NES Emulation on FPGA

Author: Oscar A Ramirez Poulat: Electrical and Computer Engineering, Carnegie Mellon University,
Diego Rodriguez: Electrical and Computer Engineering, Carnegie Mellon University,
Nikolai Lenney: Electrical and Computer Engineering, Carnegie Mellon University

***Abstract*—Our capstone project is a hardware emulator of the Nintendo Entertainment System (NES) synthesized on the DE2-115 FPGA board. This system loads and runs NES game ROMs. It allows for up to two users to send inputs via the original NES controllers connected the FPGA's GPIO pins. Visuals are displayed on a monitor using a VGA controller and audio is transmitted through the AUX port with the on-board audio codec. Up to 16 games can be loaded into SRAM and toggled between them with the board's switches.**

I. Introduction

More than 30 years ago the NES made its debut and it became the bestselling video game console of its time. Fast-forward to present day and the NES is still very much loved. Most notably, it can be observed with Nintendo's release of the NES Classic, a modern NES emulator, that was sold-out almost instantaneously. Still, this was a software emulator and many like these exist ready to be downloaded. We saw this as a call to design and implement a hardware NES emulator on an FPGA. We were motivated by our love for retro-gaming and our want to challenge ourselves with building and integrating a full system. Moreover, this was a perfect opportunity to familiarize ourselves with peripherals such as video and audio. But we wanted to do more than simply recreate the NES. These software emulators had the pleasant feature of save states and we wanted to add that as well. We wanted to see how the NES was made and make it our own.

II. Design Requirements

- PPU - the Picture Processing Unit will be running at 5.36 MHz
  - the PPU will render frames 100% pixel-accurate.
  - the PPU will have MMIO registers in the CPU's address space to manage communication with the CPU.
  - The communication through the MMIO registers will also be cycle accurate, including the DMA of the OAM.
  - the PPU will have several internal memory blocks: 256 bytes of OAM (stores sprites), 2KB of VRAM (stores background tiles), 32 bytes of palette RAM (stores color information).
  - As in the original hardware, our system will support a maximum of 8 sprites per scanline.
  - Static frame rendering based on a VRAM dump will match cycle accurate emulator (Mesen)

- CPU - the Central Processing Unit will be running at 1.78 MHz
  - The CPU will run all documented instructions correctly
  - The CPU will run instructions 100% cycle accurate
  - The CPU will support IRQ and NMI
  - The CPU does not need to implement undocumented instructions

- APU - the Audio Processing Unit will be running at 1.78 MHz
  - The APU will have the five channels: pulse 1 & 2 , triangle, noise, and data modulation
  - The APU will fire IRQ's when the DMC finishes its samples
  - The APU will receive channel control signals from the APU in MMIO registers at addresses 0x4000-0x4017 in shared RAM
  - The APU will use a non-linear mixer to create a final wave without distortion
  - The frame counter will generate the channel clocks to keep output waves in phase and in their corresponding frequency
    S

- VGA
  - System will target the 640x480 @ 60Hz industry standard
- Controllers
  - Our games will be controlled with two original NES controllers
  - Controllers will be mapped to specific MMIO addresses on the CPU's address space
- SD Card
  - The system will support loading ROMs from an SD card
  - The system will also support saving/loading game progress to/from an SD card
- Save States
  - The save state module will be able to save the state of the system into memory and then load the data back into the system later such that the system will have the exact same state that it had before
  - The user will be able to save and load states with buttons on the FPGA

From the list above, some of the requirements are worth noting.

We required that the CPU should be 100% cycle accurate. The motivation for this is that the system is driven by the CPU, and many of the interactions (especially between the PPU and CPU) have very specific timing requirements. If these timing requirements are not held, it is much more likely that the game running on the system will deviate from the expected behavior. This could lead to glitches, and possibly even have the game freeze.

The cycle accurate requirement for the PPU register interface is very important because it ensures timed events on the CPU are accurate. Keeping these requirements allows us to run more complex games such as F1 Race that update a scrolling background mid frame. This is a very time sensitive operation that could not be accomplished if the register interface was not cycle accurate.

To stay true to the original NES's specs, our system's major parameters are summarized in the following table:

Master Clock Speed	21.477272 MHz
CPU Clock Speed	1.79 MHz (Master / 12)
APU Frame Counter Rate	60 Hz
PPU Clock Speed	5.36 MHz (Master / 4)
Height of Picture	240 Scanline (corresponds to 240 pixels)
Length of Vertical Blanking	20 scanlines
Total number of CPU cycles per frame	89341.5 / 3 = 29780.5
Vertical scan rate	60 Hz
Horizontal scan rate	31 KHz

At the very least our system should be able to: load the original Donkey Kong from an SD card, allow the player to use original NES controllers to play the game, let the player click a button on the FPGA to save their game state to the SD card, let the player click an alternate button on the FPGA to load their game state from the SD card.

III. Architecture and/or Principle of Operation

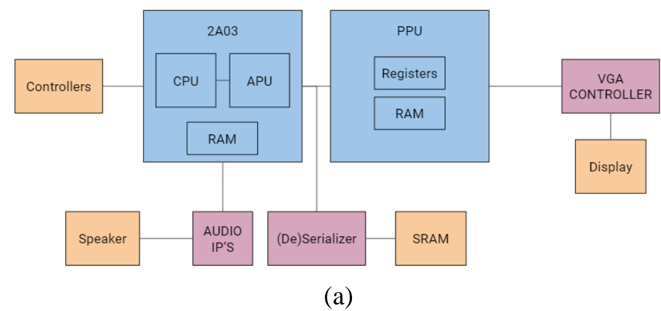
The overall system architecture is comprised of 3 major components: the CPU, the PPU and the APU.

When a game ROM gets loaded, the CPU will start reading and executing instructions. The instructions could correspond to reading user input from the controller, modifying sprites or background tiles to make a change on screen, changing the audio levels of the APU or arithmetic operations for updating game state. Most games implement their game engine as an IRQ handler that runs whenever the PPU finishes rendering a frame. The controllers will be connected to the FPGA via GPIO pins that the CPU will read and interpret accordingly.

