FPGA Implementation of the Nintendo Entertainment System (NES)

Team Name: Four People Generating A Nintendo Entertainment System

Pavan Holla, Eric Sullivan, Jonathan Ebert, Patrick Yang

Architectural Document

University of Wisconsin - Madison

Spring 2017

Table of Contents

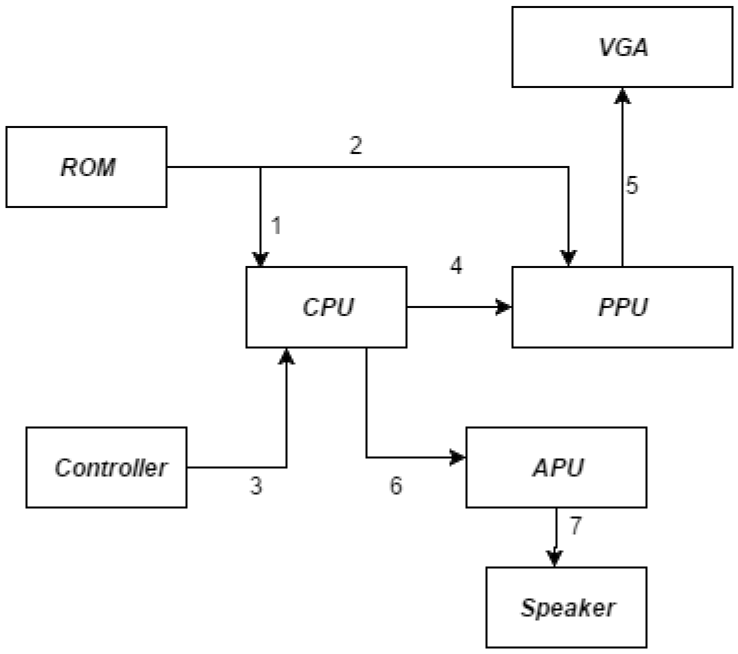
1. Introduction

Following the video game crash in the early 1980s, Nintendo released their first video game console, the Nintendo Entertainment System (NES). Following a slow release and early recalls, the console began to gain momentum in a market that many thought had died out, and the NES is still appreciated by enthusiasts today. A majority of its early success was due to the relationship that Nintendo created with third-party software developers. Nintendo required that restricted developers from publishing games without a license distributed by Nintendo. This decision led to higher quality games and helped to sway the public opinion on video games, which had been plagued by poor games for other gaming consoles.

Our motivation is to better understand how the NES worked from a hardware perspective, as the NES was an extremely advanced console when it was released in 1985 (USA). The NES has been recreated multiple times in software emulators, but has rarely been done in a hardware design language, which makes this a unique project. Nintendo chose to use the 6502 processor, also used by Apple in the Apple II, and chose to include a picture processing unit to provide a memory efficient way to output video to the TV. Our main goal is to recreate the CPU and PPU in hardware, so that we can run games that were run on the original console. In order to exactly recreate the original console, we will also need to include memory mappers, an audio processing unit, a DMA unit, a VGA interface, and a way to use a controller for input. In addition, we will be writing our own assembler for the 6502 that will allow us to create simple programs to test our implementation.

Due to the complexity of the project, work will start on the CPU and PPU. A few weeks in we will consider how much progress has been made to the PPU, and if we don't think that we will be able to finish it in a timely manner, we will instead use another implementation that we found online. By spending time creating our own PPU, we will gain valuable insight that will allow us to better integrate the whole console together. From here, we will begin working on other blocks of the project, including the controller, the memory mapper, and the APU (if time permits). Ultimately, the goal is to get an NES game running on our FPGA.

