Team Wet Dogs

NES Project

Original Goal

We set out to create a fully working NES completely in hardware without any software support, including a cartridge reader.

Accomplished Goal

A fully working NES with minimal sound support and games flashed into Block Rams on the FPGA board.

Team Members

Jack Cheung, a Senior ECE major, had taken computer architecture and a verilog design class.

Mark Ma, also a Senior ECE major, had taken computer architecture and a digital computation class in verilog.

Steve Thompson, a third Senior ECE major, had taken computer architecture and computer graphics.

Ryan Walsh, the fourth ECE major, had also taken computer architecture and computer graphics.

Our team had a strong background in verilog and hardware design, so we appropriately opted for an entirely hardware project to represent our strengths.

Hardware Description

Hardware Overview

CPU: As the main processing unit of the NES this runs the code located in the cartridge memory and controls the other hardware devices through memory mapped registers.

PPU: This writes color information to the frame-buffer based off values written to its control registers from the CPU.

APU: This controls the interface with the Audio Codec on the FPGA board. It also generates and controls 5 source channels based off values written to its control registers from the CPU.

DMA/Mapping Unit: This unit takes care of all the interfacing between the CPU, PPU, and APU. It also has the code for DMA routines between CPU memory and the APU or PPU. The memory mapper routines meant to emulate the original NES cartridge were also in this module.

Controllers: On reads from the controller mapped address, this module collects each pulse from the controller and returns the appropriate value to the Mapping Unit.

VGA Driver: Writes pixel information to the VGA pins on the board. It also selects the correct values from the frame buffer to get the correct NES resolution to output to the screen.

ROM: Different ROM modules were used to hold CPU code and PPU Pattern values, in place of a cartridge.

RAM: Different RAM modules were used for the CPU and PPU private memory spaces.

6502 CPU

Overview: The 6502 is an 8-bit, non load-store architecture with 16-bit width addresses. There are 154 documented op-codes and 13 different addressing modes. There are variable length instructions, with up to 3 bytes used in a given op-code. Only a minimal amount of pipelining is done. The NES version of the 6502 permanently has Decimal Mode disabled.

Registers: There are 6 programmer visible registers: X, Y, Accumulator, Stack Pointer, Conditions, and PC. Most operations are done using Accumulator, while X and Y are used primarily for indexing.

Cycle Counts: The expected cycle count is extremely important in the NES because it is used as a mode of timing and must be synchronized with the PPU for accurate timing. Any Instruction involving addition with regards to memory addresses can result in a page cross which in most instructions causes an additional cycle to be taken. A Page Cross occurs when there is a carry out bit from the lower 8 bytes of the address plus the other operand.

Interrupts: There are three types of interrupts supported by the 6502. Interrupts are handled by acknowledging the interrupt, saving the Condition and PC registers onto the stack, and jumping to the corresponding vector table address. Each interrupt has two bytes in memory between FFFA and FFFF which contain the address of the interrupt handler. The first interrupt is a Reset Interrupt, which only occurs as a result of the NES being powered on or the reset button being pressed by the user. The second is an IRQ, which can be generated by either by software via a BRK instruction or externally, such as through cartridge mappers or the APU. This interrupt can be disabled via the SEI instruction, and then re-enabled using CLI. While Interrupts are disabled, there can be one pending IRQ upon calling CLI, but only one. The last interrupt is an NMI. This Interrupt is specifically called by the PPU finishing a frame when NMI generation is enabled. This interrupt cannot be masked.

Stall: The CPU can also be stalled at any given point of execution by a signal being asserted externally. This is used for the DMA routines.

Memory Space: The memory space is broken down into three parts: Cartridge ROMs, private RAM, and Memory-Mapped IO. The RAM is further broken down into general purpose, stack, and Zero-Page. Mirroring occurs across most of the spaces by simply

ignoring some of the bits. For example, if it notices the top three bits as 001, it just uses the lower three bits to determine which memory mapped register to use for the PPU.

