Team TNT

David Mohney
Michael Bailey
Archi Agarwal
Martin Rosenberg

18-545 Fall 2007

http://www.andrew.cmu.edu/~dmohney/TNT

Team TNT: Contents

Project Description **3**

CPU (6502) **3**
- MMU 3
- Registers 4
- Interrupts 4
- Addressing Modes 5
- Instructions 7

PPU (2C02) **8**
- PPU I/O Registers 8
- VRAM 11
  - NES Color Bits 13
  - Pattern Tables 13
  - Attribute Tables 14
  - Name Tables 15
- Datapath Details 15
  - Sprite Datapath 16
  - Background Datapath 17
- Rendering Details 18

The TNT NES Emulator **20**
- TNT Emulator Results 20
- Emulator Specific Modules 21
  - Clock Generation 21
  - VGA Output 22
  - Controller Interface 23
- Emulator Specific Implementation 23

NES Emulation Advice **23**

Individual Thoughts **25**
- Michael Bailey 25
- David Mohney 25
- Archi Agarwal 26
- Martin Rosenberg 27

Project Description

The general specification for picking a project in 18-545 was as follows: students were to design a video game. The game was required to have the following:

- Output to a video display
- Sound effects
- User input
- Support for multiple simultaneous players
- Scoring or victory conditions

Teams were then to design and implement the game in a manner that it could stand alone on the Xilinx Vertex-II Pro FPGA board. To meet these specifications, team TNT decided to create a hardware emulator for the Nintendo Entertainment System (NES). The original goal of the team was to create a full hardware implementation of an emulator which could read games in from a cartridge and take user input from original NES controllers.

The CPU (6502)

The processing for the NES is handled by the 8-bit 6502 processor. The processor had 56 different instructions, and 13 addressing modes, for a total of 151 different unique instruction / addressing mode combinations (see 'Instructions'). The 6502 is little-endian, meaning that data is stored least-significant byte first. 16-bit addresses are sent from the CPU to a memory mapper to determine the physical location of the memory at that area. While 16 KB of memory are addressable, many of the addresses are actually mapped to the same location. The mapping of the memory is described below.

MMU (Memory Mapping Unit)

The 16-bit CPU addresses are sent to the MMU to determine the physical location specified by the address. The memory map of the address space is as follows:

$0000 - $07FF: Internal CPU RAM (mirrored at locations $0800-$1FFF)
$2000 - $2007: PPU I/O Registers (mirrored at locations $2008-$3FFF)
$4000 - $4017: Internal APU Registers
$4018 - $5FFF: Cartridge Expansion Area
$6000 - $7FFF: Cartridge SRAM Area
$8000 - $FFFF: Cartridge PRG-ROM Area

The MMU uses the address from the CPU as an index into a lookup table, converts the address to a different form depending on which region it maps to, and sets control signals (read/write) to various portions of the NES, again depending on which

region the address maps to.

Registers

The CPU has 6 8-bit registers which it uses for various purposes. They are as follows:

PC: The 8-bit program counter, which stores the address of the instruction currently being executed.

SP: The 8-bit stack pointer, which points to the bottom of the stack.

X: The 8-bit X register, used to offset addresses for various addressing modes.

Y: The 8-bit Y register, used to offset addresses for various addressing modes.

A: The 8-bit accumulator register, used to perform mathematical and logical operations in various instructions.

P: The 8-bit status register, used to store various information about the status of the processor. The information in the status register is as follows:

Bit 7: N (set if the result of the last operation is negative)
Bit 6: V (set if the result of the last operation overflows)
Bit 5: Unused (always returns 0 on read)
Bit 4: B (Break, indicates if a break command has been executed, causing an IRQ interrupt)
Bit 3: D (Decimal mode: switches the 6502 into and out of BCD mode)
Bit 2: I (Interrupt disable: masks IRQ interrupts)
Bit 1: Z (set if the result of the last operation is zero)
Bit 0: C (set if the result of the last operation has a carry-out)

Interrupts

The 6502 can receive and handle three different types of interrupts: Non-Maskable Interrupt (NMI), Maskable interrupts (IRQ), and reset interrupts. They are described below.

NMI: Non-Maskable interrupt. This interrupt is sent by the PPU to the CPU when the PPU enters its VBlank stage. While the NMI cannot be masked, clearing bit 7 of PPU I/O register $2000 will disable the PPU from sending an NMI to the CPU on VBlank. When the CPU receives an NMI, the current state is saved and execution jumps to the address specified by the NMI handler address. This address is the value at address $FFFA

and $FFFB in the program. Execution continues through the handler until an RTI (return from interrupt) instruction is reached, and then the previous state of the CPU is restored and execution continues.

IRQ: Maskable interrupt. IRQ interrupts are sent to the CPU by various memory mappers. In addition, the BRK (break) instruction will cause an IRQ interrupt to be sent to the CPU. The CPU can mask IRQ interrupts by setting the interrupt disable flag (bit 2 of register P). When an IRQ is received, if interrupts are not disabled, the CPU saves the current state and jumps to the location specified by the IRQ handler address. This address is the value at address $FFFE and $FFFF in the program. Execution continues through the handler until an RTI is reached, and then the previous state is restored and execution continues as normal.

Reset: Reset interrupt. Reset interrupts are sent to the CPU when the system first starts and when the user presses the reset button on the NES. Reset interrupts cannot be masked by the CPU. When a reset interrupt is received by the CPU, the current state is saved and execution jumps to the reset handler, specified by the program at location $FFFC and $FFFD. The interrupt handler is followed until an RTI instruction is reached, at which point the previous state is restored and execution continues as normal.

Addressing Modes

Zero Page: Zero page addressing mode takes a single operand which is the lower byte of the address. The upper byte of the address is assumed to be zero (on the 'zero page').

Indexed Zero Page (X): Indexed zero page X addressing mode takes a single operand which is the lower byte of the base address. The provided byte is offset by the contents of the X register, and the resulting one byte (wrapped around on overflow) is the lower byte of the address. The upper byte of the address is assumed to be zero.

Indexed Zero Page (Y): Indexed zero page Y addressing mode takes a single operand which is the lower byte of the base address. The provided byte is offset by the contents of the Y register, and the resulting one byte (wrapped around on overflow) is the lower byte of the address. The upper byte of the address is assumed to be zero.

Absolute: Absolute addressing mode takes two operands which form the two bytes of the absolute address. Because the 6502 is little-endian, the lowest byte of the address is provided first.

Indexed Absolute (X): Indexed absolute X addressing mode takes two

operands, which form the two bytes of the base address (the first byte provided forms the lower byte of the base). The base is then offset by the contents of the X register to form the address (the sum wraps around on overflow).

