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*— A system capable of running NES (Nintendo Entertainment System) ROMS of games such as Donkey Kong and Super Mario Bros. Our hardware implementation of the NES console will allow players to use original controllers to play and the ability to load and save game state data onto an SD card. Saving game state data is common in software emulators however it is rarer in hardware emulation, our system will try to bridge that gap.**

***Index Terms*—Emulation, NES, Retro Video Games, FPGA, SD card, RTL**

I. Introduction

We live in an era when video games are getting more complex every year. Graphics are improving, player counts are getting larger, and audio fidelity is at an all-time high. There is also a recent renaissance of retro-style games such as Shovel Knight, Celeste, Sonic Mania, and Cuphead. We thought it would be a good time to revisit the rebirth of modern video games, by studying and building an emulator for the NES system. When the NES came out, the video game industry had just experienced a crash in North America, however it quickly became the best-selling gaming console of its time. It revitalized the industry and helped propel it forward to what we have now. The NES has some of the most memorable gaming experiences such as Super Mario Bros 1, 2 and 3, The Legend of Zelda, Metroid, Donkey Kong, amongst many others. Even though there are many software emulators that provide cycle accurate emulation we want to use hardware to provide the user a retro feel closer to the original console, by using original controllers, outputting a CRT-like video signal, and offering a cycle accurate emulator on an FPGA board. One of the most attractive features of software emulators is saving game progress. Save states were restricted to a handful of games that had non-volatile memory in the cartridge, however most games did not support them. Saving game states is a nice feature because you can create your own checkpoints and avoid restarting the game from the beginning. Most FPGA emulators we found online, do not support this feature so we would like to offer this as a feature in ours.

II. Design Requirements

Our implementation of the NES will try to match the original console as close as possible. To accomplish this, we have the following requirements:

- PPU - the Picture Processing Unit will be running at 5.36 MHz
    - the PPU will render frames 100% cycle-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
    - Controller's will be mapped to specific MMIO addresses on the CPU's address space
    - the CPU will run instructions 100% cycle accurate
    - the CPU is based on the 6502 processor and will support IRQs
    -
- 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
- VGA
    - System will target the 640x480 @ 60Hz industry standard
- 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

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.

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

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 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 System Verilog 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 System Verilog 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 set of benchmark tests to verify that every instruction and addressing mode works adequately.

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

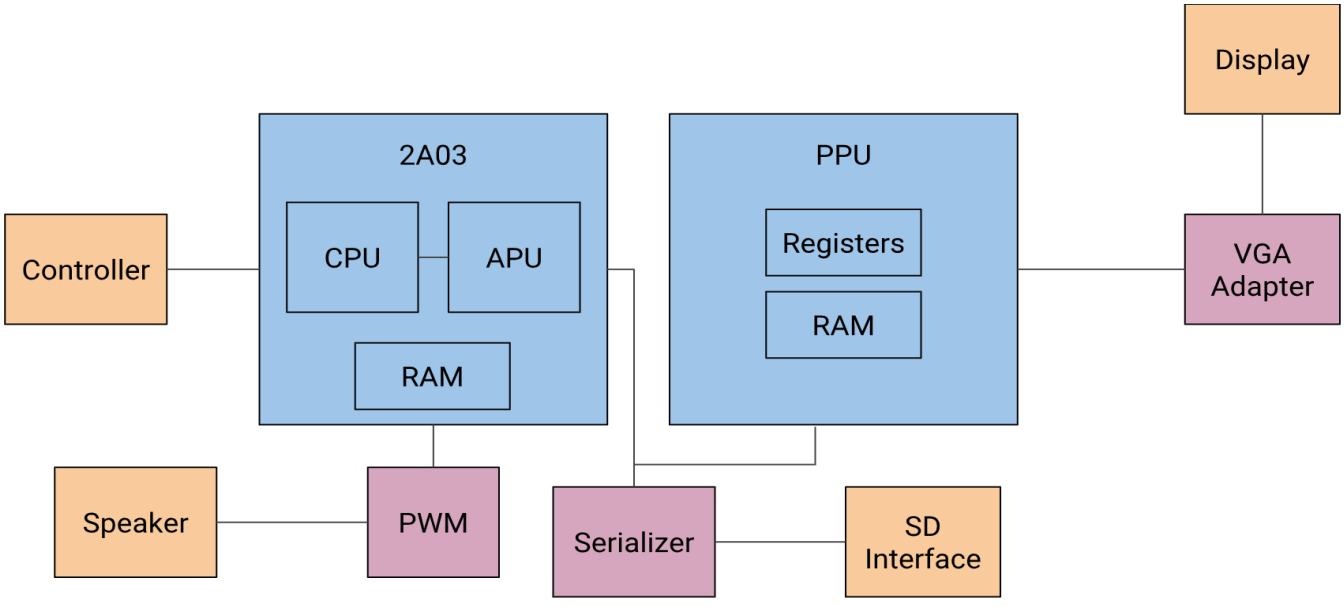
III. Architecture and/or Principle of Operation

