

Index

Descriptions	Tabs/Pages
Project Report	Red
-Project Description	2
-Problem Statement	3
-System Design	4
-Program Design	6
-Construction of a Prototype	28
-Implementation	36
-Testing	42
-Maintenance Plan	45
-Conclusions	46
-References	47
-Acknowledgements	48
Requirements Specification: First Draft	White
Requirements Specification: Final Draft	Blue
Individual Project Reports (Essays)	Orange
Appendices	Yellow
-Resumes	
-Presentation Slides	
-Source Code	
-CD Wallet (see readme.txt)	

Project Descriptions

Overview

Our project consists of recreating the Nintendo Entertainment System on an FPGA. Using a Terasic DE2i-150 development board, we wish to design an NES emulator from the ground up. The process involves implementing the system in terms of units. This means we will set up all appropriate drivers and controllers needed in a certain module before moving on to the next step. This ensures an efficient use of the limited time we have.

Who Are We?

We are a group of Computer Science and Engineering students from University of California, Riverside. As part of our Senior Design class in our Summer Study Abroad course in Lausanne, Switzerland, we decided to tackle something that deals with both hardware and software interfacing. Since the concentration deals with Computer Architecture and Embedded Systems, we felt that designing an emulator on an FPGA would be a great project for the course.

The Team

Sergio Morales

Handled coding the RP2A03 CPU from scratch, and the Altera Nios II SOPC interface between any peripherals.

Hector Dominguez

Dealt with coding the Picture Processing Unit for video rendering (built from scratch), and the SD Card interface for reading NES games.

Omar Torres

Worked with the hardware, NES/SNES Controllers, VGA signals, and boot loader.

Randy Truong

Worked with hardware, handled coding of the NES/SNES Controllers as well as worked on the VGA signals.

Kevin Mitton

Worked with hardware, started into the APU interface

Problem Statement

Time Constraints

The first priority in our project is to have a working prototype within the timespan we have. Building a working NES Emulator on a Terasic DE2i-150 FPGA board requires a large amount of time spent researching before actually programming and implementing each component. A certain amount of time spent learning to use the development tools, such as Eclipse with Altera plugins, Quartus Web Edition, and Git for version control, so we'd have to account for that as well.

Previous Experiences

The group has sufficient experience through previous courses to be able to work on designing, prototyping, and testing. For example, the team may utilize experience in VHDL from Computer Architecture course to learn Verilog for FPGA programming and running testbenches.

Access to Required Software/Hardware

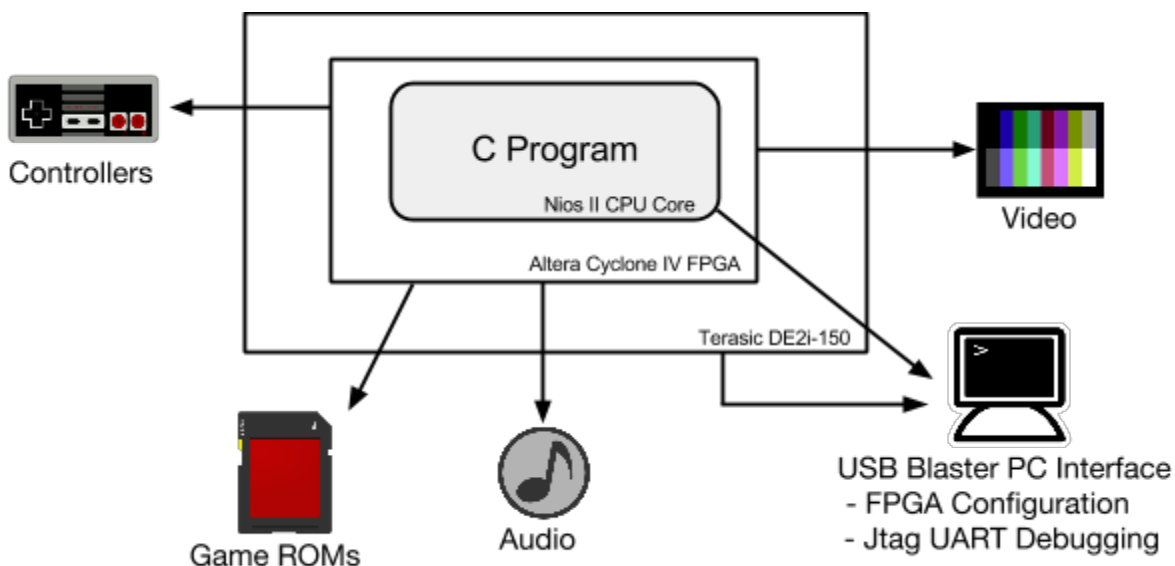
In terms of hardware and software, we have access to everything we would need to complete the project. This includes the FPGA board, the Altera's editor software, and any other necessary hardware such as a monitor, NES Controller, and SD cards to build the NES FPGA Emulator.

