_pointer] <= 0;

                            end else begin
                                // Write new sprite data to output
                                sprites_to_render_out[sprite_write_pointer] <= {
                                    // 2 pattern table bytes
                                    tile_byte_1: tile_byte_1,
                                    tile_byte_2: tile_byte_2,

                                    // Relevant parts of the attribute byte
                                    color_data: sprite_in.color_data,
                                    background_priority: sprite_in.background_priority,

                                    x_position: sprite_in.x_position
                                };
                            end

                        end

                        if(cycle_info_in.memory_fetch_state == VRAM_RECIEVE) begin
                            sprite_write_pointer <= sprite_write_pointer + 1; // Increment
sprite_write_pointer regardless of queue state so that it overflows correctly every 8 sprite
prefetches
                        end
                    end


                    TILE_FETCH_IDLE: begin
                        // Should never be in this state
                    end
                endcase
```

```
            end

            // Clear the secondary oam on one of the background prefetch cycles to reset it
for the next scanline
            secondary_oam_clear_out <= ppu_h_count_in ==
VISIBLE_SCANLINE_LAST_BACKGROUND_PREFETCH_CYCLE;

            // Ensure this is only asserted for one cycle
            if(new_background_data_available_out == 1) begin
                new_background_data_available_out <= 0;
            end

            // Ensure this is only asserted for one cycle
            if(secondary_oam_pop_out == 1) begin
                secondary_oam_pop_out <= 0;
            end

        end

    end

endmodule
```

Background.sv

```
import PPU_Types::*;
module background(  input clock,
                    input reset,
                    input BackgroundDataToRender background_data_to_render_in,
                    input new_data,
                    //input[2:0] fineX,
                    input [8:0] hcount_in,
                    input [8:0] vcount_in,
                    input PPUCycleInfo ppu_cycle_info_in,

                    output logic [4:0] background_pixel_data,
                    output logic [8:0] hcount_out,
                    output logic [8:0] vcount_out,
                    output logic drawing

    );
        logic [15:0] PatternReg1;
        logic [15:0] PatternReg2;
        logic [7:0] attribute1;
        logic [7:0] attribute2;
        logic new_attribute1;
        logic new_attribute2;

        always_ff @(posedge clock) begin
            if (reset) begin
                background_pixel_data <= 0;
                hcount_out <= 0;
                vcount_out <= 0;
                drawing <= 0;
            end else begin
                case(ppu_cycle_info_in.vstate)
                    PRERENDER: begin
                        case(ppu_cycle_info_in.hstate)
                            HORIZONTAL_IDLE: begin
                                drawing <= 1'b0;
                                if (new_data) begin
                                    //When new data is ready load it into the correct latches
and registers
                                    new_attribute1 <=
background_data_to_render_in.attributes[0];
                                    new_attribute2 <=
background_data_to_render_in.attributes[1];
                                    PatternReg1 <=
{PatternReg1[15:7],background_data_to_render_in.tile_byte_1};
                                    PatternReg2 <=
{PatternReg2[15:7],background_data_to_render_in.tile_byte_2};
                                end
                            end
                            BACKGROUND_DRAW: begin
                                drawing <= 1'b0;
                                if (new_data) begin
                                    //When new data is ready load it into the correct latches
and registers
                                    new_attribute1 <=
background_data_to_render_in.attributes[0];
                                    new_attribute2 <=
background_data_to_render_in.attributes[1];
                                    PatternReg1 <=
{PatternReg1[14:0],background_data_to_render_in.attributes[0]};
                                    PatternReg2 <=
{PatternReg2[14:0],background_data_to_render_in.attributes[1]};
```

```
                                    PatternReg1 <=
{PatternReg1[14:8],background_data_to_render_in.tile_byte_1,1'b0};
                                    PatternReg2 <=
{PatternReg2[14:8],background_data_to_render_in.tile_byte_2,1'b0};
                                end else begin
                                    //Shifting all of the registers one bit
                                    PatternReg1 <= {PatternReg1[14:0],1'b0};
                                    PatternReg2 <= {PatternReg2[14:0],1'b0};
                                    attribute1 <= {attribute1[6:0],new_attribute1};
                                    attribute2 <= {attribute2[6:0],new_attribute2};
                                end
                            end
                            SPRITE_PREFETCH: drawing <= 1'b0;
                            BACKGROUND_PREFETCH:begin
                                drawing <= 1'b0;
                                if (new_data) begin
                                    //When new data is ready load it into the correct latches
and registers
                                    new_attribute1 <=
background_data_to_render_in.attributes[0];
                                    new_attribute2 <=
background_data_to_render_in.attributes[1];
                                    PatternReg1 <=
{PatternReg1[14:0],background_data_to_render_in.attributes[0]};
                                    PatternReg2 <=
{PatternReg2[14:0],background_data_to_render_in.attributes[1]};
                                    PatternReg1 <=
{PatternReg1[14:8],background_data_to_render_in.tile_byte_1,1'b0};
                                    PatternReg2 <=
{PatternReg2[14:8],background_data_to_render_in.tile_byte_2,1'b0};
                                end else begin
                                    //Shifting all of the registers one bit each clock cycle
                                    PatternReg1 <= {PatternReg1[14:0],1'b0};
                                    PatternReg2 <= {PatternReg2[14:0],1'b0};
                                    attribute1 <= {attribute1[6:0],new_attribute1};
                                    attribute2 <= {attribute2[6:0],new_attribute2};
                                end
                            end
                        endcase
                    end
                    VISIBLE: begin
                        case(ppu_cycle_info_in.hstate)
                            HORIZONTAL_IDLE: begin
                                drawing <= 1'b0;
                                if (new_data) begin
                                    //When new data is ready load it into the correct latches
and registers
                                    new_attribute1 <=
background_data_to_render_in.attributes[0];
                                    new_attribute2 <=
background_data_to_render_in.attributes[1];
                                    PatternReg1 <=
{PatternReg1[15:1],background_data_to_render_in.attributes[0]};
                                    PatternReg2 <=
{PatternReg2[15:1],background_data_to_render_in.attributes[1]};
                                    PatternReg1 <=
{PatternReg1[15:8],background_data_to_render_in.tile_byte_1};
                                    PatternReg2 <=
{PatternReg2[15:8],background_data_to_render_in.tile_byte_2};
                                end
                            end
                            BACKGROUND_DRAW: begin
                                drawing <= 1'b1;
                                if (new_data) begin
                                    //When new data is ready load it into the correct latches
and registers
```