The PPU's job is to look at VRAM and output the corresponding frame pixel-by-pixel. The pixel-by-pixel rendering is fed into a VGA module that converts the pixel's color to RGB values to output to the display. Additionally, the PPU has to perform reads and writes of VRAM on behalf of the CPU. The PPU also provides status information via MMIO registers on the CPU's address space so that the CPU knows when it is safe to continue execution. The CPU restarts execution of the game code when the PPU raises the VBlank IRQ, which occurs when the PPU has finished rendering a frame and is no longer accessing VRAM so that CPU can modify it. Consequently, a game only has about 2273 CPU cycles to perform game updates before the PPU takes over control and starts rendering the updates.

The APU is controlled by the CPU writes to memory mapped registers 0x4000-0x4017. These writes are intended to change the frequency, length, and volume of the APU channels. The APU channels that were implemented include the two pulse channels, noise, and triangle channel. The channels function independently from one another and are responsible for continuously generating their corresponding wave. The outputs of the channels are then passed into a non-linear mixer to determine the final audio. All the while, the frame counter is in charge of clocking and timing for the APU internals. It also raises IRQs under special circumstances. The DMC channel is the fifth channel in the APU, but it was not finalized and removed due to unwanted effects on audio.

The ROM was originally going to be loaded from an SD card using the NIOS II softcore, however due to time constraints we decided to load games from SRAM instead. In the end we wrote a script that took several game ROMs and combined them into one hex file that we then loaded on the FPGA using Terasic's control panel for the DE2-115. Once on the chip we used switches on the board to select what game to play and pressing one of the keys copied the game from SRAM to internal BRAMs used by the PPU and CPU. Initially we also wanted to support save states however we couldn't meet this goal.



(a)

Fig. 1. System picture. (a) Block diagram

IV. Design Trade Studies

Since our goal for capstone was to recreate the NES, many of our design choices were limited. We needed to strictly follow the expected behavior of the three major subsystems: CPU, PPU, and APU. If we were to diverge from this, the likelihood of properly emulating the NES hardware would be slim. Still, due to some vagueness in the documentation we had the liberty of inferring our own designs.

One of the more interesting choices we had to make was how to handle the clock inside of our CPU. The MOS 6502 has a two-phase clock, which means that is drives two clock signals, PHI1 and PHI2, which are mutually exclusive; one is high while the other is low. Most of the internal registers are clocked by the clock signal PHI1, but the memory bus signals are clocked on PHI2. In implementing the CPU, we could either use this two clock setup, or instead have one clock, and ensure that memory data arrives at the correct time relative to what the registered elements observe. This ended up being as simple as delaying memory by one cycle. While this wasn't the true implementation of the original device, our system would not be able to tell any differences because of it, and it was far simpler to add this memory delay than to deal with two clock signals in a closely interconnected system.

In the original system the PPU shared a memory bus with the CPU which meant that the CPU could use leftover values on the bus, however this behavior is only used by very few games and in order to simplify out design we had disjoint memory buses for the CPU and PPU this made the memory controllers simpler to manage and debug, while still being compatible with the vast majority of games.

Initially the PPU read background data differently than the original. Specifically, it didn't share the PPUADDR register between the background rendering and the register interface. This decoupling of the PPUADDR made it so that initial versions of the background rendering modules were easier to develop and debug. This decoupling of the register meant that scrolling games would not work because they rely on this exact behavior. Once the entire system was connected and non-scrolling games worked, I went back and rewrote background rendering so that it more closely matched the original implementation.

The CPU and PPU require strict cycle accuracy to achieve proper game rendering. Some games require communication

between these two units while the monitor is updating the image. Being one cycle off would result in noticeably incorrect pixel placement and color. However, the CPU and APU communication is significantly more lenient. The APU has a literal array of registers that hold the values written by the CPU before they are seen by the APU's components. This means that the APU channels do not see the most recent changes until the next CPU cycle.

This design choice is acceptable since it is essentially delaying the audio output for less than a millisecond so the difference would not be noticeable. Although having the CPU write directly into the APU's channels was doable, at the time of implementation it was much easier to reason over and debug the system by having all the signals stored in this register array. It is worth noting that the frame counter interfaced directly with the CPU's writes. The reason for this is because the frame counter could raise IRQ's. Some games uses these IRQ's for timing and synchronization so cycle accuracy was crucial for the frame counter.

A major difference between our implementation and the original was in the mixing scheme. The NES had 5 Digital to Analog Converters (DAC), one for each audio channel. The FPGA board we were synthesizing on only has one DAC that was part of 24-bit audio codec that could drive the audio jack. One option was to purchase some more DACs and drive them with the FPGA GPIO pins. This would require wiring some external components on a breadboard and then driving a speaker. Even though this would most closely capture the audio of the NES, we preferred if we could drive a speaker with a standard AUX cable. So, we opted to use a lookup table to approximate the non-linear mixing of the DACs and use the audio codec for its single DAC and capability to drive the AUX port.

One final design tradeoff we considered was how we would store our games and how where we would put the RAMs for our PPU and CPU. For our RAMs we had two options: Block RAM or SRAM. The advantages of BRAM is that we can use two BRAM blocks, so the CPU and PPU don't need to worry about stealing the memory ports from each other's. The advantage of using SRAM is that it is larger, and that the game ROM would likely be loaded into SRAM as well. The main drawback of using SRAM is that it only has one set of input wires, which means that the CPU and PPU could not use it simultaneously. It would still be possible to use SRAM in this way, since we could interleave these devices accesses to it on the cycles when each device is inactive. We decided against this since this interleaving would likely lead to problems, and the BRAM was more than large enough to fit our RAMs.

For storing our games, we had three main options: store them in Block RAM, SRAM, or on an SD card. The BRAM would have been very easy to use since the game ROM can be synthesized directly into BRAM, and we were already familiar with using BRAM since our RAMs are also in BRAM. It is also good because it means that the CPU and PPU don't have to worry about competing for any data lines since the ROM can be distributed into different BRAMs for program ROM and character ROM. The main drawback is that the overall capacity is small and would limit the number of games and save files we could store on the board at one time. This limitation is even worse if we were using games outside of mapper 0, since they tend to be larger. The SRAM is advantageous because we can fit more games in it since is has a 2MB capacity, but it is disadvantageous because it's a little harder to program with Quartus, and because the CPU and PPU can't access is simultaneously. The main advantage of the SD card is that is is persistent, which means we don't need to reprogram it, and that it could even maintain save data. The drawback of the SD card is that there is a lot of overhead in communicating with it, and that the components of our system wouldn't be able to communicate with it quickly enough for our games to run cycle accurately. In the end we ended up loading our games onto SRAM, and then selectively loading one active game at a time into the BRAM.

