Nintendo Entertainment System Hardware Emulation

Daniel Klahn, Sidne Gregory, Israel Bonilla

6.111 Fall 2019

Overview

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, 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 intend to reproduce the NES hardware as faithfully as possible as an exercise in legacy hardware emulation

Overall System

The Nintendo Entertainment System consists of three major internal components to be emulated, with input from the game cartridge and NES controller, and 256 by 240 pixel component video output . Those elements to be reproduced are a central processing unit (CPU), pixel processing unit (PPU), and audio processing unit (APU). Core functionality of the games depend solely on the CPU and PPU (see figure 1), which will therefore be the base goals for the project. The CPU is the heart of the device; I\O is entirely mapped to addresses in the CPU memory, as well as communication between the CPU and PPU. The PPU responds to commands from the CPU to render the frame in question. This system needs less than 64kB of BRAM for the CPU and less than 64kB of BRAM for the PPU, as well as around 400kB of ROM space for storing games.

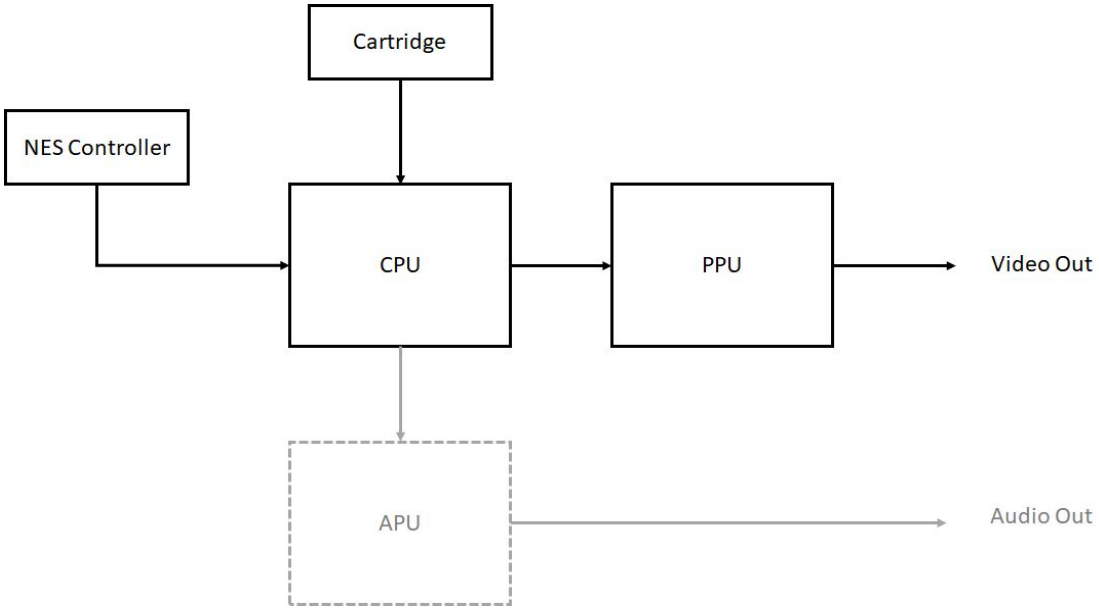

Figure 1: Overall System Block Diagram

CPU

The NES CPU is a Ricoh 2A03, which is based off of the Intel 6502. The 2A03 uses almost the same instruction set as the 6502, only without some Binary Coded Decimal (BCD) instructions. The 6502 is an 8-bit microprocessor that runs at 1.79 MHz. It supports 56 instructions in several different addressing modes. The 6502 is fairly simple in that it is not pipelined, has very few internal registers, and does not support complicated operations like multiplication. The 6502 uses memory mapping to communicate with external devices including the PPU, APU, internal RAM, the cartridge ROM, and the NES controller.

One thing that may be tricky to implement is processor interrupts. The 6502 can be interrupted by the reset pin, the PPU (the PPU interrupts the CPU at the beginning of the VSYNC period), and certain other memory mapped devices. Getting the interrupt sequence correct and to have the right timings might be pose a challenge, but there is a lot of documentation on the subject. Israel will be taking the lead in building the CPU.