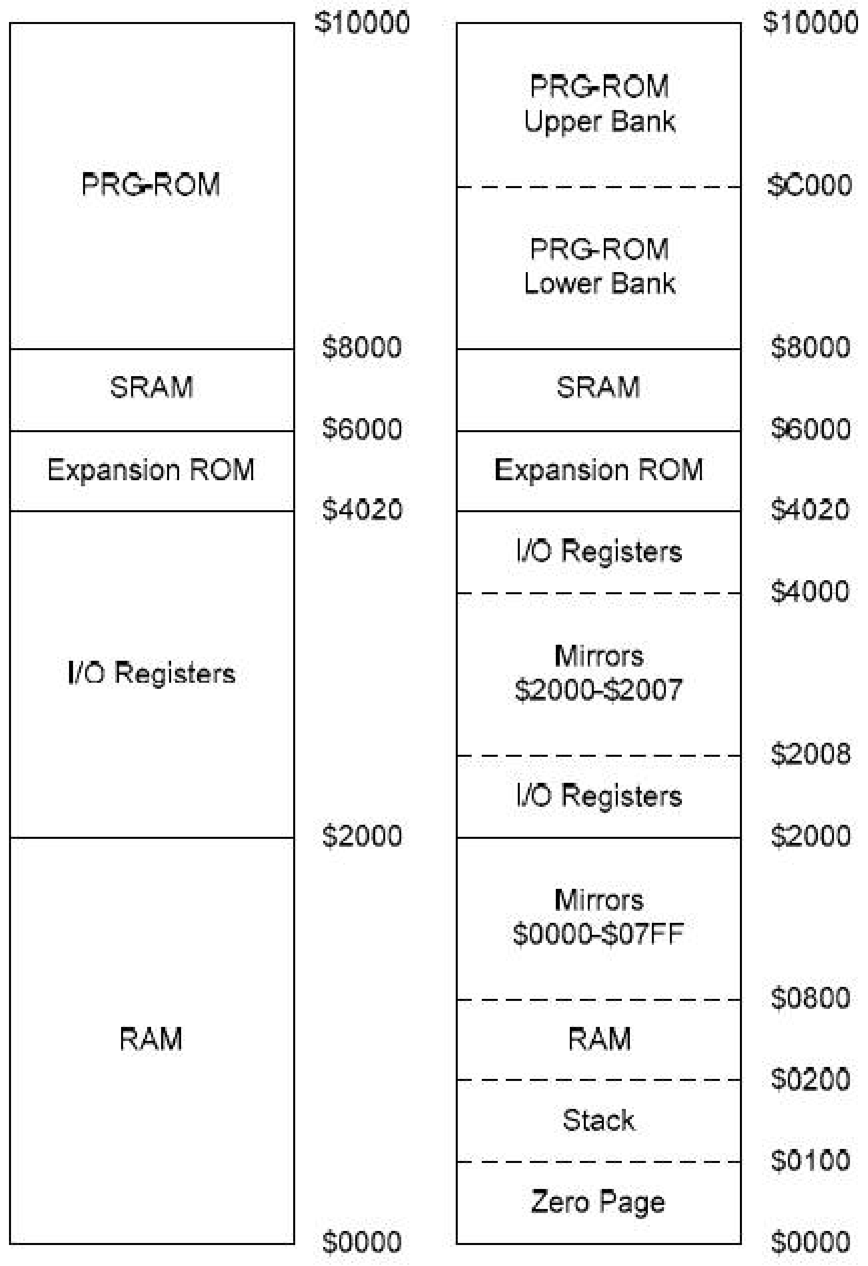
2. Hardware



These arrows are to be explained in the "Interface document".

2.1. CPU

2.1.1. Memory Map



2.1.2. CPU Registers

The CPU of the NES is the MOS 6502. It is an accumulator plus index register machine. There are five primary registers on which operations are performed:

1. **PC**
2. **Accumulator(A)**
3. **X**
4. **Y**

5. **Stack pointer**
6. **Status Register**

2.1.3. CPU ISA

The ISA may be classified into a few broad operations:

- Load into A,X,Y registers from memory
- Perform arithmetic operation on A,X or Y
- Move data from one register to another
- Program control instructions like Jump and Branch
- Stack operations
- Complex instructions that read, modify and write back memory.

2.1.4. CPU Addressing Modes

Additionally, there are thirteen addressing modes which these operations can use. They are

- **Accumulator** – The data in the accumulator is used.
- **Immediate** - The byte in memory immediately following the instruction is used.
- **Zero Page** – The Nth byte in the first page of RAM is used where N is the byte in memory immediately following the instruction.
- **Zero Page, X Index** – The (N+X)th byte in the first page of RAM is used where N is the byte in memory immediately following the instruction and X is the contents of the X index register.
- **Zero Page, Y Index** – Same as above but with the Y index register
- **Absolute** – The two bytes in memory following the instruction specify the absolute address of the byte of data to be used.
- **Absolute, X Index** - The two bytes in memory following the instruction specify the base address. The contents of the X index register are then added to the base address to obtain the address of the byte of data to be used.
- **Absolute, Y Index** – Same as above but with the Y index register
- **Implied** – Data is either not needed or the location of the data is implied by the instruction.
- **Relative** – The content of sum of (the program counter and the byte in memory immediately following the instruction) is used.
- **Absolute Indirect** - The two bytes in memory following the instruction specify the absolute address of the two bytes that contain the absolute address of the byte of data to be used.
- **(Indirect, X)** – A combination of Indirect Addressing and Indexed Addressing
- **(Indirect), Y** - A combination of Indirect Addressing and Indexed Addressing

2.1.5. CPU Interrupts

The 6502 supports three interrupts. The reset interrupt routine is called after a physical reset. The other two interrupts are the non_maskable_interrupt(NMI) and the general_interrupt(IRQ). The general_interrupt can be disabled by software whereas the others cannot.

The below table outlines the ISA.

The opcode table can be found at http://www.e-tradition.net/bytes/6502/6502_instruction_set.html