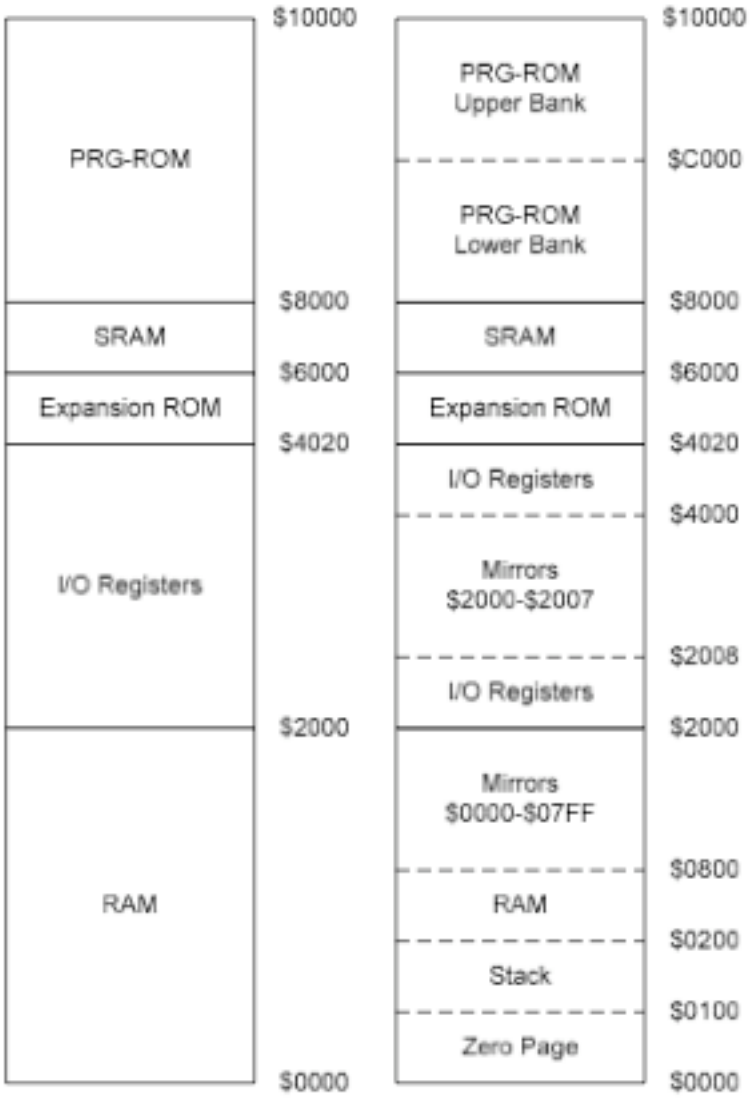
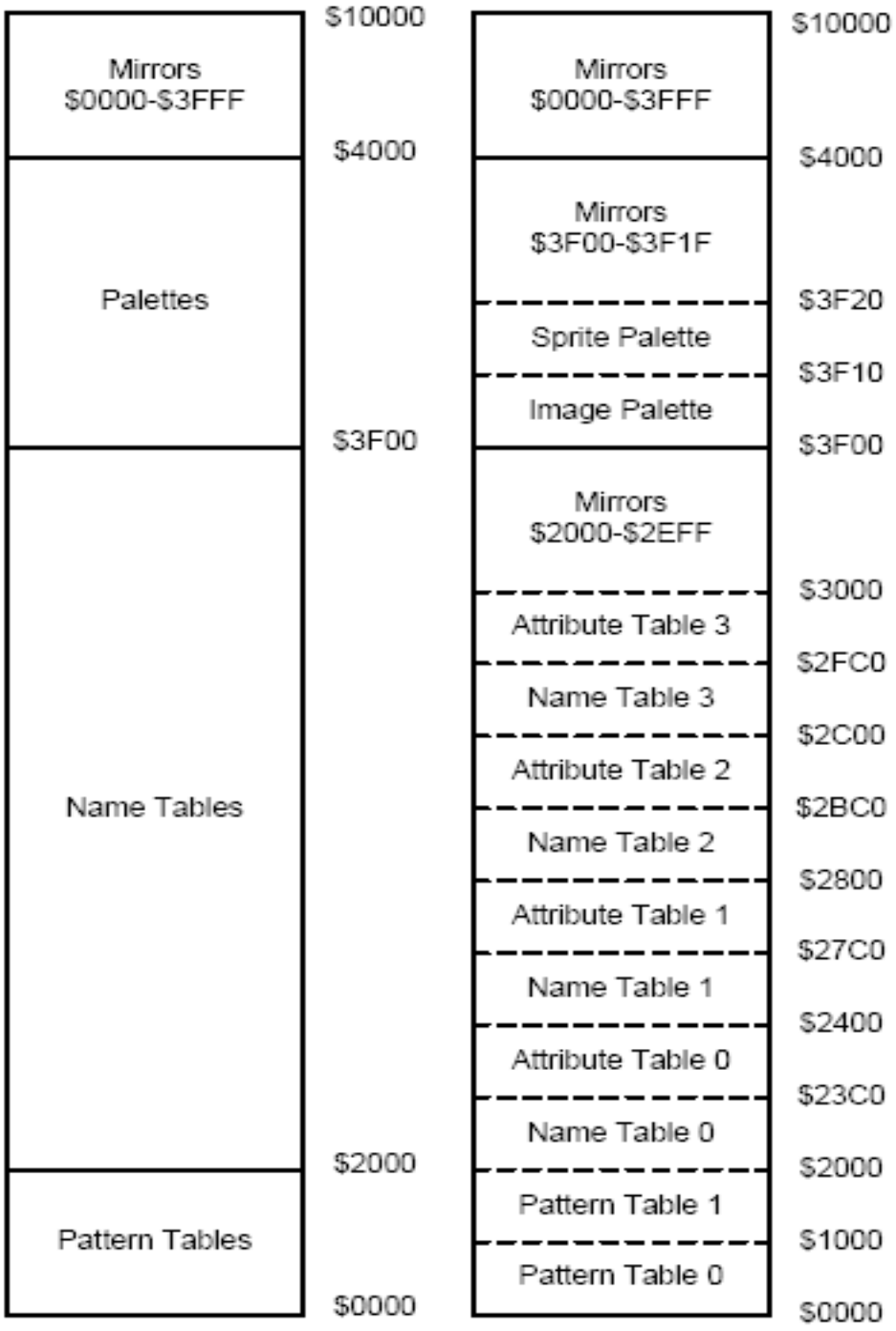


Figure 2-3. CPU memory map.

Picture Processing Unit (PPU)

PPU Memory Layout:

The NES's graphics unit is referred to as a "Picture Processing Unit" that is powered by another one of Ricoh's chip, the 2C02. Like the CPU, the PPU can address up to 64KB but only has 16KB of physical RAM. However, our implementation only used 2KB since much of the space is used for mirroring. The mapping of the PPU's memory space is laid out below.



The PPU uses 8 registers to communicate with the CPU. These registers are located in the $2000-$2007 address space of the PPU. The functions of these registers will be laid out below.