```
                                    new_attribute1 <=
background_data_to_render_in.attributes[0];
                                    new_attribute2 <=
background_data_to_render_in.attributes[1];
                                    PatternReg1 <=
{PatternReg1[14:0],background_data_to_render_in.attributes[0]};
                                    PatternReg2 <=
{PatternReg2[14:0],background_data_to_render_in.attributes[1]};
                                    PatternReg1 <=
{PatternReg1[14:8],background_data_to_render_in.tile_byte_1,1'b0};
                                    PatternReg2 <=
{PatternReg2[14:8],background_data_to_render_in.tile_byte_2,1'b0};
                                end else begin
                                    //Shifting all of the registers one bit
                                    PatternReg1 <= {PatternReg1[14:0],1'b0};
                                    PatternReg2 <= {PatternReg2[14:0],1'b0};
                                    attribute1 <= {attribute1[6:0],new_attribute1};
                                    attribute2 <= {attribute2[6:0],new_attribute2};
                                end
                                //Creating the pixel output, the zero signifies that is should
be addressing to background palettes
                                background_pixel_data <= {1'b0, attribute2[7], attribute1[7],
PatternReg2[15], PatternReg1[15]};
                                //outputting current hcount and vcount with pixel to give
pixel location
                                hcount_out <= hcount_in;
                                vcount_out <= vcount_in;
                            end
                            SPRITE_PREFETCH: drawing <= 1'b0;
                            BACKGROUND_PREFETCH:begin
                                drawing <= 1'b0;
                                if (new_data) begin
                                    //When new data is ready load it into the correct latches
and registers
                                    new_attribute1 <=
background_data_to_render_in.attributes[0];
                                    new_attribute2 <=
background_data_to_render_in.attributes[1];
                                    PatternReg1 <=
{PatternReg1[14:0],background_data_to_render_in.attributes[0]};
                                    PatternReg2 <=
{PatternReg2[14:0],background_data_to_render_in.attributes[1]};
                                    PatternReg1 <=
{PatternReg1[14:8],background_data_to_render_in.tile_byte_1,1'b0};
                                    PatternReg2 <=
{PatternReg2[14:8],background_data_to_render_in.tile_byte_2,1'b0};
                                end else begin
                                    //Shifting all of the registers one bit each clock cycle
                                    PatternReg1 <= {PatternReg1[14:0],1'b0};
                                    PatternReg2 <= {PatternReg2[14:0],1'b0};
                                    attribute1 <= {attribute1[6:0],new_attribute1};
                                    attribute2 <= {attribute2[6:0],new_attribute2};
                                end
                            end
                        endcase
                    end
                    POSTRENDER: drawing <= 1'b0;
                    VBLANK: drawing <= 1'b0;
                endcase
            end
        end
endmodule
```

Sprite_pixels.sv

```
`timescale 1ns / 1ps
import PPU_Types::*;

module sprite_pixels(
                        input clock,
                        input reset,
                        input [8:0] hcount_in,
                        input [8:0] vcount_in,
                        input SpriteDataToRender[7:0] sprites_data_in,
                        /*
                            // 2 pattern table bytes
                            logic[7:0] tile_byte_1;
                            logic[7:0] tile_byte_2;

                            // Relevant parts of the attribute byte
                            logic[1:0] color_data;
                            logic background_priority;

                            logic[7:0] x_position;
                        */
                        input PPUCycleInfo cycle_info_in,
                        //sprite_pixel_data bit 5 background priority, bit 4 = 1 to direct to
sprite palettes, bit 3&2 attribute palette data, bit 1&0 color data from pattern tables
                        output logic [5:0] sprite_pixel_data,
                        output logic [8:0] hcount_out,
                        output logic [8:0] vcount_out,
                        output logic drawing
    );

    //creating registers x locations of sprites
    logic [7:0] location_sprite0;
    logic [7:0] location_sprite1;
    logic [7:0] location_sprite2;
    logic [7:0] location_sprite3;
    logic [7:0] location_sprite4;
    logic [7:0] location_sprite5;
    logic [7:0] location_sprite6;
    logic [7:0] location_sprite7;

    //creating 8 latches with attribute info of sprites bit 0&1 color data bit 2 background
priority
    logic [1:0] attribute_sprite0;
    logic [1:0] attribute_sprite1;
    logic [1:0] attribute_sprite2;
    logic [1:0] attribute_sprite3;
    logic [1:0] attribute_sprite4;
    logic [1:0] attribute_sprite5;
    logic [1:0] attribute_sprite6;
    logic [1:0] attribute_sprite7;

    //creating 16 8-bit shift registers for pattern table bytes, 2 for each sprite
    logic [7:0] first_pattern_sprite0;
    logic [7:0] first_pattern_sprite1;
    logic [7:0] first_pattern_sprite2;
    logic [7:0] first_pattern_sprite3;
    logic [7:0] first_pattern_sprite4;
    logic [7:0] first_pattern_sprite5;
    logic [7:0] first_pattern_sprite6;
    logic [7:0] first_pattern_sprite7;
    logic [7:0] second_pattern_sprite0;
    logic [7:0] second_pattern_sprite1;
    logic [7:0] second_pattern_sprite2;
```