Is This a Significant Design Problem?

The project has several components to implement: the CPU, APU, PPU, VGA Controller, NES Controller, and several interfacing controllers on the board itself. We were required to learn the software given to us for the FPGA board and to learn Verilog to implement these components. Implementation will require the use of certain state machines and memory management to correctly emulate NES games. This why the design is non-trivial. Certain decisions in how we design, for example, the CPU, will affect performance immensely. This is because the NES design scheme is based purely off hardware, whereas we are offloading some of this hardware description in software, specifically in C.

System Design

We took advantage of the many peripherals the Terasic DE2-i150 board featured, therefore, we tried to focus the design around the FPGA board, leaving only a few components to be used outside of the board. Even then, we utilized the general purpose I/O (GPIO) pins. As an overview of what we used, the follow diagram represents our entire system design:



High level view of our design.

Controllers

We used the GPIO pins to connect the controller interface for our emulator. From there, we wrote a Verilog module that is able to keep track of which buttons are pressed, by storing the 8-bit value in a register.

Game ROMs

We use an SD card to store the NES game roms. This allows our emulator to let the user choose from a selection of roms that are stored on the card.

Audio

Using the GPIO pins and a generic speaker, we plan to send NES audio through here.

Video

NES graphics are displayed through the board's VGA ports. This is controlled by the Nios II CPU core, since there was an available VGA controller IP (intellectual property) core handy.

USB Blaster PC Interface

This was used to configure the FPGA through Altera's 'Quartus II Web Edition'. In addition, we used Eclipse (with some Altera plugins) as our software development platform for writing certain components in C. This also allowed us to be able to debug our emulator, as a terminal in Eclipse are available to use for printing and parsing through bugs.

Program Design

Describing the design of our emulator is broken up into the following categories:

- NIOS II SOPC Interface
- NES CPU - The RP2A03 Processor
- NES PPU - The Picture Processing Unit
- NES APU - The pseudo-Audio Processing Unit
- SD Card Interface
- VGA Interface
- Controller Interface

NIOS II SOPC Interface

In order to facilitate rapid development of our NES emulator prototype, we decided it would be best to use Altera's Nios II IP core, in order to off load heavy programming into C, rather than the harder-to-debug-and-verify Verilog language. This is one of the main components used throughout our component design in Qsys. In a nutshell, Nios II is a 32 bit RISC processor, which, combined with the Eclipse package, allows us to compile C programs, and be able to run them on the FPGA. Note that this is only possible after we've configured the FPGA with the Nios II IP core.

Nios II also comes with very useful features and libraries that we've taken advantage of. This is specifically when it comes to SD Card interfacing, VGA controlling, and accessing certain registers we stored on the FPGA in our main Verilog module from our C program. An example of such a register would be the button input from our controllers.

Since the C program needs to be stored somewhere in memory for the Nios processor to be able to run it, we use the SDRAM on-board as the primary storage for the entire C program, and anything that needs to be pushed to the stack and heap. Another great aspect of Qsys, is that we are able to have memory-mapped I/O registers that connect from within our Verilog code, to the Nios II processor, allow possible communication between our C program to any exposed modules. One example of our use of this is controller input, which will be covered more in detail later.