$2000: PPU Control Register

Bit 7: Enable NMI on VBlank (0=Disabled, 1=Enabled)
Bit 6: Not Used for NES
Bit 5: Sprite Size (0=8x8, 1=8x16)
Bit 4: Background Pattern Table Select (0=$0000, 1=$1000)
Bit 3: 8x8 Sprite Pattern Table Select (0=$0000, 1=$1000)
Bit 2: VRAM Address Increment Amount (0 =Increment by 1,
1 =Increment by 32)
Bits [1:0]: Base Name Table Address (0 - 3 = $2000, $2400, $2800, $2C00)

$2001: PPU Control Register

Bits [7:5]: Color Emphasis
Bit 4: Sprite Enable (0 =Not Displayed, 1 =Displayed)
Bit 3: Background Enable (0 =Not Displayed, 1 =Displayed)
Bit 2: Sprite Clipping (0=Hide in leftmost 8-pixel column, 1=No Clipping)
Bit 1: Background Clipping (0 =Hide in leftmost 8-pixel column, 1=No Clipping)
Bit 0: Monochrome Color

$2002: PPU Status Register

Bit 7: VBlank (0=Not VBlank 1=VBlank)
Bit 6: Sprite 0 Hit Flag (0=No Collision, 1=Sprite0 /Background Collision)
Bit 5: Sprite Overflow Flag (0 = 8>=sprites, 1= 8 < sprites, on a scanline)
Bits [4:0]: Not Used

$2003: SPR-RAM Address Register

Bits [7:0]: Address to access in SPR-RAM

$2004: SPR-RAM Data Register

Bits [7:0]: Data to write to SPR-RAM at address specified by $2003

$2005: PPU Background Scrolling Offset

1st Write: Bits [7:0]: Horizontal scroll index (X value 0 - 255)
2nd Write: Bits [7:0]: Vertical scroll index (Y value 0 - 239)

$2006: VRAM Address Register

1st Write: Upper 8bits of address into VRAM

2nd Write: Lower 8bits of address into VRAM

$2007: VRAM Data Register

Bits [7:0]: Data to write to / Data read from VRAM at address specified by $2006

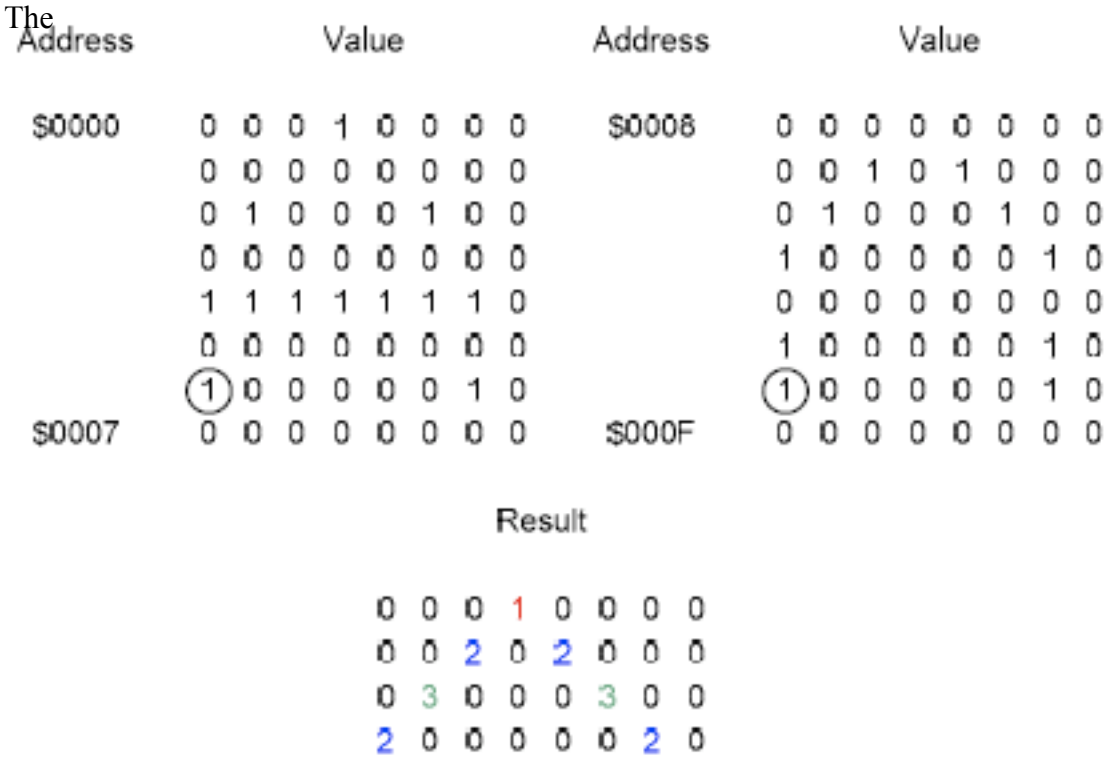
There is also a separate area of 256 Bytes for the sprite memory. While this can loaded through $2003/$2004, however it is mostly done through Direct Memory Access through the CPU's address $4014. One write to this address can load the entire sprite ram, instead of having to do multiple writes to $2003/$2004.

PPU Color Palette:

There are two different color palettes in the NES, one for images and one for sprites. These both have sixteen entries, thirteen of which are unique. There are located at $3F00-$3F0F (image palette) and $3F10-$3F1F (sprite palette) and $3F04, $3F08, $3F0C, $3F10, $3F14, $3F18 and $3F1C are just copies of $3F00. The values in these are not actually the color values, but rather indices into the system color palette, which contains 64 entries and 52 unique colors. At any one point a max 25 of these colors can be displayed n the screen.

