FPGA NES

Alex Underwood
Brandon Wong
Clay Patterson
Daniel Bothwell

Oklahoma State University
College of Electrical and Computer Engineering
Capstone Design

Abstract (Alex)

The Nintendo Entertainment System (NES) is Nintendo's 1980-era console that paved the way for the gaming industry over the following decades. The FPGA NES is a senior project that aims to seamlessly recreate the original NES using SystemVerilog while updating the audio and video outputs to a modern interface. The system's functionality was broken out into 3 major subsystems (the CPU, the APU, and the PPU) and put on a development board. In the end, the system was recreated to meet all initial specifications we set for ourselves while also implementing additional mappers to allow more than 500 games to be played.

Abstract (Alex) **1**

Introduction (Clay) **3**

Project Specifications (Daniel) **4**

CPU (Daniel) **5**

APU (Clay) **8**

PPU & VGA (Brandon & Alex) **10**

Controllers (Alex, Brandon, & Clay) **12**

Cartridge and Mappers (Alex) **13**

Potential Improvements (Clay) **16**

Conclusion (Daniel) **16**

Acknowledgements (All) **17**

References (All) **18**

Appendix I - Code **19**

Appendix II - Pictures **19**

Introduction (Clay)

In 1985 Nintendo released a revolutionary gaming console called the NES (Nintendo Entertainment System). The NES was the first true color console to be released, and featured a 6-bit color scheme. This, coupled with other state of the art features, led the console to sell millions of units and become one of the most emulated consoles in history.

The FPGA NES aims to recreate the original feel of the NES while also improving the outputs to modern interfaces. Using Xilinx's Vivado, SystemVerilog instructs the compiler on how to synthesize the design. The program then creates the digital logic gates and places/routes them to make an actionable circuit.

For the project we selected Digilent's Zybo Zynq-7010 Development board due to the onboard FPGA's capabilities as well as the peripheral connections it had available. The Zynq-7010 integrates a dual-core ARM processor with a Xilinx -7 FPGA which allowed for in-depth debugging of the design's operation.

The project was divided into three major subsystems with two additional sections that were worked on in unison. These components are the Central Processing Unit (CPU), Audio Processing Unit (APU), and the Picture Processing Unit (PPU) along with the minor sections being the Controllers and the Memory Mappers. The CPU takes the data from the cartridges and converts it into actionable instructions that are passed through to the APU and PPU. The APU takes these instructions and uses its four channels to produce 8-bit audio and the PPU uses them to create the sprites and background to be outputted as a video signal.

Project Specifications (Daniel)

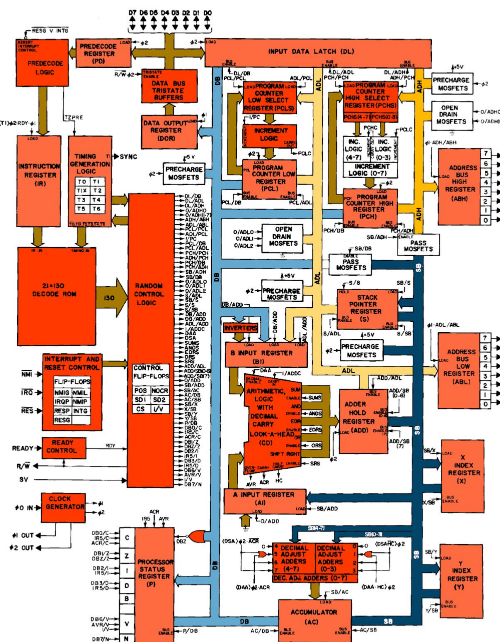
Project Specifications		Achieved?
1. A SystemVerilog implementation of an NES on an FPGA		
	1.1 Must be virtually indistinguishable from an original NES during operation	✓
	1.2 Capable of playing a variety of official NES games	✓
	1.3 Capable of being transported easily	✓
2. Output on modern interfaces		
	2.1 Video output to VGA with upscaling	✓
	2.2 Audio output to the standard 3.5mm	✓
3. Use industry standard VLSI development software		
	3.1 Vivado and ModelSim	✓

Figure(1) project specifications

CPU (Daniel)

The original CPU for the NES was based off of the MOS 6502. A child company called Ricoh developed a modified version of the 6502 for use by Nintendo. The original 6502 has 256 opcodes; however, the Ricoh implementation for Nintendo only had 56, and of those 56, not every one was used.