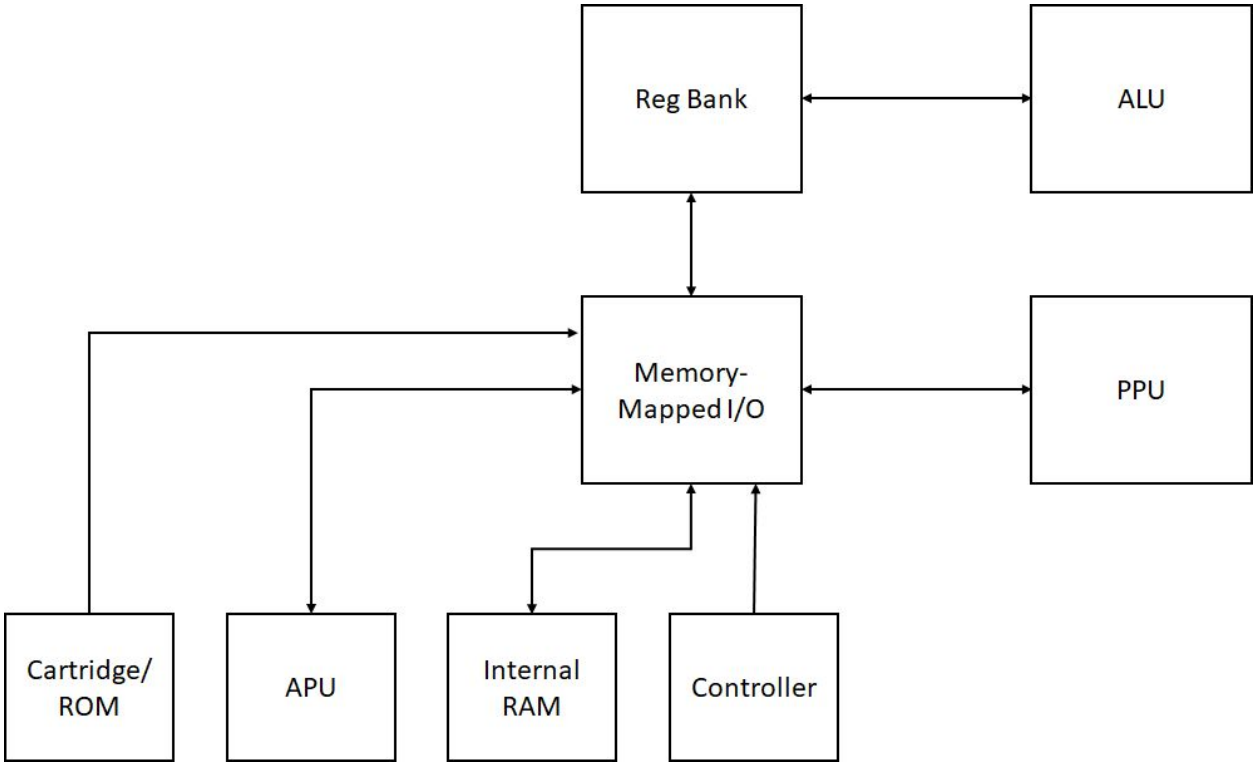

Figure 2: CPU Block Diagram

PPU

The PPU for the NES is a Ricoh 2C02. The PPU interfaces with the CPU through 8 memory-mapped registers. This component is responsible for defining both the shape and color of the background and sprites that make up the 256x240 pixel NES output.

The background and the sprites are created separately and then must be combined together while not breaking the systems restrictions of having more than 25 colors per frame or having more than 8 sprites per scanline. They must also work together to make sure the correct pixels are being shown, such that when a sprite is in a certain location the sprite pixel is on top of the background and is shown instead of the background pixels.

Both the sprite and background modules use the pattern tables stored in a portion of the 2kB internal VRAM (Video RAM). These pattern tables define the shape and color of the 8x8 tiles that make up both the background and the sprites. Each tile is 16 bytes and is defined by two 8 byte planes, the first of which hold bit 0 of color for each pixel, the second holding bit 1 of color. Each module has its own color palette that addresses to the main NES color palette.

The background processor also makes use of name tables and attribute tables to create its. The name table stores which tiles from the pattern tables will go in which place in the screen and the attribute table which is appended to the end of the name table tells which palette each tile uses. The PPU has enough storage to hold 2 name and attribute tables but can address up to 4 which can then be stored on extra storage included in game cartridges.

Sidne and Daniel will be responsible for building the PPU, one taking the lead on the background module while the other is in charge of the sprites module and they will work together to integrate their two modules together.