2.1.6. ISA Table

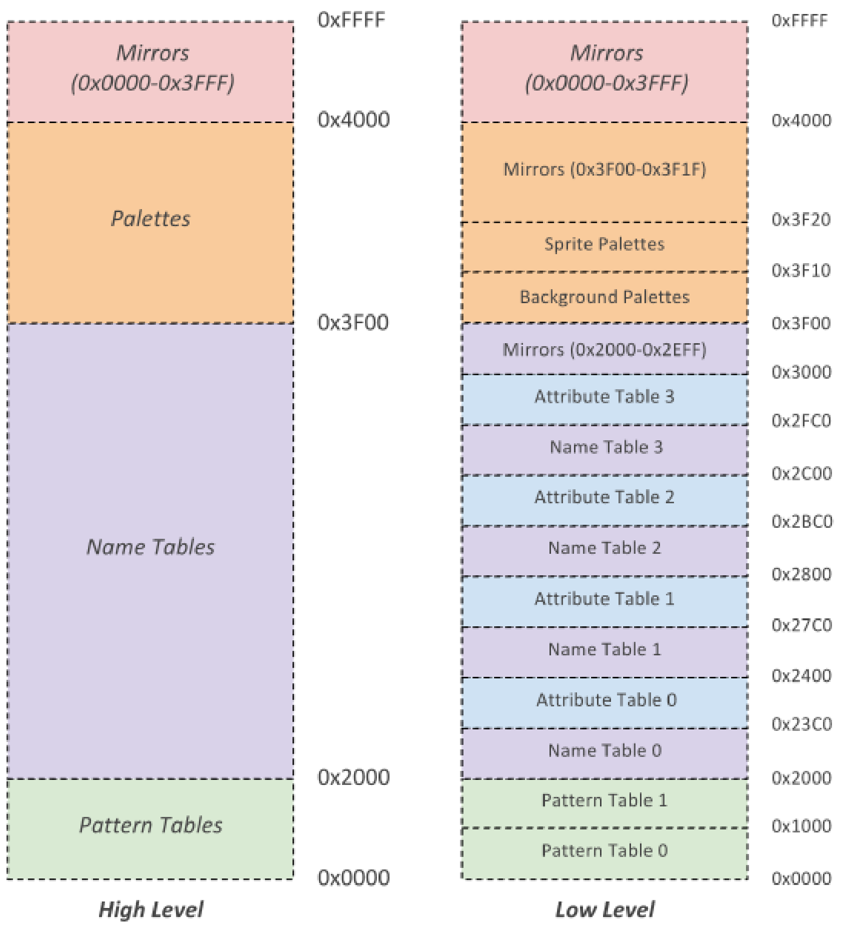
ADC	**add with carry**	JSR	jump subroutine
AND	and (with accumulator)	**LDA**	**load accumulator**
ASL	arithmetic shift left	LDY	load X
BCC	branch on carry clear	LDY	load Y
BCS	branch on carry set	LSR	logical shift right
BEQ	**branch on equal (zero set)**	NOP	no operation
BIT	bit test	ORA	or with accumulator
BMI	branch on minus (negative set)	PHA	push accumulator
BNE	branch on not equal (zero clear)	**PHP**	**push processor status (SR)**
BPL	branch on plus (negative clear)	PLA	pull accumulator
BRK	interrupt	PLP	pull processor status (SR)
BVC	branch on overflow clear	ROL	rotate left
BVS	branch on overflow set	ROR	rotate right
CLC	clear carry	**RTI**	**return from interrupt**
CLD	clear decimal	RTS	return from subroutine
CLI	clear interrupt disable	SBC	subtract with carry
CLV	clear overflow	SEC	set carry
CMP	compare (with accumulator)	SED	set decimal
CPX	compare with X	SEI	set interrupt disable
CPY	compare with Y	**STA**	**store accumulator**
DEC	decrement	STX	store X
DEX	decrement X	STY	store Y
DEY	decrement Y	**TAX**	**transfer accumulator to X**
EOR	exclusive or (with accumulator)	TAY	transfer accumulator to Y
INC	**increment**	TSX	transfer stack pointer to X
INX	increment X	TXA	transfer X to accumulator

INY	increment Y	TXS	transfer X to stack pointer
JMP	jump	TYA	transfer Y to accumulator

2.2. Picture Processing Unit (PPU)

The PPU is responsible to all of the drawing logic for video output. It contains data about both background tiles and sprite data. To draw this data to the screen the background and sprite data are fetched for every pixel and then a mux decides what data to send to the video out. Overall the logic of how the drawing is done is not too complex, but the protocols for obtaining this information from RAM/ROM is difficult because the PPU has to work between the two different clock domains of the CPU and memory system.

2.2.1. PPU Memory Map



2.2.2. PPU Registers

- Control registers are mapped into the CPUs address space ($2000 - $2007)
- The registers are repeated every eight bytes until address $3FFF