V. Testing and Verification

In order to test our overall system, we will first test our smaller subcomponents to ensure they work correctly. It will be crucial to do individual testing first because our overall system will likely not work if any of the individual components fail, especially the CPU and the PPU. Also the complexity of testing the system as a whole is much more complicated to do automatically. There are three major metrics we are going to benchmark: frame accuracy, cycle accuracy, and memory accuracy.

Most of our PPU testing for frame accuracy will be heavily reliant on the Mesen emulator. We chose it because it has a very good debugger and it is cycle accurate. It allows you to analyze every component of the NES at runtime, set breakpoints and most importantly for us it lets you copy the entire PPU memory. This is how we generate static VRAM dumps which we then feed to our hardware implementation to generate a frame. To generate a frame in our hardware we have a testbench that uses SystemVerilog to write the color of every pixel in our frame to a text file. We then have a python script that takes in this text file and compares it with a reference frame generated by the Mesen emulator. This way we can create any number of test vectors by loading any game ROM to the emulator, pausing at a frame that has behavior we are testing, and copying the static VRAM to our hardware emulator. Some of the behavior we are looking to test: frames with no sprites, frames with sprites, frames with more than 8 sprites on a scanline (this was a restriction on the original NES), frames with a scrolled background horizontally, frames with a scrolled background vertically. So far we have 5 tests of frames with sprites and no sprites we are using in development, but we plan on creating quite a few more when we get to more complicated parts of the PPU. In terms of PPU cycle accuracy, we plan on using counters and SystemVerilog assertions to ensure that all the signals get triggered on the correct cycle. We also need to ensure that our rendering process takes the exact number of cycles as in the original even if our implementation differs slightly. We will also ensure that every operation on the PPU's registers takes the exact amount of cycles as in the original spec, again we will

use assertions to verify this. For instance, DMA for the PPU needs to take 513 cycles if on an even CPU cycle or 514 cycles if on an odd CPU cycle.

In terms of accuracy, for the CPU we will benchmark our implementation against a reference implementation of the 6502. We will run a handpicked set of benchmark tests to verify that every instruction and addressing mode works adequately.

For the APU we can also use the Mesen emulator to look at its registers and compare them to our own to ensure we are producing the correct sounds. Similar to the PPU testing we can load any game we would like and probe its values at a given cycle.

Some of our frame accuracy results:

Test	Accuracy
balloon_fight_trace0	99.99%
balloon_fight_trace1	99.43%
balloon_fight_trace2	100.00%
donkey_kong_3_trace0	100.00%
donkey_kong_3_trace1	100.00%
donkey_kong_3_trace2	100.00%
tag_team_wrestling0	100.00%
tag_team_wrestling1	100.00%
tag_team_wrestling2	100.00%
super_mario_bros0	100.00%
super_mario_bros1	100.00%
super_mario_bros2	100.00%

These results represent how closely our generated frames match a known cycle accurate emulator such as Mesen. Most games we reached the intended accuracy, however for some of our balloon fight traces we did not generate some pixels correctly. I did not look too deeply as to why this was occurring but I believe it was an issue with how I acquired the traces.

We wrote a testbench that tested specific registers in the PPU here are our results.

Test Name	Description	Status
OAMDMA	Tests the OAMDMA register	Passed
OAMDATA	Tests the OAMDATA register	Passed
OAMDMA timing	Tests the timing of OAMDMA	Passed
PPUDATA	Tests the PPUDATA register	Passed

We used tests from the NES wiki to verify that our sprite 0 hit was correct.

Test Name	Description	Status
01.basic	Tests basic sprite 0 detection	Passed
02.alignment	Tests alignment of the detection with cpu cycle	Passed
03.corners	Ensures correct sprite 0 behavior in the corners	Passed
04.flip	Ensures correct sprite 0 behavior when fliping a sprite	Passed
05.left_clip	Ensures correct sprite 0 behavior when left side of the screen is clipped out	Passed
06.right_edge	Ensures correct sprite 0 behavior in the right edge	Passed
07.screen_bottom	Ensures correct sprite 0 behavior in the bottom of the screen	Passed
08.double_height	Ensures correct sprite 0 when using 8x16 sprites	Failed
09.timing_basics	Ensures sprite 0 hit timing to within 12 or so PPU clocks.	Failed
10.timing_order	Tests sprite 0 hit timing in case where multiple sprites hit	Failed
11.edge_timing	Tests sprite 0 hit timing for which pixel it first reports hit on when some pixels are under clip, or at or beyond right edge.	Passed

For the APU we used NES tests as well:

Test Name	Description	Status
reset	Tests APU power and reset state	Passed
pulse div	Tests frequency timer of pulse waves	Passed
env	Tests envelopes	Passed
sweep	Tests the overflow and cutoff of sweep unit	Passed
timers	Tests the frequency of all channels	Failed: noise, dmc
tri lin	Tests triangle linear counter	Passed
volume	Tests the volume of all 5 channels	Failed
apu test	Array of tests	Inconclusive

To verify the correctness of the CPU we decided to use a ROM that tested each instruction in the 6502 against a reference implementation. The test is called nestest.nes and can be found in the NES wiki. We passed the entire test.

Here is a list of games that the CPU can run:

Games that are playable	Load but still have issues	Don't Load
Balloon Fight	B-Wings	Galaxian
Bomberman		
Donkey Kong		
Donkey Kong 3		
Donkey Kong Jr		
Donkey Kong Jr Math		
Ice-Hockey		
Galaga		
Mario Bros		
Pac-Land		
Pac-Man		
Space Invaders		
Super Mario Bros		
Soccer		
Spelunker		
Tennis		
Xevious		
Tag Team Wrestling		
Ice Climbers		
1942		
Baseball		
Excitebike		
Kung Fu		
F-1 Race		

VI. System Description

A. CPU

Our CPU is a recreation of the MOS 6502, with some slight differences. The main differences from the MOS 6502 are that we will not support decimal mode addition, since this feature was removed on the NES, and we will not support undocumented opcodes, since only a small subset of games use these.