Indexed Absolute (Y): Indexed absolute Y addressing mode takes two operands, which form the two bytes of the base address (the first byte provided forms the lower byte of the base). The base is then offset by the contents of the Y register to form the address (the sum wraps around on overflow).

Indirect: Indirect addressing mode takes two operands which form a 16-bit address. The address is then dereferenced, and the two bytes following the dereferenced address form the new address. When forming both addresses, the first byte is the byte with lowest priority.

Implied: Implied addressing mode takes no operands. The address is assumed by the name of the instructions.

Accumulator: Accumulator addressing mode takes no operands. Instructions using accumulator addressing mode operate directly on the accumulator register.

Immediate: Immediate addressing mode takes one operand. Instructions using immediate addressing mode operate directly on the one operand (no address is calculated).

Relative: Relative addressing mode takes one operand which is interpreted as a signed integer in the range -128 to 127. This integer is used as an offset from the current value of the program counter (after it is incremented following the instruction) to form the address. Relative addressing mode is used in branch instructions.

Indexed Indirect: Indexed Indirect addressing mode takes a single operand which forms the base address. The contents of the X register are then added to the base address with wraparound to form the lowest byte of the intermediate address. The upper byte of the intermediate address is assumed to be 0. The intermediate address is then dereferenced, and the two bytes immediately following the dereferenced location form the lowest and highest bytes of the address, respectively.

Indirect Indexed: Indirect indexed addressing mode takes in a single operand which serves as the lowest byte of a base address. The upper byte of the base address is assumed to be 0. The base address is then dereferenced, and the dereferenced value is offset by the contents of the Y register. The result forms the address.

Instructions

There are 56 unique instructions that can be executed by the CPU. They are listed below:

ADC Add Memory to Accumulator with Carry
AND "AND" Memory with Accumulator
ASL Shift Left One Bit (Memory or Accumulator)

BCC Branch on Carry Clear
BCS Branch on Carry Set
BEQ Branch on Result Zero
BIT Test Bits in Memory with Accumulator
BMI Branch on Result Minus
BNE Branch on Result not Zero
BPL Branch on Result Plus
BRK Force Break
BVC Branch on Overflow Clear
BVS Branch on Overflow Set

CLC Clear Carry Flag
CLD Clear Decimal Mode
CLI Clear interrupt Disable Bit
CLV Clear Overflow Flag
CMP Compare Memory and Accumulator
CPX Compare Memory and Index X
CPY Compare Memory and Index Y

DEC Decrement Memory by One
DEX Decrement Index X by One
DEY Decrement Index Y by One

EOR "Exclusive-Or" Memory with Accumulator

INC Increment Memory by One
INX Increment Index X by One
INY Increment Index Y by One

JMP Jump to New Location
JSR Jump to New Location Saving Return Address

LDA Load Accumulator with Memory
LDX Load Index X with Memory
LDY Load Index Y with Memory
LSR Shift Right One Bit (Memory or Accumulator)

NOP No Operation

ORA "OR" Memory with Accumulator

PHA Push Accumulator on Stack
PHP Push Processor Status on Stack
PLA Pull Accumulator from Stack
PLP Pull Processor Status from Stack

ROL Rotate One Bit Left (Memory or Accumulator)
ROR Rotate One Bit Right (Memory or Accumulator)
RTI Return from Interrupt
RTS Return from Subroutine

SBC Subtract Memory from Accumulator with Borrow
SEC Set Carry Flag
SED Set Decimal Mode
SEI Set Interrupt Disable Status
STA Store Accumulator in Memory
STX Store Index X in Memory
STY Store Index Y in Memory

TAX Transfer Accumulator to Index X
TAY Transfer Accumulator to Index Y
TSX Transfer Stack Pointer to Index X
TXA Transfer Index X to Accumulator
TXS Transfer Index X to Stack Pointer
TYA Transfer Index Y to Accumulator

The PPU (2C02)

The graphics processing for the NES is handled by the 2C02 graphics processing unit, known as the Picture Processing Unit (PPU). The PPU stores information about the graphics in a 16-kilobyte memory called VRAM. Further information about sprites is stored in a 256 Byte memory called SPR-RAM. The CPU can write to VRAM and SPR-RAM, as well as accessing other information relating to the PPU, through use of an 8 Byte register file, which maps to the region $2000-$2008 in the CPU's memory map. The registers are explained in detail below, as well as read/write accessibility by the CPU.

PPU I/O Registers

$2000: PPU Control Register #1 (Write only)

Bit 7: Enable NMI on VBlank (0 = Disabled, 1=Enbaled)
Bit 6: PPU Master/Slave Selection (0=Master, 1=Slave) (Not used in NES)
Bit 5: Sprite Size (0=8x8, 1=8x16)
Bit 4: Background Pattern Table Select (0=VRAM $0000, 1=VRAM $1000)
Bit 3: 8x8 Sprite Pattern Table Select (0=VRAM $0000, 1=VRAM $1000)
Bit 2: VRAM Address Increment Amount (0=Increment by 1, 1=Increment by 32)
Bits 1-0: Name Table Scroll Address (0-3=VRAM $2000, $2400, $2800, $2C00)

Register $2000 contains various control information set by the CPU and used by the CPU.

$2001: PPU Control Register #2 (Write Only)

Bits 7-5: Color Emphasis (Not used in emulator)
Bit 4: Sprite Visibility (0=Not Displayed, 1=Displayed)
Bit 3: Background Visibility (0=Not Displayed, 1=Displayed)
Bit 2: Sprite Clipping (0=Hide in left 8-pixel column, 1=No Clipping)
Bit 1: Background Clipping (0=Hide in left 8-pixel column, 1=No Clipping)
Bit 0: Monochrome Mode (Not used in emulator)

Register $2001 contains various control information set by the CPU and used by the CPU.

$2002: PPU Status Register (Read Only)

Bit 7: VBlank Flag (0=Rendering, 1=VBlank)
Bit 6: Sprite 0 Hit Flag (0=No Collision, 1=Sprite0/Background Collision)
Bit 5: Sprite Overflow Flag (0= At most 8 sprites, 1=More than 8 sprites on a scanline)
Bit 4: Ignore Writes to VRAM
Bits 3-0: Not Used

Register $2002 contains various status information set by the PPU and read by the CPU. The VBlank Flag (Bit 7) is cleared when $2002 is read by the CPU. Reading also resets the $1^{st}/2^{nd}$ write flag (see $2005 and $2006).

$2003: SPR-RAM Address Register (Write Only)

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

Register $2003 contains the address used by the CPU to access SPR-RAM during VBlank and used by the PPU to access SPR-RAM during rendering. Also contains the address in SPR-RAM to start writing to during a DMA operation. $2003 is automatically incremented every time $2004 is written to (but not when it is read from).