PPU Pattern Tables:

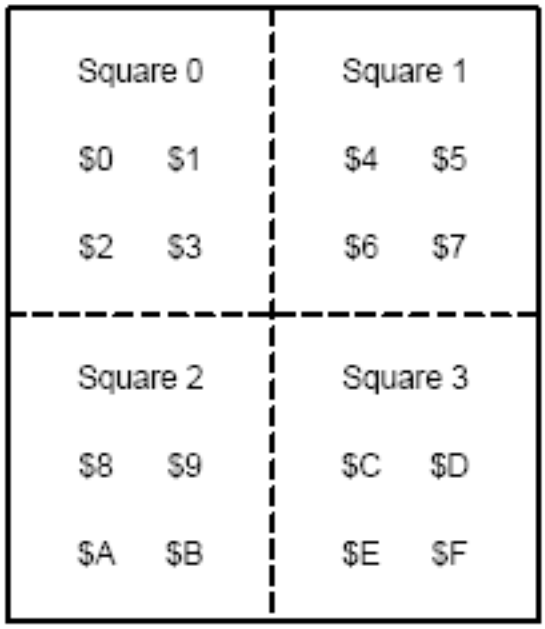
The pattern tables of the PPU are defined essentially which pixels are "on" and which ones are "off". More clearly, they define the shape of the objects that will be drawn. It can be more easily shown with a picture.



The pattern tables form 8x8 pixel tiles as the “A” that is shown above. These two bits form the lower two bits of an address into the palette table. They are used in conjunction with bits from the attribute table in order to form a 4-bit address into the palette table.

Name Tables & Attribute Tables:

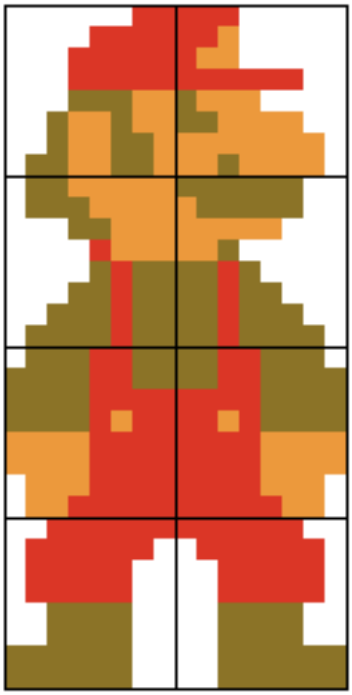
The name tables in the PPU are 32x30 tiles each of which are 8x8 pixels which makes for a name table to be 256x240 pixels, the resolution of the NES. Each of these name tables have an associated attribute table. Each byte in the attribute table represents a 4x4 group of tiles, so an attribute table is an 8x8 table of these groups. Each 4x4 group is further divided into four 2x2 squares. The 8x8 tiles are numbered $0-$F. The layout of the byte 20 is 33221100 where every two bits specifies the most significant two color bits for the specified square.



The name tables allow us to figure out which pattern table we should be using for a given tile.

Sprites:

Sprites are either going to be 8x8 pixels or 8x16 pixels. Most images are composed of multiple sprites as shown in the following break down of the infamous Mario.



There are a maximum of 64 sprites all of which are composed of 4 bytes, which are laid out as follows.

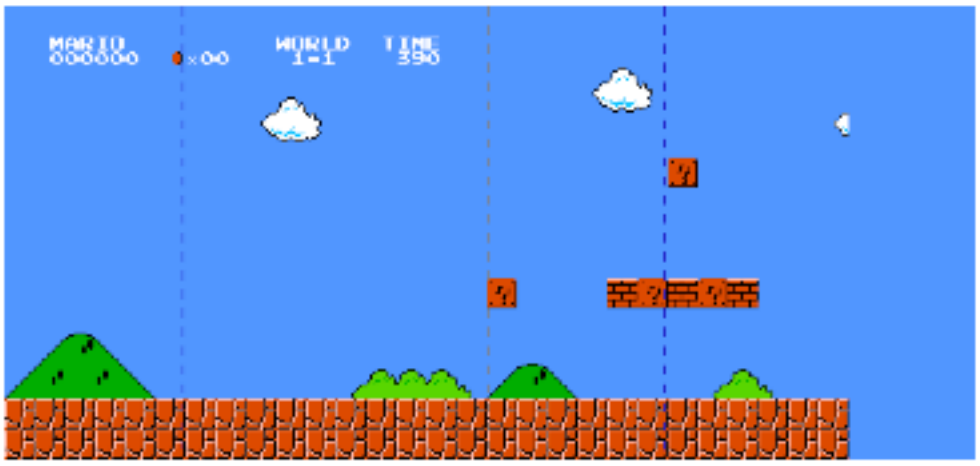
• Byte 0 - Stores the y-coordinate of the top left of the sprite minus 1.

• Byte 1 - Index number of the sprite in the pattern tables.
• Byte 2 - Stores the attributes of the sprite.
• Bits 0-1 - Most significant two bits of the colour.
• Bit 5 - Indicates whether this sprite has priority over the background.
• Bit 6 - Indicates whether to flip the sprite horizontally.
• Bit 7 - Indicates whether to flip the sprite vertically.

The sprites have a priority of whether they should be displayed or not. They are prioritized from 0-63 with 0 being the highest priority and 63 being the lowest. Due to hardware limitations only 8 sprites are allowed to be on any scanline, if there were it would cause the undesirable effect of flickering.

Scrolling:

Sprites are also used in scrolling, specifically sprite 0. Games will use the location of sprite 0 to see if it hit a non-transparent background pixel in order to start switching between name tables. This is done in two fashions, horizontal or vertical. Horizontal would be what you see in a game like the original Mario where are you run to the right the screen "scrolls" and advances to the next one. This is done through the usage of the different name tables.