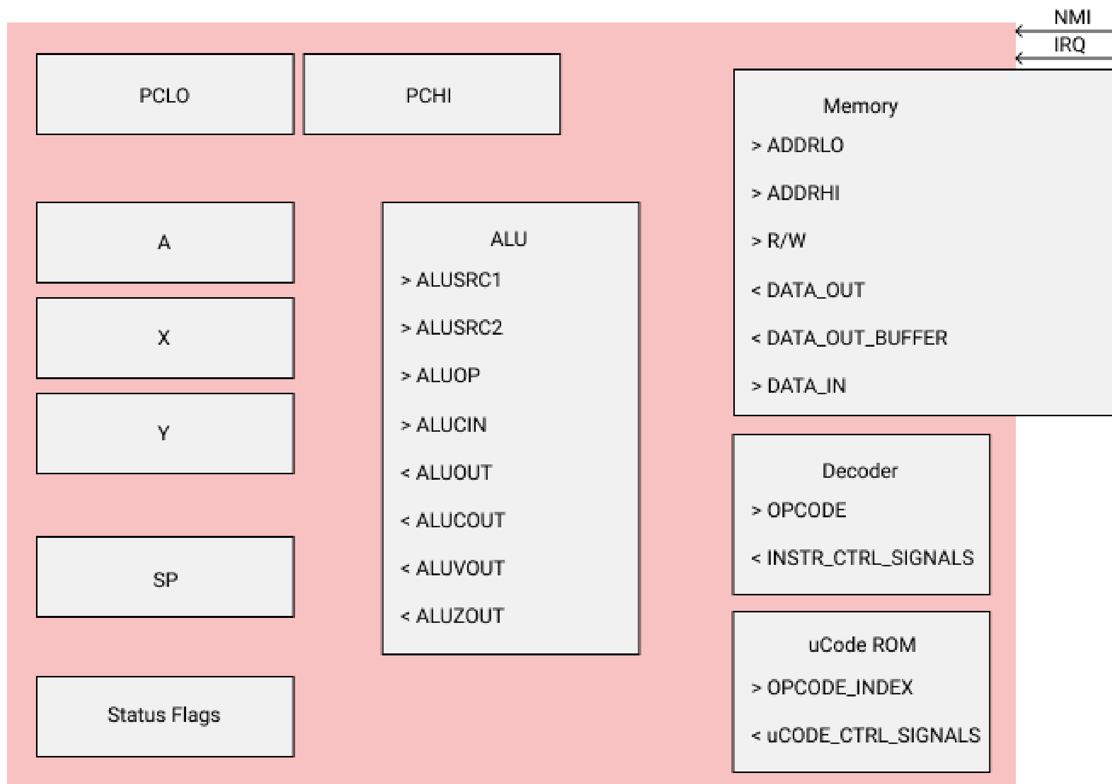
The MOS 6502 is an 8-bit, microcoded processor, with a 16-bit address space. The architectural state of the processor includes the 16-bit program counter (PC) and the following 8-bit registers: the accumulator (A), two index registers (X and Y), stack pointer (SP), and status flags. The status flags include 7 flags: the negative flag, overflow flag, break flag, decimal flag, interrupt flag, zero flag, and carry flag.

To interface with memory, the MOS 6502 has a 16-bit memory address line (ADDR), a single bit read and write line, and an 8-bit data line, which we have split into a data_out line (for reads) and a data_in line (for writes). We also added an an internal register for the previously read value to the memory controller. The read line is active on every cycle, so the CPU either reads or writes on every cycle. The address is registered, but it is also write-through, so the memory address does not need to be reasserted to access the don't need to change the address if you want to keep accessing the same address in memory, and there is only a delay of one cycle per each read, even for new addresses.

Our ALU is fairly straightforward. It takes in two 8-bit values (alu_src1 and alu_src2), a single bit source 2 invert signal (alu_src2_invert), a single bit carry-in (alu_c_in), and an operation (alu_op), and produces an 8-bit output (alu_out), a single bit carry-out signal (alu_c_out), a single bit zero-out signal (alu_z_out), and a single bit overflow signal (alu_v_out). The alu operations are add, xor, or, and, shift left, shift right, and hold. The alu does not operate combinationally, so all of the output signals are registered, which is necessary for some addressing modes. The hold operation just keeps the outputs of the alu steady.

These are the main elements of the datapath. The registers primarily interact with each other and memory through the ALU. To move a value from A to X, for example, A is moved to alu_src1 and 0 is moved to alu_src2, 0 to alu_c_in, and add to alu_op. The alu_output would then be moved into Y on the following cycle. An 8-bit value can be moved directly into the status register, or individual flags can be set, cleared, or set based on the outputs of the alu. The PC has a special 16-bit incrementor, so its value can be updated without needing to use the ALU. Also note that PC and ADDR can be split into separate 8-bit halves, since it is necessary in many cases to move an 8-bit value into just one half of these two addresses.

To manage the datapath, we need our control signals. The control signals are dictated by each instruction. Every instruction has the same first two steps: fetch and decode. At the beginning of each instruction, the only known is the PC, or



the address of the instruction, so fetch just issues a read to memory at the address of the PC and increments the PC. In decode, we have the actual opcode of the instruction available, but we still need to interpret it to figure out what actually needs to be done. Each opcode specifies an instruction and an addressing mode.

Many opcodes have a vector of control signals associated with them. These signals can specify ALU sources, ALU output destinations, ALU operations, branch conditions, flag setting conditions, etc. It is expected that the alu operation happens prior to the register write back and flag setting. The control signal vector does not specify when each operation needs to be performed, but rather, that these operations need to be done during the instruction's lifetime.

Every addressing mode has a specific sequence of control vectors (the microcode) that are used to manage which reads, writes, alu operations, and register writes happen, and when. The control vector can specify what values to move into the address line, the read signal, what values to move into the PC, whether or not to skip a line in the microcode, whether or not to halt the microcode, when to begin fetching the next instruction, etc.

A link to the CPU's microcode that we've written so far has been included in our references.

