Team AwesomeNES Final Report

Contents

Project Website:
`http://www.andrew.cmu.edu/user/rsinnott/AwesomeNES/`

Project Description

This class asked us to implement a video game system on a Xilinx Virtex 2 Pro FPGA prototyping board. The requirements for the design were as follows: must have video display, must have sound effects, must take in user input from an external device, must support multiple concurrent players, must have a scoring mechanism, and most importantly, must be fun. Other than that, the direction and design of the project were up to us.

We decided to implement a version of the original home gaming system, the Nintendo Entertainment System (NES). Though challenging, the system would definitely meet the requirements. Because so many games already existed for it, we would be able to devote all our time to the hardware rather than trying to create both a hardware system and impressive demo software. Finally, the NES has a very closed specification- either it works as the original did, or it doesn't work- and is far better documented than any other gaming console.

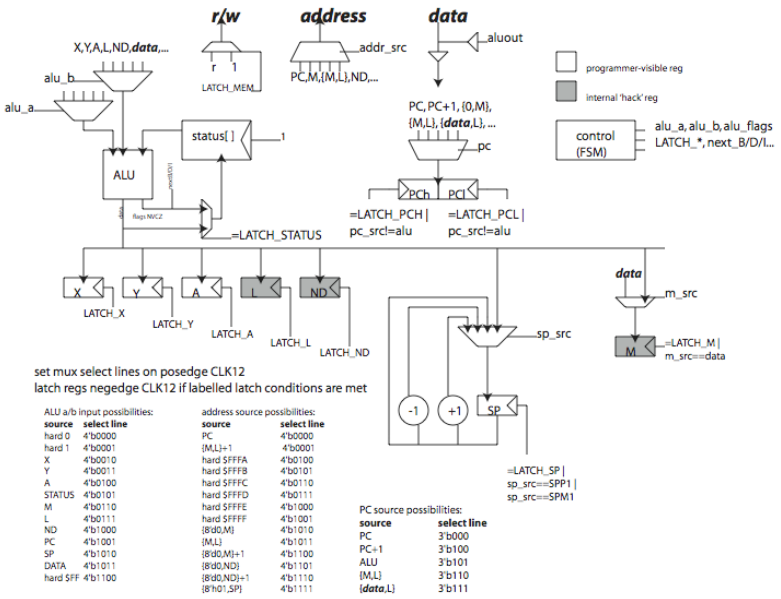
The following sections contain details on all the parts of our design, an overview of our methodology, what we learned from the attempt, and individual comments by all group members.

The NES 2A03:

Part 1a: The NMOS 6502 *(spkelly)*

The NMOS 6502 is a relatively simple 8-bit processor. It has a total of 56 instructions spanning the loading/storing any of its 3 general-purpose registers, basic control flow, basic stack management, and roughly 14 arithmetic/logical operations. It is run on a master clock at just over 21Mhz, but divides this by 12 to clock its own operation. What complicates the 6502's implementation is the fact that over half of its instructions can use many or most of 11 different addressing modes to index a 16-bit address space, and achieving correct execution cycle count for many instructions requires a highly combinational datapath capable of retrieving and using data from memory in the same cycle its address becomes available. Our 6502 implementation is not a true 6502, but rather a clone which meets all critical parts of the 6502 spec and is greatly facilitated by the use of a functional Verilog control loop.

A real 6502 utilizes a simple ALU with 8-bit addition-with-carry, subtraction-with-carry, AND, OR, and XOR functionality, as well as basic incrementers/decrementers and a shift register. Our 6502 rolls all this into a single 8-bit 12-function ALU which can take both real data inputs and a selection of hardcoded values. While this increases the bitwidth of the ALU control input, it greatly decreases the number of distinct modules we need and the number of separate values we need to set in each state of our FSM.



6502 wiring diagram (see appendix A for details)

As stated, the first difficulty with the 6502 is that many instructions can use any of numerous addressing modes, and instruction execution time will vary by the mode

used. The original 6502 uses a microcode instruction set to specify operations cycle by cycle. Our implementation uses a 40-state FSM containing two fetch/decode states followed optionally by any of 18 different multi-state execution loops, which together are suitable to match all documented 6502 functionality and a moderate subset of undocumented opcodes achieved by replacing the last (dead) microcode cycle of certain read-modify-write instructions with a different computational cycle. Two additional states were added to our 6502 late in the project to handle sprite Direct Memory Access- whenever the register at $4014 is written to, the 6502 pauses its normal execution, instead devoting its cycles to an automated read/write loop to transfer all the data between $xx00 and $xxFF, where xx is the value written to $4014, to $2004, effectively updating all sprite data in the PPU's memory space.

The second difficulty with the 6502 is that to achieve the 2-cycle execution time for certain instructions and help decrease the execution time of numerous others, memory values have to be accessible during the same cycle as their addresses are generated. This posed the biggest challenge to implementation since traditional Verilog FSM-based processors set the processor's internal registers as a direct function of FSM state, setting address lines on one cycle and reading/writing data on the next. To account for the necessity of a combinational datapath with direct access to memory lines, we decided to use a combinational datapath with direct access to memory lines. Our FSM, then, controlled not the actual internal registers but the select line values for over 10 different mulitplexor trees.

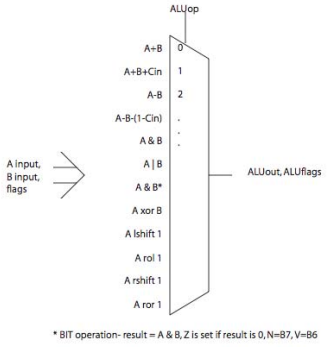
In order to allow the required memory timing, an additional hack was needed. In the first version of our 6502, we configured the datapath and memory address on the positive edge of the clock but latch actual values to their destination registers on the negative edge of the clock, with memory running on a faster clock than the CPU and consequently taking/delivering data cleanly, if multiple times, between the positive and negative edges of the CPU clock. There is a chance that this is how the real 6502 operates, if inferences can be made off the fact that the real 6502's internal clock runs at 1/12 the speed of the NES master clock and has a very explicit 15/24 duty cycle- enough to allow slow value propagation after the posedge before latching on the negedge. However, once we noticed that some PPU and controller registers change the system state even on a read, we could no longer overclock the memory/peripheral system and presume reads to be harmless. In the second version of our 6502, the datapath is configured on the positive edge of the clock, memory access happens on the negative edge, and values are latched to internal registers at the following positive edge. The hope was that register latching would happen enough faster than datapath reconfiguration that we would reliably get only the values from the previous cycle into the registers. This turned out not to be the case, so a third timing variant was developed which used a skewed clock to latch registers once and only once between the negative edge and positive edge of the CPU clock. Matters were complicated here by the fact that, unlike the real 2A03, our 6502 took in [master clock / 12] rather than taking in [master clock] and producing [master clock / 12]. After multiple petitions for a real skewed clock, Sean was able to rig a reliable hack which generated an appropriately skewed clock from a combination of the CPU clock, PPU clock, and a shift register.

The only things our 6502 is presently *not* designed to handle identically to a real 6502 are a small collection of undocumented opcodes and cycle-exact timing of NMI/IRQ interrupt handling. The undocumented opcodes *not* currently handled by our 6502 (or rather, handled by treatment as 2-byte NOPs) mostly create an output by driving multiple values to the data bus at once and letting the bus hardware determine what the actual value is. This would be difficult or impossible to capture digitally in Verilog. IRQs and NMIs are handled by our 6502 in 2 fewer cycles than on a real 6502. This should not be an issue for most games, but could be remedied by the addition of another dead state to our FSM.

Our 6502 currently synthesizes and was tested on all addressing modes, and on at least one variant of each arithmetic/logical operation. While its performance on real cartridge code is ambiguous, each individual instruction appears to be operating correctly to the extent visible in ChipScope, and Sean is reasonably convinced that any remaining bugs are either due to typos in one or two individual states, or else resultant from a fundamental difference between the real 6502 spec and the spec Sean worked out over the course of the semester. The former would be difficult to catch without a test, tedious to write and perhaps more difficult to plan, of every instruction in every addressing mode. The latter would need to be checked by further research and disassembly of commercial ROMs for comparison with the instructions seen/executed by our 6502.

Part 1b: The Arithmetic Logic Unit *(rrajan, spkelly)*

The ALU implements various basic instructions that would support the 11 addressing modes of 6502. Each of the instructions affect some flag bit. The ALU only gets data in the form of 2 operands from the CPU and will have to perform the operations as required. They do not distinguish between the various addressing modes.