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 of the game's game engine is implemented 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 ROM will be read from an SD card by using the NIOS II softcore on the FPGA and Quartus IP block for SD interface. Once the data from the SD card is read, it will be copied into SRAM so that the CPU and PPU can read it. Similarly, to store the state of the game, we will serialize all registers and the 2 RAMs (CPU and PPU). Once the data is serialized, we will store it at a predetermined position in SRAM that will then enable the NIOS II softcore to read the data from SRAM and store it in the SD card. To load a save state we will undo this operation by reading SD card data and storing it at predetermined location on SRAM so it then gets loaded into the PPU and CPU.



IV. System Description

A. CPU Subsystem

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 very small subset of games use these.

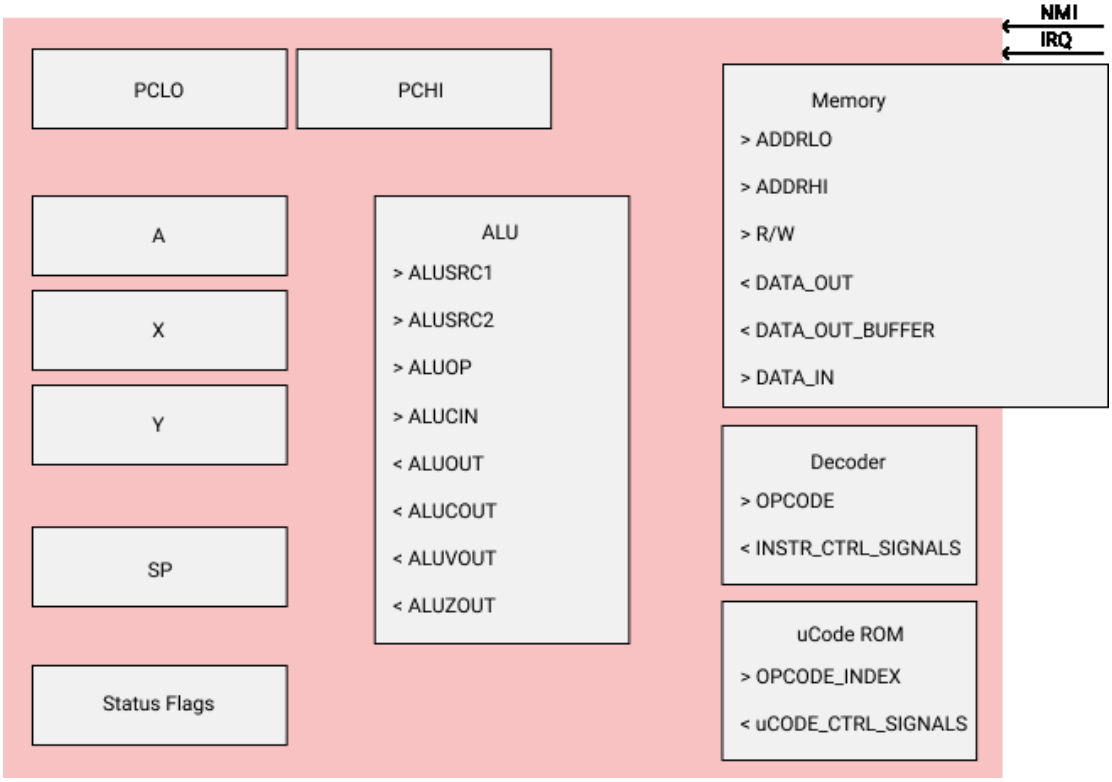
The MOS 6502 is an 8-bit, microcoded processor, with a 16-bit address space. The architectural state of the processor includes the 16-bit program counter (PC) and the following 8-bit registers: the accumulator (A), two index registers (X and Y), stack pointer (SP), and status flags. The status flags 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 need an additional internal register that holds the previously read value from the memory, which we include as a part of our memory interface. 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 you 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 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 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 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 to skip a line in the microcode, whether to halt the microde, 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 Subsystem