CPU - The Ricoh 2A03 Processor

The Nintendo Entertainment system has a Ricoh 2A03 Processor, which consists of:

- A MOS 6502 Microprocessor lacking Binary-Coded Decimal mode.
- 22 memory-mapped I/O registers
- pseudu-Audio Processing Unit
- Communication to and from Picture Processing Unit

Outline

- The NMOS 6502 Microprocessor
- Instruction Set
- Decoding Opcodes
- Power-up State and Interrupt Handling
- Memory Mapped I/O

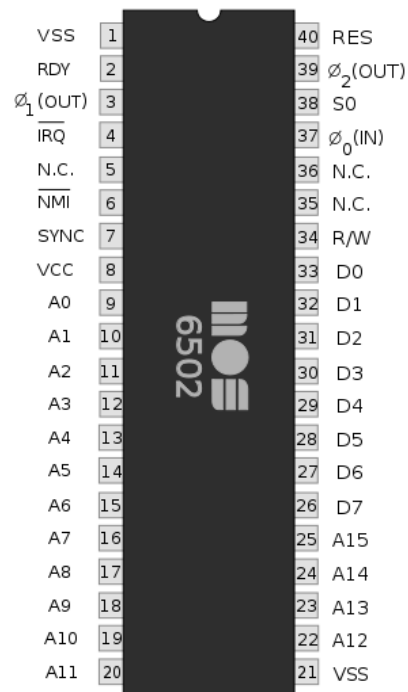
NMOS 6502 Microprocessor

The NMOS 6502, already known for being featured in Apple and Atari systems, serves as the microprocessor for running NES cartridge-based games.

It's 16-bit address bus allows for up 64kB of addressing space, without external multi memory controllers that some NES games may make use of. Data, however, is a byte long.

It's instruction set relies solely on 3 registers for arithmetic and storing values, the Accumulator (**A**), and Index registers **X** and **Y**.

Three other registers, the Program Counter (**PC**), Stack Pointer (**SP**), and Status register (also known as the processor flags, **P**), are all essential to run instructions. It's important to note that all registers are 8-bits wide, with the exception of the PC, due to the address bus.



The Program Counter keeps track of the currently executing instruction, which is incremented automatically. The Stack Pointer is an area in RAM that can be used by the NES program to store data and hold return values for interrupts and subroutines. Finally, P register holds the values for certain flags that are triggered instructions and interrupts- they are vital for some instructions, as they rely on flags that have been set by previous ones.

The following memory map of the NES A203 illustrates the main use of each memory region. Note memory from now on will be referred by '\$', in hex.

Region	Location
Upper PRG Bank	\$C000-\$FFFF
Lower PRG Bank	\$8000-\$BFFF
SRAM	\$6000-\$7FFF
Expansion ROM	\$4020-\$5FFF
I/O Registers	\$4000-\$401F
\$2000-\$2007 [M]	\$2008-\$3FFF
I/O Registers	\$2000-\$2007
\$0000-\$07FF [M]	\$0800-\$1FFF
RAM	\$0200-\$07FF
Stack	\$0100-\$01FF
Zero Page	\$0000-\$00FF

The NES CPU Memory Map

The two PRG banks represent the memory region of the NES program, which are memory mapped to the cartridge itself. SRAM and Expansion ROM are filled depending on whether the game uses them or not. I/O Registers are used primarily for joypads and the Picture Processing Unit. Note that the Stack Pointer has an implicit 0x100 added to it, since it is only 8 bits, and we would need at least 9 to access Stack memory. Finally, Zero Page is a special area of memory that would only require one byte to access because of the range.

Many instructions make use of this area, since it would require less CPU cycles to read and write here.

Another important detail of the NES CPU memory map is mirroring. Regions denoted by **[M]** are mirrored copies of previous regions. Writing or reading here would produce the same effect as writing or reading to the original region, and vice-versa to the original.

Before executing instructions, it's important to know about the Status/Processor Flags in the 6502:

Bit	Flag	Description
0	C - Carry	This flag is set to '1' if the last instruction, most likely arithmetic operations such as ADC caused a carry.
1	Z - Zero	Set to '1' if last instruction caused a result of 0.
2	I - Interrupt Disable	Set to '1' for IRQ interrupts to be disabled. See ' Power-up State and Interrupt Handling ' for more info on interrupts.
3	D - Enable BCD	Not used in NES. Normally this would cause add and subtract instructions to operate in BCD mode.
4	B - Break flag	This flag only exists in BRK operations, when the flags are pushed to the stack.
5	Unused	
6	V - Overflow	Set to '1' if last instruction resulted in an overflow.
7	N- Negative	Set to '1' if last instruction resulted in bit 7 of the result to be 1.