ALU functional diagram

The flags are indicated as
N – Negative (set if the result of the instruction is negative in signed representation)
Z – Zero (set if the result of the instruction results in a zero)
C – Carry (set if the result of the instruction results in a carry)
I – Interrupts (set if the instruction generates an interrupt)
D – Decimal (set if the result of the instruction is in the decimal format)
V – Overflow (set if the result of the instruction creates an overflow condition)

OP1 – Operand 1
OP2 – Operand 2
RES – Result register
– indicates no change
X indicates a change in the value

The instructions implemented by the ALU are

1. ADD
   Add 2 operands — N Z C I D V
   OP1 + OP2 -> C, RES — X X X - - X

2. ADC
   Add 2 Operands with Carry — N Z C I D V
   OP1 + OP2 + C -> C, RES — X X X - - X

3. ASL
   Arithmetic Shift Left — N Z C I D V
   C <- OP1 <- 0 — X X X - - X
   C <- b7 | b6 | b5 | b4 | b3 | b2 | b1 | b0| <- 0

4. BIT
   Bit Test Operands — N Z C I D V
   OP1 ^ OP2 -> RES — X X - - - X

5. CLC
   Clear Carry — N Z C I D V
   0 -> C — - - X - - -

6. CLD
   Clear Decimal — N Z C I D V
   0 -> D — - - - - X -

7. CLV
   Clear Overflow — N Z C I D V
   0 - > V — - - - - - X

8. CMP

Compare 2 operands — N Z C I D V
OP1 – OP2 = RES — X X X - - -

9. DEC
   Decrement the number — N Z C I D V
   OP1 – 1 -> RES — X X - - - -

10. EOR
    Exor the 2 operands — N Z C I D V
    OP1 ^ OP2 -> RES — X X - - - -

11. INC
    Increment the number — N Z C I D V
    OP1 + 1 -> RES — X X - - - -

12. LSR
    Logical Shift Right — N Z C I D V
    0 -> b7 | b6 | b5 | b4| b3 | b2 | b1 -> b0 — X X - - - -

13. ORA
    OR the operand with the second — N Z C I D V
    OP1 | OP2 -> RES — X X - - - -

14. ROL
    Rotate left by one bit — N Z C I D V
    C <- b7 | b6 | b5 | b4 | b3 | b2 | b1 | b0 <- 0 — X X X - - -

15. ROR
    Rotate right by one bit — N Z C I D V
    0 -> b7 | b6 | b5 | b4 | b3 | b2 | b1 | b0 -> C — X X X - - -

16. SBC
    Subtract one operand from other with carry — N Z C I D V
    OP1 – OP2 -> RES — X X X - - X

Part 2: The pAPU *(rrajan, rsinnott)*

The pAPU section is the emulation of the 2A03 processor which is basically the 6502 processor but without the decimal mode that 6502 supports.

We basically implemented only the square channel of the pAPU unit, which essentially has 5 channels, 2 square channels, 1 triangle wave, noise channel and the DMC.

The pAPU needs the values at memory addresses $4000 to $4017 mainly to function, which is provided by the 6502. The 6502 provides various addressing modes that enable the values at these addresses to be retrieved and fed into the pAPU unit.

The signal that serves as the clock to the pAPU is the 1.79 MHz clock which is the main system clock divided by 12, and this is what clocks each unit of the pAPU.

The audio processing unit consists of 5 channels.
• Square Channel 1
• Square Channel 2
• Triangular Channel
• Noise Channel
• Delta Modulation Channel

We were successful in getting one of the Square channels working.

The square channel essentially has the below given units.
• Envelop Generator
• Sweep Unit
• Timer
• Sequencer
• Length Counter
• DAC

The pAPU is clocked by 1.79 Mhz clock, which is the master clcok(21.48 Mhz) divided by 12.
The registers which control the square channel (chl 1 and chl 2)are

$4000/$4004 : ddle nnnn : duty, disable length , envelop disable, envelop period
$4001/$4005 : eppp nsss : enable sweep, period, negate, shift
$4002/$4006 : pppp pppp : period low
$4003/$4007 : llll lppp : length index, period low

Envelop Generator:

This is used to generate a constant volume. The channel's first register controls the volume. The unit is made up of a divider and a counter. The divider's period is set to n +1. The divider is clocked at each of the clock signal that it receives, except for when there has been a write to the 4^{th} register since the last clock, then the divider is reset to 0 an counter is set to 15.

Each clock that the divider outputs, the counter is decremented. Only in cases where the loop is set, and counter is 0, then it is set to 15.

The channel's volume is the value in the counter. If the disable is set then the volume would be 'n'.

In our case, we had set it to a constant value of 1111(n) and envelope was disabled.

Sweep Unit:

This is used to constantly change the frequency of the square channel. This is controlled by the 2nd register of the channel. This contains a divider and a shifter.

The period of the divider is p. The shifter calculates a result from the channel's period registers(4002 and 4003). This value is shifted right by s bits. If the negate bit is set, then the shifted value is inverted. The shifted and inverted value is then added to the current period which yields the final period. This is continuously updated on each clock.

If the sweep unit is enabled and the output of the shifter is non zero, when the divider outputs a clock and the period high and period low registers are updated with the new value. If the channel's period is less than 8 or shifter's value greater than 7ff then the output is a 0.

Timer:

The timer consists of a divider whose period is got from the period high and period low values of the channel. The divider's period will be p+1, which is an 11 bit value.

This is continuously updated by the sweep unit.

Sequencer:

This is the unit which generates some low frequency signals 60 Hz, 120 Hz, 240 Hz, 48 Hz, 96 Hz, 192 Hz. The 240 Hz clock is generated by dividing the 1.79 MHz clock by 7458 and the 120 Hz, and 60 Hz can be generated from th 240 Hz signal. The 192 Hz clock is generated by dividing the 1.79 MHz clock by 9323.

The bit 7 of $4017 controls the mode .

If mode is 0, then the 4 step sequence is generated(60,120 and 240) and if the mode is 1, the 5 step sequence is generated(48,96 and 192).

Length Counter:

This allows duration control of the channel. The 'halt' bit which is the 5th bit in the channel's first register is the one that controls the counter. If the halt bit is set, the

counting can be halted. The counter is loaded with a value indexed from a table using the higher 5 bits of the channel's 4th register.

iiii i--- length index

bits 7-4	bit 3 = 0	bit 3 = 1
0	$0A	$FE
1	$14	$02
2	$28	$04
3	$50	$06
4	$A0	$08
5	$3C	$0A
6	$0E	$0C
7	$1A	$0E
8	$0C	$10
9	$18	$12
A	$30	$14
B	$60	$16
C	$C0	$18
D	$48	$1A
E	$10	$1C
F	$20	$1E

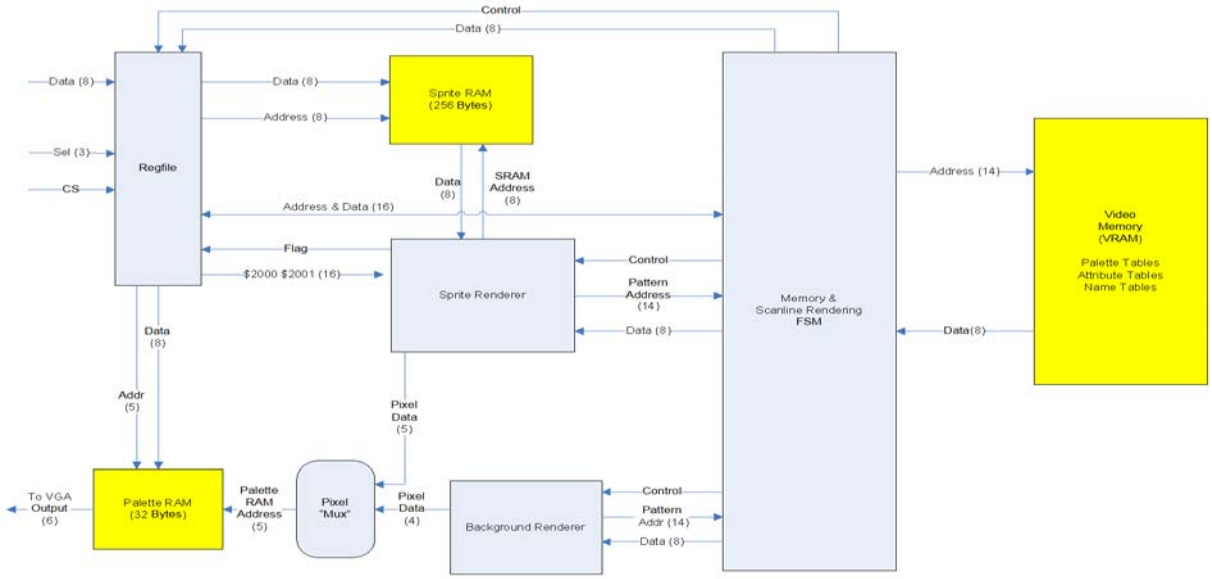
The counter can be cleared by clearing the appropriate bit in the status register, which clears the counter. When this is clocked, if the counter value is non zero and the halt flag is clear, the counter is decremented.
The bits 7 and 6 of the channel's first register also controls the duty cycle of the waves.
00 : 12.5%
01 : 25%
10 : 50%
11 : ~12.5% or 75% (with low first).
Sweep unit controls the period and hence updates the period high and period low registers. This decides the divider value in the timer and in turn controls the sequencer. On the sequencer, length and envelop being enabled and clocked, it is fed to the DAC.

Sweep ------> Timer/2 -------> Sequencer
Sequencer + Length + Envelop ------> DAC

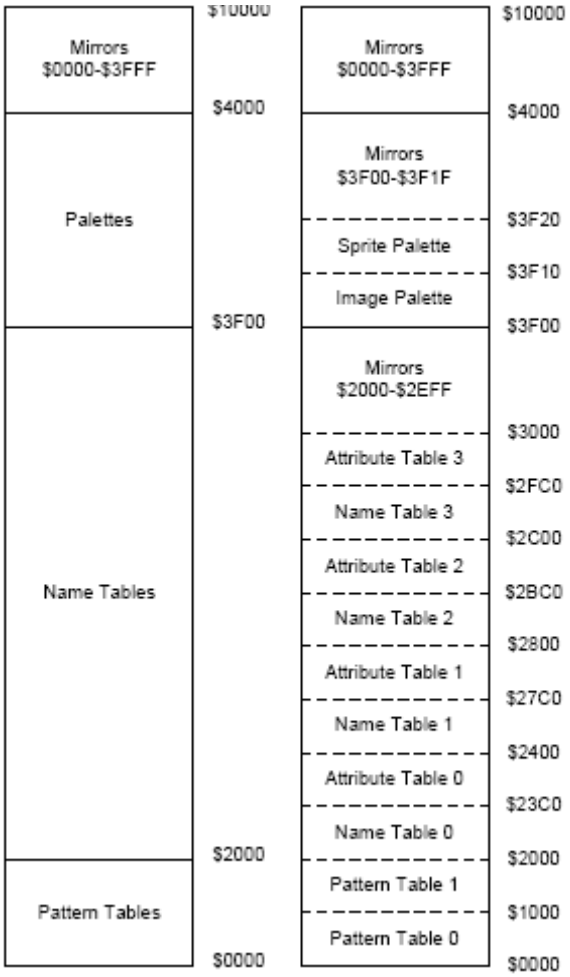
The square channels in combination with the other channels would be ideally required to produce the correct NES audio characteristics. Presently it can generate only monotones. The output was connected to a speaker and tested. Finally when the project was put together, it generated a beep as a part of the scoring mechanism.