$2004: SPR-RAM Data Register (Read/Write)

Bits 7-0: Data to write to / Data read from SPR-RAM at address specified by $2003

Register $2004 contains the data returned from SPR-RAM on a read or the data to be written to SPR-RAM on a write at the address specified by $2003. Writing to $2004 also increments the value of $2003.

$2005: PPU Background Scrolling Offset (Write Only)

First Write:
Bits 7-0: Horizontal scroll index (X value 0-255)

Second Write:
Bits 7-0: Vertical scroll index (Y value 0-239)

Register $2005 is used to modify the contents of an internal 16-bit VRAM address pointer maintained by the register file (see 'VRAM Index Pointer' for details).

$2006: VRAM Address Register (Write Only)

Bits 7-0: Address to access VRAM (First write is upper 8-bits of address, second write is lower 6 bits)

Register $2006 is used by the CPU to specify what address to write to / read from during VBlank.

$2007: VRAM Data Register (Read/Write)

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

VRAM Index Pointer:

A one-bit register is stored in the register file to determine the parity of a particular access to register $2005 and $2006. In addition, the register file stores a 16-bit register used by the background renderer to update its scroll registers. The format of the 16-bit register is as follows:

yyyn nYYY YYXX XXX

X: The horizontal index of the tile to access in VRAM (0-31)
Y: The vertical index of the tile to access in VRAM (0-31)
n: The horizontal and vertical name table origin (0-3)
y: The fine vertical scroll offset, specifies which line of the accessed tile to use (0-7)

Each write to $2005 and $2006 update the 16-bit register, with the value of the one-bit register determining how the 16-bit register is updated. The

table below describes how the 16-bit pointer is updated by the writes to $2005 and $2006:

Pointer Bit	$2005 - 1st Write	$2005 - 2nd Write	$2006 - 1st Write	$2006 - 2nd Write
Bit 15: -	-	-	$2006 bit 7	-
Bit 14: y	-	$2005 bit 2	$2006 bit 6	-
Bit 13: y	-	$2005 bit 1	$2006 bit 5	-
Bit 12: y	-	$2005 bit 0	$2006 bit 4	-
Bit 11: n	-	-	$2006 bit 3	-
Bit 10: n	-	-	$2006 bit 2	-
Bit 9: Y	-	$2005 bit 7	$2006 bit 1	-
Bit 8: Y	-	$2005 bit 6	$2006 bit 0	-
Bit 7: Y	-	$2005 bit 5	-	$2006 bit 7
Bit 6: Y	-	$2005 bit 4	-	$2006 bit 6
Bit 5: Y	-	$2005 bit 3	-	$2006 bit 5
Bit 4: X	$2005 bit 7	-	-	$2006 bit 4
Bit 3: X	$2005 bit 6	-	-	$2006 bit 3
Bit 2: X	$2005 bit 5	-	-	$2006 bit 2
Bit 1: X	$2005 bit 4	-	-	$2006 bit 1
Bit 0: X	$2005 bit 3	-	-	$2006 bit 0

Table 1: VRAM Pointer Update Values

For example, on the first write to register $2005, the 7th bit of the value written to $2005 gets inserted in the VRAM pointer bit 4, which corresponds to the upper bit of the X tile index (see 'Background Rendering'). On the second write to register $2005, the 7th bit of the value written gets inserted in the VRAM pointer bit 9, which corresponds to the upper bit of the Y tile index. In addition, bits 2-0 of the first write to $2005 update the fine horizontal counter in the background renderer.

One other oddity of the NES is that there is only one one-bit parity register to determine whether a register is on the first or second write. As a result, writing to $2005 once causes the next write to $2006 to count as the second write.

VRAM

The NES' graphics information is stored in a 16-Kilobyte memory called VRAM. the layout of VRAM is as follows:

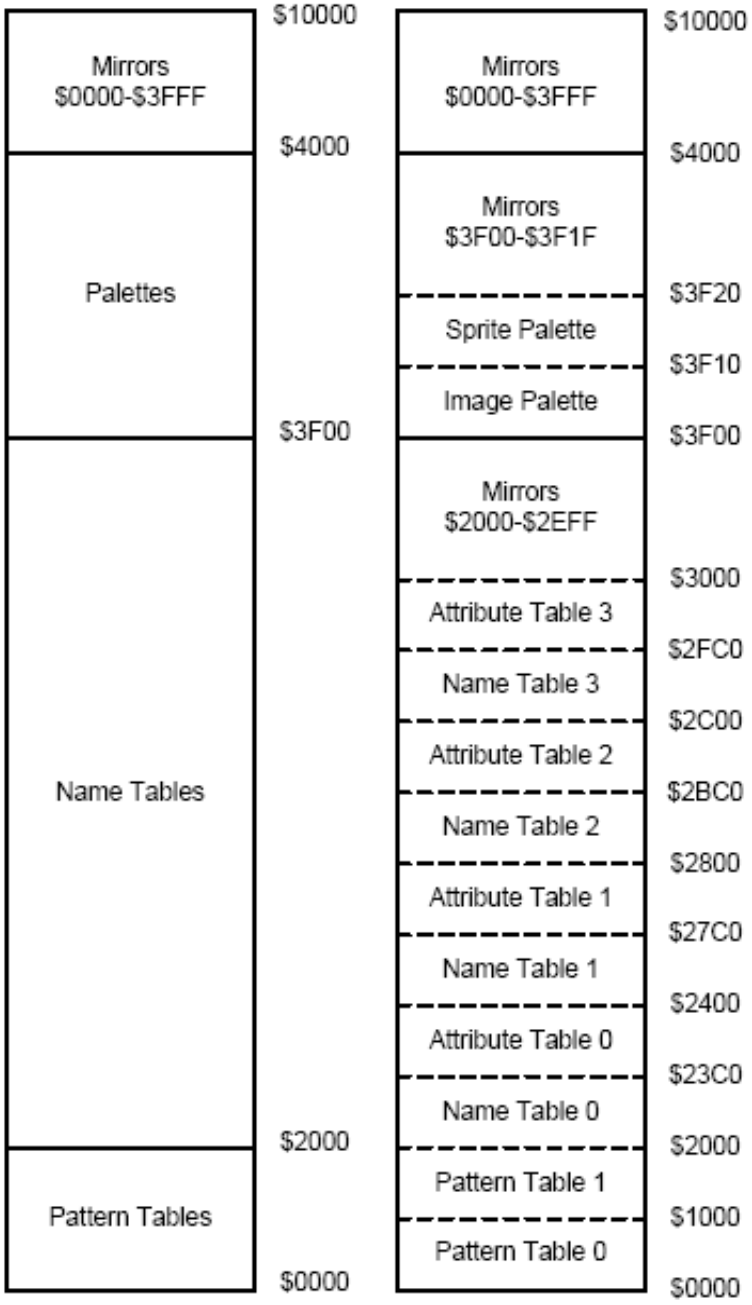

Figure 1: VRAM Memory Map

NES Color Bits

There are 64 NES system colors, specified by a 6-bit value. At any given point in time, sprites and the background can only use a subset of those colors as specified by their respective palette. The background palette consists of the 16 bytes starting at location $3F00. Each byte-long entry is the 6-bit NES system color index, as well as two bits used by the NES to specify the intensity of the color (these bits are unused by the emulator). The color at location $3F00 is the default background color, and is mirrored at locations $3F04, $3F08, and $3F0C. Therefore, the background can only use 13 colors. The sprite palette consists of the 16 bytes starting at location $3F10. The entries at locations $3F10, $3F14, $3F18, and $3F1C are used to indicate the sprite is invisible, so the sprites can therefore only use 12 colors.