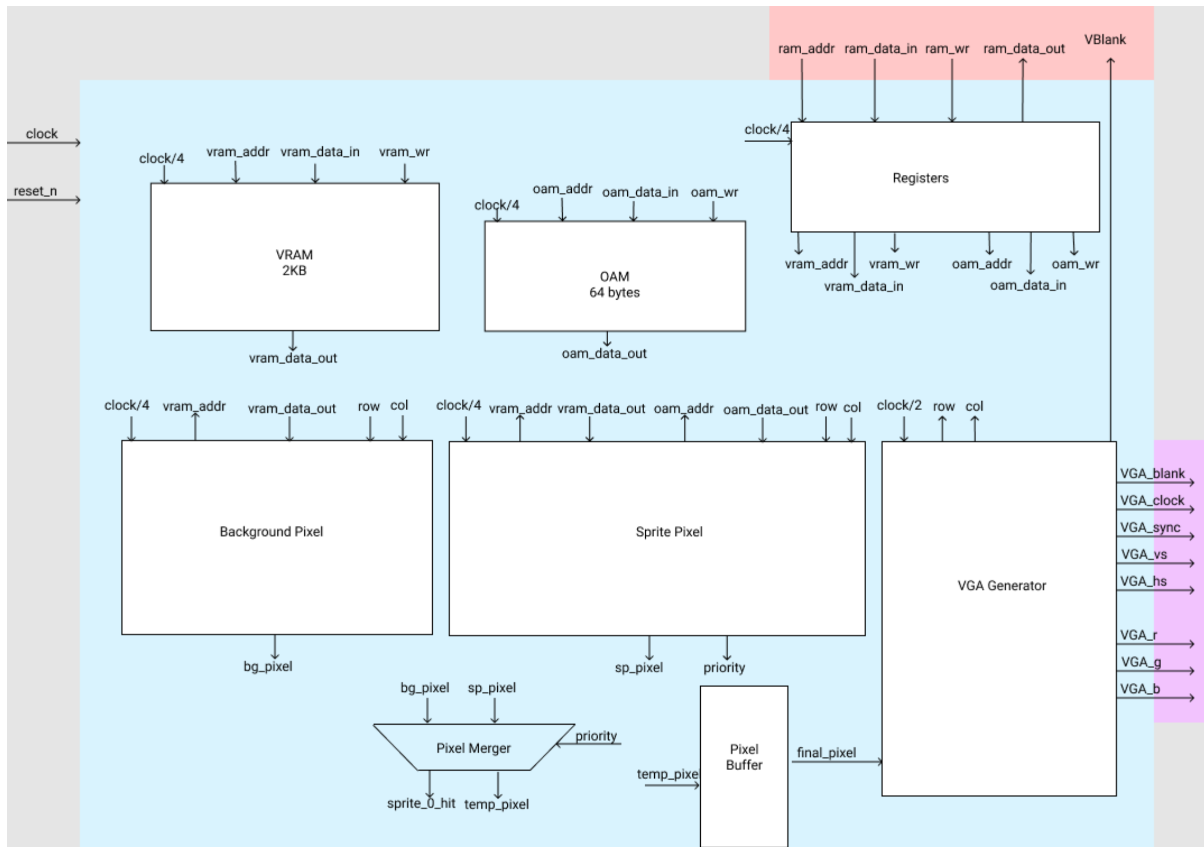
Although the MOS 6502 is a relatively simple CPU, it still has some instruction level parallelism. The CPU can fetch the next instruction, even if it still finishing up an instruction, if the instruction if it is still working on will not use memory for the duration of its execution. Some instructions which only read from memory, such as logical operations, loads, and comparisons fit this category. In many cases an instruction will run for two cycles while the succeeding instruction is in fetch and decode, though this doesn't create any data hazards since none of the registers are accessed in fetch or decode. In decode, we need to figure out what the vector of control signals should be, what line of microcode to jump to, whether we need to increment the PC again, and whether to start fetching the next instruction on the following cycle. The decoder module is meant to take in an opcode and figure out all of these. Note that some addressing modes specify operands, and in these cases we need to increment the PC while still in decode. The uCode ROM is just a memory that stores vectors of microcode control signals.

The CPU also has two interrupt signals, which are the interrupt request (IRQ) and non-maskable interrupt (NMI). These interrupts can trigger an interrupt handler to be run in the CPU in place of an instruction. These interrupts can come from the APU and PPU when they need to communicate to the CPU.

In addition to the interrupts, the CPU can share information with the APU and PPU with shared memory. The CPU shares 8 8-bit register with the PPU as part of its address space, and it shares an additional 24 8-bit registers with the APU as well. The first two Kilo-Bytes of the address space are reserved as the CPU's RAM, and will be implemented with block RAMs. The top 48 Kilo-Bytes of the CPU's address space is the cartridge space, which is where the games' instructions live. We will use the FPGA's SRAM to implement the cartridge space. The remaining space in the address space just maps to the same portions listed previously, so multiple different addresses will map to identical portions of memory.

B. PPU

The PPU (Picture Processing Unit, is responsible of rendering the game's frames and displaying them on a monitor



via VGA cable. The PPU has two main roles: create an interface to interact with the CPU in memory and display the appropriate frames on screen.

First a brief overview of the memory layout and how sprites are represented internally. The pixel information to display on-screen is not kept on a pixel by pixel basis, instead pixels are grouped into tiles which correspond to an 8x8 pixel area. These tiles are kept in the game's ROM. From the PPU's perspective, however, the tiles are kept in the first 8KB of the PPU's address space. These 8KB are split into two tables called Pattern Tables, one holds tile information for sprites and the other for backgrounds. The layout of backgrounds is kept in the Nametable, and it is on a tile basis. In other words, an entry in the Nametable corresponds to a tile index which is an address in the Pattern Table. Sprite information is kept in a table called OAM, there are 4 bytes per sprite corresponding to the x position, y position, the tile index, and sprite attributes (such as vertical or horizontal flipping). Finally, there is the Palette RAM which holds 4 palettes (sets of colors) tiles.

Communicating with the CPU is done through a set of registers that allow it to write to VRAM and OAM, which control background tiles and sprite tiles respectively. By writing to these, the CPU can modify what gets rendered. Other registers let the CPU specify an X and Y scroll so that tiles on screen give the illusion of scrolling.

The controller register (**PPUCTRL**) is at address 0x2000 (of CPU's address space). This is a write only register that allows the CPU to set the following attributes: base nametable address (where in VRAM the background tile information is read from), set the stride for how to access the PPUs VRAM either +1 or +32, specify what pattern table to use for sprites, specify what pattern table to use for background tiles, and whether the PPU should generate a Non Maskable Interrupt to the CPU at the start of the vertical blanking interval.

The mask register (**PPUMASK**) is at address 0x2001. This is a write only register that allows the CPU to control how the PPU renders sprites, backgrounds, and colors. Specifically, there are bits to control: greyscale (give pixels a greyer look), whether the PPU should render backgrounds or sprites on the leftmost 8 pixels of the screen, hide background, hide sprites, and change RGB color intensities.