2C02 Picture Processing Unit (PPU) *(rng, rsinnott)*

2C02 Picture Processing Unit (PPU)



The 2C02 was Nintendo's custom graphics processor. The PPU can address up to 16KB of memory, but only has 2KB of physical RAM. The PPU is controlled by the CPU via registers $2000-$2007. Registers $2006 and $2007 are used to write to VRAM. Interestingly, a double write to $2006 is required to assemble the address to write to. This is because the address space is 14-bits, but the register is only 8 bits wide. Additionally, the PPU had 256 bytes of separate memory for sprites. The VRAM was also located off the PPU chip, and usually a memory mapper on the cartridges determined if a particular VRAM access accessed the cartridges RAM and ROM, or if it accessed the onboard VRAM instead.



PPU Memory Map

It takes two cycles for every memory access to the VRAM. During the first phase, ALE is set, and the lower 8 bits of the address are latched to an external latch, and during the next cycle, either read or write is set to determine its behavior. This was done in order to save pins on the original NES, allowing the AD bus to be used for both addressing and data I/O.

The Following Table describes the functionality of each register:

Register	Bits	Description
$2000		PPU Control Register #1 (writable)
	7	Enable NMI on VBLANK if 1
	6	PPU Master/Slave Select (not used, there's only 1 PPU)
	5	Sprite Size: 0: 8x8, 1: 8x16
	4	Background Pattern Bitmap Table Select
	3	Sprite Pattern Bitmap Table Select
	2	PPU Address Increment: Increment by 32 if 1, by 1 if 0
	1-0	Name Table Address Select

$2001		PPU Control Register #2 (writable)
	7-5	Background color when $2001.0 is 1? Intensity on 0 (This register does some weird stuff with the intensity levels on NTSC TVs, we're not sure exactly what type of behavior this is supposed to have)
	4	Sprite Enable: Render Sprites on 1
	3	Background Enable: Render Background on 1
	2	Sprite Clipping: 1: No clipping 0: No sprites displayed in left 8-pixel column
	1	Background Clipping 1: No clipping 0: Background not displayed in left 8-pixel column
	0	Display Type 1: Mono (Black and White) display 0: Color display
$2002		PPU Status Register (read only)
	7	VBlank Flag- Gets set to 1 when entering VBLANK Reading from Register $2002 clears this flag
	6	Sprite #0 Drawn Flag (cleared on VBLANK)
	5	More Than 8 Sprites On Current Scanline Flag
	4	VRAM Write Flag – 1 if currently writing to VRAM Writes to VRAM are ignored while this flag is high
	3-0	Not used?
$2003		Sprite RAM Address Register (writable)
		Holds the 8 bit address used to access Sprite RAM
$2004		Sprite RAM Data Register (writable)
		Used to Read and Write Data from the Sprite RAM Increments $2003 by 1 when accessed
$2005		Scroll Offset Register (16 Bit Register)
		Used to scroll the background Accessing this register toggles whether the high byte or low byte is accessed next (Reading from $2002 resets the toggle bit)
$2006		VRAM Address Register (16 Bit Register)
		Specifies the 16 bit address used to access VRAM by access through $2007 Same Behavior for access as $2005 Note: Shares the same toggle bit with $2005, accessing $2005 or $2006 will set the toggle
$2007		VRAM Data Register

		Used to Read and Write from VRAM Increments $2006 on access, increment amount determined by $2000.1 If reading, the data read will be data from the previous address* $2007 acts as a read and write buffer between the CPU and VRAM *Doesn't do this when accessing addresses $3Fxx since this accesses the internal Palette RAM instead of the VRAM

Screen Rendering Overview

In order to render a frame onto the screen, the PPU goes through a series of steps depending on the status of the current scanline. The basic operations of a frame are as follows. The first 20 scanlines is referred to as the VINT period. This starts when VBLANK goes high, and stays in this state for 20 scanlines. In this state, the PPU does not do any image processing, and is time allocated for the CPU to set the control registers, and write information to the sprite RAM and VRAM. After this period is over, there is a special scanline(scanline 20) that is used to prime the pipeline. This scanline behaves just like the rest of the scanlines that will be described below, except it will not render anything to the screen. After this, for the next 240 scanlines, it will perform the rendering operations that will be described in detail below. After which, it will rest for one more scanline, and then raise VBLANK and reenter the VINT period.

Each scanline can be broken down into 4 stages, which behave as follows:

Stage 1: This is where all the rendering takes place. One pixel is drawn every clock cycle here. This stage lasts 256 clock cycles out of the 340/341 clock cycles of each scanline.
During this stage, the background renderer is making full use of VRAM, and will fetch the Name Tables, Attribute Tables, and the Pattern Bitmap data for the tiles to be drawn.
The sprite buffers are also all active during this time, and the data from the background and sprites is passed to the priority mux to determine what the outputted pixel is.
The Range Evaluator is also running at this time, to determine which sprites will be drawn in the next scanline

Stage 2: This is when the PPU enters it's HBLANK stage, where it can no longer render pixels while waiting for the gun to return to the beginning of the next scanline. This phase lasts 64 clock cycles
The background renderer lets go of control of the VRAM during this stage, and allows the sprite renderer access.
The sprite renderer fetches the appropriate bitmaps from the VRAM during this phase and loads its sprite buffers for processing of the next phase

Stage 3: This stage lasts 16 clock cycles
Control of the VRAM returns to the background renderer. This is the period where the background renderer prefetches the first two tiles to be drawn for the next scanline, and loads them into the appropriate shift registers.
The sprite renderer doesn't do anything during this stage.

Stage 4: This is the final stage. This stage lasts either 4 or 5 cycles
During this stage, the PPU retrieves two name table entries, but we are unsure of why this takes place. It doesn't seem to be used and seems to be wasted cycles.
The last cycle here, (cycle 341), is a rest cycle, which during every odd scanline 20 is skipped, in order to do some phase adjusting for the color burst related to NTSC.

Sprite Rendering

The NES is capable of drawing 64 different sprites in a single frame, but because of the limitations of time to access the VRAM, the sprite renderer is only capable of rendering 8 sprites per scanline. Sprite Information is stored in two places: the various attributes of the sprites are stored in Sprite RAM, and the pixel data is stored on the memory mapped pattern bitmap determined by $2000.3. For each sprite, 4 bytes of data is used to store the various information, and is laid out as follows:

Byte 0: Y Position of the Top Left Corner -1
Byte 1: Tile Index Address
Byte 2: Attribute Information
- Bits 1, 0: Palette Select Bits
- Bit 2-4: Not Used?
- Bit 5: Sprite Priority Bit
- Bit 6: Horizontal Inversion Bit
- Bit 7: Vertical Inversion Bit

Byte 3: X position of Top Left Corner

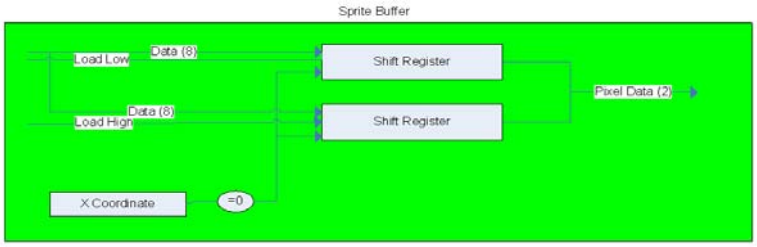
The first step in drawing a sprite is determining which sprites are in range to be drawn for the next scanline, which is then all stored in a temporary sprite memory to be used to load the data to be rendered when it is allotted time to use the VRAM. This sprite temporary memory contains 24 bits of data: 8 bits for the Tile Index Address, 8 bits for the X coordinate position, 4 bits of attribute data, bits 6, 5, 1, and 0 of the attribute information (The vertical inversion bit isn't stored because vertical inversion is already applied by this step), and the 4 bit range that was calculated. Because of such limited time allotted for retrieving this data, the PPU only has time to retrieve data for 8 sprites per scanline. If it is an 8x8 sprite, the information stored works as expected, using the appropriate pattern bitmap selected from $2000.3, the tile index, and the lower 3 bits of the range to determine the address to be used. Thus, the address used is:

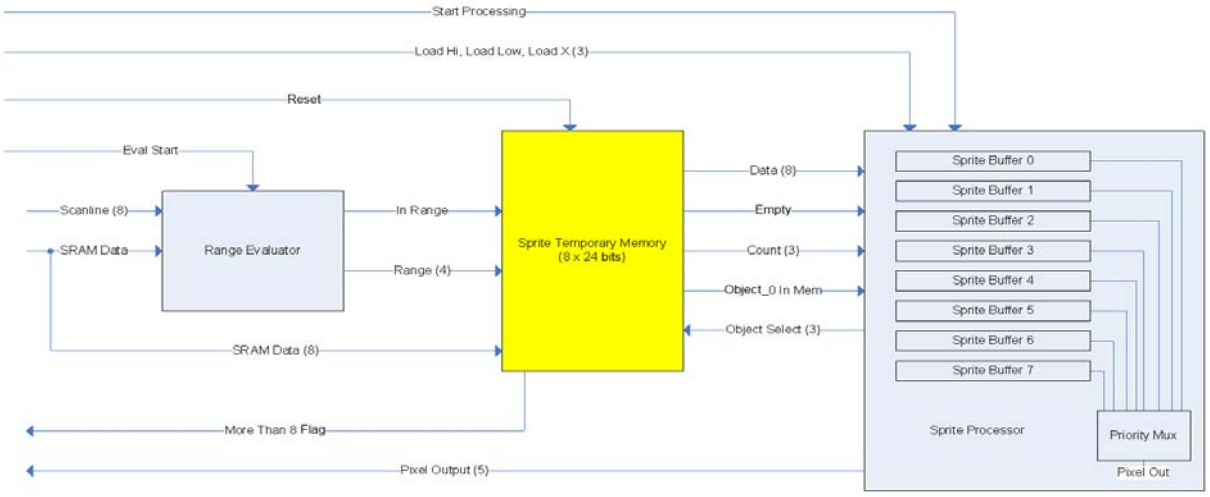
Sprite Address = {0,Pattern Table Select, Tile Index, High_bit, Range[2:0]}