The PPU (Picture Processing Unit), oversees rendering the game's frames and displaying them on a TV screen. The PPU has two major jobs: facilitate interactions with the PPU's memory and displaying the correct pixels 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 a 8x8 pixel area on screen. These tiles are kept in the game's ROM along with the code. 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 the other tile information for backgrounds. The layout of a background is kept in the Nametable, and the layout 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. Sprites 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) for sprite tiles and 4 palettes for background tiles.

The first job is done through a set of registers that allow the CPU 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 on screen. Other registers let the CPU specify an X and Y scroll so that tiles on screen give the illusion of scrolling. These registers are at the heart of the NES since they are the bridge between what gets displayed on screen and the game's code.

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 the background tile layout information is located in the PPU's VRAM), 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 stars.

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**.

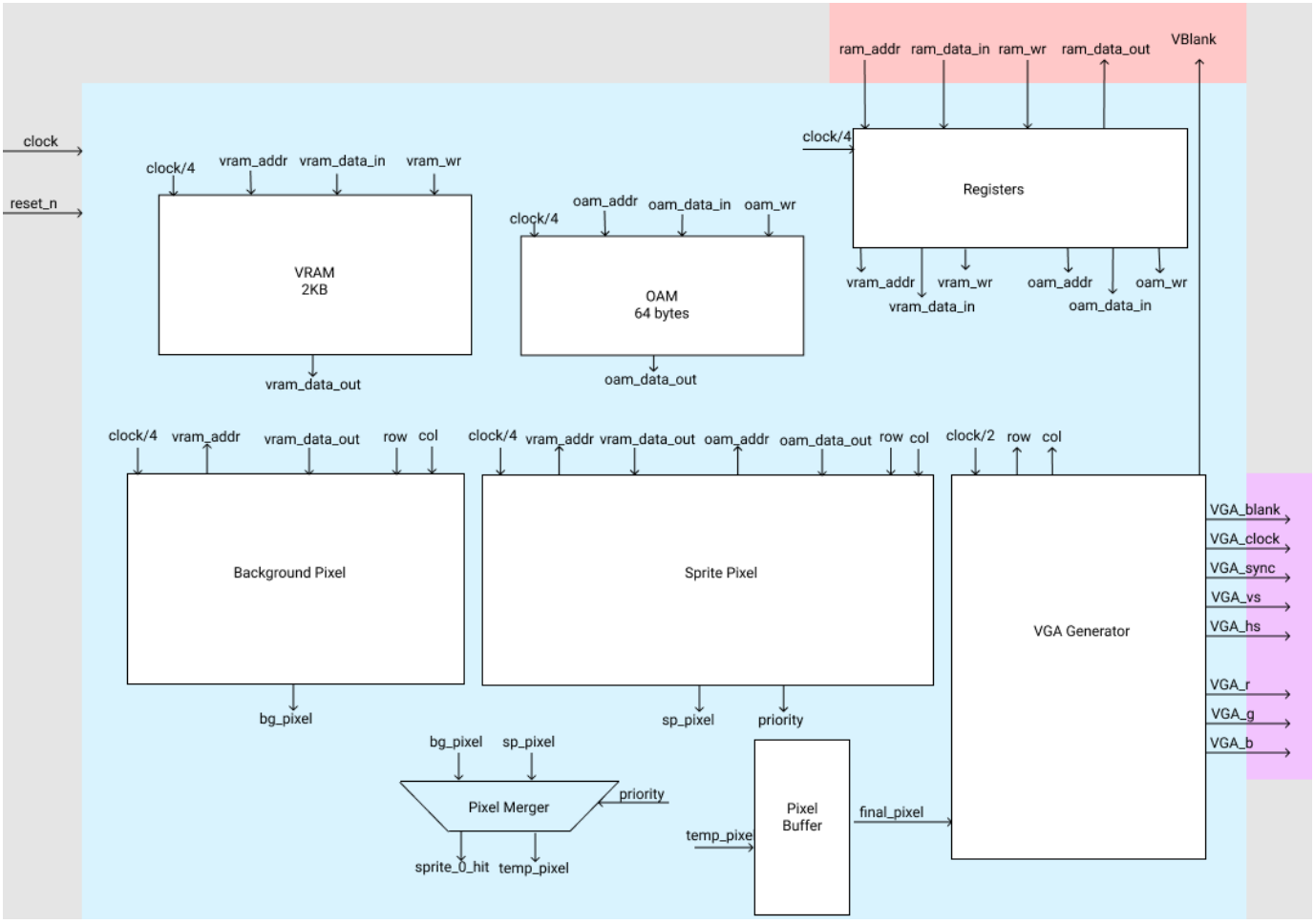
The scroll register (**PPUSCROLL**) is at address 0x2005. This is a write only register that allows the CPU to specify what the top left corner pixel should get rendered. This allows pixel-granular scrolling to work and was a great feature of the NES at the time.