```
logic [7:0] second_pattern_sprite3;
logic [7:0] second_pattern_sprite4;
logic [7:0] second_pattern_sprite5;
logic [7:0] second_pattern_sprite6;
logic [7:0] second_pattern_sprite7;

//creating latches to hold pixel data for each sprite
logic [5:0] sprite0_pixel;
logic [5:0] sprite1_pixel;
logic [5:0] sprite2_pixel;
logic [5:0] sprite3_pixel;
logic [5:0] sprite4_pixel;
logic [5:0] sprite5_pixel;
logic [5:0] sprite6_pixel;
logic [5:0] sprite7_pixel;

logic priority0;
logic priority1;
logic priority2;
logic priority3;
logic priority4;
logic priority5;
logic priority6;
logic priority7;

//latch to hold data to be outputted
logic [5:0] send_out;


always_comb begin
    if (sprite0_pixel != 6'b0 && sprite0_pixel != 6'b010000) begin
        send_out = sprite0_pixel;
    end else if (sprite1_pixel != 6'b0 && sprite1_pixel != 6'b010000) begin
        send_out = sprite1_pixel;
    end else if (sprite2_pixel != 6'b0 && sprite2_pixel != 6'b010000) begin
        send_out = sprite2_pixel;
    end else if (sprite3_pixel != 6'b0 && sprite3_pixel != 6'b010000) begin
        send_out = sprite3_pixel;
    end else if (sprite4_pixel != 6'b0 && sprite4_pixel != 6'b010000) begin
        send_out = sprite4_pixel;
    end else if (sprite5_pixel != 6'b0 && sprite5_pixel != 6'b010000) begin
        send_out = sprite5_pixel;
    end else if (sprite6_pixel != 6'b0 && sprite6_pixel != 6'b010000) begin
        send_out = sprite6_pixel;
    end else if (sprite7_pixel != 6'b0 && sprite7_pixel != 6'b010000) begin
        send_out = sprite7_pixel;
    end else begin
        send_out = 6'b0;
    end
    if (hcount_in == 0) begin
        send_out =6'b0;
    end
end

always_ff @(posedge clock) begin
    if (reset) begin
        location_sprite0 <= 0;
        location_sprite1 <= 0;
        location_sprite2 <= 0;
        location_sprite3 <= 0;
        location_sprite4 <= 0;
        location_sprite5 <= 0;
        location_sprite6 <= 0;
        location_sprite7 <= 0;

        attribute_sprite0 <= 0;
```

```
            attribute_sprite1 <= 0;
            attribute_sprite2 <= 0;
            attribute_sprite3 <= 0;
            attribute_sprite4 <= 0;
            attribute_sprite5 <= 0;
            attribute_sprite6 <= 0;
            attribute_sprite7 <= 0;

            first_pattern_sprite0 <= 0;
            first_pattern_sprite1 <= 0;
            first_pattern_sprite2 <= 0;
            first_pattern_sprite3 <= 0;
            first_pattern_sprite4 <= 0;
            first_pattern_sprite5 <= 0;
            first_pattern_sprite6 <= 0;
            first_pattern_sprite7 <= 0;

            second_pattern_sprite0 <= 0;
            second_pattern_sprite1 <= 0;
            second_pattern_sprite2 <= 0;
            second_pattern_sprite3 <= 0;
            second_pattern_sprite4 <= 0;
            second_pattern_sprite5 <= 0;
            second_pattern_sprite6 <= 0;
            second_pattern_sprite7 <= 0;

            sprite0_pixel <= 0;
            sprite1_pixel <= 0;
            sprite2_pixel <= 0;
            sprite3_pixel <= 0;
            sprite4_pixel <= 0;
            sprite5_pixel <= 0;
            sprite6_pixel <= 0;
            sprite7_pixel <= 0;

            priority0 <= 0;
            priority1 <= 0;
            priority2 <= 0;
            priority3 <= 0;
            priority4 <= 0;
            priority5 <= 0;
            priority6 <= 0;
            priority7 <= 0;

        end else begin
            case(cycle_info_in.vstate)
                PRERENDER: begin
                        drawing <= 1'b0;
                        //load all of the sprite information for each line after the sprite
prefetch
                        if (hcount_in == VISIBLE_SCANLINE_LAST_SPRITE_PREFETCH_CYCLE + 2)
begin
                            //loading first pattern table byte for each sprite
                            first_pattern_sprite0 <= sprites_data_in[0].tile_byte_1;
                            first_pattern_sprite1 <= sprites_data_in[1].tile_byte_1;
                            first_pattern_sprite2 <= sprites_data_in[2].tile_byte_1;
                            first_pattern_sprite3 <= sprites_data_in[3].tile_byte_1;
                            first_pattern_sprite4 <= sprites_data_in[4].tile_byte_1;
                            first_pattern_sprite5 <= sprites_data_in[5].tile_byte_1;
                            first_pattern_sprite6 <= sprites_data_in[6].tile_byte_1;
                            first_pattern_sprite7 <= sprites_data_in[7].tile_byte_1;
                            //loading second pattern table byte for each sprite
                            second_pattern_sprite0 <= sprites_data_in[0].tile_byte_2;
                            second_pattern_sprite1 <= sprites_data_in[1].tile_byte_2;
                            second_pattern_sprite2 <= sprites_data_in[2].tile_byte_2;
                            second_pattern_sprite3 <= sprites_data_in[3].tile_byte_2;
```