The High_bit determines if we're retrieving the upper or the lower bit of the tile from the attribute table, the range determines which row in the 8x8 tile to fetch.
If it is an 8x16 sprite, the pattern select bit from $2000.3 is ignored; the LSB of the tile index is instead used to determine which pattern table to use and the MSB of the range becomes the LSB of the tile index. The lower bits function as normal. The sprite address in this case looks like this:

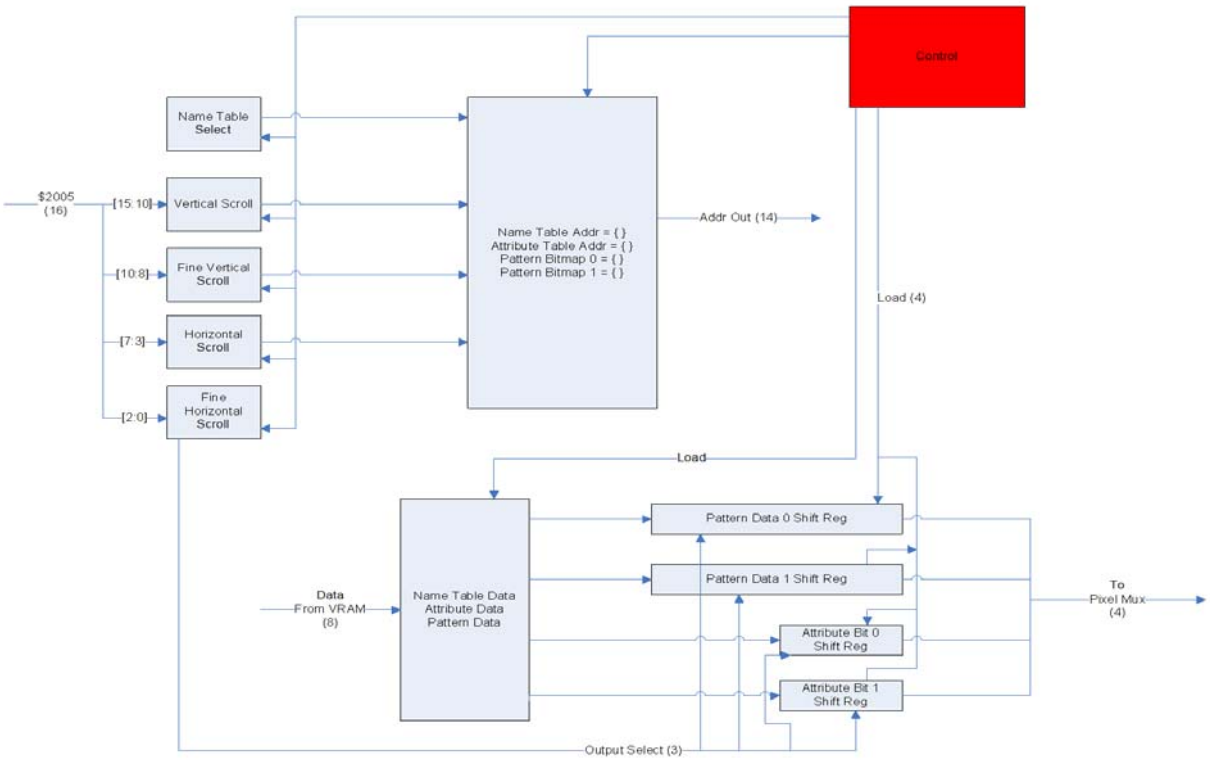
Sprite Address = {0, Tile Index[0], Tile Index[7:1], Range[3], High_bit, Range[2:0]}

This data is loaded into shift registers that will be used to draw the sprites for the next scanline. The X coordinate is also loaded into a register that decrements as every pixel is rendered, and the associated shift registers for that sprite will start shifting when this reaches 0. This allowed the sprite to start drawing at the appropriate time on the scanline, instead of at the beginning of the screen. All this information is passed through a priority mux that determines which pixel is to be drawn. The data for the PPU is arranged in such a way that the priority of the sprites is determined by their order, with sprite 0 being the most important, and sprite 64 being the least. The mux basically checks the eight sprite buffers in order, and if a sprite is transparent, it looks at the next buffer, until it sees a non-transparent sprite, which it then outputs, or reaches the end, where it will just output an all zero transparent sprite. Also passed along with the two pattern bitmap bits to the priority mux is the priority of the sprite, the two palette select bits, and whether this sprite is sprite #0 (important for sprite #0 hit detection). Note that this is why it's important to have a scanline that doesn't render to the screen in order to preload the sprites into these sprite buffers to be drawn.





Background Rendering



Background Rendering involves accessing data stored on the VRAM. The information is stored in three separate tables, and was an optimization that the NES used to overcome its severe memory limitations. Thus it takes four reads from memory to drawn a single tile, which is 8 pixels long. The first read is to the name table which determines which pattern bitmap to use, the second is to the attribute table, which contains the information about the tile accessed by the name table. This table contains the upper two bits of the background color. The last two reads are to the pattern bitmap, and contain the actual 8x8 tile that the name table is referring to. The first read determines the lower bit, the second the upper bit.

The NES has four name tables, at addresses $2000, $2400, $2800, and $2C00, and an attribute table associated with each name table. Each name table contains tile indices to 8x8 tiles stored in the pattern bitmap. The name table stores 32x30 tiles, each a byte long. Though it has four name tables, the NES only has enough physical memory for two tables, and the other two are typically mirrored by the memory controller. The types of mirroring are determined by the memory controller on the cartridges, but in effect, two of the tables just point to the other two tables used. The name tables basically serve as pointers to tiles that would be stored in ROM on the cartridges, lowering the required number of writes to the PPU by the CPU during the limited VINT period that the CPU has access to write to memory, and also served to help save memory, since a name table needed to only store 1 byte per tile, instead of the 16 bytes a pattern bitmap tile takes up.

Image taken from Jeremy Chadwick's nestech document found on nesdev:

Name	NT#0	NT#1	NT#2	NT#3	Flags
Horizontal	$000	$000	$400	$400	
Vertical	$000	$400	$000	$400	
Four-screen	$000	$400	$800	$C00	F
Single-screen					S
CHR-ROM mirroring					C

Along with each name table, it has an attribute table. This is probably the hardest to understand table that is stored in memory. Each byte of data contains data for a 4x4 tile block, with two bits of color for each tile. This was another of the Nintendo's methods to save on memory. Because the upper two bits of the four bits needed to get 16 colors are stored in this attribute table instead, you only needed to write 2 bits to the pattern bitmaps stored on the ROMs, cutting the amount of memory required on the cartridge for tiles in half.

Information copied from Jeremy Chadwick's nestech document found on nesdev:

```
It's quite confusing; two graphical diagrams may help:

  +------------+------------+
  |  Square 0  |  Square 1  |  #0-F represents an 8x8 tile
  |   #0  #1   |   #4  #5   |
  |   #2  #3   |   #6  #7   |  Square [x] represents four (4) 8x8 tiles
  +------------+------------+   (i.e. a 16x16 pixel grid)
  |  Square 2  |  Square 3  |
  |   #8  #9   |   #C  #D   |
  |   #A  #B   |   #E  #F   |
  +------------+------------+

The actual format of the attribute byte is the following (and corris-
ponds to the above example):

   Attribute Byte
     (Square #)
  ----------------
      33221100
      ||||||+--- Upper two (2) colour bits for Square 0 (Tiles #0,1,2,3)
      ||||+----- Upper two (2) colour bits for Square 1 (Tiles #4,5,6,7)
      ||+------- Upper two (2) colour bits for Square 2 (Tiles #8,9,A,B)
      +--------- Upper two (2) colour bits for Square 3 (Tiles #C,D,E,F)
```

Because of the unique way that the attribute bits are mapped to various tiles, one byte is able to contain enough data for 256 pixels or the 64 tiles, as opposed to the 1 Kbyte needed if these top two bits of color were stored the same way as the lower two bits are stored in the pattern bitmap tables. On the other hand, this also limited the amount of colors that could be represented for tiles, since four tiles shared the same two bits of color.

The Pattern Bitmap Tables as far as we can tell are usually stored on the ROMs, and start at addresses $0000 and $1000. There are two tables, one is typically used for the sprites and one for the background, though if the sprites used are 8x16 sprites, then both tables are used for sprites, and the background still uses its one table. The tile index

number will point to a specific tile on the pattern bitmap tables. A tile is 8x8 pixels, and since the pattern bitmap stores two bits of this, this takes up 16 bytes per tile. The lower 8 bytes contain the information for the lower bit, and the upper 8 bytes contain information for the upper 8 bytes. The fine vertical offset is used to determine which two bytes get retrieved for a particular memory access.