APU

Xilinx Audio Codec

The Xilinx audio codec is responsible for generating all sound. It utilizes the Intel AC97 audio codec. It has 5 relevant signals which are all serial ports.

s_data_out - Signal which accepts the data to be played as well as general settings such as sampling rate, output volume and recording volume.

s_data_in - Signal which is outputted from the audio codec, containing status information regarding the information received and whether it is valid.

bit_clock - Signal that is the clock the audio codec runs at.

synch – Signal indicates the start of a valid s_data_out data frame. It lasts for 16 cycles always.

audio_reset_b – Signal that resets the AC97 audio codec. It is crucial that this signal be pulled low for 1 us and then left high in order to reset the codec. If not, it will never begin running.

The audio codec also accepts a very strict format for all its inputs. The s_data_out signal is broken into frames, with each frame containing thirteen time slots. The first slot is

called the tag slot, while the following twelve are called data slots. The tag slot uses 16 bits and the data slots use 20 bits, resulting in 256 bits per frame. Each bit either has a specific meaning to it or must be cleared to 0 in order for the audio codec to work correctly. The specifications of the 256 bits are listed in http://cache.national.com/ds/LM/LM4550.pdf.

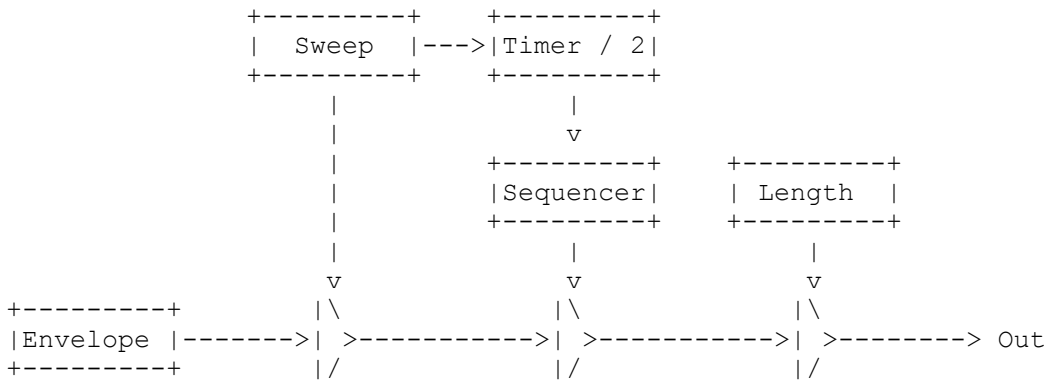
NES APU

The NES audio procession unit is comprised of 5 separate sound channels. There are two square wave channels, a triangle wave channel, a noise channel, and a delta modulation channel. In general, the two square wave channels are used primarily to play the main melodies of NES games, as well as sound effects such as beeps, lasers, and Mario jumping. The triangle wave is typically an accompaniment for the square waves, as it primarily plays scales. The noise channel creates a static-sounding noise. The delta modulation channel seems to be used to generate bass tones for the background.

Frame Sequencer

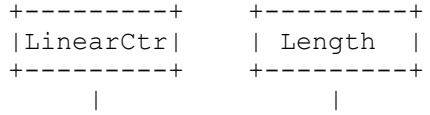
All APU modules are clocked by the frame sequencer. The frame sequencer can generate different clock frequencies between 240 Hz and 60 Hz to drive each separate APU module. It essentially takes the CPU clock, divides it, and outputs clocks accordingly to different modules.

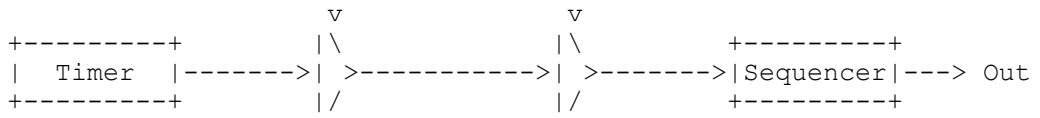
Square Channel



The square wave is created using an envelope generator, which is essentially a counter. In order for the envelope generator's value to reach the output, each one of the 3 gates must receive a non-zero input. Otherwise, the output is zero.

Triangle Channel



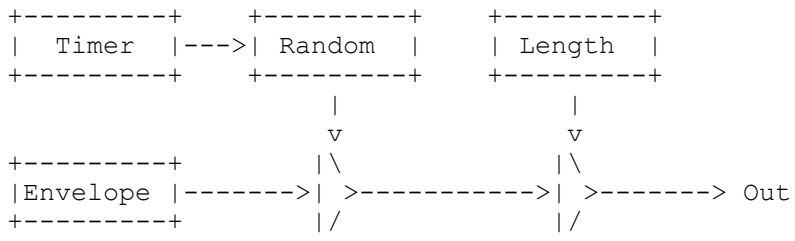


The triangle channel has a timer, which is clocked by the frame sequencer. On a given period, it outputs a signal. If both the linear counter and the length counter are not zero, the signal is passed through to the sequencer. The sequencer goes through the following sequence:

```
F E D C B A 9 8 7 6 5 4 3 2 1 0 0 1 2 3 4 5 6 7 8 9 A B C D E F
```

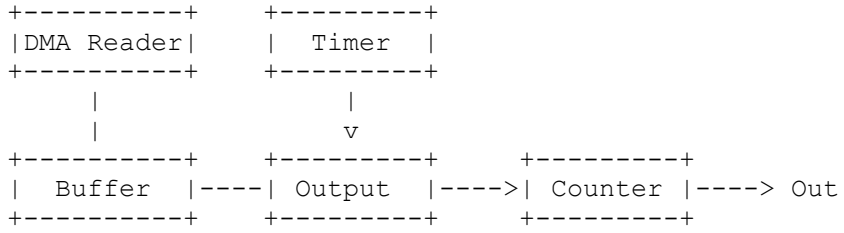
It will output this sequence of values, changing each time it is gets a signal.