```
                            second_pattern_sprite4 <= sprites_data_in[4].tile_byte_2;
                            second_pattern_sprite5 <= sprites_data_in[5].tile_byte_2;
                            second_pattern_sprite6 <= sprites_data_in[6].tile_byte_2;
                            second_pattern_sprite7 <= sprites_data_in[7].tile_byte_2;
                            //loading x location of each sprite
                            location_sprite0 <= sprites_data_in[0].x_position;
                            location_sprite1 <= sprites_data_in[1].x_position;
                            location_sprite2 <= sprites_data_in[2].x_position;
                            location_sprite3 <= sprites_data_in[3].x_position;
                            location_sprite4 <= sprites_data_in[4].x_position;
                            location_sprite5 <= sprites_data_in[5].x_position;
                            location_sprite6 <= sprites_data_in[6].x_position;
                            location_sprite7 <= sprites_data_in[7].x_position;
                            //loading color data for each sprite
                            attribute_sprite0 <= sprites_data_in[0].color_data;
                            attribute_sprite1 <= sprites_data_in[1].color_data;
                            attribute_sprite2 <= sprites_data_in[2].color_data;
                            attribute_sprite3 <= sprites_data_in[3].color_data;
                            attribute_sprite4 <= sprites_data_in[4].color_data;
                            attribute_sprite5 <= sprites_data_in[5].color_data;
                            attribute_sprite6 <= sprites_data_in[6].color_data;
                            attribute_sprite7 <= sprites_data_in[7].color_data;
                            //loading background priority data
                            priority0 <= sprites_data_in[0].background_priority;
                            priority1 <= sprites_data_in[1].background_priority;
                            priority2 <= sprites_data_in[2].background_priority;
                            priority3 <= sprites_data_in[3].background_priority;
                            priority4 <= sprites_data_in[4].background_priority;
                            priority5 <= sprites_data_in[5].background_priority;
                            priority6 <= sprites_data_in[6].background_priority;
                            priority7 <= sprites_data_in[7].background_priority;
                        end
                end
                VISIBLE: begin
                        drawing <= 1'b0;
                        //load all of the sprite information for each line after the sprite
prefetch
                        if (hcount_in == VISIBLE_SCANLINE_LAST_SPRITE_PREFETCH_CYCLE + 2)
begin //changed >= to ==
                            //loading first pattern table byte for each sprite
                            first_pattern_sprite0 <= sprites_data_in[0].tile_byte_1;
                            first_pattern_sprite1 <= sprites_data_in[1].tile_byte_1;
                            first_pattern_sprite2 <= sprites_data_in[2].tile_byte_1;
                            first_pattern_sprite3 <= sprites_data_in[3].tile_byte_1;
                            first_pattern_sprite4 <= sprites_data_in[4].tile_byte_1;
                            first_pattern_sprite5 <= sprites_data_in[5].tile_byte_1;
                            first_pattern_sprite6 <= sprites_data_in[6].tile_byte_1;
                            first_pattern_sprite7 <= sprites_data_in[7].tile_byte_1;
                            //loading second pattern table byte for each sprite
                            second_pattern_sprite0 <= sprites_data_in[0].tile_byte_2;
                            second_pattern_sprite1 <= sprites_data_in[1].tile_byte_2;
                            second_pattern_sprite2 <= sprites_data_in[2].tile_byte_2;
                            second_pattern_sprite3 <= sprites_data_in[3].tile_byte_2;
                            second_pattern_sprite4 <= sprites_data_in[4].tile_byte_2;
                            second_pattern_sprite5 <= sprites_data_in[5].tile_byte_2;
                            second_pattern_sprite6 <= sprites_data_in[6].tile_byte_2;
                            second_pattern_sprite7 <= sprites_data_in[7].tile_byte_2;
                            //loading x location of each sprite
                            location_sprite0 <= sprites_data_in[0].x_position;
                            location_sprite1 <= sprites_data_in[1].x_position;
                            location_sprite2 <= sprites_data_in[2].x_position;
                            location_sprite3 <= sprites_data_in[3].x_position;
                            location_sprite4 <= sprites_data_in[4].x_position;
                            location_sprite5 <= sprites_data_in[5].x_position;
                            location_sprite6 <= sprites_data_in[6].x_position;
                            location_sprite7 <= sprites_data_in[7].x_position;
```

```
                            //loading color data for each sprite
                            attribute_sprite0 <= sprites_data_in[0].color_data;
                            attribute_sprite1 <= sprites_data_in[1].color_data;
                            attribute_sprite2 <= sprites_data_in[2].color_data;
                            attribute_sprite3 <= sprites_data_in[3].color_data;
                            attribute_sprite4 <= sprites_data_in[4].color_data;
                            attribute_sprite5 <= sprites_data_in[5].color_data;
                            attribute_sprite6 <= sprites_data_in[6].color_data;
                            attribute_sprite7 <= sprites_data_in[7].color_data;
                            //loading background priority data
                            priority0 <= sprites_data_in[0].background_priority;
                            priority1 <= sprites_data_in[1].background_priority;
                            priority2 <= sprites_data_in[2].background_priority;
                            priority3 <= sprites_data_in[3].background_priority;
                            priority4 <= sprites_data_in[4].background_priority;
                            priority5 <= sprites_data_in[5].background_priority;
                            priority6 <= sprites_data_in[6].background_priority;
                            priority7 <= sprites_data_in[7].background_priority;