For any given pixel displayed on the screen, its index into the sprite or background palette is determined by a 4-bit value. The lower 2-bits are determined by the pattern tables (see 'Pattern Tables'), and the upper 2-bits are determined by the attribute tables (see 'Attribute Tables').

Pattern Tables

The NES has two pattern tables, one at VRAM location $0000, and one at VRAM location $1000. Eeach 16-byte entry into the pattern table describes the lower 2-bits of the index into the palette table for an 8x8 pixel tile. The first 8 bytes contain the lowest bit, and the second 8 bytes contain the highest bit, as shown in an example below:

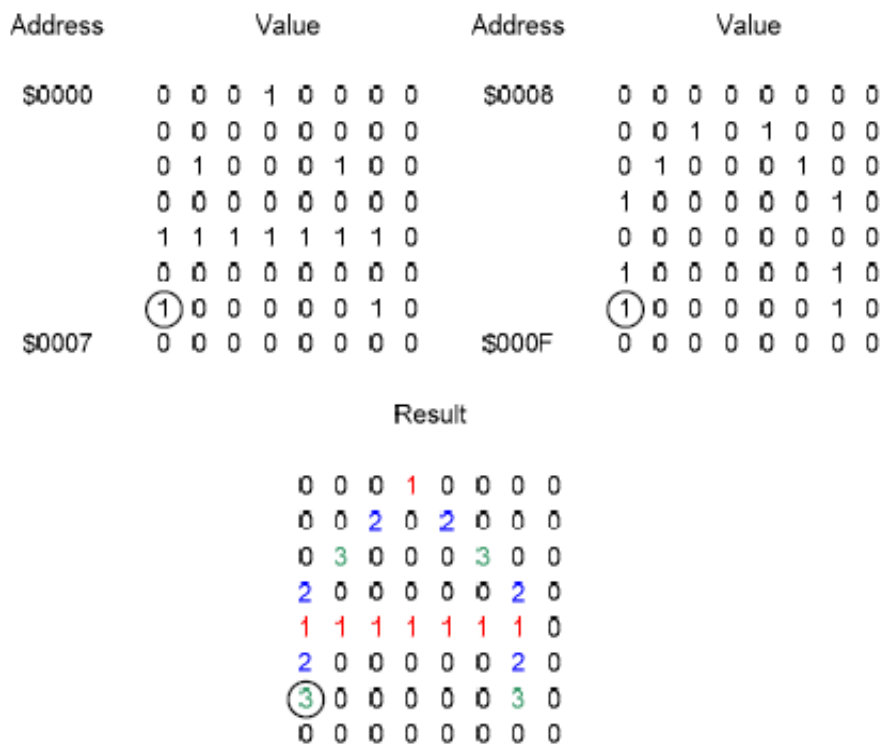

Figure 2: Pattern Table Example

As show in the example, the upper bit of the color is taken from the second 8-bytes ($0008 - $000F), and the lower bit of the color is taken from the first 8 bytes ($0000 - $0007). These two bits are concatenated together to form the lower two bits of the index into the palette table.

Attribute Tables

The upper two bits of the index into the palette tables is given by the attribute tables. There are 4 attribute tables in the NES, one at VRAM location $23C0, one at $27C0, one at $2BC0, and one at $2FC0. Each one-byte entry into the attribute table describes the upper two bits of the index into the palette table for 16 8x8-pixel tiles. The lowest two bits are used to describe the upper-left square of 4 tiles, the next two bits describe the upper-right square, the third two bits describe the lower-left square, and the highest two bits describe the lower-right square.

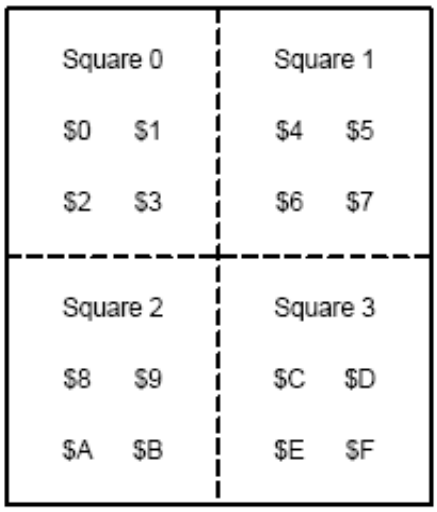

Figure 3: Attribute Table Example

In the example above, $0-$F each describe one 8x8-pixel tile. If the corresponding one-byte entry in the attribute table was 33221100 where 0, 1, 2, and 3 are arbitrary 2-bit values, then $C-$F would use 33 as the upper two bits of the index into the palette table, $8-$B would use 22, $4-$7 would use 11, and $0-$3 would use 00.

Name Tables

Each attribute table has a corresponding name table. The four name tables used by the NES are stored at VRAM locations $2000, $2400, $2800, and $2C00. Each one-byte name table entry acts as an index into either pattern table 0 or pattern table 1 to specify which pattern tile is used for an 8x8-pixel tile.

The NES only has space in its VRAM to store two name tables and two attribute tables; however, it can address four of each. Two name and attribute tables are mirrors of the other two, in a manner determined by the type of mirroring used by the game. If horizontal mirroring is being used, the first two name and attribute tables are the same, and the second two name and attribute tables are the same. If vertical mirroring is being used, name and attribute tables one and three are the same, and two and four are the same. Additionally, a game can use 4-way scrolling, in which each name table and attribute table stores unique information. In this case, the last two name and attribute tables are stored in internal memory on the cartridge.

Datapath Details

The datapath in the NES is broken up into two distinct sections, sprites and background, which are joined together at the end. The datapath for each section is described below.