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's second job (displaying pixels on screen) will be accomplished 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 ate 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 to 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.

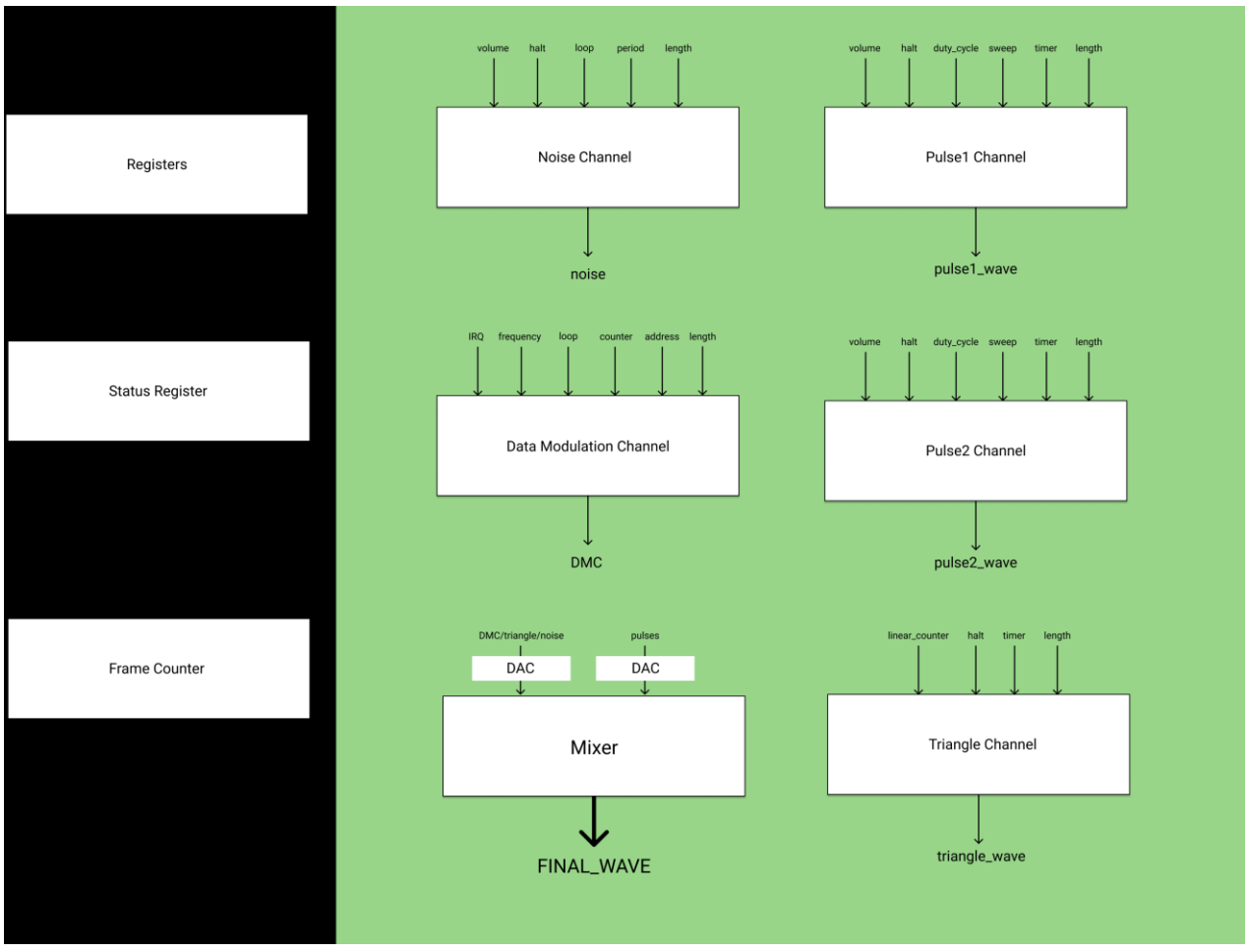
From the PPU's perspective the rendering process takes 262 scanlines, each one outputting 341 'dots' (can think of them as pixels but only the first 256 'dots' will be visible on screen) one per cycle. The first 240 scan lines oversee rendering the visible pixels, this is accomplished by rendering pixels into a buffer that is passed on to the VGA module once an entire scanline is done. 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 are simply kept to 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 the clocks they are running, 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 Subsystem*