Image taken from Jeremy Chadwick's nestech document found on nesdev:

```
 VRAM    Contents of                    Colour
 Addr   Pattern Table                   Result
------ ---------------                 --------
$0000: %00010000 = $10 --+             ...1.... Periods are used to
  ..   %00000000 = $00   |             ..2.2... represent colour 0.
  ..   %01000100 = $44   |             .3...3.. Numbers represent
  ..   %00000000 = $00   +-- Bit 0     2.....2. the actual palette
  ..   %11111110 = $FE   |             1111111. colour #.
  ..   %00000000 = $00   |             2.....2.
  ..   %10000010 = $82   |             3.....3.
$0007: %00000000 = $00 --+             ........

$0008: %00000000 = $00 --+
  ..   %00101000 = $28   |
  ..   %01000100 = $44   |
  ..   %10000010 = $82   +-- Bit 1
  ..   %00000000 = $00   |
  ..   %10000010 = $82   |
$000F: %00000000 = $00 --+

The result of the above Pattern Table is the character 'A', as shown
in the "Colour Result" section in the upper right.
```

A bit in $2000 determines which pattern bitmap backgrounds use, the name table determines which particular address gets accessed, 1 bit is used to determine if we're accessing the low or high byte, and the fine vertical offset determined which of the 8 bytes is accessed, so the memory access to the pattern bitmap looked something like this:

Pattern Bitmap Address = {0, Select bit, Tile Index(8 bits), High_bit, Fine Vertical Offset(3 bits)}

Register $2005 is used to determine the starting offsets in the name tables to use, in order to handle scrolling. The values of this register are loaded into several registers, which increment at various times to determine the next name table address to be fetched. The counter for the horizontal scroll increments once every 8 clock cycles (after all the information for the current tile is accessed), and will flip the lower bit of the name table select when it reaches 32, and reset to 0. The vertical offset is daisy chained to the fine vertical offset, which is used to determine which row in the tile to access. The fine vertical offset increments once every HBLANK. The vertical offset counter only counts to 30, and resets to 0 when it reaches 30, and flips the upper bit of the name table select when it reaches 30. It does this because a name table is only 32x30 tiles.
The pattern bitmap information retrieved is loaded into the upper 8 bits of a 16 bit shift register every 8 clock cycles during rendering. This register also shifts every clock cycle during rendering. The attribute table data is loaded serially into an 8 bit shift register, with the bit loaded serially to be determined by which tile of the 16 tiles that the attribute table refers to is currently in use. Note that this means that the same two bits are always loaded for every 8 pixels, limiting the amount of colors that can be drawn to the same 4 colors every 8 pixels. The fine horizontal offset is used to determine where in the shift

register to sample the pixel data. This data is passed to the pixel mux to be processed along with the sprite data.

Pixel “Multiplexer” (Mux)

The Pixel "Multiplexer" decides which color is displayed for each pixel on the screen. Data arrives from the Background Renderer (4 bits -background_pixel), Sprite Renderer (5 bits - sprite_pixel) and the registers ($2001.3 - $2001.7). The Sprite Data is passed into the Mux if the Sprite Enable bit ($2001.4) is set. The Background Data is passed into the Mux if the Background Enable bit ($2001.3) is set. If the Background Enable bit is not set, the Default Background ($2001.5-$2005.7) is passed to the Mux. We’re not sure if this was the correct behavior, since different documentations had conflicting information on the purpose of these three bits on $2001. The data that gets passed from the priority mux becomes the address to a lookup table in the palette ram, to determine what color is used. This color was then passed out of the chip to the TV, or in our case, to our VGA framebuffer, to be converted by a VGA lookup table.

Inside the Mux, if the Sprite Pixel is transparent, the background is selected for that pixel. If the Sprite is non-transparent and has high priority, then the Sprite is selected. If the Sprite has low priority, then it selects the Background. If the sprite drawn is sprite #0 and has a collision with the background (both are nontransparent for that pixel), then the sprite #0 hit flag is raised. Transparency was determined by seeing if the two bits that were obtained from the palette bitmaps were zero. Because of this logic for determining colors, sprites were only able to have 3 colors each, and backgrounds could have three unique colors, and one that was the default background color, which is located in the palette ram address $3F00. Because of this mapping, the background and sprites weren’t able to access all 16 colors that are technically addressable on the palette. Sprites were only able to access 12 colors, since any address with the lower two bits as 00 would be regarded as transparent, so addresses $10, $14, $18, and $1C could never be reached. Backgrounds were able to use 13 colors, since $00 was reached, and served as the default background color, but any access to $04, $08, $0C was considered transparent and mapped to $00 if a sprite wasn’t getting drawn on top of it.

Debugging the PPU

One of the biggest problems with the PPU was figuring out how we were going to debug it. The PPU, while well documented, is well documented by fans, and Nintendo never released any official information on it. Because of this, information is all from hearsay and tests of the actual system, to guess what was inside the PPU. This leads to a lot of parts that no one is what they actually did, and a lot of conflicting information. This made figuring out the actual functionality of the PPU in order to try to debug it very difficult, so we relied a lot on trying to figure out which information made the most

sense, and was the most universally agreed on to determine which functionality was the proper one, while hoping that once we got it fully working, we could determine what the appropriate behaviors were, and fix the glitches that happened from things that we interpreted badly. Unfortunately we never got to the point where we could run an actual cartridge to find and fix most of these glitches.

Very few of the modules themselves were very complicated, and the individual modules were easily debugged, but a large part of the PPU could not be debugged without the entire PPU attached and fully functional, since most of it's functionality depended on our FSMs. Even with it fully connected, we still quickly ran into a problem of how to send it enough test data that we could get appropriate output, and how we would even judge that what we were seeing was correct. In the end we decided we would simulate a black box CPU through an FSM and send in data through the registers on the PPU. This also served as a good stress test of the previously tested register file. We also decided that the easiest way to catch and fix bugs would be to get our NES to VGA conversion working properly and to use visual verification first. Because of various timing issues, we ended up having to use ModelSim extensively to find all the various timing problems that were causing the glitches and improper display on the screen. This proved to be a very time consuming process with all the signals that needed to be observed, and the fact that it took close to 100,000 clock cycles to render a single frame. Deciding to get our VGA working first, and using visual verification first proved to be a very good idea for preliminary debugging, because with it, we were able to narrow down where problems were and had a better idea of what to look for in order to find all our bugs. The bugs that proved the hardest to track down were the ones that worked perfectly in simulation but because of synthesizing it with XST into Xilinx primitives, the wrong behaviors were assumed by their primitives, leading to different behavior. Using the post synthesis verilog files that ISE generated, and comparing it with our simulation version, we were able to finally track down some of the more elusive bugs, such as that even though the synthesis tool tells you that some of our memory modules should be converted to synchronous reading memory modules instead of a synchronous reads, and hinted that it wasn't using available block RAM, which we were fine with, since the blocks they were referring to were small, and would've worked fine as just an array of flip-flops. We eventually realized that even despite the warning that we weren't making use of the faster block RAMs, it converted those modules into block RAMs anyway, and was using synchronous reads, which completely threw off our timing on some of our more timing sensitive modules. Once that was fixed, we were able to quickly find and fix most of the other glitches that we saw and found. In the end, visual verification was the most valuable tool we had to find and help debug most of the glitches that we encountered on the PPU.

Support Modules

NES Cartridge Interface *(rsinnott)*

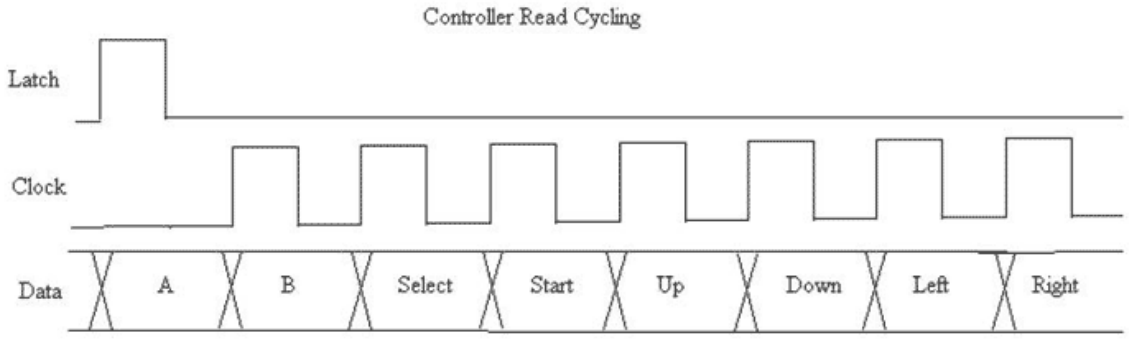
To enable the use of original NES cartridges on the system, we integrated a cartridge interface into the FPGA. We acquired a NES to Famicom 72-pin to 60 pin adapter to build the interface off of. Using two IDE cables, each of the 72 pins on the NES side of the adapter were soldered to. This cable was then plugged into the Left and Right expansion ports of the FPGA board. This provided a connection between the FPGA and any original NES cartridge which we plug into the adapter.

After construction of the adapter and appropriate additions to the Xilinx UCF file were made, we were able to successfully read data off both the PRG (program ROM) and CHR (character ROM) busses of the cartridge.

NES Controller Interface *(rng, rsinnott)*

We integrated controllers from the original NES into the FPGA board for user input. We obtained (2) Yobo Super 8 gamepads, as well as 1 authentic Nintendo controller. These were wired in on the Breadboard we had plugged into the High Speed Expansion port of the FPGA board.

The NES controller connects to the FPGA via 5 wires. They are Power, Ground, Latch, Clock (also referred to as 'Pulse'), and Data. Power and ground are tied to +5 volts and ground, respectively. To read user input, the controller first receives a single pulse from the system on the 'Latch' line, which latches the state of the 8 buttons into a shift register in the controller. The state of the buttons is then returned serially as the 'Clock' line is pulsed.



(image from http://seb.riot.org/nescontr/)

As a design decision, we decided to poll the controllers at a static rate of 60 times per second in hardware. An FSM is used to read the data from the controller. The FSM sends a pulse to the

When the CPU requests data from the controllers, it expects to receive the data serially. Since we chose to read the controllers at a static rate, we hold the controllers'

data in registers, which are re-serialized when the CPU makes a request to the controllers' memory location.