Noise Channel



The noise channel follows the same pattern as the square wave, as it is created with an envelope generator. The only different is that there is a random generator mode that is also set, which creates the static-sounding effect.

Delta Modulation Channel



The DMC is the most complex out of all the different channels. It requires direct memory accesses from the CPU to fill the buffer. It then takes the data in the buffer, shifts it's values out bit by bit, and increments or decrements the counter accordingly based on the shifted bit. Whenever the DMC requires a memory access, it sends a signal to the CPU, stalling the CPU for a set number of cycles.

APU Output

```
output = square_out + tnd_out


                      95.88
square_out = -----------------------
                    8128
```

$$\frac{95.88}{\frac{8128}{\text{square1} + \text{square2}} + 100}$$

$$\text{tnd\_out} = \frac{159.79}{\frac{1}{\frac{\text{triangle}}{8227} + \frac{\text{noise}}{12241} + \frac{\text{dmc}}{22638}} + 100}$$

The APU's output is determined using the above equation. The value gained is between 0.0 and 1.0, and is then scaled against 2^18, producing an 18 bit value to feed into the AC97 audio codec. To simulate this equation, a lookup table was used which holds every possible output, mapped to the equivalent 18 bit result.

Mapping / DMA Unit

CPU/PPU Registers: The PPU ran at a clock speed 4 times greater than the CPU. To make sure that needless multiple writes or reads weren't created, an enable signal would be asserted when the address matched a PPU mapped memory address and for only one PPU cycle. NMI requests were also generated based on PPU output and sent to the CPU until they were acknowledged.

CPU/APU Registers: The CPU and APU are on the same clock cycles, so no special consideration was needed other than detecting when the address was a valid APU mapped address.

Sprite DMA: A sprite DMA routine is initiated on a write to $4016. The value written to that address is used as the upper bits of the CPU address from which to copy data over to the Sprite Ram. Every DMA copies the whole 256 byte page over to the Sprite Ram that is used by the PPU.

APU DMA: An APU DMA gets requested by the APU, always samples one byte at a time, and takes a total of four cycles to complete.

Cartridge Mappers: Our unit has to take into consideration memory mapping techniques used by various cartridges. We provided support for four different mapping styles which all work on the same basic principle: On a write to a given ROM address, take the value that is being written and either swap the lower page of the CPU ROM, or both Pattern ROM pages. The mapping styles we supported were: no mapping, only pattern ROM swaps, only CPU ROM page swaps, or both pattern and CPU ROM swaps. Beyond this, they would have required a real amount of effort to implement, such as creating an IRQ timer.

NES Controllers

The controllers for the NES are actually pretty simple once you read the documentation for it. The NES console simply polls the controller 60 times a second and records the state of the controller and whenever the game program requests to read the controller data it shifts out one bit of the controller state each time.

To save time and troubleshooting, I took the one of the previous year's NES FSM module since it had already worked, I just had to adapt it to our NES. To connect the

controller to the FPGA, I initially took apart one controller and soldered new wires onto the NES controller board and connected them onto a breadboard that was connected to the FPGA through the high speed expansion port. Later I learned I could just thread the wires into the original controllers pin slots and into the low speed expansion ports as well.

The controller in more detail works as follows:

1) The hardware polls the controller first by initiating a LATCH signal

2) After LATCH stays on for a period of time, the controller outputs the state of the 'A' button (0 for pressed 1 for unpressed).

3) The hardware begins to strobe the controller 7 more times for the rest of the buttons, recording each button state after each strobe. The order of the buttons after 'A' goes 'B START SELECT UP DOWN LEFT RIGHT'.

4) The program reads this controller state by first writing a '1' to the $4016/7 register and then a '0'. This initiates the register for reading.

5) Each read at $4016/7 returns the state of one button, in the same order as previously stated above, A B START SELECT UP DOWN LEFT RIGHT.

This worked out pretty well, as long as the controllers were polled a lot faster than the whatever the 6502 was clocked at, you would obtain pretty accurate controller information.

Design Approach

Task Assignments

CPU: Mark and Steve started designing the CPU very early on in the project. Most of the design process involved finding documentation, and then deciding which of the

conflicting documents we found to use. Once that was done, Mark got a 6502 simulator and compiler up and running and started writing tests while Steve wrote the verilog. Mark then finished off the testing in simulation before trying to port it to the board.

PPU: Ryan started working on the PPU and pouring through all the different fan-written documents and designed a nice overall framework for how the PPU would run. When the CPU design phase was done, Steve started to help with the PPU design as well.

DMA / Mapping Unit: This was done originally by Steve in simulation with the CPU, and was added onto by Ryan and Steve as the integration began.

VGA Code: Jack took the code used by Xilinx in one of the provided demos and modified it to suit our purposes. As our first foray into the hardware side of the board, this was a considerable task that helped us with our later parts. It also allowed for much easier and more intensive testing for the PPU.

Controllers: Mark worked with interfacing the controllers with the FPGA board and getting them working in actual hardware, as well as writing the appropriate logic to forward the correct data back to the Mapping Unit.

APU: Jack wrote the APU module and figured out how to interface with the Audio Codec on the board.

Testing

A large part of the formal verification was done by Mark, especially with regards to the CPU. All of the CPU code could be tested in simulation, and was thus easy to have fully working before we put it on the board. The PPU presented a different problem of processing so much data and continuously outputting lots of data that making a simulation would have required a considerable amount of work, not something we were sure that we had.

When we were not testing in simulation, we were testing code that was actually placed on the FPGA board and running real NES games, specifically Tetris. We downloaded NesTen and FCEU NES emulators. FCEU provided useful debug output of the CPU state and a nice feature of logging code that was getting run. NesTen was particularly useful when we got to the sound part, as it would allow us to isolate source waves to hear what they sounded like.