                        end
                        //calculating if a sprite is there and drawing that pixel
                        else if (hcount_in < VISIBLE_SCANLINE_LAST_BACKGROUND_DRAW_CYCLE)
begin //2 is added because of clockcycle offset
                            if (hcount_in >= location_sprite7 && hcount_in < location_sprite7
+ 8'd8 ) begin
                                sprite7_pixel <= {priority7, 1'b1,
attribute_sprite7,second_pattern_sprite7[7],first_pattern_sprite7[7]};
                                first_pattern_sprite7 <= {first_pattern_sprite7[6:0],1'b0};
                                second_pattern_sprite7 <= {second_pattern_sprite7[6:0],1'b0};
                            end else begin
                                sprite7_pixel <= 6'b0;
                            end
                            if (hcount_in >= location_sprite6 && hcount_in < location_sprite6
+ 8'd8 ) begin
                                sprite6_pixel <= {priority6, 1'b1,
attribute_sprite6,second_pattern_sprite6[7],first_pattern_sprite6[7]};
                                first_pattern_sprite6 <= {first_pattern_sprite6[6:0],1'b0};
                                second_pattern_sprite6 <= {second_pattern_sprite6[6:0],1'b0};
                            end else begin
                                sprite6_pixel <= 6'b0;
                            end
                            if (hcount_in >= location_sprite5 && hcount_in < location_sprite5
+ 8'd8 ) begin
                                sprite6_pixel <= {priority6, 1'b1,
attribute_sprite6,second_pattern_sprite6[7],first_pattern_sprite6[7]};
                                first_pattern_sprite5 <= {first_pattern_sprite5[6:0],1'b0};
                                second_pattern_sprite5 <= {second_pattern_sprite5[6:0],1'b0};
                            end else begin
                                sprite5_pixel <= 6'b0;
                            end
                            if (hcount_in >= location_sprite4 && hcount_in < location_sprite4
+ 8'd8 ) begin
                                sprite6_pixel <= {priority6, 1'b1,
attribute_sprite6,second_pattern_sprite6[7],first_pattern_sprite6[7]};
                                first_pattern_sprite4 <= {first_pattern_sprite4[6:0],1'b0};
                                second_pattern_sprite4 <= {second_pattern_sprite4[6:0],1'b0};
                            end else begin
                                sprite4_pixel <= 6'b0;
                            end
                            if (hcount_in >= location_sprite3 && hcount_in < location_sprite3
+ 8'd8 ) begin
                                sprite6_pixel <= {priority6, 1'b1,
attribute_sprite6,second_pattern_sprite6[7],first_pattern_sprite6[7]};
                                first_pattern_sprite3 <= {first_pattern_sprite3[6:0],1'b0};
                                second_pattern_sprite3 <= {second_pattern_sprite3[6:0],1'b0};
                            end else begin
```

```
                                sprite3_pixel <= 6'b0;
                            end
                            if (hcount_in >= location_sprite2 && hcount_in < location_sprite2
+ 8'd8 ) begin
                                sprite6_pixel <= {priority6, 1'b1,
attribute_sprite6,second_pattern_sprite6[7],first_pattern_sprite6[7]};
                                first_pattern_sprite2 <= {first_pattern_sprite2[6:0],1'b0};
                                second_pattern_sprite2 <= {second_pattern_sprite2[6:0],1'b0};
                            end else begin
                                sprite2_pixel <= 6'b0;
                            end
                            if (hcount_in >= location_sprite1 && hcount_in < location_sprite1
+ 8'd8 ) begin
                                sprite6_pixel <= {priority6, 1'b1,
attribute_sprite6,second_pattern_sprite6[7],first_pattern_sprite6[7]};

                                first_pattern_sprite1 <= {first_pattern_sprite1[6:0],1'b0};
                                second_pattern_sprite1 <= {second_pattern_sprite1[6:0],1'b0};
                            end else begin
                                sprite1_pixel <= 6'b0;
                            end
                            if (hcount_in >= location_sprite0 && hcount_in < location_sprite0
+ 8'd8 ) begin
                                sprite6_pixel <= {priority6, 1'b1,
attribute_sprite6,second_pattern_sprite6[7],first_pattern_sprite6[7]};
                                first_pattern_sprite0 <= {first_pattern_sprite0[6:0],1'b0};
                                second_pattern_sprite0 <= {second_pattern_sprite0[6:0],1'b0};
                            end else begin
                                sprite0_pixel <= 6'b0;
                            end
                            if (send_out == 6'b0) begin
                                drawing <= 1'b0;
                            end else begin
                                drawing <= 1'b1;
                            end
                            sprite_pixel_data <= send_out;
                            if (hcount_in == 0) begin
                                hcount_out <= 0;
                            end else begin
                                hcount_out <= hcount_in - 9'd1 ;
                            end
                            vcount_out <= vcount_in;
                        end
                end
                POSTRENDER: drawing <= 1'b0;
                VBLANK: drawing <= 1'b0;
            endcase
        end
    end
endmodule
```

Pixel_mux.sv

```
module pixel_mux(
                input clock,
                input reset,
                input [8:0] sprite_hcount,
                input [8:0] sprite_vcount,
                input [5:0] sprite_pixel_data,
                input [4:0] background_pixel_data,
                //possibly need this input
                input background_drawing,
                input sprite_drawing,

                output logic [4:0] pixel_to_render,
                output logic [8:0] hcount_out,
                output logic [8:0] vcount_out,
                output logic drawing
    );

    logic [4:0] background_hold;
    logic background_drawing_latch;

    always_ff @(posedge clock) begin
        if (sprite_hcount == 9'd255)  begin
            background_hold <= 0;
            background_drawing_latch <= 0;
        end else begin
            background_hold <= background_pixel_data;
            background_drawing_latch <= background_drawing;
        end

        //when checking for transparency in these if statements only need to check
        //bits 1&0 because 00 in each color palette is the transparent color
        if (sprite_pixel_data == 6'b0) begin
            pixel_to_render <= background_hold;
        end else if (sprite_pixel_data[5] == 1'b0 && background_hold[1:0] != 2'b0) begin
            pixel_to_render <= background_hold;
        end else if (sprite_pixel_data[5] == 1'b0 && background_hold[1:0] == 2'b0) begin
            pixel_to_render <= sprite_pixel_data[4:0];
        end else if (sprite_pixel_data[5] == 1'b1 && sprite_pixel_data[1:0] == 2'b0) begin
            pixel_to_render <= background_hold;
        end else if (sprite_pixel_data[5] == 1'b1 && sprite_pixel_data[1:0] != 2'b0) begin
            pixel_to_render <= sprite_pixel_data;
        end
        hcount_out <= sprite_hcount;
        vcount_out <= sprite_vcount;
        if (background_drawing_latch == 1'b1 || sprite_drawing == 1'b1) begin
            drawing <= 1'b1;
        end else begin
            drawing <= 1'b0;
        end
    end
endmodule
```

PaletteRAM.sv