Clock Generation *(rng, rsinnott)*

Designing the timing of the system was a challenge, as we had to fit within many different constraints. The original NES was clocked entirely off of the NTSC crystal inside the NES. The NTSC crystal provided a 21.48 MHz clock, which the CPU divided by 12 to obtain its clock speed of 1.79 MHz. The PPU divided the 21.48 MHz clock by 4 to obtain its clock speed of 5.37 MHz, three times faster than the CPU.

The timing scheme of the system was driven by the constraints imposed by the components of the Xilinx board. First, the Xilinx board's sole clock source is its 100 MHz system clock. Additionally, since we opted to utilize Xilinx's DCMs (Digital Clock Managers), we were subject to the restrictions imposed by them. The first major limitation of the Xilinx DCMs is they have a minimum input frequency of 24 MHz. Additionally, Xilinx DCMs can only divide a frequency by integers up to 16, and half-integers up to 7.5.

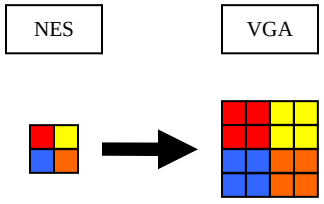
Given these constraints, we had to establish a series of clock multiplications and divisions to get the appropriate frequencies. Multiplying the 100 MHz clock by 2 with a DCM gives a 200 MHz clock. Dividing the 200 MHz clock by 7.5 with another DCM yields a 26.666 MHz clock, which becomes the VGA Pixel Clock. Using two more DCMs, the 26.666 MHz clock is divided by 5 and 15 yield 5.333 MHz and 1.777 MHz clocks, respectively. These become our CPU and PPU clocks.

VGA Adapter & Framebuffer *(rng, rsinnott)*

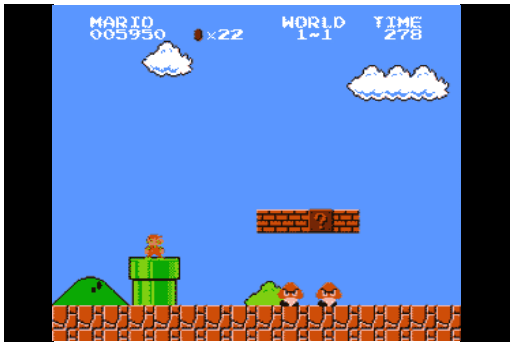
After discovering integrating the hardware for an NTSC display into the FPGA would have been a large headache, mostly due to timing issues, we decided to write a module that would translate the output from the NES to display on a VGA monitor.

Our system uses two framebuffers which are continually swapped between the NES and the VGA module. When a pixel is rendered by the NES, it is stored as the 6-bit NES color code in a framebuffer until the NES has finished rendering an entire frame. When the NES completes a frame, it switches which framebuffer it is writing to and begins the process of filling the framebuffer over again.

The VGA module reads each pixel the NES has written to the framebuffer and, in real-time, converts the NES color values to VGA color values for the monitor (see appendix B for VGA lookup table). The VGA module renders each pixel twice on each scanline and renders each scanline twice. This translates each NES pixel to 4 pixels on the VGA monitor, thereby producing a 512x480 image. Since VGA has a resolution of 640x480, the additional 128 pixels are split into 2 black bars which frame the NES image.



Pixel and Scanline Doubling



VGA output with black bars

When the VGA module finishes rendering a screen, it checks to see which framebuffer the NES is writing into. The framebuffer the NES is not writing into will contain the most recent, complete frame. The VGA module switches to the appropriate framebuffer, and begins rendering a screen all over again. By ensuring the VGA and NES are never reading and writing the same region of the same framebuffer, we eliminate visual artifacts and image tearing on the screen.

The Memory Mapper *(spkelly)*

We wrote a separate module to tie all the various components together. Not to be confused with on-cartridge memory mappers which determine which banks of cartridge ROM/RAM are connected to the cartridge address and data lines, our mapper module handles the assembly of all memories and peripherals into the address space seen by the 6502. The mapper module has ports for address, data and access-enable lines to the 6502, cartridge PRG, onboard RAM, and PPU, as well as inputs for the current continuously-polled status of both controllers, and an internal register file for registers $4000-$401F, many of which are output directly to the pAPU. Operation is fairly straightforward, with address-based multiplexor logic determining which lines to route to and from the 6502. Registers $4016 and $4017 are a little more difficult due to their multi-function nature. Reading from $4016 returns the current status of one button on the Player 1 controller and advances that controller's shift register to the next button to return. Reading from $4017 does the same thing for Player 2's controller. Writing to $4016 sends a strobe signal to both controllers to reset their internal shift registers, and writing to $4017 affects flags in the pAPU. Initially the logic regarding $4016 and $4017 was hellish, and likely incorrect. Once controller access was relegated to a separate, continuously-polling module and the mapper no longer needed to interface directly with the NES controller pins, however, the logic fell into place far more cleanly.

The limitations of our current mapper are mainly in how it handles on-cartridge memory mappers which extend the valid 6502-accessible cartridge address space. We could find no solid documentation on how the real NES handles, for instance, $600x writes to on-cartridge save RAM. $600x is well outside the $8000-$FFFF range mapped to cartridge PRG, and there are only 15 dedicated cartridge PRG address lines. There is a pin which takes the NAND of a 16th address line and the CPU clock, and possibly some

other logic, but we could not find any conclusive documentation on how it actually affected what was visible on the data lines. As such, we only anticipate "mapper 0" (one or two banks of CHR and PRG wired directly to the cartridge address and data lines) games will work with our design.

Note: we were able to read data from a Super Mario Bros. / Duck Hunt cartridge, which does use a custom mapper, as well as a couple "mapper 1" games, but because "mapper 0"cartridges weren't even running cleanly, it is difficult to tell whether these non-0 mappers were being handled correctly, or whether we were only seeing instructions from a PRG region not relying on any mapper functionality.

Methodology

The NES is inherently a hardware system, so we planned for a dominantly Verilog implementation, with possible emulation on the PowerPC processors if necessary. The design tools available and their varying degrees of usability (namely Xilinx ISE being far less complex and more reliable than EDK) pushed us all the way over to hardware. The NMOS 6502 base CPU, the PPU graphics chip and the pAPU audio component of the CPU were seen as the biggest modules required, so we planned to implement them individually to spec over the course of the semester, and wire them together at the end. The PPU in particular ended up involving nontrivial independent clock generation and VGA encoding modules. We saw the need for, but put off until last, an interconnect module to do address manipulation and construct the actual NES memory map from the various memories, coprocessors and peripherals.

From the very beginning we took a hint from previous NES attempts and dedicated the strongest members of our group to the PPU for the whole semester. The other half was assigned the CPU and, time permitting, the pAPU. Modules were written in Verilog, simulated with NC Verilog, Verilog XL, and the ModelSim design simulator, and implemented on a Xilinx University Program Virtex II Pro FPGA board. Each module was tested independently in simulation as far as possible using nonsynthesizable functional Verilog testbenches, then adjusted for synthesis and run on the board alongside mock-ups of any other hardware needed (e.g. a mock 6502 to send a fixed string of instructions to the PPU to display an image). Finally, the plan was to assemble all completed components on the board and iron out interfacing bugs. In reality, the PPU and CPU subteams approached a fully unified design from their own directions, and each wound up with a unified design that supported their own components perfectly, and the others in a theoretically sound but relatively untested fashion.

The PPU, CPU and memory-mapper interfacing modules were all completed and operating independently by late November. By December 7th, a top-level module integrating the CPU, PPU, pAPU and memories was completed, and even executed code from a real cartridge, however the top-level module designed by the PPU subteam, containing a mock CPU, PPU, pAPU and controller interfacing was decidedly more impressive on-screen due to as-of-yet unidentified discrepancies between a real NES and our design which caused real cartridge execution to derail before any image could be drawn. The most likely candidates for these discrepancies are cumulative minor 6502 bugs (execution looked good, but routinely ended up in questionable areas of PRG) and erroneous/absent NMI masking (NMI was always enabled, even when some carts cleared $2000, and execution frequently derailed at the end of NMI routines with non-RTI return methods). In order to run a simple commercial game, our 6502 would need to be double-checked for adherence to the original 6502 specification, and the PPU would need to have its wiring double-checked to ensure it could see and use a real cartridge CHR bank and respect all flags (interrupt masking, sprite/background enable/disable, color adjustment modes). While not strictly necessary, the pAPU could also be substantially improved from the most recently tested version.

Conclusions & Lessons

In the words of group leader Randy, the NES is “a fun project, but hard as hell!”

It was ultimately a good decision to partition the design as we did, and given the effort we all put into our respective portions, we would recommend future attempts take the same approach. However, do not rely on all the documents you find on the NES being in agreement, or being thorough from a hardware standpoint. Much of what is out there on the NES, even on Nesdev, is targeted at emulator authoring and ROM hacking rather than the nitty-gritty of hardware implementation. We wrote all of our component modules as clones of what functional specification we could find, which worked out well enough and probably gave us each a deeper understanding of our respective components than simply copying hardware diagrams would have.

We went in expecting all Xilinx software/hardware to be thoroughly documented and bug-free. This, as most groups found, was terribly naïve. Do not expect development hardware/software to work the way you expect all the time, and particularly do not expect to get any answers to specific problems out of any official Xilinx help system.