For this project we knew we had to verify the NES CPU core and make sure before we put it on the board that it was functional through simulation and to get rid of any bugs and smooth out any little details before synthesizing it. At first when we started writing out the 6502 the testing was pretty simple as we hardcoded the opcodes and instructions into our memory module by hand. But later the instructions and address modes got too complicated to figure out by hand to write more thorough test cases. So I

had to search for a 6502 compiler and assembler. The tricky part was trying to find one that wasn't a few decades old and only ms-dos and windows since all our simulation was being done on the ECE linux machines. I eventually found one, called P65, http://hkn.eecs.berkeley.edu/~mcmartin/P65/, that worked on the ECE machines. With this in installed, I wrote up two scripts on the machines, one that would take our assembly code and use the assembler to generate the binary files for our CPU core, and another to loaded up the binary file into our CPU core and ran the simulation.

Since this was our own project and their wasn't a golden 6502 module we could simulate against, I found it rather tricky to write another script to compare the output of our 6502. Besides, I had to check to make sure our 6502 was inline with the original specs, timing, cycle counts, etc…, so I did this part all by hand. To truly verify the 6502, I used two different 6502 emulators I found on the web; one was a online javascript based 6502 emulator, http://e-tradition.net/bytes/6502/, and a windows based one I could run in the windows clusters in the ECE lab, called 6502SIM. Both were tremendously useful in debugging the 6502 for various reasons, not just in terms of correctness. If there was a strange error in our version, I could step by step with the same test in both simulators and pinpoint the exact error and correct it. The simulators were also helpful in debugging the various addressing modes and page crosses that the 6502 might do as well as well as some vague instructions the 6502 had, such as Subtract-Without-Carry and the Bit instructions. Writing the tests took a bit of time too since there were a lot of corner cases I had to ensure worked, such as page crossing and status flags, which took a lot more consideration and time. All in All, it took about three weeks in the cluster of just fully testing and verifying the CPU because Steve was still writing as I was testing and I was fixing while I was testing.

Debugging things once they were on the board turned out not to be too bad. We found out most of the errors from observing our variables through ChipScope and comparing them with what we knew about how Tetris code ran. The same process was often used to figure out the proper way to interface the CPU and PPU, as documentation was sometimes unclear or incorrect.

After the NES reached a certain level of completion, the easiest method for testing simply became to run games and verify that it looked/played correctly. We were happy to oblige.

Current Status

As a team, we are extremely satisfied with what we got accomplished. While it would have been nice to have a fully working APU, or a ROM interface, we believe we did very well with the time constraints we had. Currently the APU only can correctly play one source generator at a time. They all work correctly independently, but are getting combined in such a way that one takes priority over another and is all you hear, unless it gets disabled, in which case you hear the secondary channel. All games using the first four memory mapping styles are currently supported, just about the best we could do with the size constraints of the FPGA Block RAM's without the use of some sort of Cartridge reader. Other than those two things, we discovered no errors in our code, even after running some of the more ridiculous tests designed for software emulators.

Reflections

We really had two regrets in terms of the way we went about designing and testing our project. The first was that we didn't come up with a full simulation model for our code, something that would at the very least dump pictures periodically that we could look at. The wait time between sometimes trivial changes could be almost an hour, seriously hampering our productivity. If we had a better simulation model, we could have at least avoided synthesizing and waiting on minor changes to our code.

The second regret is that we didn't spend more time at the beginning figuring out the way board worked, particularly the VGA and Audio Codec. If these had been figured out much sooner, the PPU could have undergone testing sooner, and the APU would probably have been finished. More time spent playing on the low level side of the board would be extremely worthwhile in future versions of the class.

Individual Reports

Ryan Walsh:

Emulating the NES was one of the most time consuming and fun projects I have worked on during my time at CMU. I was tasked at the beginning of developing the PPU. During the first half of the semester I went through a plethora of documents that attempted to describe how the PPU worked. The problem with this was that the people who were writing the documents weren't always clear in the way they explained things and furthermore a lot of the documents contradicted each other. This was a sore point in designing the PPU our team could not have possibly reverse engineered it and emulated it in one semester. But, picking and choosing what seemed to make sense I created a semi-functional PPU that handled the backgrounds; sprites had not yet been implemented since I figured it was far better to get backgrounds working and then add in the sprite code. I did some minimal testing on the PPU because extensive testing in simulation would have required writing a far more complicated VGA module and wasn't worth the time. It then came time to hook the PPU up to the CPU and see what happens. This is where Steve and I starting making mistakes. The first time we synthesized our hopes for a fully working interaction between the CPU and PPU obviously did not come to fruition. Instead of going back to simulation to test minor things that we thought might be wrong, we continued testing using synthesis. This was a bad idea because synthesis took a long time due to the fact we had to load the ROM files into block ram. This made us cautious of what changes we made; however, it created a lot of down time for us waiting for the synthesis to complete. Eventually, after feeling like I had wasted too much time on synthesizing I moved back to simulation for a little while to correct a find and correct a few errors we had noticed during synthesis. Eventually we got backgrounds working after modifying the code we had originally started with and many hours in the lab. After the backgrounds were mostly working (still had a few glitches here and there, but those were mostly related to us not having the right ratio of clocks) it was debugging through chipscope to find errors where the code from our modules matched those that were output by an emulator. We used FCEUX extensively since it allowed us to not only see the code the CPU should be producing but also the layout of the name tables, the pattern tables, and the palettes. I attempted to add the sprite code into our project but I was thinking about it the wrong way and Steve ended up rewriting it much more elegantly (although

still pretty ugly). The rest of the time was spent tracing through code trying to find errors in the CPU or PPU, both of which we found more than we would've liked to.