6502 Processor Flags/Bits

6502 Instruction Set

Although the NES doesn't utilize Binary-Coded Decimal feature of the 6502, there are still a variety of instructions for the programmer to choose from. There are 56 total unique instructions to use. Note that there are different addressing modes that the CPU may use to access or store data. Taking this into account, there are over 150 possible

instructions/opcodes. Instruction addressing modes can be broken down into 13 different types, as shown in the table below:

Addressing Mode	Description	Example
Accumulator	Instructions that execute directly with or on register A.	LSR A
Absolute	These instructions have a 16-bit address as the operand.	LDA \$2002
Absolute, X	Take a 16-bit address as well, but adds register X to the end.	LDA \$2002, X
Absolute, Y	Same as the latter, except it uses Y.	AND \$6000, Y
Immediate	Takes in the operand as a constant byte.	LDA, #\$50
Implicit	No operand required to perform instruction.	CLC
Indirect	Used only for JMP instruction; operands identify location of the where the actual instruction will be executed.	JMP
Indexed Indirect	Fetches location of value from Zero Page memory using the operand byte + X, and operand byte + X + 1.	LDA (\$FF, X)
Indirect Indexed	Fetches location of value from ZP memory using operand byte, and operand byte + 1, then adds Y to give full address.	STA (\$FF), Y
Relative	Used for branch instructions. Operand is used as signed offset to next instruction to jump to.	BNE \$E5
Zero Page	One operand is used for memory accesses, because Zero Page only ranges from \$0000-\$00FF.	LDA \$F5
Zero Page, X	Uses the operand byte + X as memory location for value.	AND \$10, X
Zero Page, Y	Uses the operand byte + Y as memory location for value	LDX \$10, Y

The different addressing modes in the NES Instruction Set

Note, as mentioned before, the 6502 uses 8-bit registers. Since we need 16-bits to address memory, many operations handle addresses or offsets in terms of 2 byte chunks. The 6502

is **little endian**. This means the lower byte of an address is always read first when working with memory. Because of these different addressing modes, each individual instruction has its own number of clock cycles needed to finish. They range from being 2 to 7 cycles.

Decoding Opcodes

There are about 50 unique instructions. Each of them may have several variations based on the addressing modes, but the general execution is the same. All instructions start off through the CPU decoding the first byte as the instruction (the contents of the PC), incrementing the PC, and starting execution.

Execution almost always involve memory reading and writing, storing or manipulating registers, or branching. Rows represent lower bits while columns represent upper bits. ORing them together leads to the Opcodes. The following table describes all possible opcodes:

	0x00	0x01	0x02	0x03	0x04	0x05	0x06	x07
0x00	BRK b	ORA (d,X)				ORA d	ASL d	
0x10	BPL r	ORA (d),Y				ORA d,X	ASL d,X	
0x20	JSR a	AND (d,X)			BIT d	AND d	ROL d	
0x30	BMI r	AND (d),Y				AND d,X	ROL d,X	
0x40	RTI	EOR (d,X)				EOR d	LSR d	
0x50	BVC r	EOR (d),Y				EOR d,X	LSR d,X	
0x60	RTS	ADC (d,X)				ADC d	ROR d	
0x70	BVS r	ADC (d),Y				ADC d,X	ROR d,X	
0x80		STA (d,X)			STY d	STA d	STX d	
0x90	BCC r	STA (d),Y			STY d,X	STA d,X	STX d,Y	
0xA0	LDY #	LDA (d,X)	LDX #		LDY d	LDA d	LDX d	
0xB0	BCS r	LDA (d),Y			LDY d,X	LDA d,X	LDX d,Y	