For this project, design of the CPU was crucial for further development of other key systems. The CPU needed to have nearly perfect timing in order to properly interact the the PPU, or else the system would fail. Therefore, it was key in the design of the CPU that the original Ricoh 6502 design is followed. The block diagram used for the initial planning of the development of the CPU is below in Figure().



Figure(2) Block diagram from Ricoh

The first major implementation issue to address is the timing of the CPU. The original Mos and Ricoh chips ran at 1.79 Mhz, which is not a native clock speed for our FPGA. Therefore, a clock divider must be used. By taking a 100Mhz clock signal and dividing it by 56 (the dividing factor used), a 1.785 Mhz signal can be reached. This speed, while not exactly perfect, allows for a perfect timing implementation of the Ricoh 6502 on the FPGA. Now that the timing of the CPU has been addressed, the logical components of the CPU can be laid out.

A major step in completing the CPU involves defining all necessary inputs and outputs. The major signals coming into the CPU are the Clock, Reset, Ready, NMI, NRES, NIRQ, and Data signals. The Clock signal is exactly as it sounds, a 100Mhz system clock generated by the FPGA is passed into the CPU to be divided by the internal clock divider designed. This can be seen in the Verilog code listed in the Appendix. The reset signal is dictated by a button located on the FPGA. When the button is pressed, a signal is sent into the CPU which resets the state and cycle back to their initial ready states. The Ready signal is passed in to denote when the CPU should begin its logic. For the CPU, the NMI signal, non-masking interrupt, is a passive low signal used by the PPU. While for the PPU, the NMI is an output signal. The PPU will send this signal when it needs to communicate with the CPU. The NRES signal, reset interrupt signal, is an active low signal used similarly to the reset signal. The Data signal is an 8 bit signal that brings in any incoming data the CPU needs. This can include information from the cartridges, and other sources.

The major signals coming out of the CPU are Data, Address, and Read/Write signals. As the most important signal of the CPU, the Data output bus is an 8 bit signal that sends the calculations done by the CPU to all supporting components. Without this signal, the CPU would do nothing. The Address bus signal dictates which registers the data is saved to. It is a 16 bit signal due to all of the registers it must address. Lastly, the Read/Write signal is a flag that denotes what state the CPU is in. It is a 1 bit signal.

Now that all of the input and output signals have been declared and defined for the CPU the bulk of the CPU can be designed. Utilizing the block diagram, the CPU can be broken up into a 7-stage pipeline, an ALU, registers, control logic, wire declarations, and opcode declarations. The wire and opcode declarations need to be discussed first, as they are the most numerous parts.

For the CPU, roughly 500 various wires and opcodes were declared. All registers must have a Q and D wire in order to hold the data, which alone accounts for a large amount. The opcodes used 139 wire declarations. While it was stated above that there are only 56 opcodes, each individual opcode can have a different addressing type. The addressing types are absolute, immediate, zero page, and indirect. Once this is done the registers can be created.

For the Ricoh and MOS 6502, there are three general purpose registers, along with countless other registers for timing. Register A, the accumulator register, is used for overflow detection. This register takes in the a signal from the adder, and outputs to both the Status bus and Data bus if they buses are enabled. The X index register and Y index register define the addressing type for the Status Bus. Outside of these three registers are a

grab bag of other registers used for data holding and timing. By following the datapath of the block diagram in Figure(), all the wires defined d_name are the input wires for a register, and all wires defined q_name are the output wires. Vivado synthesizes these wires into registers automatically, therefore, register modules are not needed to be individually created. Instead, by using logic blocks in the code Vivado will optimize the registers as seen fit.

After the registers are done, the control logic and pipeline needs to be implemented. When the cartridge or PPU sends a certain signal/instruction, the CPU will take that instruction and determine the output signal based off of the control logic. Now that all parts of the CPU have been discussed, the code can be written, synthesized, and tested.

Simulation and testing for the CPU was done via brute force methods. Initially, there was no testing until the entire file had been written. From a top down approach, it seemed easiest to get the entire CPU coded before testing. This caused a few issues due to mislabeled wires; however, in the long run, it allowed for sanity to remain in check. Once the file had no wire issues and was able to be simulated by ModelSim, the initial stages of testing could begin. By simulating input logic of various possible instructions the CPU would receive, it was possible to tell if the control logic was correct. Once manual ModelSim simulations had been done, full scale testing using test ROMs needed to be done. The test ROM simulated every aspect of the CPU, and tested for every possible instruction and opcode. The first trial of this was a success, and allowed for the project to move forward.