The Audio Processing Unit (APU) is responsible for generating the sound for the NES. The CPU controls the APU by writing control signals at addresses 0x4000-0x4017 in the RAM they share. These addresses are wired to the APU's registers. As for the outputs, the APU produces a final wave that drives the speakers and can produce an IRQ.

The APU produces its sub waves with its 5 channels: pulse_1, pulse_2, triangle, data modulation, and noise. The pulse channels produce square waves with specified duty cycle, frequency, duration, sweep, and volume. Moreover, the volume can be steady or an envelope. The triangle channel produces a triangle wave with specified duration and frequency. The data modulation channel outputs a 7-bit PCM signal with the specified sample start address and sample size. When the DCM finishes playing its sample then it will fire an IRQ. Finally, the noise channel produces pseudo random bits with the exact same



control signals as the pulse channels.

Finally, the frame counter is responsible for creating clocks that will drive the different channels. The outputs of the DMC, triangle, and noise channel are joined together as well as the pulse waves. These two new waves are then passed through a digital to analog converter and then mixed to create the final wave.

V. Project Management

A. Schedule

See last page for a detailed view of our schedule

B. Team Member Responsibilities

Nikolai - Nikolai is primarily in charge of implementing the CPU, including a software simulation and hardware implementation, and verification of both implementations. Nikolai will work with Oscar to integrate the CPU and the PPU. It is also his responsibility to create an SD card controller that can be used by Oscar to load game data from the SD card into SRAM, and that Diego can use to load, and store save states from the SD card into the system. It is also his responsibility to create an interface between the NES controllers and the design.

Diego - Diego will share the workload of the PPU with Oscar during the earlier stages of the PPU development but will shift his focus towards implementing the APU. Diego will also assume the responsibility of integrating the APU with the CPU. He will also oversee interfacing with our speaker. It will also be Diego's responsibility to serialize and deserialize the save state data for the system, which includes moving data back and forth between the SRAM and SD card, after Nikolai creates an SD card interface.

Oscar - Oscar oversees the PPU design and implementation, though Diego will help him during the earlier phases of research and development. Oscar and Nikolai will work together to integrate the CPU and the PPU once both are completed. Oscar will also create the VGA interface for the design. It is also his responsibility to load game ROMs from the SD card into the SRAM, after Nikolai creates an SD card interface.