- **PPUCTRL[7:0]** ($2000) WRITE

        - [1:0]: Base nametable address (Also something about PPU scroll)
            - 0: $2000
            - 1: $2400
            - 2: $2800
            - 3: $2C00
        - [2]: VRAM address increment per CPU read/write of PPUDATA
            - 0: Add 1 going across
            - 1: Add 32 going down
        - [3]: Sprite pattern table for 8x8 sprites
            - 0: $0000
            - 1: $1000
            - Ignored in 8x16 sprite mode
        - [4]: Background pattern table address
            - 0: $0000
            - 1: $1000
        - [5]: Sprite size
            - 0: 8x8
            - 1: 8x16
        - [6]: PPU master/slave select
            - 0: Read backdrop from EXT pins
            - 1: Output color on EXT pins
        - [7]: Generate NMI interrupt at the start of vertical blanking interval
            - 0: off
            - 1: on
    - **PPUMASK[7:0]** ($2001) WRITE
        - [0]: Use grayscale image
            - 0: Normal color
            - 1: Grayscale
        - [1]: Show left 8 pixels of background
            - 0: Hide
            - 1: Show background in leftmost 8 pixels of screen
        - [2]: Show left 8 piexels of sprites
            - 0: Hide
            - 1: Show sprites in leftmost 8 pixels of screen
        - [3]: Render the background
            - 0: Don’t show background
            - 1: Show background
        - [4]: Render the sprites
            - 0: Don’t show sprites
            - 1: Show sprites
        - [5]: Emphisize red
        - [6]: Emphisize green
        - [7]: Emphisize blue
    - **PPUSTATUS[7:0]** ($2002) READ
        - [4:0]: Nothing?
        - [5]: Set for sprite overflow which is when more than 8 sprites exist in one scanline (Is actually more complicated than this to do a hardware bug)
        - [6]: Sprite 0 hit. This bit gets set when a non zero part of sprite zero overlaps a non zero background pixel
        - [7]: Vertical blank status register
            - 0: Not in vertical blank

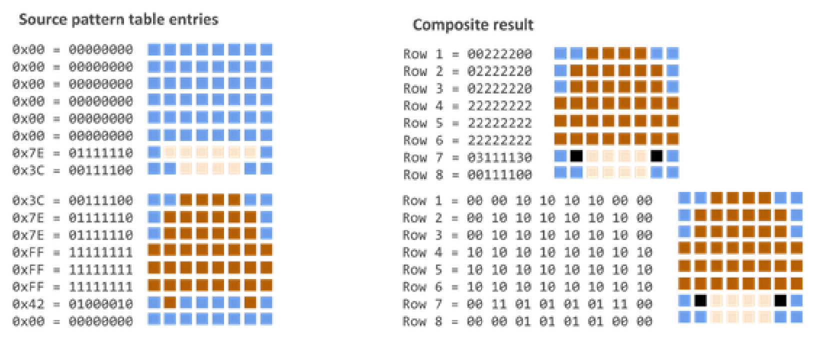
            - 1: Currently in vertical blank
    - **OAMADDR[7:0]** ($2003) WRITE
        - Address of the object attribute memory the program wants to access
    - **OAMDATA[7:0]** ($2004) READ/WRITE
        - ???
    - **PPUSCROLL[7:0]** ($2005) WRITE
        - Tells the PPU what pixel of the nametable selected in PPUCTRL should be in the top left hand corner of the screen
    - **PPUADDR[7:0]** ($2006) WRITE
        - Address the CPU wants to write to VRAM before writing a read of PPUSTATUS is required and then two bytes are written in first the high byte then the low byte
    - **PPUDATA[7:0]** ($2007) READ/WRITE
        - Writes/Reads data from VRAM for the CPU. The value in PPUADDR is then incremented by the value specified in PPUCTRL
    - **OAMDMA[7:0]** ($4014) WRITE
        - A write of $XX to this register will result in the CPU memory page at $XX00-$XXFF being written into the PPU object attribute memory

2.2.3. PPU CHAROM

- ROM from the cartridge is broken in two sections
    - Program ROM
        - Contains program code for the 6502
        - Is mapped into the CPU address space by the mapper
    - Character ROM
        - Contains sprite and background data for the PPU
        - Is mapped into the PPU address space by the mapper

2.2.4. PPU Rendering

- Pattern Tables
    - $0000-$2000 in VRAM
        - Pattern Table 0 ($0000-$0FFF)
        - Pattern Table 1 ($1000-$2000)
        - The program selects which one of these contains sprites and backgrounds
        - Each pattern table is 16 bytes long and represents 1 8x8 pixel tile
            - Each 8x1 row is 2 bytes long
            - Each bit in the byte represents a pixel and the corresponding bit for each byte is combined to create a 2 bit color.
                - Color_pixel = {byte2[0], byte1[0]}
            - So there can only be 4 colors in any given tile
            - Rightmost bit is leftmost pixel
    - Any pattern that has a value of 0 is transparent i.e. the background color

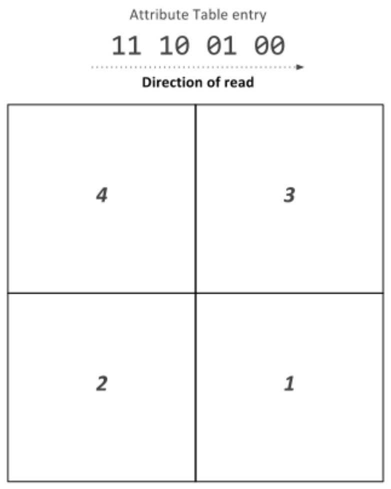

Source pattern table entries

```
0x00 = 00000000
0x00 = 00000000
0x00 = 00000000
0x00 = 00000000
0x00 = 00000000
0x00 = 00000000
0x7E = 01111110
0x3C = 00111100

0x3C = 00111100
0x7E = 01111110
0x7E = 01111110
0xFF = 11111111
0xFF = 11111111
0xFF = 11111111
0x42 = 01000010
0x00 = 00000000
```