APU (Clay)

The NES APU produces the sound for the system through a combination of the two pulse channels, the triangle channel, and the noise channel. Through variable-rate timers and modulators driven by the frame counter, each channel's sound is fed to a mixer and then output. The APU operates at the half the frequency of the CPU clock which is accomplished through a simple counter to divide the CPU clock.

The top-level APU file is called by the CPU and takes the CPU clock, a reset, channel mute, address in, data in, and a read/write control as inputs. In return, the CPU is given the output of the pulse width modulation of the mixed signals, and a data out signal that denotes which channels are active. Signals from the CPU instantiate each of the signals as well as the frame counter which is used to drive the envelope, sweep, and length counters in order to control the overall sound.

The pulse channel uses the envelope generator, a sweep-controlled timer and sequencer, and then the length counter to send its sound to the mixer. The triangle channel takes a timer output created through divisions of the CPU clock, modified by the linear and length counters and sends it to the sequencer before the sound reaches the mixer. This implementation had the triangle channel's timer as a division of the CPU clock instead of the APU clock. The noise channel takes a period input and produces the pseudo-random noise through a look-up table which clocks a shift register.

Each channel is combined in the mixer which combines a division on the sums of the pulse channels and a separate division on the triangle and noise channels and then uses pulse code modulation (PCM) to create analog audio that is output to the 3.5mm jack.

Simulation and testing for the APU took place iteratively as each channel was written, compiled and synthesized. The first channel created was the pulse channel, which is stimulated through writes to 2 registers in memory. To test its functionality, Modelsim was used to view the waveform after forcing the values it reads and then stepping through. After verification, the other channels went through a similar process before the higher level modules were constructed.

With the channels individually functioning, the higher level modules were written to decode CPU instructions and control the overall pitch, length, and variation of the 4 channels' sound. Successful simulation on Modelsim led to implementation on Vivado to verify correctness audibly. Additionally, each channel's 8-bit audio output was attached to

mute switches on the development board to allow for individual channel hearing tests.

To verify overall correctness of the APU, a combination test ROMs and aural tests were utilized. The test ROMs allowed individual channel stimulation but side-by-side comparison with Mesen proved to be the most valuable test because it allowed us to listen to the differences and identify which portions of the signals were not correct. Once the APU passed the aural comparison with Mesen, the section was deemed complete and work transitioned to the controller and mappers.

PPU & VGA (Brandon & Alex)

The NES PPU, the Picture Processing Unit, was a chip developed in-house at Nintendo to take the data from the CPU and use it to display video on an NTSC TV. The NES PPU creates the images through a combination of different tables: nametables for the background and sprites, and the attribute tables. The nametable for the background is responsible for holding the data for the background of a frame and similarly the nametable for the sprites holds the data for the sprites in a frame. The attribute tables holds the position of each sprite and any data associated with the sprite like whether to render the sprite or the background first. The PPU uses a multiplexer to determine which table should be outputted to each pixel and then shifts this value out one bit at a time. The final output of the PPU is 6 bits which represent the colors that should be displayed on the screen.

The PPU interfaces with the CPU through two registers, the PPUADDR and the PPUDATA. Due to this design, the CPU must output all data serially to the PPU, one byte at a time. Furthermore, data is usually only be written during a period known as VBLANK, a period of about 2250 clock cycles. The reason for this is so the data writes don't mess up the picture displayed on the screen. The period called VBLANK happens on pixels that are outside the viewable area of the screen. While it is possible to write to the PPU during the rendering of the frame, most games didn't do this. For the couple of games that did implement writing to the PPU during the rendering of the frame, it was up to the game developers to solve the problem of the messed up pixels during the write.

One problem in designing the PPU was that it is hard to test standalone. In order to verify that all the tables are being updated correctly and that the timing is correct, it needs to be fed correct data from the CPU. Initially, a basic testbench was created that would test a couple inputs to see if all the internal and outputted signals were correct, but this wasn't able to test the full module of the PPU. In order for full testing, the PPU had to be hooked up to the CPU to properly test if the timing was correct and if everything was recreated properly.