Our scheduling was a little lax, involving some completion goals, but no formal plan for what to do once things really did start coming together. The 6502 in particular was a week behind our initial schedule but with promising outlook at mid-semester, but once it simulated successfully, the CPU/pAPU subteam never quite committed to either taking the CPU into serious synthesis work or trying to develop the pAPU in a reduced-from-initial-plan timeframe, and ultimately split to do less efficient work in both directions at once. Our advice to future teams is set goals for yourselves, including bailout points for dropping particular components and coming to a group consensus on whether to finish the whole design or aim for a safer partial demo that will better fit your remaining timeframe. A few more pointers: eliminate impediments to synthesis of individual modules as early as possible, leave at least 2 weeks for combination of individually synthesized components, and budget as much time to deal with the combination of clock generation, peripheral interfacing, memory mapping, and memory generation/preloading as you would for any other major Verilog component- controller interfacing and preloading Xilinx memories without impeding the rest of a design are particularly nontrivial.

In all, building a working hardware NES in FPGA logic in a semester may not be an unreachable goal, however it wouldn’t be an easy one. We would not advise attempting it without at least four thoroughly committed group members, all with at least a year or two of formal Verilog training, and all taking no more than one other course requiring significant effort. Our PPU subteam made phenomenal progress, but worked many 10+ hour days in lab to achieve a working demo. Our CPU subteam was impeded at the start of the project as Reshmi learned a semester of hardware design methodology in a 3-week team-assisted crash course while Sean worked solo to complete the bulk of the 6502, and then the whole design got resultantly stalled late in the semester when Sean, the only member with a working knowledge of the 6502, got bogged down by a

wave of other classwork just when the CPU needed to be synthesized. By the end of the semester, Randy and Ray, who had pushed for a NES most strongly at the start, were putting in seriously unhealthy amounts of time in a panic to get a 100% working NES, Sean had given up hope of anything better than a concept tech demo and was routinely bailing on NES lab sessions after 6-7 hours in order to cut his losses in two other programming courses, and Reshmi was increasingly alienated, having dedicated most of her time to the pAPU, which was deemed questionable in early November and more or less dropped by December in the attempt to get the CPU and PPU to meet up. This snowballing from initial functional roadblocks to rising group tension stemming from different project goals, different amounts of available time, looming deadlines, and lack of sleep is what ultimately made a real FPGA NES impossible for *us*, as close as we came in many respects. If a future team can learn from our mistakes and start in with the knowledge we only gained from experience, they may just be able to pull it off.

Last Words

Sean Kelly

Alright. Agreeing to a full NES in a semester alongside two programming labs may not have been the best choice if I was aiming to hold or raise my ~3.5 GPA. I think (fingers crossed) I managed to pull it off, but it did hurt my time commitment to 18-545 individually.

The original plan was for Reshmi and I to do the 6502 and the pAPU, but since Reshmi was still learning Verilog for much of the semester, I wound up writing all of the 6502 proper, and doing coaching/cleanup on the 6502 ALU for as long as it would have taken me to write it myself (although I don't begrudge Reshmi the learning experience). I also wrote the memory mapper to tie all parts of the design together, and indeed we only got everything wired together at the end because I got exasperated at Ray's contentedness to sit in lab for an hour falling asleep while waiting for Randy to show up and pulled everything we had together through my mapper into a single project.

That said, if I averaged 12 hours a week in this course it was only because I pulled ~6 lab hours every other day since Thanksgiving, and a 22 hour stint the day before final presentations, which I am well aware pales in comparison to what Randy and Ray contributed. The 6502 took about a month to research/write, in 2-3 hour sittings every few days plus a couple 6-hour debugging sessions in the cluster and a few 1-hour e-mail compositions to Reshmi over the ALU. It took one few-hour sitting in lab to get synthesizing, and 3-ish few-hour revisions to find a timing scheme that worked. The memory mapper was drafted in a single ~3 hour sitting, tested in simulation for a few hours, and took another few hours in lab to get synthesizing. The self-preloading BRAM memories to *connect to* the memory mapper in lieu of real cartridge data in and of themselves took 2 attempts and a total of ~10 hours. One such memory remains in the final design as onboard RAM.

As far as project time not spent on hardware, I sunk easily 10 hours into attempting to track down 9-pin "pirate" Famicom controllers, and then finding a site at which I could buy both a cartridge adaptor and a pair of controllers without having to buy a complete Famiclone system. I also did the entire proposal report and compiled the final report.

As far as course improvements, I mentioned this in my FCE, but it would be nice, potentially, to have a "member trade day" some time shortly after project proposals. While there are certainly some things AwesomeNES as a whole, and myself in particular could have done or not done to have made better progress over the semester, it did hurt us that we were doing a hardware-only project with a member who had never done any hardware development, both in that we had to devote time to teaching and taking on extra work, and in that at the end we had fewer people who knew any given module well enough to even *use* it, let alone debug it. If projects can be brainstormed on the basis of what resources groups have, but groups can be adjusted after it becomes obvious what resources projects actually need, I think all groups might have a better chance at meeting their goal.

Ray Ng

What I worked on for the project was the Picture Processing Unit of the NES. What I found trickiest about working on this part of the project was figuring out what the actual unit did. All the documentation on it was hard to understand, and often contained conflicting information. Randy and I spent the first month just trying to figure out and translate the documentation to an actual architecture. I would say at this point in the project, we spent approximately 10-14 hours a week going through sites trying to find more detailed information, drawing out possible datapaths, realizing that we completely misinterpreted things quite a few times, and having to start over with trying to figure out how things worked. The way that the NES handled backgrounds was particularly confusing, with all the clever tricks they used in order to save on memory, to maximize what they could do with the technology they had available to them, and a lot of the documentation seemed to assume that we had a firm understanding of how to program an NES game, so had more of an understanding of how it worked than we actually did. Figuring out all this was certainly a learning experience, and we learned to appreciate just how clever a piece of hardware the PPU really was.

By the time it came to starting to put everything together and synthesizing and testing on the FPGA board, the amount of time that we spent on this grew dramatically. Randy and I spent at least 10 hours a day in here starting at the beginning of November, and my estimates was we worked approximately 40-60 hours a week for the first few weeks of November. I also spent all of Thanksgiving Break here debugging the PPU. By the week of DR3, we were spending even more time in here, and I would guess we were spending 70-80 hours a week in here until the day of the public demo.

I had a lot of fun on this project, and I found the project to be very interesting. My biggest regret was that we were never able to get it fully working by the time of the public demo, despite getting so close.

Reshmi Rajan

I have had a very good learning experience with this project and hope to keep the process going. And learning Verilog was the first step towards the project that I took. I believe that I have improved considerably over the 3 months that we were working on the project.

First, I had been assigned the work of implementing the ALU. Once that was done, I started work on th pAPU. The documentation given on the net was of immense
help. Initially, our plan was to finish the entire sound module. But, later decided that we would try and finish the square channel first and then if time permits, get the others done. And by the project demo we had that square channel working.

My team mates Randy, Ray and Sean helped me a lot in improving my Verilog, using the simulator, and they were really patient enough to double check what I had done. I also got to learn certain nuances of the Xilinx ISE.

We hope to keep the working going on the project and try and get the

entire NES emulator together.

Randy Sinnott

This project had many obstacles we had to overcome. First, the PPU has never been fully documented, so we had to reconcile many documents and take educated guesses in places. Secondly, we ran into issues with the Xilinx tools synthesizing our hardware with different functionality than the Verilog we had coded. Third, as the only purely hardware project in the class, much of our learning had to be done on our own, as many members of the course had not seen the issues we were encountering.

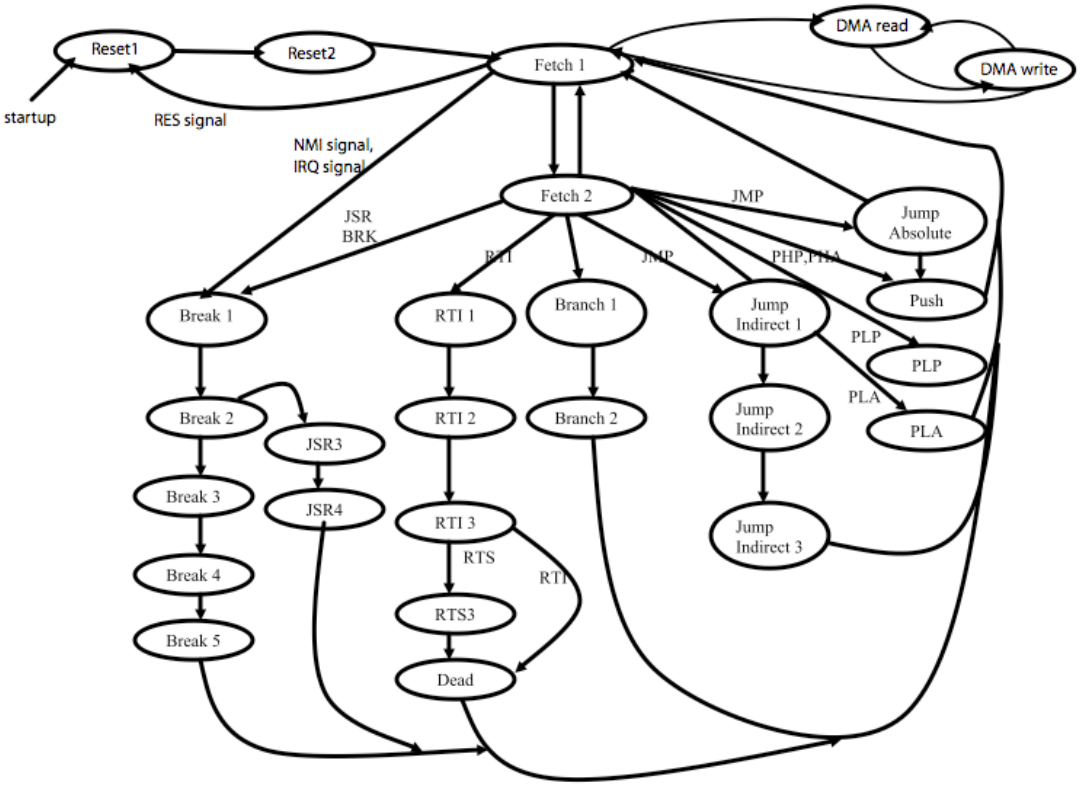
The components of the project I contributed to were: the PPU, the VGA module, the framebuffer module, the controller peripherals, the cartridge peripheral, and clock generation module. I also assisted Reshmi with debugging the pAPU's square wave module, which was the on.