```
module PaletteRAM(
        input logic clock_in,
        input logic reset_in,

        // The pixel palette address being input
        input logic palette_bank_select_in,
        input logic[1:0] palette_select_in,
        input logic[1:0] color_select_in,

        // The position on the display of the pixel being input
        input logic[8:0] pixel_h_count_in,
        input logic[8:0] pixel_v_count_in,

        output PPUColor pixel_color_out,
        output logic[8:0] pixel_h_count_out, // The h_count position of the color being output
        output logic[8:0] pixel_v_count_out // The v_count position of the colot being output
    );

    PPUColor transparent_color;
    assign transparent_color = 6'h21; // light grey

    PPUColor[3:0][3:0] background_palettes;
    assign background_palettes = {
        {6'h08, 6'h18, 6'h17, transparent_color},
        {6'h18, 6'h17, 6'h08, transparent_color},
        {6'h17, 6'h08, 6'h18, transparent_color},
        {6'h06, 6'h07, 6'h17, transparent_color}
    };

    PPUColor[3:0][3:0] sprite_palettes;
    assign sprite_palettes = {
        {6'h06, 6'h37, 6'h02, transparent_color},
        {6'h06, 6'h37, 6'h02, transparent_color},
        {6'h06, 6'h37, 6'h02, transparent_color},
        {6'h06, 6'h37, 6'h02, transparent_color}
    };

    always_ff @(posedge clock_in) begin
        if(reset_in) begin
            // Reset output
            pixel_color_out <= 0;
            pixel_h_count_out <= 0;
            pixel_v_count_out <= 0;

        end else begin
            pixel_color_out <= palette_bank_select_in ?
sprite_palettes[palette_select_in][color_select_in]:
background_palettes[palette_select_in][color_select_in];

            // This provides a one-stage pipeline so that the position information follows the
pixel information correctly
            pixel_h_count_out <= pixel_h_count_in;
            pixel_v_count_out <= pixel_v_count_in;

        end
    end

endmodule
```

FrameBuffer.sv

```
import PPU_Types::*;

/*
    Module notes:

    This module serves as a frame buffer between the output of the PPU and the input to the
VGA module.

    This module contains two memories that each store an entire frame of video. While the PPU
is writing
    the next frame to one buffer, the VGA module is reading the current frame data from the
other.

    Note that the output pixel data lags the VGA v and h count by one clock cycle.

    Also note that vga_v_count_in and ppu_v_count_in should range from 0 to 239 inclusive and
    vga_h_count_in and ppu_h_count_in should range from 0 to 255 inclusive
*/
module FrameBuffer(
        input logic reset_in,

        input logic frame_num_in,               // This determines which of the two internal
frame buffers to read from

        // Interface with the video out unit
        input logic vga_clock_in,
        input logic[8:0] vga_v_count_in,
        input logic[8:0] vga_h_count_in,
        output PPUColor pixel_color_out,

        // Interface with the PPU
        input logic ppu_clock_in,
        input logic[8:0] ppu_v_count_in,
        input logic[8:0] ppu_h_count_in,
        input PPUColor pixel_color_in
    );

    // Calculate the address to write the new pixel data
    logic[15:0] ppu_write_addr;
    assign ppu_write_addr = (ppu_v_count_in * NUM_HORIZONTAL_PIXELS) + ppu_h_count_in;

    // Calculate the address to read the next pixel data
    logic[15:0] video_out_read_addr;
    assign video_out_read_addr = (vga_v_count_in * NUM_HORIZONTAL_PIXELS) + vga_h_count_in;

    PPUColor buffer1_pixel;
    ppu_frame_buffer buffer1(
        .clka(frame_num_in ? vga_clock_in : ppu_clock_in),
        .wea(~frame_num_in), // Write to this frame buffer when frame_num_in is high
        .addra(frame_num_in ? video_out_read_addr : ppu_write_addr),
        .dina(pixel_color_in),
        .douta(buffer1_pixel)
    );

    PPUColor buffer2_pixel;
    ppu_frame_buffer buffer2(
        .clka(frame_num_in ? ppu_clock_in : vga_clock_in),
        .wea(frame_num_in), // Write to this frame buffer when frame_num_in is low
        .addra(frame_num_in ? ppu_write_addr : video_out_read_addr),
        .dina(pixel_color_in),
        .douta(buffer2_pixel)
    );
```

```
    always_ff @(posedge vga_clock_in) begin
        pixel_color_out <= frame_num_in ? buffer1_pixel : buffer2_pixel; // When frame_num_in
is high, output the data read from buffer1
    end

endmodule
```

PPUColorMapper.sv

```
import PPU_Types::*;

module PPUColorMapper(
        input PPUColor ppu_color_in,

        output VGAColor vga_color_out
    );