0xC0	CPY #	CMP (d,X)			CPY d	CMP d	DEC d	
0xD0	BNE r	CMP (d),Y				CMP d,X	DEC d,X	
0xE0	CPX #	SBC (d,X)			CPX d	SBC d	INC d	
0xF0	BEQ r	SBC (d),Y				SBC d,X	INC d,X	
	0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	x0F
0x00	PHP	ORA #	ASL A			ORA a	ASL a	
0x10	CLC	ORA a,Y				ORA a,X	ASL a,X	
0x20	PLP	AND #	ROL A		BIT a	AND a	ROL a	
0x30	SEC	AND a,Y				AND a,X	ROL a,X	
0x40	PHA	EOR #	LSR A		JMP a	EOR a	LSR a	
0x50	CLI	EOR a,Y				EOR a,X	LSR a,X	
0x60	PLA	ADC #	ROR A		JMP (a)	ADC a	ROR a	
0x70	SEI	ADC a,Y				ADC a,X	ROR a,X	
0x80	DEY		TXA		STY a	STA a	STX a	
0x90	TYA	STA a,Y	TXS			STA a,X		
0xA0	TAY	LDA #	TAX		LDY a	LDA a	LDX a	
0xB0	CLV	LDA a,Y	TSX		LDY a,X	LDA a,X	LDX a,Y	
0xC0	INY	CMP #	DEX		CPY a	CMP a	DEC a	
0xD0	CLD	CMP a,Y				CMP a,X	DEC a,X	
0xE0	INX	SBC #	NOP		CPX a	SBC a	INC a	
0xF0	SED	SBC a,Y				SBC a,X	INC a,X	

The full opcode table for the 6502. Blank spaces are illegal/unofficial opcodes

Power-up State and Interrupt Handling

One important thing to note is that the 6502's power-up state starts off by executing an interrupt, that is, it performs a Reset interrupt. For each interrupt, there is a reserved area in memory that contains the starting point of the handler, or subroutine. The NES utilizes three main interrupts:

- **Interrupt Service Request (IRQ)** - This interrupt in the NES is simulated in software by calling the BRK instruction, except that the PC is loaded with the contents the IRQ vector.
- **Reset** - This interrupt is nearly always the starting point for NES games. It is up to the game programmer to decide how the game starts, whether they initialize memory to 0, or they skip straight to the game. A very important note about Reset is that nothing in memory is actually 'reset', merely the PC is set to start back to where the game previously started. This is how some games were able to have special features that let the user restart the game and still have some sort of memory saved.
- **Non-maskable Interrupt** - This interrupt, as the name suggests, cannot be ignored, like IRQ can. In the NES, this is controlled by the Picture Processing Unit. Normally when the screen is done rendering, NMI will be enabled, causing the game to perform a subroutine by setting the PC to the contents of the NMI vector. This can be used for situations where the programmer needs to wait for the screen to finish rendering before running a certain piece of code.

Interrupt	Location
NMI	\$FFFA-\$FFFB
Reset	\$FFFC-\$FFFD
IRQ	\$FFFE-\$FFFF

Locations of Interrupt vectors in CPU memory

Memory Mapped I/O

The Ricoh A203 bridges communication between all peripherals through memory mapping with the following memory regions:

- Audio: Locations \$4000-\$4015 serve as registers between audio, mainly for volume, function generators, delta modulation channel, etc.
- Controllers: \$4016 and \$4017 are player 1 and 2's controllers. Any button input is directly mapped to these registers, available for the game program to read from.
- Video: In order to communicate between the NES' Picture Processing Unit, there is a set of registers used between both units use for rendering data:

Register	Description
\$2000 'PPUCTRL'	Write only. Contains flags for controlling PPU.
\$2001 'PPUMASK'	Write only. Controls screen rendering and color.
\$2002 'PPUSTATUS'	Read only. Used to determine rendering timing.
\$2003 'OAMADDR'	Write only. Refers to the address to which you wish to write data to the Sprite table (OAM).
\$2004 'OAMDATA'	Read/Write. Sprite data is written here. Note that the OAMADDR is incremented every time a write happens.
\$2005 'PPUSCROLL'	Write (twice) only. Register is use for scrolling the the screen.
\$2006 'PPUADDR'	Write (twice) only. Refers to the address to which you wish to write data to the Nametable, or the background data.
\$2007 'PPUDATA'	Read/Write. Data to be written/read at specific address in PPU memory is in this register. Note, writes to this register trigger an auto-increment to the PPU address. Reading from this register grab data from a post-buffer, which means 2 reads are required to actually grab the right data specified at the PPU address.