Besides working on the PPU and debugging I was also in charge of writing scripts to dump ROM files to a useful format to be loaded into the block ram. This required using the hex editor "Hex Workshop" and a perl script I wrote.

I think that this project taught me quite a lot about time management and working in a team. I knew Mark and Steve prior to working on the project and was an acquaintance with Jack. This allowed for a much better work environment than if we had not known each other. This is because we were aware of each other's strengths and weaknesses, which allowed us to divide up the work efficiently. Overall it was a great experience working on emulating the NES. Our goals were not quite what we set out to do, but we achieved a great result and I am very proud to have worked on it.

Mark Ma:

The class was pretty fun, learning about the NES and all the hardware by itself was pretty interesting. Although there were lots of places I might have gotten stuck at, Steve would somehow always brainstorm an idea through and get us going again on the CPU. Verifying and Testing the CPU was a pretty draining task which was it took three weeks, since it was being written at the same time by Steve. I was also fixing the current CPU I was testing at the same time and uploading these changes to Steve, which then I had to test any new features he had finished writing, and thus repeating the task of verification. Luckily enough this paid off since during synthesizing the CPU basically had very few errors, around two to four that were easily caught and fixed though. I spent the rest of the time in the lab doing the controllers and helping any other team members with their part of the project as well. I would say though the worst part about this class was the lack of the TA since it sucked having noone to teach us how to use the Platform Studio software and its quirks, especially the negedge reset for the hardware, that took a day at least for figure out for everyone. I was the reverse of everyone in my group, since when we started writing the CPU I spent up to maybe 40 hours a week in the lab debugging and testing, but afterwards I didn't have to since I didn't have much to fix when synthesizing. I spent more time later doing the controllers and later just testing out the NES games. This project really required a good dynamic team that could work with each others skills and for something like this you cant have any one person lacking in verilog skills at all, but luckily everyone in our group was pretty proficient in it. Plus I think we actually understood the NES hardware.

Jack Cheung:

I have very mixed feelings regarding this class. The project was itself very rewarding and interesting to work on. However the interface with Xilinx was one of the most frustrating experiences of my life. Coupled with that was the lack of a TA knowledgeable with Xilinx and the FPGA who could inform us of simple quirks and mistakes that instead took us days to debug.

Interestingly enough, I worked almost exclusively with figuring out the format of the FPGA, as I worked with both the VGA frame buffer and the AC97 audio codec. Both took an unnecessarily large amount of time to understand, as there were so many minor specifications that needed to be exact or else the entire thing would fail. In fact, I would go so far as to say more time was spent attempting to get the Xilinx board to work than actually coding and designing.

I also worked on creating the APU. This part of the NES was probably the most alien to all of us, as none of our group members are particularly informed when it comes to signals and sound. As a result, we made several very unintelligent decisions regarding implementation that were a direct result of our lack of knowledge in the field. It also made it difficult to debug errors in the waves, as we did not understand the correlation between the real sound and the wave. This ended up wasting a lot of time.

Overall, this project was probably the most time consuming thing I have ever done in my college experience. In the first month, I spent about 20 hours a week working. For the second month, I was easily in lab 40 hours a week. For the last month or so, I was in the neighborhood of 50-60 hours. This of course can be attributed to the extremely long synthesis times of our entire project, as well as the lack of Xilinx knowledge which resulted in wasted debugging time.

In my opinion, the class would be fine given a knowledgeable TA staff that could answer simple questions and provide us with useful advice. For example, wasting several days on debugging our BRAM module, only to find out Xilinx does not support BRAM written in Verilog. Without a TA, the course is more discovering the quirks of Xilinx than designing a game.

Steve Thompson

First off, I want to say that I'm extremely proud of the way our team worked tirelessly on this project and how we supported each other so well. Every person on our team took at least one individual project and ran it to completion and verification. I'm even willing to go so far as saying every person in our group was in that lab more than any person in any other group. Even though it was a lot of work, I feel like overall we had a lot of fun doing something that turned out so well.

I fairly quickly started going after writing the CPU code in verilog. I broke the instructions down into groups and started completing groups at a time. For the first few sets of opcodes, I spent some time thinking about timing and size optimizations. I soon realized, however, that we were trying to meet a 1.8 MHz clock speed and had a whole FPGA to put our NES onto, so I just started writing code that was logically correct without bogging myself down too much with optimizations. This proved to work pretty

well, as the CPU was written a quarter of the way through the semester, and verified and synthesized not long after. Mark was a great resource for testing and only missed three errors in my code.

After the CPU was written, I spent time working with Xilinx tools trying to get the correct clock speeds that I needed for the NES, and trying to figure out how to synthesize initialized Block RAM. This proved to be an impossible task in verilog, but fortunately I had picked up VHDL over the summer and it recognized reading data in from a file in VHDL. I feel like I wasted too much time in this section, and should have been helping more with the PPU and CPU interface.

Once this was done, I started reading up on the PPU and soon helped Walsh with the verilog for the PPU. After flailing around for a while trying to come up with a decent validation method in simulation, we decided to just try things on the board with the FPGA. We caught a lot of bugs before we could actually see anything on the Monitor using Chipscope, but in the end this turned out pretty well, and soon it became a matter of identifying on error based on the visual output to the FPGA, and trying to fix it and repeating the process. I'd say I stopped really learning anything about the project a month before it was done, and it became more busy work.

Overall, I had a great experience and learned a lot about delegation and relying on other people, something I've had a problem with in the past. I'm not sure I could do this for another semester though. Some of my other classes didn't get quite the attention they deserved.