    // Buckle up
    // There's probably a better way to do this, but idk
    logic[11:0] color_map[63:0]; // 64 x 12 bit array
    assign color_map[0] = 12'h777;                          // 0x00
    assign color_map[1] = 12'h00F;
    assign color_map[2] = 12'h00B;
    assign color_map[3] = 12'h42B;
    assign color_map[4] = 12'h908;
    assign color_map[5] = 12'hA02;
    assign color_map[6] = 12'hA10;
    assign color_map[7] = 12'h810;
    assign color_map[8] = 12'h530;
    assign color_map[9] = 12'h070;
    assign color_map[10] = 12'h060;
    assign color_map[11] = 12'h050;
    assign color_map[12] = 12'h045;
    assign color_map[13] = 12'h000;
    assign color_map[14] = 12'h000;
    assign color_map[15] = 12'h000;                         // 0x0F
    assign color_map[16] = 12'hBBB;                         // 0x10
    assign color_map[17] = 12'h07F;
    assign color_map[18] = 12'h05F;
    assign color_map[19] = 12'h64F;
    assign color_map[20] = 12'hD0C;
    assign color_map[21] = 12'hE05;
    assign color_map[22] = 12'hF30;
    assign color_map[23] = 12'hE51;
    assign color_map[24] = 12'hA70;
    assign color_map[25] = 12'h0B0;
    assign color_map[26] = 12'h0A0;
    assign color_map[27] = 12'h0A4;
    assign color_map[28] = 12'h088;
    assign color_map[29] = 12'h000;
    assign color_map[30] = 12'h000;
    assign color_map[31] = 12'h000;                         // 0x1F
    assign color_map[32] = 12'hFFF;                         // 0x20
    assign color_map[33] = 12'h3BF;
    assign color_map[34] = 12'h68F;
    assign color_map[35] = 12'h97F;
    assign color_map[36] = 12'hF7F;
    assign color_map[37] = 12'hF59;
    assign color_map[38] = 12'hF75;
    assign color_map[39] = 12'hFA4;
    assign color_map[40] = 12'hFB0;
    assign color_map[41] = 12'hBF1;
    assign color_map[42] = 12'h5D5;
    assign color_map[43] = 12'h5F9;
    assign color_map[44] = 12'h0ED;
    assign color_map[45] = 12'h777;
    assign color_map[46] = 12'h000;
    assign color_map[47] = 12'h000;                         // 0x2F
    assign color_map[48] = 12'hFFF;                         // 0x30
    assign color_map[49] = 12'hAEF;
    assign color_map[50] = 12'hBBF;
```

```
    assign color_map[51] = 12'hDBF;
    assign color_map[52] = 12'hFBF;
    assign color_map[53] = 12'hFAC;
    assign color_map[54] = 12'hFDB;
    assign color_map[55] = 12'hFEA;
    assign color_map[56] = 12'hFD7;
    assign color_map[57] = 12'hDF7;
    assign color_map[58] = 12'hBFB;
    assign color_map[59] = 12'hBFD;
    assign color_map[60] = 12'h0FF;
    assign color_map[61] = 12'hDDD;
    assign color_map[62] = 12'h000;
    assign color_map[63] = 12'h000;              // 0x3F

    logic[11:0] color_bits = color_map[ppu_color_in];
    assign vga_color_out = {
        r: color_bits[11:8],
        g: color_bits[7:4],
        b: color_bits[3:0]
    };

endmodule
```

VGA.sv

```
//////////////////////////////////////////////////////////////////////////////////
// Update: 8/8/2019 GH
// Create Date: 10/02/2015 02:05:19 AM
// Module Name: xvga
//
// xvga: Generate VGA display signals (1024 x 768 @ 60Hz)
//
//                              ---- HORIZONTAL -----     ------VERTICAL -----
//                              Active                    Active
//                    Freq      Video   FP  Sync   BP      Video   FP  Sync  BP
//   640x480, 60Hz    25.175    640     16    96   48       480    11   2    31
//   800x600, 60Hz    40.000    800     40   128   88       600     1   4    23
//   1024x768, 60Hz   65.000    1024    24   136  160       768     3   6    29
//   1280x1024, 60Hz  108.00    1280    48   112  248       768     1   3    38
//   1280x720p 60Hz   75.25     1280    72    80  216       720     3   5    30
//   1920x1080 60Hz   148.5     1920    88    44  148      1080     4   5    36
//
// change the clock frequency, front porches, sync's, and back porches to create
// other screen resolutions
//////////////////////////////////////////////////////////////////////////////////
module vga_1024(input vclock_in,
            output reg [10:0] hcount_out,    // pixel number on current line
            output reg [9:0] vcount_out,     // line number
            output reg vsync_out, hsync_out,
            output reg blank_out);

   parameter DISPLAY_WIDTH  = 1024;      // display width
   parameter DISPLAY_HEIGHT = 768;       // number of lines

   parameter  H_FP = 24;                 // horizontal front porch
   parameter  H_SYNC_PULSE = 136;        // horizontal sync
   parameter  H_BP = 160;                // horizontal back porch

   parameter  V_FP = 3;                  // vertical front porch
   parameter  V_SYNC_PULSE = 6;          // vertical sync
   parameter  V_BP = 29;                 // vertical back porch

   // horizontal: 1344 pixels total
   // display 1024 pixels per line
   reg hblank,vblank;
   wire hsyncon,hsyncoff,hreset,hblankon;
   assign hblankon = (hcount_out == (DISPLAY_WIDTH -1));
   assign hsyncon = (hcount_out == (DISPLAY_WIDTH + H_FP - 1));  //1047
   assign hsyncoff = (hcount_out == (DISPLAY_WIDTH + H_FP + H_SYNC_PULSE - 1));  // 1183
   assign hreset = (hcount_out == (DISPLAY_WIDTH + H_FP + H_SYNC_PULSE + H_BP - 1));  //1343

   // vertical: 806 lines total
   // display 768 lines
   wire vsyncon,vsyncoff,vreset,vblankon;
   assign vblankon = hreset & (vcount_out == (DISPLAY_HEIGHT - 1));   // 767
   assign vsyncon = hreset & (vcount_out == (DISPLAY_HEIGHT + V_FP - 1));  // 771
   assign vsyncoff = hreset & (vcount_out == (DISPLAY_HEIGHT + V_FP + V_SYNC_PULSE - 1));  //
777
   assign vreset = hreset & (vcount_out == (DISPLAY_HEIGHT + V_FP + V_SYNC_PULSE + V_BP - 1));
// 805