Once the CPU was completed, it was hooked up to the PPU to verify that both of those modules were working as expected. We were able to take a NES ROM and use the binary values as the input for the CPU like it would normally expect as a test vector in the testbench. We were then able to compare the output of the PPU to the values of the PPU in what is considered the best emulator, Mesen.

Once we were able to verify that the PPU was working correctly, we then focused on a way to display those outputs on a screen. After reading a lot about the VGA interface, we were

able to create basic hardware that would take 6-bit values and upscale it to 16-bit color before outputting each pixel to the screen. In order to display on monitors, the VGA interface needs 25.175 MHz. The only clock value we could get was 25 MHz; although, the VGA interface tends to be a little lax and our clock did work.

One key difference between the original NES PPU and our implemented version is the clock speed. Our implementation runs at 100 MHz while the original ran at 5.37 MHz. The PPU was intentionally clocked faster to allow for a better resolution of 512 x 480 over the original of 256 x 240. Since reading the value from a register was not destructive, the VGA module was able to read each value twice to double the resolution; this would double each pixel and double each scanline.

Controllers (Alex, Brandon, & Clay)

The controllers for the FPGA NES use the original NES's ports and interface with our system through the PMOD pins. The output of the controller's onboard shift register was initially passed to the FPGA and then executed, however, after many revisions the implementation was changed.

Initially, the controller's data was passed straight into the joypad module which then fed into a register for the CPU to see. This design passed simulation in Xilinx's Vivado but failed upon deployment presumably due to minute issues in timing. The result was that the end of the shift register's value was duplicated to the beginning, meaning that individual buttons interfered with others when pressed. In our case, the A button would be triggered whenever the right button was pressed.

To resolve this issue, as well as account for potential differences in third-party manufacturer's controllers, the onboard ARM processor is handed the controller's data. The value of the controller shift register would be clocked into the ARM processor through GPIO pins. In the C program, each bit would have bitwise logic associated with it to ensure that the shift register was outputting each button in the correct sequence, then the C program would output a byte value with each bit representing the status of a different button. This value was then fed into the register for the CPU to see.

Cartridge and Mappers (Alex)

NES Cartridges are what contained the games that the NES played. In terms of hardware, a cartridge is essentially a large memory bank (usually Read-Only Memory, but this was not always the case) divided into two parts: the Program ROM (PRG-ROM) and the Character ROM (CHR-ROM). PRG-ROM stores data that the CPU will access such as the actual instructions to execute the game as well as data the game will call on during runtime. CHR-ROM stores commonly used graphic elements and is directly accessible by the PPU, enabling this memory bank to hold commonly used backgrounds for quick reference without requiring CPU intervention.

To emulate these memory banks in this project, the cartridge module uses the FPGA's Block RAM (BRAM) modules, which allow for the creation of highly scalable and resizable general purpose memory chunks. When hard coding a game into the system, such as what was done with Super Mario Bros for ease of both testing and demonstration purposes, the values of locations in BRAM were preset to the data as would be found on the Super Mario Bros game cartridge so the system initialized with the game loaded and ready to play. For the microSD game-loading feature, the memory is initialized without a value and thus generally contains all zero data values, which are quickly overwritten with the contents of the microSD that contains the actual game cartridge data.

One major issue with the NES and its cartridges is the amount of available address space. The NES CPU uses 16-bit memory addresses, giving the CPU access to 64KB of memory. The first 32KB of the CPU address space is used for working space RAM for the CPU, PPU register references, and control registers for using the APU and controllers. This leaves only 32KB of memory space left for the PRG-ROM half of the cartridge, which acts as an extension of the CPU's memory by taking over the upper half of the CPU's memory address space. Games like Super Mario Bros, Donkey Kong, and Galaga use 32KB of PRG-ROM or less work fine, but game developers at the time wanted to push the system further and get even more complex games working.

To get around this memory limitation, the NES uses what are called "mappers". A mapper is a way putting a game with more than 32KB of PRG-ROM on a cartridge and then allowing the CPU to interface with that larger memory by switching which 32KB of PRG-ROM is visible by the CPU's addressable memory space throughout the runtime of the game. Mappers were added not by hardware in the NES itself, but by additional hardware located in the cartridges. By instructing the CPU to write to the usually Read-Only Memory of the cartridge, additional ICs on the cartridge such as shift registers could store address prefixes that would be appended to the next memory read the CPU performs on the cartridge,

essentially allowing the CPU to construct addresses larger than 16 bits and therefore work with even more memory than the system originally was intended to.