Sprite Datapath

Information about the sprites is stored in SPR-RAM, a 256-byte memory that can be written to and read from by the CPU during VBlank. Each sprite is described by a contiguous 4-byte section, for a total of 64 sprites possible. The sprites are stored in order of priority, such that if two sprites contain overlapping non-transparent sections, the sprite that is first in SPR-RAM will contain priority over the other. Only 8 sprites can be displayed on a scanline; the first 8 sprites in SPR-RAM that are determined to be in range in a given scanline are used. The layout of the sprites in SPR-RAM is described as follows:

SPR-RAM

BYTE 0: Sprite Y-Position (Minus 1)

This byte stores the Y-Coordinate of the sprite on the screen, minus 1. The reason one is subtracted is so that the in-range evaluation can be simplified, as it happens one scanline before the sprite is actually rendered (see 'Rendering Details').

BYTE 1: Pattern Table Tile Index

This byte stores the index into the pattern tables to determine the bitmap for this sprite. For 8x8 sprites, bits 7-0 determine the index directly, and bit 3 of register $2000 determines which table to use. For 8x16 sprites, bits 7-1 determine the upper 7 bits of the index (both of the patterns at that index are used), and bit 0 determines which pattern table to use.

BYTE 2: Sprite Attribute Information

This byte stores various attribute information about the sprite, as described below:

Bit 7:	Flip Sprite Vertically (1=Flip, 0=Don't flip)
Bit 6:	Flip Sprite Horizontally (1=Flip, 0=Don't flip)
Bit 5:	Background Priority (1=Sprite is behind background, 0=Sprite is in front)
Bits 4-2:	Not used (always 0)
Bits 1-0:	Upper two bits of palette entry (correspond to attribute table information)

BYTE 3: Sprite X-Position

This byte stores the X-Coordinate of the sprite on the screen.

During the first stage of each scanline (see 'Rendering Details'), the sprite information is later transferred into a 24-byte internal region of temporary storage. Each sprite in this area is described by three contiguous bytes, for a total of eight sprites (corresponding to the 8 sprites that can be in range in each scanline). The information stored for each sprite is described as follows:

Sprite Temporary Storage

BYTE 0: Pattern Table Tile Index

This byte stores the index into the pattern tables to determine the bitmap for this sprite. This is copied directly from byte 1 in SPR-RAM.

BYTE 1: Sprite X-Position

This stores the X-Coordinate of the sprite on the screen. This is copied directly from byte 3 in SPR-RAM.

BYTE 2: Sprite Attribute Information

This stores various attribute information about the sprite. This attribute information is slightly different than the attribute information stored in SPR-RAM. It is described below:

Bits 7-4:	Lowest 8-bits of the in-range comparison (see 'Rendering Details')
Bit 3:	Flip Sprite Horizontally (1=Flip, 0=Don't Flip)
Bit 2:	Background Priority (1=Sprite behind background, 0=Sprite in front)
Bits 1-0:	Upper two bits of palette entry

From the temporary storage, the pattern is looked up in VRAM. The address of the sprite pattern in VRAM is determined as follows (depending on whether sprite size is 8x8 or 8x16):

8x8 sprites: {1'b0, $2000.3, Tile Index [7:0], High_Bit, Range[2:0]}
8x16 sprites: {1'b0, Tile Index[0], Tile Index[7:1], High_Bit, Range[2:0]}

High_Bit is a control signal from the FSM, which will first load the lowest bit of the palette table index, and then the highest bit. Range corresponds to the difference calculated in the in-range evaluation unit (see 'Rendering Details').

Afterwards, the pattern (along with the X-Coordinate, the attribute information, and the priority bit) are transferred into a series of buffers which hold the data until rendering occurs.

Background Datapath

During the render period and during the background prefetch (see 'Rendering Details'), the background renderer must look up first the pattern index in the name table, then the attribute information in the attribute table, and lastly it must look up the upper and lower byte describing the particular pattern tile indexed.

A series of counters determine which index in the name table to access in VRAM; the following describes the counters and their bit-widths:

FH[2:0]: The 3-bit fine horizontal offset used to determine which pixel of the pattern to select
FV[2:0]: The 3-bit fine vertical offset used to determine which line of the pattern to use
HT[4:0]: The 5-bit horizontal offset used to determine which horizontal tile to fetch
VT[4:0]: The 5-bit vertical offset used to determine which line to fetch a tile from
H: The 1-bit counter used to determine the lowest bit of which name table to use
V: The 1-bit counter used to determine the highest bit of which name table to use

The address looked up in the name table corresponds to the following:

{2'b10, Tile index (from name table) [7:0], High_Bit, FV[2:0]}

Because the fetching of sprites and background tiles occurs at different times, the High_Bit signal can be shared between the two.

The counters are chained together in the following order: FH, H, FV, V, HT, VT. When one rolls over to its max value (all ones for each counter except VT, which rolls over at 239), the next one in the list is incremented. In addition, the counters are updated on writes to $2006 and $2007 by the CPU, as determined by the VRAM pointer. The 'XXXXX' of the VRAM pointer corresponds to the HT counter, the 'YYYYY' corresponds to the VT, the 'yyy' corresponds to the FV, and the 'nn' corresponds to the FV and HV, respectively.

After the appropriate byte is fetched from the name table, it is stored in an 8-bit register, and the corresponding entry is fetched from the attribute table. At this point, the VT and HT counters select which two bits of the attribute byte to use, and the two attribute bits are stored in a 2-bit register. Then the two corresponding bytes are fetched from the pattern tables, as selected by the tile index from the name table and from the FV counter. Each of these pattern bytes are stored in 8-bit buffers. After all of the values are fetched, the attribute bytes, along with the two pattern bytes, are moved to a buffer. During render, the appropriate bits of the buffer are determined by the FH counter, which is incremented every render cycle. When the FH counter reaches 7, the next patterns and attribute bits are loaded from the registers into the buffers to be rendered.

Rendering Details

The rendering of each fram in the PPU is broken up into 261 scanlines. The first 20 scanlines (0-19) are referred to as the VBlank period. This is the only time when the CPU may access SPR-RAM and VRAM. The PPU rests during this period, and nothing

is rendered on the screen. The next scanline (20) is used by the PPU to load data from SPR-RAM into the Sprite temporary memory, and then to look up the patterns in VRAM and load the sprite buffers so the pixels will be available at the start of scanline 21. The first two tiles of the background for scanline 21 are also looked up in scanline 21. Nothing is rendered on the screen during scanline 20). The following 240 scanlines (21-260) are used to actually render pixels to the screen. The next scanline (261) is the last scanline. During this scanline, the PPU rests and prepares to enter the VBlank period.