The status register (**PPUSTATUS**) is at address 0x2002. This is a read only register that allows the CPU to know what the state of the PPU is. This register is often used for timed events, specifically by knowing when the PPU has reached a specific pixel on the screen. The most important flags in this register are the Sprite 0 hit, which gets triggered when a special background tile overlaps another special sprite tile, and the Vblank flag, which gets triggered when vertical blanking phase begins.

The OAM address register (**OAMADDR**) is at address 0x2003. This is a write only register that specifies at what location of the OAM (memory that holds sprite information) the CPU wants to write to.

The OAM data register (**OAMDATA**) is at address 0x2004. This is a read and write register that specifies the data you want to write to address in **OAMADDR**. **OAMADDR** is incremented after each write to **OAMDATA**.

The scroll register (**PPUSCROLL**) is at address 0x2005. This is a write only register that allows the CPU to specify what the top left corner pixel should get rendered. This allows pixel-granular scrolling to work and was a great feature of the NES at the time.

The address register (**PPUADDR**) is at address 0x2006. This is a write only register that allows the CPU to specify at what address in VRAM to write data to. This register is used in conjunction with the **PPUDATA** register to fill in the PPU's VRAM with background tile layouts.

The data register (**PPUDATA**) is at address 0x2007. This is a read and write register which allows the CPU to specify the data to write to VRAM. After each write the **PPUADDR** register is incremented by 1 or 32 depending on the stride bit specified in the **PPUCTRL** register.

The OAM DMA (**OAMDMA**) is at address 0x4014. This is a write register that allows the CPU to perform DMA on the PPU's VRAM. The CPU only has to write one byte YY to this address and the PPU will copy the data from range 0xYY00 - YYFF of the CPU's memory into the PPU's internal OAM.

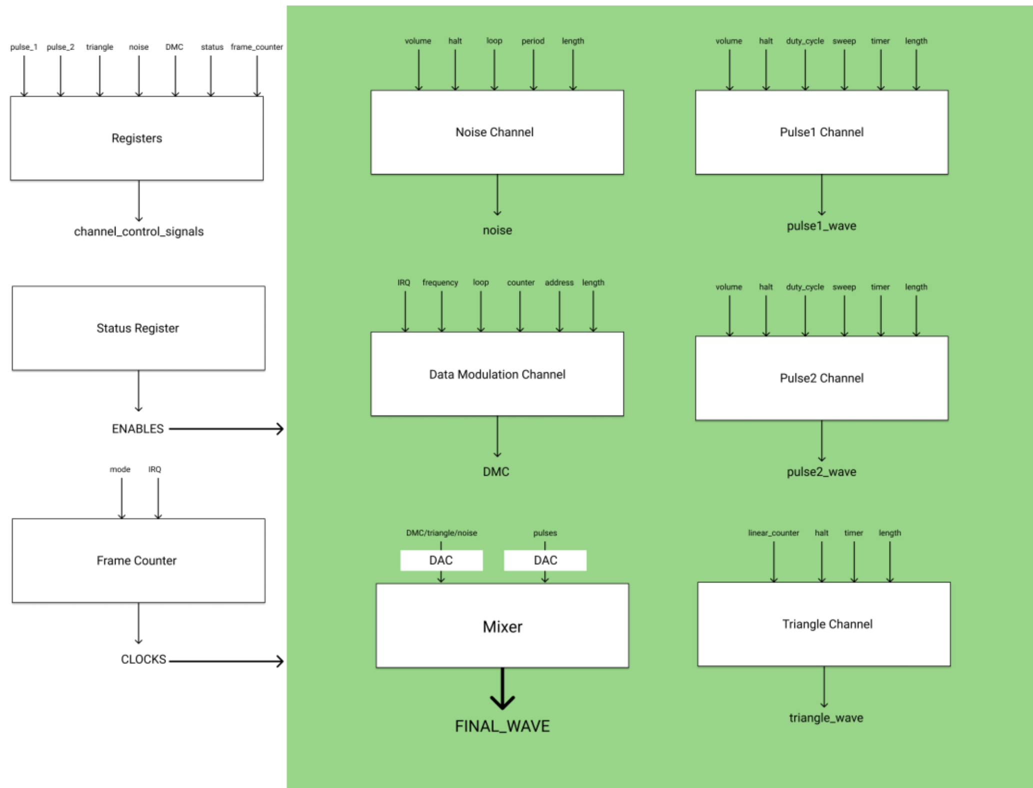
The PPU will display pixels on a monitor through VGA. Depending on your geographic location, the original NES was engineered to display images on the NTSC or PAL video standards. The two video standards have distinct frequencies they run at. The original NTSC video standard has a horizontal refresh rate of 15KHz, that is horizontal scan lines are fed to the display at 15000 per second. However, modern VGA has a horizontal refresh rate of 31KHz, so we have designed our system to render frames at the original 15KHz but output them at 31KHz. This is done by running the VGA module at twice the frequency of the PPU, and either: outputting the same scanline twice at double the speed or outputting the scanline at double the speed followed by a black scanline. The latter option will give our games a retro CRT-like look, so we will be opting for this initially. If we notice that the image is too dark, and we don't like how it looks we will revert to the first option. In order to accommodate the VGA interface and to use the resources we have on the FPGA more effectively we modified the rendering process of the original NES, while keeping the overall cycle counts the same.

From the PPU's perspective the rendering process takes 262 scanlines, each one outputting 341 'dots' (can think of them as pixels but only the first 256 'dots' will be visible on screen) one per cycle. The first 240 scan lines are in charge of rendering the visible pixels, this is accomplished by rendering pixels into a buffer that is passed to the VGA module after a scanline. For each pixel at a particular position (x,y) we calculate its color by looking up the tile and color information in VRAM, OAM, and palette RAM based on x and y. We then write this color value into the previously mentioned buffer and we move on to the next pixel. For each scanline the first 256 cycles correspond to visible pixels and the remaining 85 cycles are used for the VGA's horizontal sync. The 241st and 242nd scan lines will be idle scanlines. They maintain the same cycle counts as in the original NES. Scanlines 243 - 262 will correspond to the VGA's vertical sync.

From the VGA's perspective, the process is similar. However, instead of having 262 scanlines, we will have 512 to compensate for twice the PPU's frequency. Each scanline will also have 341 'dots'. After a PPU scanline is done rendering, we will pipeline it to another buffer in the VGA which will start outputting the pixels. Because of their different clocks, one PPU scanline corresponds to two VGA scanlines, thus the VGA will display the PPU's rendered scanline twice or once followed by a black scanline, depending on the look we want. Since the VGA will output scanlines at double the frequency of the PPU we will achieve the 31KHz horizontal refresh rate needed for VGA protocol.