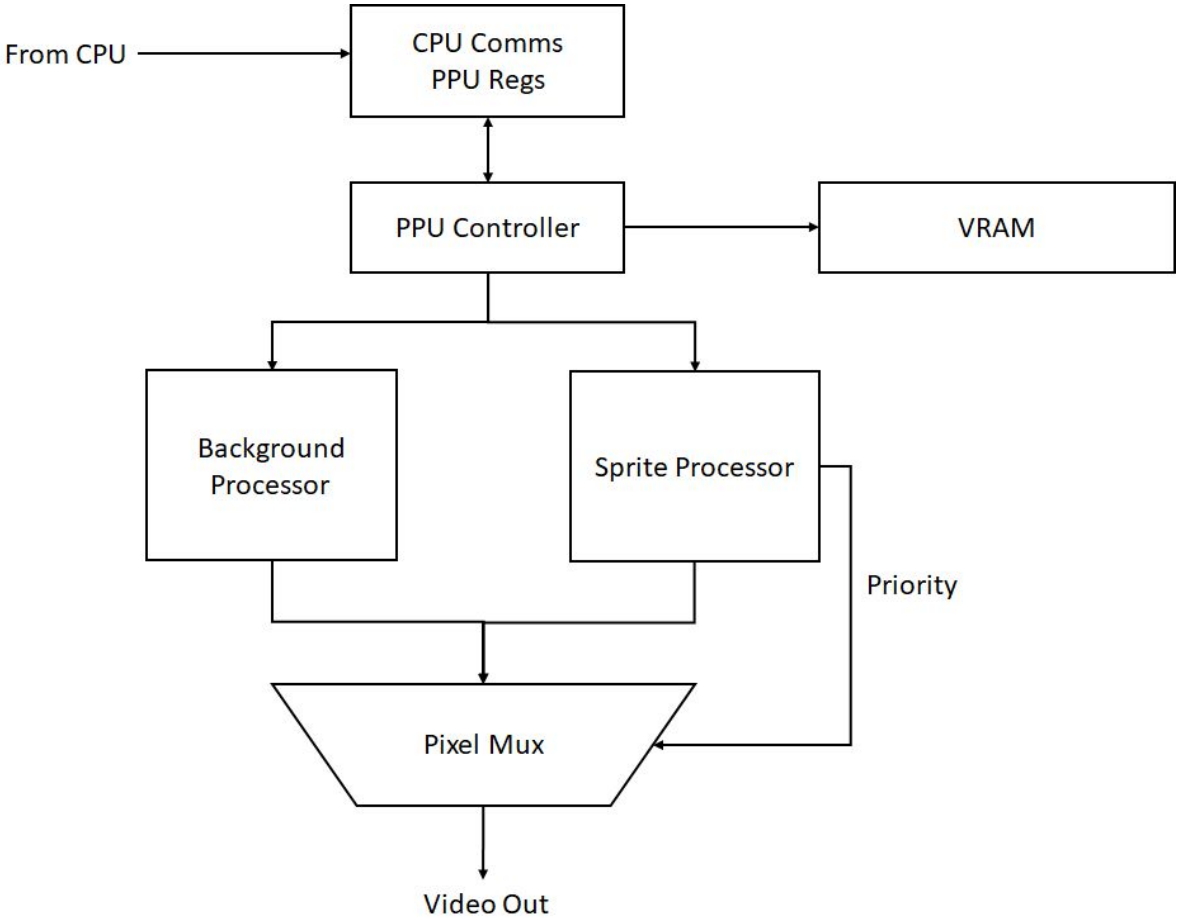

Figure 3: PPU Block Diagram

Communication between the CPU and PPU

The CPU and PPU communicate via a central data bus. The CPU talks to the PPU (and all other external devices) by writing data to special memory locations (this is known as memory mapping). In this way, the CPU is able to command the PPU to perform complex operations in parallel to its own operation. However, sometimes the CPU needs to transfer a large amount of data to the PPU and it would be slow for the CPU to individually copy every byte to the PPU. To address this (pun intended), there exists a special CPU instruction which initiates the transfer of a large block of data to the PPU from internal RAM. While this is occurring, the CPU can perform other tasks but, because the data bus is busy during this time, can not access the memory.

Stretch Goals

While our base goal for this project is to have a working emulated CPU and PPU that allows us to use NES game ROMs to play NES games on the FPGA, we would like to implement a few additional features time permitting. One such feature would be emulating the NES APU (Audio Processing Unit) so we have the game sounds and music in addition to our video output. Another feature we would like to add to this project would be the ability to interface with actual NES game cartridges. Ideally, we would like to be able to create some hardware that allowed us to plug an original NES game into our system and be able to play the game off of it.

Program Storage

While one of our stretch goals is to be able to interface an actual NES cartridge with our FPGA, we can get a minimum viable product by storing the NES games on easier mediums. For initial testing, we will obtain the original NES game ROMs and store them in BRAM. The CPU can then load instructions from BRAM at a rate of 1.79 million 8-bit instructions per second. The next option we may consider is utilizing the SD card slot on the Nexys board. This would allow us to load games programmed onto SD cards (would also require a memory access rate of 1.79 million 8-bit instructions per second) which would more closely resemble inserting a cartridge into the NES. Finally, we would like to build a hardware connector to interface the NES cartridge to the FPGA so that our system is compatible with original hardware.

Testing and Debugging

Throughout our design process, it is critical that we are able to effectively test and debug our system. One method is by writing test benches in Verilog to simulate different instructions and programs being run on our CPU/PPU. We have also found interactive 2A03 (NES CPU) and 2C02 (NES PPU) hardware emulators online that we can use to verify that the behavior of our system is correct. We can use existing software emulators to confirm that the behavior of our hardware is consistent with that of the original hardware. Obtaining an original NES, remove the CPU and/or PPU, and connect the NES to our FPGA such that our system stands in for the CPU or PPU would be another feasible method of testing. This would allow us to test the CPU alongside a working PPU and vise versa.