   // sync and blanking
   wire next_hblank,next_vblank;
   assign next_hblank = hreset ? 0 : hblankon ? 1 : hblank;
   assign next_vblank = vreset ? 0 : vblankon ? 1 : vblank;
   always_ff @(posedge vclock_in) begin
      hcount_out <= hreset ? 0 : hcount_out + 1;
```

```
      hblank <= next_hblank;
      hsync_out <= hsyncon ? 0 : hsyncoff ? 1 : hsync_out;  // active low

      vcount_out <= hreset ? (vreset ? 0 : vcount_out + 1) : vcount_out;
      vblank <= next_vblank;
      vsync_out <= vsyncon ? 0 : vsyncoff ? 1 : vsync_out;  // active low

      blank_out <= next_vblank | (next_hblank & ~hreset);
   end

endmodule
```

PPUTypes.sv

```
/*
    Enums and constants used for the PPU
*/

package PPU_Types;

parameter NUM_VERTICAL_PIXELS = 240; // The number of pixels tall the drawn window is
parameter NUM_HORIZONTAL_PIXELS = 256; // The number of pixels wide the drawn window is

parameter NUM_SCANLINES = 262;
parameter NUM_CYCLES_PER_SCANLINE = 341;


// The boundaries for the different scanline phases
parameter LAST_VISIBLE_SCANLINE = 239; // The last scanline number of the visible phase
parameter LAST_POSTRENDER_SCANLINE = 240; // The last scanline number of the postrender phase
parameter LAST_VBLANK_SCANLINE = 260; // The last scanline number of the vblank phase
parameter LAST_PRERENDER_SCANLINE = 261; // The last scanline number of the prerender phase


// The boundaries for the different cycle phases during the visible scanlines
parameter VISIBLE_SCANLINE_LAST_BACKGROUND_DRAW_CYCLE = 255; // The number of the last cycle
of the background draw phase
parameter VISIBLE_SCANLINE_LAST_SPRITE_PREFETCH_CYCLE = 319; // The number of the last cycle
of the sprite data prefetch phase
parameter VISIBLE_SCANLINE_LAST_BACKGROUND_PREFETCH_CYCLE = 335; // The number of the last
cycle of the background data prefetch phase
parameter VISIBLE_SCANLINE_LAST_IDLE_CYCLE = 340; // The number of the last cycle of the
ending idle phase


/*
    VISIBLE: scanlines [0,239]
    POSTRENDER: scaneline 240
    VBLANK: scanlines [241,260]
    PRERENDER: scanline 261
*/
typedef enum {
    PRERENDER, VISIBLE, POSTRENDER, VBLANK
} VerticalState;


/*
    BACKGROUND: When the PPU is drawing the background. Cycles [0, 255]
    SPRITE_PREFETCH: When the PPU is pre-fetching the sprite data for the next frame. Cycles
[256,319]
    HORIZONTAL_IDLE: When the PPU isn't actually drawing anything. Cycles [320,324]
    BACKGROUND_PREFETCH: When the PPU is pre-fetching the background tiles for the next frame.
Cycles [325,340]
*/
typedef enum {
    HORIZONTAL_IDLE, BACKGROUND_DRAW, SPRITE_PREFETCH, BACKGROUND_PREFETCH
} HorizontalState;


/*
    TILE_FETCH_IDLE: the PPU isn't fetching tile data
    NAMETABLE_READ: the PPU is fetching data from a nametable
    ATTRIBUTE_READ: the PPU is fetching data from an attribute table
    PATTERN_READ_1: the PPU is fetching the first byte of tile data
    PATTERN_READ_2: the PPU is fetching the second byte of tile data
*/
```

```
typedef enum {
    NAMETABLE_FETCH, ATTRIBUTE_FETCH, PATTERN_FETCH_1, PATTERN_FETCH_2, TILE_FETCH_IDLE
} TileFetchState;


/*
    MEM_FETCH_IDLE: the PPU isn't fetching tile data
    VRAM_REQUEST: the PPU is asserting an address to the VRAM
    VRAM_RECIEVE: the PPU is reading the data returned by the VRAM
*/
typedef enum {
    VRAM_REQUEST, VRAM_RECIEVE, MEMORY_FETCH_IDLE
} MemoryFetchState;


typedef struct packed {
    VerticalState vstate;
    HorizontalState hstate;
    TileFetchState tile_fetch_state;
    MemoryFetchState memory_fetch_state;
} PPUCycleInfo;


/*
    This is the data type representing a sprite as it is stored in Object Attribute Memory
(OAM)
*/
typedef struct packed {
    logic[7:0] x_position;
    logic[7:0] y_position;
    logic[7:0] tile_index;

    // Attribute byte
    logic[1:0] color_data;
    logic background_priority;
    logic horizontal_flip;
    logic vertical_flip;
} OAMSprite;


/*
    This is the data type defining a pixel from a sprite as it is passed to the rendering
module
*/
typedef struct packed {
    // 2 pattern table bytes
    logic[7:0] tile_byte_1;
    logic[7:0] tile_byte_2;

    // Relevant parts of the attribute byte
    logic[1:0] color_data;
    logic background_priority;

    logic[7:0] x_position;
} SpriteDataToRender;


/*
    This is the data type defining a background pixel as it is passed to the rendering module
*/
typedef struct packed {
    // 2 pattern table bytes
    logic[7:0] tile_byte_1;
    logic[7:0] tile_byte_2;

    logic[1:0] attributes;
```

```
} BackgroundDataToRender;


typedef logic[5:0] PPUColor;


typedef struct packed {
    logic[3:0] r;
    logic[3:0] g;
    logic[3:0] b;
} VGAColor;

endpackage
```

Sources

Diskin, Patrick. “Nintendo Entertainment System Documentation.” nesdev.com, 1.0, August 2004. http://www.nesdev.com/NESDoc.pdf