Composite result

```
Row 1 = 00222200
Row 2 = 02222220
Row 3 = 02222220
Row 4 = 22222222
Row 5 = 22222222
Row 6 = 22222222
Row 7 = 03111130
Row 8 = 00111100

Row 1 = 00 00 10 10 10 10 00 00
Row 2 = 00 10 10 10 10 10 10 00
Row 3 = 00 10 10 10 10 10 10 00
Row 4 = 10 10 10 10 10 10 10 10
Row 5 = 10 10 10 10 10 10 10 10
Row 6 = 10 10 10 10 10 10 10 10
Row 7 = 00 11 01 01 01 01 11 00
Row 8 = 00 00 01 01 01 01 00 00
```

- Name Tables
  - $2000-$2FFF in VRAM with $3000-$3EFF as a mirror
  - Laid out in memory in 32x30 fashion
    - Resulting in a resolution of 256x240
  - Although the PPU supports 4 name tables the NES only supplied enough VRAM for 2 this results in 2 of the 4 name tables being mirror
    - Vertically = horizontal movement
    - Horizontally = vertical movement
  - Each entry in the name table refers to one pattern table and is one byte. Since there are 32x30=960 entries each name table requires 960 bytes of space the left over 64 bytes are used for attribute tables
  - Attribute tables
    - 1 byte entries that contains the palette assignment for a 2x2 grid of tiles



- Sprites
  - Just like backgrounds sprite tile data is contained in one of the pattern tables
  - But unlike backgrounds sprite information is not contained in name tables but in a special reserved 256 byte RAM called the object attribute memory (OAM)
- Object Attribute Memory
  - 256 bytes of dedicated RAM
  - Each object is allocated 4 bytes of OAM so we can store data about 64 sprites at once
  - Each object has the following information stored in OAM
    - X Coordinate

        - Y Coordinate
        - Pattern Table Index
        - Palette Assignment
        - Horizontal/Vertical Flip
  - Palette Table
    - Located at $3F00-$3F20
      - $3F00-$3F0F is background palettes
      - $3F10-$3F1F is sprite palettes
    - Mirrored all the way to $4000
    - Each color takes one byte
    - Every background tile and sprite needs a color palette.
    - When the background or sprite is being rendered the the color for a specific table is looked up in the correct palette and sent to the draw select mux.
  - Rendering is broken into two parts which are done for each horizontal scanline
    - Background Rendering
      - The background enable register ($2001) controls if the default background color is rendered ($2001) or if background data from the background renderer.
      - The background data is obtained for every pixel
    - Sprite Rendering
      - The sprite renderer has room for 8 unique sprites on each scanline.
      - For each scanline the renderer looks through the OAM for sprites that need to be drawn on the scanline. If this is the case the sprite is loaded into the scanline local sprites
        - If this number exceeds 8 a flag is set and the behavior is undefined.
      - If a sprite should be drawn for a pixel instead of the background the sprite renderer sets the sprite priority line to a mux that decides what to send to the screen and the mux selects the sprite color data.

2.3. APU

The NES included an Audio Processing Unit (APU) to control all sound output. The APU contains five audio channels: two pulse wave modulation channels, a triangle wave channel, a noise channel (for random audio), and a delta modulation channel. Each channel is mapped to registers in the CPU's address space. Each channel runs independently of each other. The outputs of all five channels are then combined using a non-linear mixing scheme. The APU also has a dedicated APU Status register. A write to this register can enable/disable any of the five channels. A read to this register can tell you if each channel still has a positive count on their respective timers. In addition, a read to this register will reveal any DMC or frame interrupts.

2.3.1. APU Registers

$4000	First pulse wave	DDLC VVVV	Duty, Envelope Loop, Constant Volume, Volume
$4001	First pulse wave	EPPP NSSS	Enabled, Period, Negate, Shift
$4002	First pulse wave	TTTT TTTT	Timer low
$4003	First pulse wave	LLLL LTTT	Length counter load, Timer high
$4004	Second pulse wave	DDLC VVVV	Duty, Envelope Loop, Constant Volume, Volume

$4005	Second pulse wave	EPPP NSSS	Enabled, Period, Negate, Shift
$4006	Second pulse wave	TTTT TTTT	Timer low
$4007	Second pulse wave	LLLL LTTT	Length counter load, Timer high
$4008	Triangle wave	CRRR RRRR	Length counter control, linear count load
$4009	Triangle wave		Unused
$400A	Triangle wave	TTTT TTTT	Timer low
$400B	Triangle wave	LLLL LTTT	Length counter load, Timer high
$400C	Noise Channel	--LC VVVV	Envelope Loop, Constant Volume, Volume
$400D	Noise Channel		Unused
$400E	Noise Channel	L--- PPPP	Loop Noise, Noise Period
$400F	Noise Channel	LLLL L---	Length counter load
$4010	Delta modulation channel	IL-- FFFF	IRQ enable, Loop, Frequency
$4011	Delta modulation channel	-LLL LLLL	Load counter
$4012	Delta modulation channel	AAAA AAAA	Sample Address
$4013	Delta modulation channel	LLLL LLLL	Sample Length
$4015 (write)	APU Status Register Writes	---D NT21	Enable DMC, Enable Noise, Enable Triangle, Enable Pulse 2/1
$4015 (read)	APU Status Register Read	IF-D NT21	DMC Interrupt, Frame Interrupt, DMC Active, Length Counter > 0 for Noise, Triangle, and Pulse Channels
$4017	APU Frame Counter	MI-- ----	Mode (0 = 4 step, 1 = 5 step), IRQ inhibit flag