C. APU

The Audio Processing Unit (APU) is responsible for generating the sound for the NES. The audio output is non - linear combination of the five channels: pulse1, pulse2, triangle, noise and the DMC. Each one these channels work independently and continuously output amplitudes that are then joined by the non-linear mixer to output its final value. Our system does not have the DMC. It was removed before demo because it was failing to reliably produce its audio signals and began to interfere with the overall quality of the sound. By removing it, the four remaining channels were clearly heard without being drowned out by the DMC.



As for the channels that were implemented, a brief description of each is provided below. The pulse channels basic functionality was to produce a square wave. Additionally, it had a sweep and envelope unit. The sweep unit helped vary the frequency and the envelope varied the volume to produce waves that resembled as sawtooth wave. The only difference between pulse1 and pulse2 was the addresses the registers were mapped to and how the seep unit calculated its target frequency. The triangle channel, as the name suggests, produced a triangle wave. Finally, the noise channel created pseudorandom noise.

All these channels can be written to individually. The signals that are controlled in all these channels are the frequency and the length. The frequency is represented with a timer that is essentially a variable clock divider. When the timer reaches zero it attempts to clock the channel's sequencer. It is the sequencer that determines what is the next value for the channel to output. Note, I mentioned that when the timer is zero it attempts to clock the sequencer. Any timer pulse must go through gates. When the gates are activated, they allow for the value to go through, otherwise they force the output to zero. Whether these gates are activated or not are most often controlled by the length and linear counters. If the counts are non-zero then the gate is activated and allows for the channel output to change. Otherwise, zero is outputted and thus the channel is muted.

Then there is the frame counter that clocks the five channels. More specifically, it produces the clock for their linear and length counters. Furthermore, it clocks the envelope and sweep unit. The frame counter controls how often channels need to be updated and when they are to be updated. In addition to timing, the frame counter is also responsible of raising IRQs. This interrupt is connected directly to the CPU and must be handled immediately.

The outputs of the channels are then passed into a non-linear mixer. It is a lookup table that converts the outputs of the many channels into a single amplitude. This table is meant to mimic the weights given to the channels in the NES.

VII. Project Management

A. Schedule

The final schedule can be found on the last page of the report

The schedule shifted dramatically due to the rearrangement of responsibilities. Originally, Oscar and Diego were meant to work together on developing the PPU under the assumption that it could be done concurrently and that the APU was much simpler. Early on it became clear to see that working side-by-side on the PPU was inefficient and that the APU could be crucial for the system's timing so Diego switched to work solely on the APU.

Originally all three team members had slack in the final weeks of the schedule. Difficulties with using the on-board audio codec put testing of the APU behind schedule. Thus, development of the APU was stalled significantly and Diego's slack disappeared. As a result, Nikolai was then placed in charge of handling save states, but the late finalization of the subsystems left too little time to setup all registers for loading and saving. Finally, Oscar struggled with a combination of version control issues and game scrolling bugs which extended his expected schedules into the final weeks. His slack time was replaced with setting up SRAM to boot 16 games for demo.

B. Team Member Responsibilities

Nikolai – Responsible for the development of the CPU

including a software simulation and hardware implementation, and their verification. Nikolai worked with Oscar to integrate the CPU and the PPU. He also led the interfacing of the system with the controllers. His final task was adding the save states.

Diego – Oscar and Diego shared the workload during the earlier stages of the PPU development, but Diego shifted to APU development. Diego also assumed the responsibility of integrating the APU with the CPU. He was also responsible for creating the interface to setup and use the audio codec for driving the DAC and AUX port.

Oscar - Oscar oversaw the PPU design and implementation, though Diego helped him during the earlier phases of research and development. Oscar and Nikolai worked together to integrate the CPU and the PPU once both were completed. Oscar also created the VGA interface for the design. It was also his responsibility to load game ROMs into the SRAM and transfer it into BRAM to run multiple game ROMs.

All – All three members worked together on the various reports and presentations throughout the semester.

C. Budget

The Bill of Materials can be found after the References shown in Table 1.

D. Risk Management

For the CPU, the greatest Risk that exists is not properly implementing interrupts. The NES has two crucial interrupts: the VGA VBLANK and the APU IRQ. The former signals to the CPU that it is time for PPU memory to be before it is time to the PPU to render the next frame. If the CPU is unable to handle this interrupt, then essentially nothing on the display will be updated. Thus, the NES would be unusable. As for the APU IRQ, it is fired whenever the DCM finishes playing its sample. Although this is related to audio, some NES titles play "nothing" to transform the IRQ into a timer to setoff events in the game. The latter interrupt is not as fatal for the project but can limit the number of games that can be ran on our emulator.

Furthermore, both interrupts require appropriate context switching. Even if the CPU manages to handle the interrupts correctly it must be able to return to the process that was originally interrupted. If this process context is not properly saved or restored then the game can crash or lead to some undefined behavior. To avoid this risk, Nikolai will prioritize implementing and testing interrupts using small benchmarks to verify that the running process can be interrupted, the handler is triggered, and context is correctly restored. As for Oscar and Diego, they will prioritize the DMC channel implementation to have the IRQ fire. Moreover, the VGA controller will produce dummy images and require refreshing the image to create the VBLANK signal.

One component of our project that ended up being a much higher risk factor than initially anticipated was the save state feature. This feature seemed simple enough to add, as it essentially just required us to stream data from our system into SRAM, or from SRAM into our system. The tricking part was routing data throughout our system. We maintained a spreadsheet listing where every single registered value in our system was (we need to know every single value to preserve, or else the state of the NES would be incorrect when reloaded), and used this to generate a Verilog Header file that assigns each registered signal an address. When streaming data to or from the system, we use this address to choose which register's data we read, or which register we right data to. The reason this was a challenge is because this meant we had to crawl through the entire system, and modify every sequential logic block to allow values to take values from an external source. Additionally, we had to add ports to almost every module in the system so that each sequential logic block could see the signals coming out of the save state module. This becomes painful when he have over a dozen modules and over a hundred registered signals.