Each scanline lasts 341 clock cycles. The PPU must look up 170 bytes in VRAM per scanline (except for scanlines 0-19 which look up nothing, scanline 20 which only looks up sprites and 2 background tiles, and scanline 261 which does nothing). Since each VRAM access takes two clock cycles, the scanlines can be evenly divided into 171 stages, one for each lookup plus a rest cycle at the end. The stages are described as follows:

Memory stages 1-128:

This is the period during which the PPU is actually rendering pixels on the screen. It lasts 256 clock cycles, during which one horizontal line (256 pixels) is drawn on the screen. During these memory stages, the background renderer is reading from VRAM to determine the name, attribute, and pattern table entries corresponding to each tile on the current scanline (except the first two tiles, which are fetched during stages 161-168 on the previous scanline). The address into the name tables in VRAM is determined by a series of scroll regesters (see 'Background Datapath' above). Once fetched, the background data is stored in the background buffers, where it is displayed. The fine horizontal scroll determines which pixel from the last buffer will actually be displayed on the screen.

The X-Coordinate value in the sprite buffers is decremented each cycle during this phase until it equals zero. At this point, the sprite starts displaying on the screen.

Also during these stages, the sprite renderer reads through SPR-RAM to determine which sprites are in range to be drawn during the NEXT scanline. The first 8 sprites determined to be in range are loaded into the sprite-temporary memory.

The in-range evaluation starts from the first sprite in SPR-RAM, and ends with the 64^{th} (each sprite is evaluated, even after 8 in-range sprites are found). The evaluation of each sprite takes 4 cycles, corresponding to reading each byte in SPR-RAM associated with that sprite. The following difference is calculated as an 'in-range test':

(Current_Scanline – 21) – (Sprite_Y_Coordinate + 1)

If this value is between 0 and 7, the sprite is determined to be in-range. It is then moved into sprite temporary storage, along with the lowest four bits of the in-range

test difference. If the vertical inversion bit is set, the lowest three bits of this difference are flipped to produce the vertical inversion effect. The first 8 sprites determined to be in range are stored into the memory at this point.

Memory stages 129-160:

During these stages, the sprite renderer loads the buffers with the bitmap pattern data. For each of the 8 sprites in temporary memory, the renderer first looks up the bitmap pattern in VRAM using the index stored at the first byte in temporary memory. For each sprite, two garbage VRAM accesses occur (these happen so that the Nintendo can reuse the hardware for the background fetching). Then the bitmap corresponding to the lowest bit is fetched, and then the bitmap corresponding to the highest bit is fetched. At this point, horizontal inversion is applied, and then the patterns are stored in the sprite buffers along with the attribute data and the priority bit.

Memory stages 161-168:

This is when the background renderer fetches the first two tiles to be displayed on the screen during the next scanline. It does this so that they can be loaded into the buffer and be ready to start displaying at the beginning of the render period. The fetching of these two tiles works the exact same way as it does in the first 128 memory stages: first the name table byte is retrieved from VRAM, then the attribute table byte, next the pattern table entry corresponding to the lower bit of the index into the palette table, and last the pattern corresponding to the higher bit.

Memory stages 169-170:

During this stage, the background renderer fetches two bytes from the name table. It is unclear as to why these two bytes are fetched from VRAM.

After stage 170 (clock cycle 341):

After the 170th memory fetch, the PPU rests for one clock cycle before beginning the next scanline.

The TNT NES Emulator

TNT Emulator Results

The original goal of the project was to be able to play NES games from a cartridge that connects to the board. In the end, support for cartridges was not able to be added. In their place, ROMs are downloaded onto the Desktop. These ROMS are run through a program that translates them into the PRG-ROM and CHR-ROM sections. The CHR-ROM and VRAM modules are then initialized to the values specified in the

files before the project is synthesized onto the board.

In addition, the pAPU (2A03) was not fully implemented. The following describes the role of the pAPU and team TNT's final progress in the module.

The 2A03 was the sound processor used by the NES. It actually contained the 6502 CPU, and had a several wrappers that controlled various frequency generators, and other components. The CPU memory addresses $4000 through $4020 are handled by the 2A03. Most of them simply control the audio processor, setting the volume, envelope decay, frequency, duration, etc. $4014 is special; it doesn't actually control the pAPU exactly, but it does initiate a DMA sequence lasting 512 cycles. The pAPU sections of the 2A03 are also responsible for generating the level sensitive IRQ's. Team TNT implemented the DMA engine as a module between the CPU and the MMU, with the pAPU as a separate module that would be accessed via the MMU.
As of right now most of the individual modules for controlling the pAPU have been written (duty cycle generator, triangle wave generator, envelope generator, frequency sweep generator, timing control, frequency control). Missing are a random noise generator and a decoder that has some interesting properties. Most of the components haven't been hooked up, and they are missing the dual ported ability to communicate with the CPU. After solving a similar problem with the PPU and CPU communicating with each other, any timing issues with the pAPU can be solved in a relatively small amount of time.

The NES emulator is designed to run on the Xilinx FPGA board. As a result, several components (such as the VGA connection) were unique to the emulator. Those modules are described below.

Emulator-Specific Modules

Clock Generation

The Xilinx board has one system clock that runs at 100 MHz. All other frequencies are generated through the use of the Digital Clock Manager (DCM) module (provided by Xilinx for use in XPS, adapted to be used in ISE). The various clocks used by the emulator, along with the method of generating them, is listed below.

VGA Clock: The VGA output is designed to run at 25 MHz; this is the least flexible clock in the design of the emulator. The VGA clock is generated by dividing the board's system clock by 4. The VGA base clock then serves as the system clock for the NES (although the NES' sytem clock ran at only 24 MHz).

PPU Clock: The PPU clock is obtained by dividing the system clock by 6, yielding a 4.17 MHz clock. While this is slower than the NES' PPU

clock frequency of 5.1 MHz, this is not a problem as the VGA RAM has 3 buffers to accommodate different clock frequencies.

CPU Clock: The CPU clock is obtained by dividing the system clock by 12, yielding a 2.53 MHz clock. This is only 2 times slower than the PPU, as opposed to the 3 times slower that the 6502 clock is from the 2C02 clock. This will also not cause a problem; it will only allow the CPU more time to write to VRAM during a VBlank.

VGA Output

The Xilinx boards are hooked up to a 640 x 480 VGA computer monitor. The monitor is hooked up to a 25 MHz clock. In addition to the clock, it has 8-bit ports for red, green, and blue pixel values, as well as one-bit ports for an h-sync signal and a v-sync signal. During each of the first 640 VGA clock cycles, the value received by the red, green, and blue ports will determine the pixels displayed on the screen. After 640 cycles (corresponding to 640 horizontal pixels or one line), an the h-sync signal is asserted and the red, green, and blue values must be zero. Once the h-sync signal goes low, the screen starts accepting red, green, and blue values corresponding to the pixels on the next line. Once all 480 lines have been drawn, the v-sync signal is asserted. Again during this time, the red, green, and blue values must be zero. If any of the color values are nonzero during h-sync or v-sync, the monitor will not receive a video signal.