2.4. Memory Mapper

Cartridges are a Read-Only Memory that contains necessary data to run games. However, it is some cases that a cartridge holds more data than the CPU can address to. In this case, memory mapper comes into play and changes the mapping as needed so that one address can point to multiple locations in a cartridge. For our case, the end goal is to get the game Super Mario Bros. running on our FPGA. This game does not use a memory mapper, so initially we will not be working on any memory mappers. If time permits, we might add support for the other memory mapping systems so that we can play other games.

2.5. DMA

Direct Memory Access from ROM to PPU is essential in the design. DMA allows PPU to read necessary sprite data independently from CPU. This allows PPU to operate at higher clock frequency than CPU's. However ROM doesn't support simultaneous reads from CPU and PPU, therefore CPU's read request has to wait until PPU's is done.

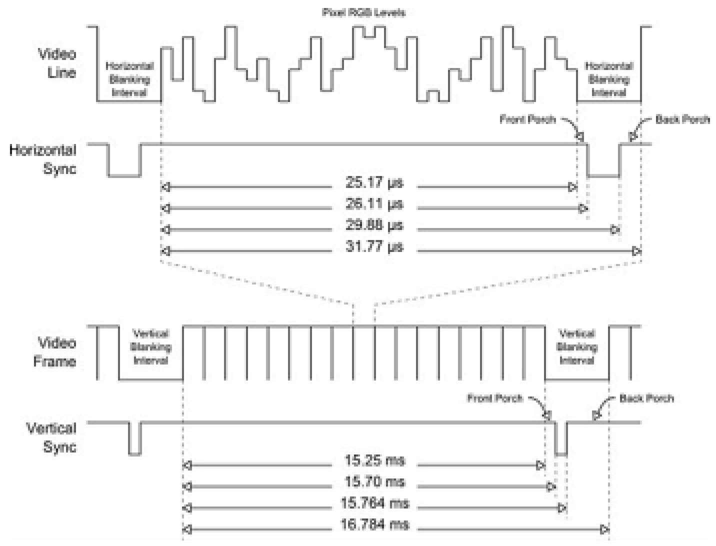
For most NES programs, the CPU will write a copy of the sprite OMA somewhere in the CPU addressable RAM. The DMA is then used to copy the 256 bytes into the OAM using the OAMDMA Register mentioned above. Writing XX to this register causes the DMA to fully initialize the OAM by writing to the 256 bytes to the OAMDATA register. The DMA will retrieve this location at the addresses $XX00 - $XXFF. The CPU is suspended while this transfer is taking place, however, this results in 4x faster process. The total process will take 513 (or 514 for odd CPU cycles) cycles to complete. First, there are one (or two) idle cycles followed by 256 pairs of alternating read write cycles. The DMA transfer will begin at the address specified in OAMADDR and increment from there.

2.6. Controller

Controller is the input from the user to play the the game. We will be using the keyboard instead of the joypad as the controller. Keyboard will be communicating with FPGA, using SPART communication method. The input is then memory mapped on $4016 and $4017 for CPU to read every short time frame.

2.7. VGA

The VGA interface consists of sending the pixel data to the screen one row at a time from left to right. In between each row it requires a special signal called horizontal sync (hsync) to be asserted at a specific time when only black pixels are being sent, called the blanking interval. This happens until the bottom of the screen is reached when another blanking interval begins where the interface is only sending black pixels, but instead of hsync being asserted the vertical sync signal is asserted.



The main difficulty with the VGA interface will be designing a system to take the PPU output (a 256x240 image) and converting it into a native resolution of 640x480 or 1280x960.

3. Software

3.1. Assembler

In order to test and debug our NES implementation, we will need to write an assembler to assemble code for the 6502 processor. Because the NES was released in the 1980's, the system has a relatively archaic amount of memory. It had a total of 2kb of RAM, expandable up to 8kb with the cartridges, and only 256 bytes of this memory was reserved for the stack. Due to the small amount of memory on the system, it is unnecessary to create a high-level compiler. In fact, a compiler likely be too inefficient for the low-memory NES. There are already many 6502 assemblers available, but we will be creating our own implementation to better understand the processor. The instructions have already been provided in the above CPU section, but the below table specifies the possible modes and the hex values of each of the opcodes. In addition, we plan to include support for labels to make writing software easier. The assembler will take in a file written in assembly and output either binary or hex files.