The two main reasons we were unable to implement save states is because when fell behind schedule and couldn't find the time to catch up, and because it required all members of the group to be active participants. Since each leg of the project was written by a different person, that section of the project had to be updated by that person for save states, which meant that everyone needed to set aside time to work on this one feature while we had other things to work on in our project. A safer option would have been a feature that someone could have done without as much active participation from the other members, such as an additional Mapper to showcase more impressive games.

VIII. Summary

By the project deadline, we were able to meet many of our requirements, but not all. Our CPU seemed to meet the correctness and timing requirements we laid out, though our testing framework was not robust enough to prove this. Our PPU had rare issue where the screen would suddenly flash =, most likely caused by a Sprite 0 bug. As demonstrated in the metrics and validation section the APU passed several tests hosted on NESDEV. Still, it failed some cycle accuracy test for the frame counter which may have been the cause for some noticeable glitches, such as sustained sound effect and missing short length sounds, Despite these deviations, we were still able to play more than 20 games without any game breaking glitches (this number is likely higher, but we didn't test the whole library).

The most noticeable requirements that we did not reach were our SD card and save state requirements. The SD card was mainly intended to facilitate loading the game ROMs onto the FPGA. We were able to load multiple game together onto the board's SRAM. Another goal of using the SD card was to keep save data, but this point is moot since we didn't have any save data. As for the save data, while we had a save state module written that worked in simulation for saving and loading the CPU's data, we were not able to extend this feature to the APU and PPU as well, which is critical for the state of the game. Overall, our NES emulator matched the original the vast majority of the time with some small rare glitches and was

missing some additional features.

A. Future work

For the time being there are no concrete plans for future development of the emulator. If we were to continue with the project, we would finish adding save states. Also, correct the edge case tests that were being failed to resolve the strange and rare glitches. The most exciting prospect would be to add support for other Mappers beside Mapper-0. It would be rewarding to have iconic and impressive titles such as the Legend of Zelda and Super Mario bros 2 with the addition of support for Mapper-1 and Mapper-4 games.

B. Lessons Learned

Throughout our time working on this project we learned a few lessons. There are some of the more obvious lessons, such as plan your overall design from the start. This means answering questions about what the different parts in the system need to do, and exactly how they need to communicate with each other. We learned that simulators can be helpful tools when trying to understand how a design is supposed to work, especially when you have a test framework for the simulator, but they can take longer than expected to develop; while a simulator may take a long time to develop, it may still be worthwhile, since it means you have a better understanding of your design at an earlier point, and it can make the development of the actual RTL very quick.

For FPGA projects, it is a good idea to budget a lot of time for getting IPs to work. They may be advertised as "plug and play", but they often take an unreasonable amount of effort to get working. It is also a good idea to try and debug in simulation as much as you can. Synthesis is a very slow process, so spending time on a testing framework can save you the time you'd waste waiting for a design to compile and synthesize.

One lesson that is more specific to this project is that save states can be surprisingly hard to implement. The main reason for this is because it requires you to crawl through your whole system and make changes at every level so that every registered value can "see" the save state module. This requires a lot of effort from every team member and cannot be rushed in at the end. It may be fruitful to design your modules around adding this feature, so you don't need to crawl through later.

References

http://wiki.nesdev.com/w/index.php/Nesdev_Wiki
http://tinyvga.com/vga-timing/640x480@60Hz
http://tinyurl.com/mymicrocode
http://www.dustmop.io/blog/2015/04/28/nes-graphics-part-1/
http://www.dustmop.io/blog/2015/06/08/nes-graphics-part-2/
http://www.dustmop.io/blog/2015/12/18/nes-graphics-part-3/
http://nintendoage.com/pub/faq/NA/index.html?load=nerdy_nights_out.html
https://tresi.github.io/nes/

Item	Price	Quantity	Shipping	Did we use it?	Total Paid
(3rd Party) Nes Controller	$7.99	2	$2.99	No	$18.97
(1st Party) Nes Controller	$8.78	2	$0.00	Yes	Found in a bin in 1307
8-bit 2 pin Speaker	$3.95	2	$7.81	No	$15.71
Controller Adapter	$9.99	3	$0.00	Only 1	$29.97
DE2-115	$590.00	1	$0.00	Yes	Supplied by our Department
VGA Monitor	~$100.00	1	$0.00	Yes	Personal Item
Stereo Speakers	~~$50.00	1	$0.00	Yes	Personal Item
Synopsis VCS	~25,000	1	$0.00	Yes	Supplied by our Department
Quartus Prime	$3,995	1	$0.00	Yes	Supplied by our Department

Table 1: Project Bill of Materials

Week of	Nikolai	Diego	Oscar	Notable dates					
						0	Slack		
02/11	1	12	12			1	Research CPU		
02/14	1	34	34			2	Write Micro Code	24	Design Doc
02/18	2	13	35			3	Implement C Simulator	25	Design Presentation
02/21	2	14	36			4	Implement SV Implementation	26	Final Report
02/25	3	11	37			5	Test CPU instructions	27	Final Presentation
02/28	24	24	24			6	Create controller interface	28	Integrate CPU and PPU
03/04	3	11	36	Design Doc		7	Test controller interface	29	Integrate CPU and APU
03/07		11	37			8	Write Save State Module	30	Integrate entire system
03/11	0	0	0	Spring break		9	Update Registers for Save States	31	Synthesize APU
03/14	0	0	0	Spring break		10	Test serilization	32	Implement frame coutner
03/18	3	41	38			12	PPU research	33	Implement noise channel
03/21	3	40	17			13	Generate Foreground sprites	34	PPU Sprite Simulator
03/25	5	40	16			14	Generate Background sprites	35	PPU Timing analysis
03/28	4	32	23			15	Create Frame by combining sprites	36	PPU Background rendering FPGA
04/01	5	31	28			16	Test generated frames	37	PPU Sprite rendering FPGA
04/04	28	42	28			17	VGA module	38	PPU Memory Mapped Registers
04/08	6	19	22			11	Research APU	39	Debug APU
04/11	7	39	23			18	APU to CPU interface	40	Set up CODEC interface
04/15	30	39	30			19	Implement mixer	41	Implement triangle channel
04/18	30	33	30			20	Test APU	42	Implement pulse
04/22	30	43	30			21	PWM module for speaker	43	Implement DMC
04/25	8	39	27			22	Load ROM into SRAM		
04/29	27	27	27	Final Presentations		23	Testing framework with Mesen emulator		
05/02	9	39	30						
05/06	9	39	22	Demo					
05/08	26	26	26	Final Report Due					