All - All three members will equally share the responsibilities of drafting the project proposal, design review, and final report. They will also equally share work on presentation slides for the three presentations. With respect to the design itself, all three members will work on system integration during the final weeks of the project.

C. Budget

For our project we're using several different software tools, hardware platforms, and peripheral devices.

In terms of software we're using Synopsys VCS for testing our designs in simulated environments. We have been using this tool for a few years in a few different courses, so we are very comfortable with using it to test hardware designs.

For synthesizing our design, we're using Quartus Prime 16. We're using Quartus because we're very familiar with it, and because the board we've selected is an Altera board. We've chosen this version of Quartus for the relative ease of use of IP blocks over other versions of Quartus that we've used.

Our FPGA board is a Terasic DE2-115. We picked this board for several reasons. One factor was our familiarity with the board and the Altera tool-chain, since learning Vivado would definitely have quite a bit of overhead for us. We also didn't have to pay for this board out of our budget, since the department fortunately has units to spare. Most importantly, this FPGA has all of the resources we need for this project. This board has 2 MB of SRAM, an SD card slot, 4 push buttons for managing our save states, and VGA output. On the FPGA chip there are over 114 thousand LUTs and 432 MK9s, which are each about 1 Kilo-Byte of memory. In other words, this will be enough for us to build our modestly sized system. I will also note that this was not the only contender we had for our FPGA. We were also considering the DE0-CV, since it also had most of the same resources in the quantities we required, however, its VGA port is 4-bit, as opposed to the 8-bit VGA of the DE2-115. With a 4-bit VGA we would only have 32 different colors to display, but the NES requires that we have at least 64 different colors, so the DE0-CV was ultimately an infeasible choice.

In terms of peripherals we have our controllers, our controller adapters, our speaker, and our VGA monitor. Our controllers are replicas of the original NES controllers, and behave identically. Our adapters take in NES controller output on one end and give us GPIO on the other, which we can easy attach to our board and interface with. Our speaker will serve as our audio output and our monitor will serve as our digital output. These don't really need to be any specific speaker or VGA monitor, as long as they can connect to the board.

Item	Price	Quantity	Shipping	Total
Nes Controller	$7.99	2	$2.99	$18.97
Speaker	$3.95	2	$7.81	$15.71
Controller Adapter	$9.99	2	$0.00	$19.98
DE2-115	$590.00	1	$0.00	Supplied To us
VGA Monitor	~$100.00	1	$0.00	Supplied To us
Synopsis VCS	~25,000	1	$0.00	Supplied To us
Quartus Prime	$3,995	1	$0.00	Supplied To us

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.

Furthemore, 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.

REFERENCES

[1] National Semiconductor Inc., www.national.com.
[2] NES Dev Wiki, http://wiki.nesdev.com/w/index.php/Nesdev_Wiki
[3] VGA Protocol Timing, http://tinyvga.com/vga-timing/640x480@60Hz
[4] Our CPU's uCode, http://tinyurl.com/mymicrocode

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	5	11	37		8 Serialize game state data	30 Integrate entire system
03/11	0	0	0	Spring break	9 Deserialize game state data	31
03/14	0	0	0	Spring break	10 Test serilization	32
03/18	4	19	38		12 PPU research	33
03/21	5	19	16		13 Generate Foreground sprites	34 PPU Sprite Simulator
03/25	28	19	28		14 Generate Background sprites	35 PPU Timing analysis
03/28	28	20	28		15 Create Frame by combining sprites	36 PPU Background rendering FPGA
04/01	44	18	16		16 Test generated frames	37 PPU Sprite rendering FPGA
04/04	45	29	17		17 VGA module	38 PPU Memory Mapped Registers
04/08	6	8	22		11 Research APU	39
04/11	7	9	23		18 APU to CPU interface	40
04/15	30	10	30		19 APU to speaker mixing scheme	41
04/18	30	30	30		20 Test APU	42
04/22	30	30	30		21 PWM module for speaker	43
04/25	27	27	27		22 Load ROM into SRAM	44 Implement SD Card Interface
04/29	26	26	26	Final Presentations	23 Testing framework with Mesen emulator	45 Testing SD Card interface
05/02	26	26	26	Demo		
05/06				Final Report Due		