3.1.1. Assembler Opcode Table

Opcode	Mode	Hex	Opcode	Mode	Hex	Opcode	Mode	Hex
ADC	Immediate	69	DEC	Zero Page	C6	ORA	Absolute	0D
ADC	Zero Page	65	DEC	Zero Page, X	D6	ORA	Absolute, X	1D
ADC	Zero Page, X	75	DEC	Absolute	CE	ORA	Absolute, Y	19
ADC	Absolute	6D	DEC	Absolute, X	DE	ORA	Indirect, X	01
ADC	Absolute, X	7D	DEX		CA	ORA	Indirect, Y	11
ADC	Absolute, Y	79	DEY		88	PHA		48
ADC	Indirect, X	61	EOR	Immediate	49	PHP		08
ADC	Indirect, Y	71	EOR	Zero Page	45	PLA		68
AND	Immediate	29	EOR	Zero Page, X	55	PLP		28
AND	Zero Page	25	EOR	Absolute	4D	ROL	Accumulator	2A
AND	Zero Page, X	35	EOR	Absolute, X	5D	ROL	Zero Page	26
AND	Absolute	2D	EOR	Absolute, Y	59	ROL	Zero Page, X	36
AND	Absolute, X	3D	EOR	Indirect, X	41	ROL	Absolute	2E

AND	Absolute, Y	39	EOR	Indirect, Y	51	ROL	Absolute, X	3E
AND	Indirect, X	21	INC	Zero Page	E6	ROR	Accumulator	6A
AND	Indirect, Y	31	INC	Zero Page, X	F6	ROR	Zero Page	66
ASL	Accumulator	0A	INC	Absolute	EE	ROR	Zero Page, X	76
ASL	Zero Page	06	INC	Absolute, X	FE	ROR	Absolute	6E
ASL	Zero Page, X	16	INX		E8	ROR	Absolute, X	7E
ASL	Absolute	0E	INY		C8	RTI		40
ASL	Absolute, X	1E	JMP	Indirect	6C	RTS		60
BCC		90	JMP	Absolute	4C	SBC	Immediate	E9
BCS		B0	JSR		20	SBC	Zero Page	E5
BEQ		F0	LDA	Immediate	A9	SBC	Zero Page, X	F5
BIT	Zero Page	24	LDA	Zero Page	A5	SBC	Absolute	ED
BIT	Absolute	2C	LDA	Zero Page, X	B5	SBC	Absolute, X	FD
BMI		30	LDA	Absolute	AD	SBC	Absolute, Y	F9
BNE		D0	LDA	Absolute, X	BD	SBC	Indirect, X	E1
BPL		10	LDA	Absolute, Y	B9	SBC	Indirect, Y	F1
BRK		00	LDA	Indirect, X	A1	SEC		38
BVC		50	LDA	Indirect, Y	B1	SED		F8
BVS		70	LDX	Immediate	A6	SEI		78
CLC		18	LDX	Zero Page	B6	STA	Zero Page	85
CLD		D8	LDX	Zero Page, Y	AE	STA	Zero Page, X	95
CLI		58	LDX	Absolute	BE	STA	Absolute	8D
CLV		B8	LDX	Absolute, Y	A2	STA	Absolute, X	9D
CMP	Immediate	C9	LDY	Immediate	A0	STA	Absolute, Y	99
CMP	Zero Page	C5	LDY	Zero Page	A4	STA	Indirect, X	81
CMP	Zero Page, X	D5	LDY	Zero Page, X	B4	STA	Indirect, Y	91
CMP	Absolute	CD	LDY	Absolute	AC	STX	Zero Page	86

CMP	Absolute, X	DD	LDY	Absolute, X	BC	STX	Zero Page, Y	96
CMP	Absolute, Y	D9	LSR	Accumulator	4A	STX	Absolute	8E
CMP	Indirect, X	C1	LSR	Zero Page	46	STY	Zero Page	84
CMP	Indirect, Y	D1	LSR	Zero Page, X	56	STY	Zero Page, X	94
CPX	Immediate	E0	LSR	Absolute	4E	STY	Absolute	8C
CPX	Zero Page	E4	LSR	Absolute, X	5E	TAX		AA
CPX	Absolute	EC	NOP		EA	TAY		A8
CPY	Immediate	C0	ORA	Immediate	09	TSX		BA
CPY	Zero Page	C4	ORA	Zero Page	05	TXA		8A
CPY	Absolute	CC	ORA	Zero Page, X	15	TXS		9A
						TYA		98

3.1.2. Assembly Format

- Immediates
  - LDA #$20
- Absolute
  - LDX $2000
  - Value at address is loaded into X
- Zero Page
  - LDX $20
  - Value at address $0020 loaded into X
- Relative
  - LDX $20
  - Value at address PC + $20 is loaded into X
- Absolute Indexed
  - AND $2000, X
  - Value at address $2000 + X is and’ed with the destination register
- Zero Page Indexed
  - AND $20, X
  - Value ate address $0020 + X is and’ed with the destination register
- Zero Page Indexed Indirect
  - STA ($20, X)