CPU-PPU I/O Registers

PPU - The Picture Processing Unit

The Picture Processing Unit is a microprocessor that the original Nintendo (NES) used to generate image frames from data stored in memory.

PPU Subtopics

- Memory Map
- Pattern Tables
- Nametable/Attribute Tables
- Color Palettes
- Object Attribute Memory (OAM)
- Registers
- Timing

Internally the PPU runs three times faster than the CPU. Therefore, for every CPU cycle the PPU executes three cycles. The PPU is controlled via eight registers (\$2000-\$2007), which are visible from the CPU address space. The CPU sends information and data to these registers. Next, they are read by the PPU onto the PPU memory map.

PPU Memory Map		
Address	Size	Description
\$0000	\$1000	Pattern Table 0
\$1000	\$1000	Pattern Table 1
\$2000	\$3C0	Name Table 0
\$23C0	\$40	Attribute Table 0
\$2400	\$3C0	Name Table 1
\$27C0	\$40	Attribute Table 1
\$2800	\$3C0	Name Table 2
\$2BC0	\$40	Attribute Table 2
\$2C00	\$3C0	Name Table 3

\$2FC0	\$40	Attribute Table 3
\$3000	\$F00	Mirror of 2000h-2EFFh
\$3F00	\$10	BG Palette
\$3F10	\$10	Sprite Palette
\$3F20	\$E0	Mirror of 3F00h-3F1Fh

Pattern Tables

The pattern table holds 8x8 Pixel tiles that are used to draw images on the screen. One tile set holds 256 tiles, which is equivalent to 4096 bytes per set. Each individual tile holds 2 bits per pixel. Therefore, since one tile is 8x8 pixel and each pixel uses 2 bits, so to draw one tile we need a total of 16 Bytes. The PPU uses two pattern tables, one for the background and the other for sprites. However, the PPU allows sprites to access both Pattern tables.

Pattern Tables

Address	Description
\$0000	Background
\$1000	Sprite

Remember we need 16 bytes in order to draw one tile. In order to display one tile from the pattern table, first we need to grab the initial byte. After we have the initial byte, the next 15 bytes plus the initial byte will be the data for the tile.

Address	Value	Address	Value
\$0000	00010000	\$0008	00000000
\$0001	00000000	\$0009	00101000
\$0002	01000100	\$000A	01000100
\$0003	00000000	\$000B	10000010
\$0004	11111110	\$000C	00000000
\$0005	00000000	\$000D	10000010
\$0006	10000010	\$000E	10000010
\$0007	00000000	\$000F	00000000

In the table above, it shows the first 16 bytes from the pattern table (first tile). It works by combining the first 8 bytes with the second 8 bytes. Thus forming one 8 byte Tile. For example, for the first iteration we obtain the address \$0000 and \$0008. The bits from the first address will serve as bit 0 and the bits on the second address will serve as bit 1. After concatenating these two values we will have the following in terms of decimal form:

Result
00010000
00202000
03000300
20000020
11111110
30000020
30000030
00000000

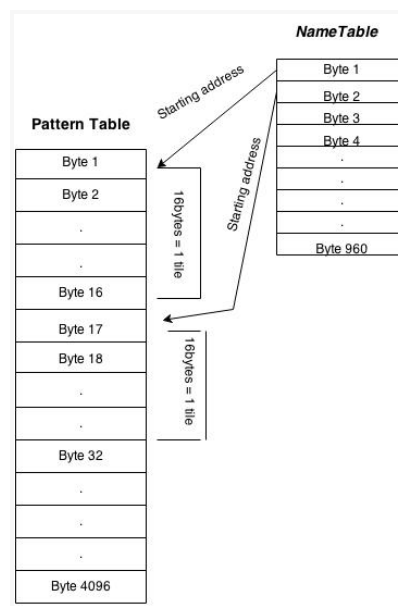
Each byte in the new eight-byte tile is composed of eight 2-bit sequences as seen below.

0 0, then the 2 bit sequence would 0
 0 1, then the 2 bit sequence would 1
 1 0, then the 2 bit sequence would 2
 1 1, then the 2 bit sequence would 3

These 2 bits represent the least significant bits of the 4-bit number, which is needed to identify the image or sprite palette entry, then it will be used to draw that specific pixel on the screen.