In addition, the timing of the signals must correspond to a preset timing pattern. Xilinx provides a module in the XPS libraries to generate the correct timing signals. Because the module is designed for XPS, the module was ported from the XPS libraries to work in ISE and trimmed to fit the needs of the emulator. Most of the unnecessary signals were stripped out (although some still remain), and the VGA RAM module used by XPS was modified to fit the needs of the emulator.

The VGA RAM module is implemented as a series of 256 x 240 x 6-bit dual-ported block rams. Each 6-bit entry corresponds to the NES system color to be represented at that location on the screen. The PPU writes to the RAM module during its render stage, and after it has written valid data to one entire buffer, the VGA module reads from that buffer. Because the VGA reads quicker than the PPU writes to the RAM, three buffers are necessary to completely prevent tearing. The VGA module reads from one buffer behind the buffer that the PPU is writing to, sometimes rendering the same buffer twice in a row to stay behind. When the PPU is done writing a buffer, it starts writing to the next buffer while the VGA finishes rendering from its current buffer, when it moves on to the one the PPU has just finished writing to.

Because the NES graphics are at a 256 x 240 resolution, each pixel is rendered 4 times to create a 512 x 480 resolution. The remaining 128 horizontal pixels on the screen are rendered in black, 64 before the start of the NES screen and 64 after.

Controller Interface

The CPU obtains user input information by querying to memory mapped I/O ports (address locations $4016 and $4017). To read the controller input data the CPU must first write a 1 and then a 0 to the address. After these signals have been asserted the CPU can then read the data back from the I/O port. The data is returned serially with one button data value returned per read. The data is return in the following order: {A, B, SELECT, START, UP, DOWN, LEFT, RIGHT}.

To actually obtain the data from the input controller continually poll the controller and store the input data in registers that the CPU will have access to when reading from the I/O ports. To accomplish polling an NES controller input finite state machine implementation was taken from a previous mit project. The source used can be found at the following link (http://web.mit.edu/6.111/www/s2004/PROJECTS/2/nes.htm). The general state progression from this source was correct but some alterations to the code had to made for the module to work for our implementation.

Emulator-Specific Implementation

The main goal when developing the NES emulator was to make it as similar as possible to the original NES; however, some portions of the NES were either undocumented or unclear, and therefore the emulator differs from the original system in some ways. Differences are listed in this section.

CPU: The main difference between TNT's emulator and the actual NES are cycle counts for instructions. While most instructions maintain the same cycle counts as the original system there are a few which do match the specified cycle count. As with most disparities in the emulator implementation, given more time the these discrepancies could easily be removed.

PPU: The emulator's PPU follows documented datapath fairly closely. Some features were omitted to allow time to get a simple ROM working. Most notable, sprite 0 hit detection and some of the memory mirroring was not implemented.

Final Scope: Due to time constraints the final emulator implementation lacked sound capability and the game cartridge interface. The emulator can however run downloaded game ROMs in the .NES format. A graphics demo and a maze game were successfully downloaded and ran on the emulator. Further game ROM testing was not completed due to lack of time.

NES Emulation Advice

The two most difficult parts of creating a hardware emulator of the NES are understanding how the NES works and testing the pieces of the emulator. Before starting to create an emulator, it is important to have thoroughly read through and understood as much documentation about the NES as possible. It is also helpful to notice where discrepancies exist in various documentations; that way, when creating the emulator, the designer can be mindful as to what may or may not be correct.

It is not vital to completely understand everything about the NES before starting to write the emulator; because the original NES was designed for performance, there are many hacks that will not make sense until the emulator designer has gotten to a particular point in the emulator design. It is however important to understand at a higher level how the NES is put together, the components involved and how they interface with one another.

When testing the various components of the NES, it is a good idea to thoroughly test each component alone and to be confident that each component works before starting to integrate the various components. Sufficient time should be left for integration; once the PPU and the CPU are communicating through the PPU's I/O registers, it is very likely that there will be many bugs that were not found during unit testing.

There were several pieces of information that became apparent throughout the course of the project. Among them are:

- Synthesis is not simulation. Often times, a particular component will simulate properly and will appear to work on ModelSim. Unfortunately, this does not mean that it will synthesize correctly. Furthermore, even if it does synthesize correctly, it is not necessarily true that it will perform the same before and after synthesis. Therefore, it is important to test thoroughly after synthesis to ensure that the components are fitting together correctly. In addition, the XST synthesizer will attempt to optimize away parts of the design that are unnecessary. Therefore, when testing an individual module with a stub to simulate the rest of the design, the stub will often be optimized away and this will sometimes cause the appearance of the design that is being tested not working. The lesson to be learned is that synthesis can change many parts of a design, and it is important to read the synthesis report to understand what exactly it is doing.

- Read the documentation carefully. In many cases, especially in the PPU, the functionality of the NES (or emulator) depends on a small detail of the design that could easily be misunderstood or even ignored. Reading many different documents allows the emulator's designer to catch as many of these details as possible. While designing, documentation should be referred to often.

- Read the XST synthesis documentation. There is a very good document describing synthesis in XST. The most important part of the document is the portion that describes how to design for the synthesizer. Particularly, RAMs and FSMs should be designed in a manner such that the synthesizer will recognize them and synthesize them in an optimal fashion.

Individual Thoughts

Michael Bailey (mbailey)

The NES project was a very interesting and very fun project. For this project I worked primarily on the PPU. Dave and I co-wrote the entire PPU datapath and control over the first half of the semester. To accomplish this, we spent a significant portion of time sifting through documentation and trying to understand all of the memory optimizations the NES system used. Once we completed this portion of the project, I began working on the controller interface and the PPU/CPU interface. The controller took about a week to get working. This was mostly due to using a broken controller at the beginning of testing. The PPU-CPU I/O interface took about the same amount of time. It was difficult at first managing the interface with two clocks, but eventually everything was debugged and working properly. After these tasks were completed, we began overall integration. I worked with archi debugging some to the interrupt functionality of the CPU and then finally debugging the PPU. This took lots of time especially since synthesis took close to twenty minutes per try. Early in the semester I worked about 10 to 20 hours per week on the project and in the last month I put in about 60 hours per week. Also after the public Demo Dave and I put in about 20 hours worth of work to get the ROMs to actually work. Not all of this time was spent working however since synthesis took so long.

I think the class was set up very well and the TAs were especially helpful. My only regret was not knowing enough about the FPGA board when specifying our initial design. If we would have specified our RAM modules to target the block RAMs from the beginning of the project, we would have had much less to debug and would have been able to demo better games. However, due to this lack of foresight, we scrambled over the last three weeks to tweak all of our modules and timings so we could utilize the block RAMs and no just the logic fabric. This was a tramendous setback, and made the first half of semesters work useless.