3.2. Simulator

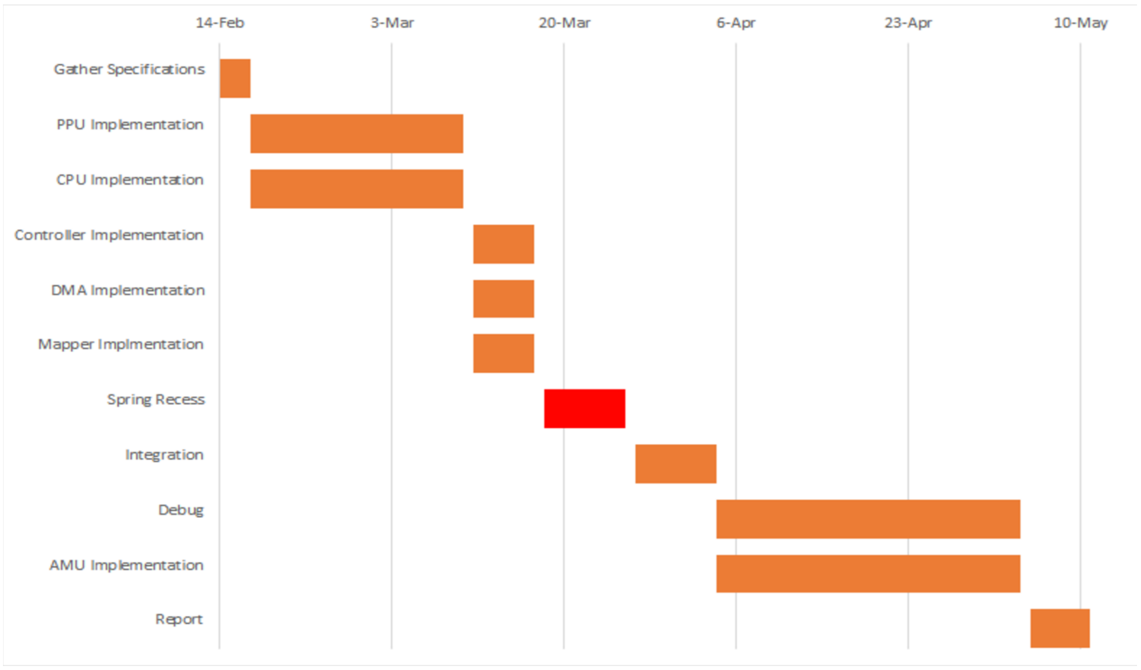
We plan to use a 6502 CPU simulator to debug and test our processor implementation. Luckily, in projects done at other institutions, students have already created their own simulators. Due to time constraints, we plan to use a simulator called 6502SIM[1]. We will be able to input binary files and hex files into the simulator. Then, we can set breakpoints and step through the file on a simulated 6502. This simulator also displays a disassembly of memory, the CPU registers, the stack, and the memory, which are all updated dynamically as the code is executed so you can watch the results. You can also modify registers and memory locations during execution.

3.3. Application

Initially, while we are still working on getting the CPU and the PPU fully functional, we will write our own software in assembly language to help debug the CPU. Our assembler will assemble the code into machine language to run on our FPGA. We will likely, include all of the instructions and each mode of the instructions in our initial testing application. This will allow us to verify that our CPU is cycle accurate for each instruction. Once we have verified that the CPU is functioning properly, we will move on from our own testing application to a NES ROM. These games are available online as object files, and loading these onto our FPGA can help to verify the correctness of the PPU and eventually will be fully playable.

4. Timeline

We will begin by working on the Central Processing Unit and Picture Processing Unit. Once we get two or three weeks into the project, we will be evaluating how much progress has been made on the two blocks and will decide whether or not we will continue working on the PPU. If we are behind, we will instead use an open source implementation that we found online. Once we get the PPU and CPU working and integrated, we will start working on the smaller blocks. We hope to get all of the main blocks finished by the end of spring break. The remainder of the time will be used for integration of and debugging. Time permitting, we may be able to spend some time on the APU. However, this is a stretch goal that we do not expect to finish.



5. Contracts

I, Pavan Holla, vow to develop the 6502 CPU unit used in the NES. This will require me to do extensive research on how the CPU was designed and worked at the hardware level. I will do my best to have a working implementation by the end of March and then begin the integration with the PPU team after spring break.

I, Patrick Yang, vow to develop the 6502 CPU unit used in the NES. This will require me to do extensive research on how the CPU was designed and worked at the hardware level. I will do my best to have a working implementation by the end of March and then begin the integration with the PPU team after spring break.

I, Eric Sullivan, vow to head the development of the NES PPU unit. This will require me to do extensive research on how the PPU was designed and worked at the hardware level. I will do my best to have a working implementation by the end of March and then begin the integration with the CPU team after spring break.

I, Jonathan Ebert, vow to assist Eric Sullivan in the development of the Picture Processing Unit for our project. This role includes the research of the NES specs, the programming of each of the individual blocks, and the integration of the Picture Processing Unit with the rest of the NES design. In addition, I will be involved with the general implementation of each of the blocks of our NES. This will require me to assist in developing any needed blocks that are not included in the CPU or PPU, and will also require me to gain a better understanding of how each of the blocks in the NES interface with each other.

6. References

1. http://www.atarihq.com/danb/6502.shtml#6502SIM
2. http://web.mit.edu/6.111/www/f2004/projects/dkm_report.pdf
3. http://www.e-tradition.net/bytes/6502/6502_instruction_set.html
4. http://www.thealmightyguru.com/Games/Hacking/Wiki/index.php/6502_Opcodes
5. https://opcode-defined.quora.com/
6. http://nesdev.com/
7. http://www.masswerk.at/6502/6502_instruction_set.html
8.