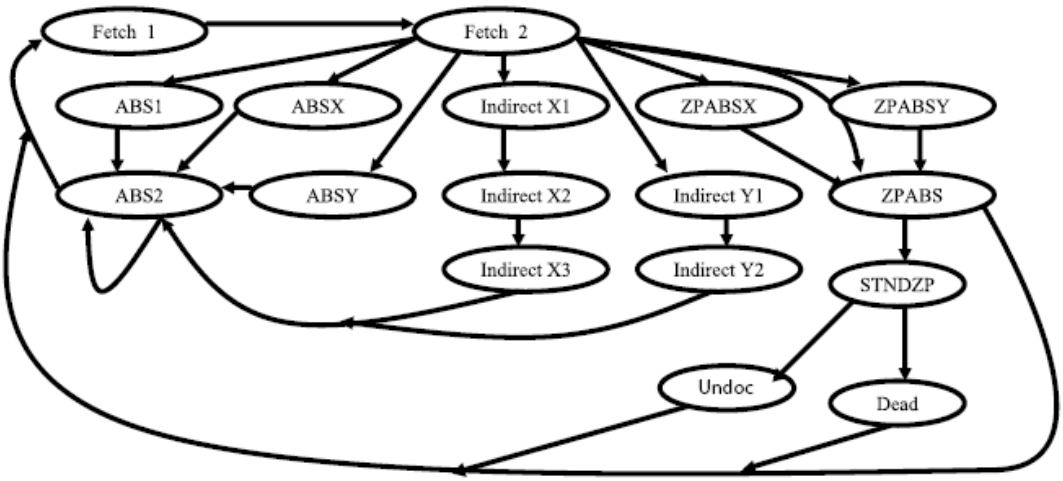
In round numbers, I spent about 10 hours per week during the months of September and October on the project. In November, we started working with the actual board, as opposed to purely simulation. From the beginning of November until the week of DR3, I would estimate I worked 30 to 60 hours per week. For the last two weeks of the project, I spent about 70 hours on the project each week.

I think this class was a good experience, and introduced me to some new areas I had not yet explored (using an FPGA, concepts computer architecture). On the downside, this project took more hours then we ever would have imagined.

One of the biggest improvements I can suggest would be a greater variety of labs. We wound up having to do a lot of learning about how to use Xilinx tools we were not exposed to in lab since we were doing a purely hardware implementation. Labs that demo using ISE instead of EDK, UCF files, and interfacing peripherals would have been helpful. We wound up reverse engineering many tools provided by other universities using the same board to learning how to do many of the things we did.

Appendix A: 6502 State Info





6502 State Function Summary

Key

($xxxx) = value at address
$xx = value

PC = program counter
PCL = lobyte of PC
PCH = hibyte of PC
IR = instruction reg
M = temp register (generic)
ND = temp register (usually for indirect addressing)
L = temp register (usually for low byte of 16-bit address)
DMA_hi = temp register storing the last value written to $4014
DMA_lo = temp counter register- goes from 00 to FF in DMA loop
Data = data inout bus
[comp] = result of IR-dependent computation on any 2 of X, Y, A, data
N/C/Z/V/D/I/B = individual status flags from ALU flag outputs
Flags = any or all flags from ALU flag outputs, dependent on IR
Status = entire status flag register to/from ALU data in/out

State	Function
Fetch 1	((PC) -> IR, PC++) or (set internal flags and enter NMI/IRQ) or (enter DMA loop)
Fetch 2	[comp] -> (M or X or Y or L or SP)/flags, PC++ if necessary
ZP-absolute	[comp] -> (M or X or Y or ND or data)/flags, PC++
ZP-abs/X	M+X->M
ZP-abs/Y	M+Y->M
Absolute 1	M+0->L
Absolute/X	M+X->L, PC++
Absolute/Y	M+Y->L, PC++
Absolute 2	(if L[8] then M+L->M/L else [comp]->(M or X or Y or ND or data)/flags), PC++
Indirect/X 1	M+X->ND
Indirect/X 2	(ND)+0->L
Indirect/X 3	(ND+1)+0->M
Indirect/Y 1	(M)+Y->L
Indirect/Y 2	(M+1)+0->M
Branch 1	L+0->PCL
Branch 2	(PCH)+1->PCH
Push	SP-1->SP
PLP	(SP)+0->status
PLA	(SP)+0->A/flags
BRK 1	PCL+0->(SP), SP--

BRK 2	PCH+0->(SP), SP--
BRK 3	status+0->(SP), SP--
BRK 4	(($FFFE) or ($FFFA))+0->PCL
BRK 5	(($FFFF) or ($FFFB))+0->PCH, *hack: clears internal interrupt flags*
JSR 3	(PC)+0->M, PC++
JSR 4	PC={M,L}
RTI 1	(SP)+0->status, SP++
RTI 2	(SP)+0->PCH, SP++
RTI 3	(SP)+0->PCL, clears internal in-interrupt flags
RTS 3	PC++
JMP absolute	(PC)+0->M, PC={data,L}
JMP indirect 1	(PC)+0->M
JMP indirect 2	({M,L})+0->PCL
JMP indirect 3	({M,L}+1)+0->PCH
StoreND	ND+0->((M) or ({M,L}))
Undoc	[comp usually involving ((M) or ({M,L}))]->(X or A or ((M) or ({M,L})))
Dead	
Reset 1	($FFFC)+0->PCL
Reset 2	($FFFD)+0->PCH
DMA read	({DMA_hi,DMA_lo})->M
DMA write	M->($2004), DMA_lo->DMA_lo+1

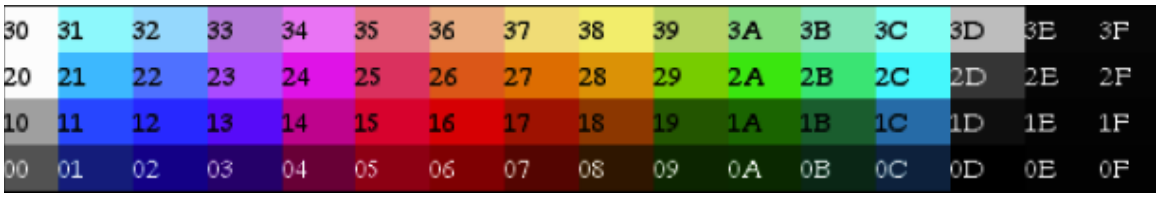
6502 Datapath Configuration Values:

Address Sources	Instruction Sources	SP Sources	PC
PC	ALU	SP+1	Hold
{M,L[7:0]}+1	Instr (hold)	SP-1	RST
DMA		SP (hold)	PC+1(PC_1)
Hard $2004		ALU	{M,L[7:0}
Hard $FFFA			ALU
Hard $FFFB			{data,L[7:0]}
Hard $FFFC			
Hard $FFFD			
Hard $FFFE			
Hard $FFFF			
{8'h00,M}			
{M,L[7:0]}			
{8'h00,M+1}			
{8'h00,ND}			
{8'h00,ND+1}			
{8'h01,SP}			

M Reg Sources	ALU Inputs	ALU Ops	ALU Output Latch-To
ALU	Hard 8'h00	ADD	NONE
Data	Hard 8'h01	CMP	X
	X	ORA	Y
	Y	AND	A
	A	EOR	Status
	Status	ADC	M
	M	SBC	L
	L	ASL	ND
	ND	ROL	PC (lo)
	PC (lo and hi)	LSR	PC (hi)
	SP	ROR	SP
	Data		Instr
	Hard 8'hFF		Memory*

* Not an actual latch. When LATCH_MEM is specified, ALU output is routed to the data lines and read-enable is lowered so that next memory access, ALU output will be written to memory.

Appendix B – NES to VGA color conversion



All Values are in Hexadecimal

NES Color	VGA Red	VGA Green	VGA Blue		NES Color	VGA Red	VGA Green	VGA Blue
00	80	80	80		**20**	FF	FF	FF
01	00	00	BB		**21**	00	95	FF
02	37	00	BF		**22**	6F	84	FF
03	84	00	A6		**23**	D5	6F	FF
04	BB	00	6A		**24**	FF	77	CC
05	B7	00	1E		**25**	FF	6F	99
06	B3	00	00		**26**	FF	7B	59
07	91	26	00		**27**	FF	91	5F
08	7B	2B	00		**28**	FF	A2	33
09	00	3E	00		**29**	A6	BF	00
0A	00	48	0D		**2A**	51	D9	6A
0B	00	3C	22		**2B**	4D	D5	AE
0C	00	2F	66		**2C**	00	D9	FF
0D	00	00	00		**2D**	66	66	66
0E	05	05	05		**2E**	0D	0D	0D
0F	05	05	05		**2F**	0D	0D	0D
10	C8	C8	C8		**30**	FF	FF	FF
11	00	59	FF		**31**	84	BF	FF
12	44	3C	FF		**32**	BB	BB	FF
13	B7	33	CC		**33**	D0	BB	FF
14	FF	33	AA		**34**	FF	BF	EA
15	FF	37	5E		**35**	FF	BF	CC
16	FF	37	1A		**36**	FF	C4	B7
17	D5	4B	00		**37**	FF	CC	AE
18	C4	62	00		**38**	FF	D9	A2
19	3C	7B	00		**39**	CC	E1	99
1A	1E	84	15		**3A**	AE	EE	B7
1B	00	95	66		**3B**	AA	F7	EE
1C	00	84	C4		**3C**	B3	EE	FF
1D	11	11	11		**3D**	DD	DD	DD
1E	09	09	09		**3E**	11	11	11
1F	09	09	09		**3F**	11	11	11