NameTable and Attribute Table

Nametables have the first byte of every tile that will be rendered to the screen during the specific timing. The PPU can render 32 tiles horizontally and 30 tiles vertically. Thus, a full render frame would consist of 960 tiles. The nametable stores 960 bytes, where each byte is the first byte of a tile to be found in the pattern table. Therefore, to draw each individual tile we have to look for the nametable byte in the pattern table. Once the byte has been found on the pattern table, then the following 15 bytes (on the pattern table) plus the byte found will create that tile.



Each nametable has an associated attribute that holds the upper two bits of the color for the tile. Remember that after concatenating the 16 bytes from the pattern table into an 8-byte tile, we obtain the two lower bits of the 4-bit pixel color for every 2-bit sequence. The attribute table is 64 bytes long. There is not enough bytes to represent all 960 bytes in the nametable. So, every byte in the attribute is used for 16 tiles of the nametable, and each nametable byte represents one tile. However, the 16 tiles that the attribute byte controls are in 4x4 tiles. For example in the diagram below, it shows the nametable tiles in order from left to right, and top to bottom. Each attribute byte corresponds to 16 nametable tiles. The first attribute byte is used for the tiles shown in yellow in the diagram below. As you can see, the

sixteen tiles that the attribute byte is used for are in 4x4 tile squares. The second attribute byte is used for the tiles shown in red. This continues from left to right, and top to bottom.

Nametable Tiles (Screen Frame)

1	2	3	4	5	6	7	8	11	12	.	.	30	30	31	32
33	34	35	36	37	38	39	40	43	44	.	.	61	62	63	64
65	66	67	68	69	70	71	72	75	76	.	.	93	94	95	96
97	98	99	100	101	102	103	104	105	106	.	.	125	126	127	128
129	130	131	132	133	134	135	136	137	138	.	.	157	158	159	160
161
.
.
.
.
.	928
929	930	931	932	933	934	935	936	960

The first attribute byte is used for the 4x4 tiles colored in yellow in the diagram above. Lets grab these tiles to explain how the attribute byte works.

1	2	3	4
33	34	35	36
65	66	67	68
97	98	99	100

The diagram above displays the tiles that represent the first attribute byte. The attribute byte has 8 bits and we have 16 tiles. We also know that upper two color bits for each pixel in the tiles comes from the attribute table. So we are going to divide these 4x4 tiles into four 2x2 sections. The diagram below shows the sections divided by color.

1	2	3	4
33	34	35	36
65	66	67	68
97	98	99	100

We have divided the table into 4 sections of 2x2 tiles. We are going to assign the corresponding bits.

Attribute Byte example: 1 1 1 0 0 1 0 0

The way this works is that each 2x2 table within the 4x4 tiles will use the same 2 bits. For example tiles 1, 2, 33 and 34 are a 2x2 tile subset of the 4x4 tile. These tiles will be sharing the same two bits from the attribute byte.

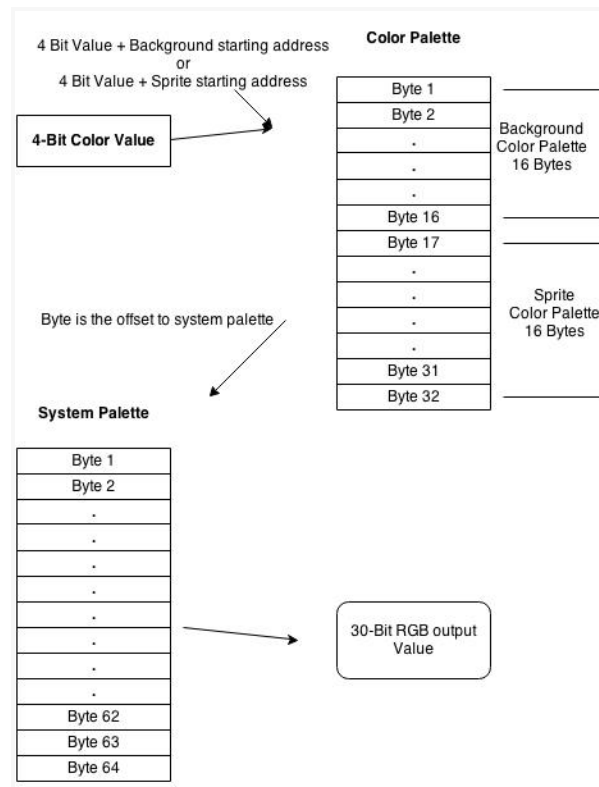
Color from example above	Bits from Attribute example above	Tile Numbers
RED	00	1, 2, 33, 34
GREEN	01	3, 4, 35, 36
PURPLE	10	65, 66, 97, 98
BLAKCK	11	67, 68, 99, 100