The only change I would suggest is maybe a lecture dedicated to verilog design specific to the FPGA where FSM and RAM design is discussed. This may not be necessary for most groups, but any group using ISE would find this information very helpful. Having this knowledge early in the semester would have been invaluable.

David Mohney (dmohney)

Before taking this course, I had very little (240 level) experience with synthesizable verilog. Because our project was implemented largely in Verilog, the course helped me grasp the difference between simulatable verilog and synthesizable verilog.

From the start of the semester, team TNT split up into the PPU team, which I was a part of, and the CPU team, which I was not. Michael and I worked together in implementing the PPU. The first approximately third of the class was spent getting used to how the PPU works, and reading through various different documentations and noting discrepancies in them. Once we actually started coding the PPU, things went pretty quickly. Both Michael and I worked on most components of the PPU together, and spent a few hours per week throughout the beginning of the semester. Starting about a week or two before Thanksgiving break, we started spending considerably more time in lab, pulling the occasional all-nighter working on various parts of the PPU. I then started working on the VGA output while Michael started working on the controllers and the PPU IO registers. During the last week of the project, the PPU and CPU teams worked together to attempt to integrate the PPU and CPU together, and to debug the various problems we encountered during integration. The team spent most of the entire week in lab during this week. In the week following the demo (during which we did not demo because our project was broken), Michael and I continued to work on the project to get ROMs to work. In the final couple of days, Michael continued to debug the PPU while I compiled the final report and created the team's website.

This class was a very positive experience for me and introduced me to the exciting field of FPGA design. If I had the semester to do over again, there is absolutely nothing that I would change.

Archi Agarwal (archia)

I had a good learning experience with this project. I took this course so as to improve my verilog coding skills. I had done verilog coding before but this was the first time I was doing a big project in verilog. But soon I realized that there was much more in this project than just verilog coding. This class was good and introduced me to FPGA concepts and also computer architecture.

My teammate Marty helped me a lot with Verilog coding. Initially we discussed the state diagram of the entire CPU. The document provided on 6502 on web is of immense help. I initially wrote the verilog code for ALU and tested it by simulating on Modelsim and writing small test benches in machine code. After ALU, I started with CPU. Starting with 140 states we finally reached 25 states and in this Marty was of great help to me. I simulated the whole CPU and initially tested each addressing mode and each instruction separately. After that I wrote small test cases like generating multiple of 2 or generating a particular sequence of numbers and tested the CPU against these test benches. Initially I started by working a couple of hours per day. However this was my first experience of verilog coding on such a big project so I had to work nearly 6-7 hours everyday. It took me a long time to figure out even

small problems. I believe my verilog coding has improved a lot over this semester.

Then I started working on interrupts. I wrote verilog for IRQs and NMI interrupts and simulated the code using Modelsim and wrote a testbench for it. I also helped a bit in the integration of CPU and PPU working mainly on the CPU part, debugging using modelsim. At the last moment CPU verilog was changed by Marty so as to fix the timing problem and I tested it against small testcases. I do wish I knew how to use Xilinx FPGA before. Since I had never worked on FPGA, I could not work much on the board. However Marty, David and Michael helped me a lot in understanding the working of board and I am thankful to them for that. I also wish we had labs that gave demo as to use ISE. After doing almost all labs we had to learn ISE on our own.

Martin Rosenberg (mjrosenb)

For the 18-545 group this semester, I was on the subteam that created the 6502 processor. The 6502 was first created in the late 1970's. As such, they did a large number of things that probably haven't been done since the mid 1980's. Initially our sub-team divided the processor into three main components, a decoder, an ALU, and a fsm/datapath. I spent about two days translating a .GIF of the ISA into a decoder that was woefully incomplete. After this, archi and I spent a few days attempting to get a general overview of how we were going to create the processor. This presented some interesting challenges, because the CPU had not only an ISA that we had to follow, but it also had a timing restrictions. Since the processor isn't a MIPS style processor that has limited addressing modes, it was kind of obvious that we needed to have a FSM & datapath as opposed to the single cycle processor that is created in 18-447 that has no internal state that isn't visible to the programmer. Never having created such a processor before, I was at a bit of a loss as how to go about designing a datapath and FSM. I tried designing a datapath first, then make an FSM using the datapath. There were some issues related to not having enough of the datapath to use in the FSM, or not being able to meet timing restrictions with the FSM. In the end, we opted to write all of the operations in a RTL form explicitly specifing a number of temporary registers, then deriving an FSM from the list of all instructions, and the datapath from the set of operations that were being done. There were a bunch of instances where in order to make our work easier, we didn't try to optimize our design, and used 3 adders when 1 would have been sufficient. The primary goal was to get each instruction to run in an equal number of cycles if not lower. The though process behine this was that we could always add a few cycles of 'do nothing' to the processor. Initially, we had designed the processor to have zero cycle memory access time, which in reality just meant that it would take any amount of time less than a cycle. Based on last year's group using DDR memory, I thought that this was acceptable. It wasn't until a while later that we were able to figure out how to make everything work with a combination of pre-fetching and adding nops into the FSM. After the CPU had passed a suite of tests that we wrote, I fixed the CPU so that it could work on both modelsim and on xilinx. The fun step was adding in a memory mapper so that the CPU thought it was always talking directly to a bank of RAM, when in fact depending on the addresses, it would talk to RAM, ROM, the PPU, or

other memory mapped registers. Since Archi didn't have a large amount of experience writing verilog, I let her do most of the testing, while I continued to implement new instructions (I also procrastinated on writing both BRK and RTI [return from interrupt]). The NMI and IRQ lines were a bit of a hack at the end, where rather than latching the next instruction and state, I'll replace that with the middle of the BRK instruction, and jump into the apropriate state, rather than ever entering the decode state. After the CPU was moved onto the FPGA, I focused primarialy on debugging. I found that due to the lack of a traditional FSM, and datapath, it was easy to make changes to it's structure. There were relatively few bugs added in by the switch from DDR to traditional RAM. I spent a decent amount of the time during the semester just making various small modules, such as the clock modules, the interconnects, and reformatting the top module into a more sane format. Unfortunately, there were a few weeks where I wasn't able to do any work (family emergencies, job interviews, more job interviews, and some computer problems). For the last day or two, there was nothing noticably wrong with the processor, and I felt that I was at a bit of a loss to help with the PPU. There were several times throughout the semester that I was too tired to reliably work on the cpu, so I started writing the APU. I had all of the basic code for the APU, however there were some interesting timing issues that existed with the audio processor. The system clock ran several timmes faster than some of the internals of the audio, yet there were registers that needed to be written to by both the audio processor and the CPU. At this point, I think it would take me a couple of days to complete the APU and integrate it into the rest of the NES.