A similar situation exists with the CHR-ROM of the PPU. The PPU only allows for 8KB of CHR-ROM so games that wanted more would have to use mappers as described above, only this time the faster PPU was accessing the memory being swapped despite the slower CPU issuing the swapping cartridge writes, requiring developers to carefully manage their instruction timing when making games. To solve this, as well as make the CHR-ROM space even more versatile, many mappers opted to replace all 8KB of the CHR-ROM with CHR-RAM, meaning it was capable of holding whatever graphics information was important at that moment during game runtime, rather than a static set of information held constant throughout the game or bank-swappable at the CPU's leasure.

One major issue of implementing these mappers on the FPGA stems from where the mapper hardware was on the original NES: the cartridges themselves. Because this system loads games off of a microSD, the microSD is incapable of containing the same bank-switching hardware that an NES cartridge would. Thus, the implementation of the NES running on the FPGA must contain the hardware for all possible mappers and switch to the correct one at boot by reading additional data located in the header of the iNES ROM files loaded off of the microSD card. From the user's perspective this means nothing noticeable, but within the FPGA is a large amount of mapper hardware that gets disabled every time a game boots because the FPGA contains all possible supported mappers and switches which one is active.

For this NES implementation, 3 major mappers are supported:

- NROM (Mapper #0)
  - Essentially “no mapper” mode, this is the mapper used when a game’s PRG-ROM does not exceed 32KB in size and its CHR-ROM does not exceed 8KB in size nor use a RAM module. With this setup, all of the game’s data fits in the default space the CPU and PPU address spaces can handle and no bank-swapping is required.
- UxROM (Mapper #2)
  - This mapper used a single register that stored a 3-bit address prefix for use on every read from the CPU to the PRG-ROM, allowing for up to 256KB of PRG-ROM data. By writing data to the PRG-ROM, the value in the address prefix register would be overwritten with the 3 least significant bits of the data being written, thereby switching with 32KB bank the CPU could see. This mapper also used CHR-RAM instead of ROM.
- AxROM (Mapper #7)
  - This mapper used a similar system to UxROM, but used the 4th bit of the 8-bit data being written to the address prefix register to change the PPU mirroring settings. Changing this setting on the fly, which is usually hardware-defined by the cartridge connections, gave games that used this mapper to the ability to drastically change their playstyles in between levels or segments of the game without using complex and generally slow programming tricks.

With the mappers this system supports, the FPGA NES is capable of running over 500 officially licensed games.

Potential Improvements (Clay)

Future iterations of this project could include:

- More mappers to allow full support of the official NES game library
- Output to the HDMI video and audio standard
- Better resolution with a frame buffer
- Addition of on-system volume control
- More integration with the development board to allow selection of ROMs from memory
    - This would require ARM code to allow for selection of which NES ROM to load on boot.
- Video color palette options - to have the original color or more bright color
- Custom PCBs that do not rely on development boards for the peripherals
    - Would require rework of the SystemVerilog code as well as intensive soldering for the FPGA

Conclusion (Daniel)

The final design implemented is a nearly identical recreation of the original NES. The CPU, PPU, and APU all exactly portray the original hardware used on the NES, even down to the glitches in the original system. Our implementation has virtually the same clock speed, same controller input, and same game capabilities as the original NES. The only differences are that our system outputs on modern output signals, and our system comes in a 3D printed case.

In terms of design specifications, we met, if not exceeded, all expectations set for ourselves. We have the functionality to play over 500 NES games through the swappable SD cards. Our system outputs video over VGA, upscaling from 256x240 pixels to 512x480 pixels. We have audio out over a 3.5mm headphone jack, which can output sound to any modern speakers or headphones. All of these specifications were achieved through the design process using Vivado and ModelSim, which checks off the last of our design specifications.

Acknowledgements (All)

The FPGA NES team would like to acknowledge Dr. Stine for being all-knowledgeable and a deity to us throughout the project conception and design process.

Additionally the team would like to acknowledge Nintendo for creating a system and allowing educational recreations of the hardware.

References (All)

Nesdev (wiki.nesdev.com)
Digilent Forums (forum.digilentinc.com)
Xilinx Forums (forums.xilinx.com)

Appendix I - Code

For code please refer to the Github repository located at https://github.com/BW0ng/FPGA-NES.

Appendix II - Pictures