The two bits obtained for the tiles from the attribute byte are the upper two color bits. The lower two bits are obtained from the pattern table after combining the 16 bytes.

Color Palettes

The PPU has two color palettes one for the background tiles, and for the sprite tiles. The background color palette begins at address \$3F00 and the sprite color palette begins at address \$3F10.

Earlier, we learned how to obtain a 4-bit hex value that represents the color bits. The lowest two bits were obtained from the pattern table and the upper two bits were obtained from the attribute table (background). This 4-bit value represents an offset of the palette table. If the color is for a background tile, then we obtain the data stored on the background starting address plus the 4-bit offset. The same applies for sprite. The data stored in this address will be an offset to the system color palette that we hard coded into the system. The data stored on the system color palette are 30-bit values that we designed to represent the NES colors to be output on the DE2i-150 board in RGB. The NES does not output in RGB format, instead it outputs in NTSC format and we did not understand how that worked, so we built our own NTSC to RGB controller.



Object Attribute Memory (OAM)

The OAM is extra memory block outside the PPU and CPU memory maps. This memory block contains all of the sprite information to be displayed at that exact rendered frame. The OAM is similar to the nametables because nametables only handle Background. Therefore, the OAM only handles sprites. Sprites are moveable characters that are drawn onto the screen. They can be either in format of 8x8 pixels or 8x16 pixels. Most characters in the NES are formed of multiple sprites. As explained earlier, the sprite data is stored on the pattern tables while the sprite attributes are stored on the extra memory block.

The process of outputting sprites onto the screen is similar to outputting the background tiles. The OAM stores a total of up to 64 sprites that will be rendered to the screen at that specific time. For every screen frame, this table will update to the new data similar to the nametables. Each sprite in the OAM is composed of 4 bytes. Thus, the whole memory block has a total of 256 Bytes long. Each sprite has the following information,

Byte 0: Y Position of the top left of sprite minus one
 Byte 1: The Tile index number within the pattern table
 Byte 2: Attributes of the sprite (palette, priority, flipping)
 Byte 3: X Position of left side of the sprite to be rendered

Sprites that are 8x16 pixels use different pattern tables based on their index number. If the index number is even, the sprite data is in the first background pattern table, otherwise it is on the sprite pattern table. Sprites are given priority on their position in the extra memory block. On each scanline, the system calculates which sprites are going to be drawn on that scanline. Lowest priority sprites are the first to be drawn to ensure the highest priority sprites are drawn on top. However, only eight sprites can be displayed per scanline.

Registers (I/O)

The PPU uses eight internal I/O registers to communicate with the CPU. These registers are located in the CPU memory map (\$2000-\$20007). These registers are used to transfer data to the PPU, which changes the output on the screen and vice versa. The description of every register byte is explained below. (The following information was obtained from www.nesdev.com)

Register \$2000

7654 3210

		++- Base nametable address
		(0 = \$2000; 1 = \$2400; 2 = \$2800; 3 = \$2C00)
	+---	VRAM address increment per CPU read/write of PPUDATA
		(0: add 1, going across; 1: add 32, going down)
	+----	Sprite pattern table address for 8x8 sprites
		(0: \$0000; 1: \$1000; ignored in 8x16 mode)
	+-----	Background pattern table address (0: \$0000; 1: \$1000)
	+-----	Sprite size (0: 8x8; 1: 8x16)
	+-----	PPU master/slave select
		(0: read backdrop from EXT pins; 1: output color on EXT pins)
	+-----	Generate an NMI at the start of the vertical blanking interval (0: off; 1: on)

Register \$2001

7654 3210

||| | |||

||| | ||| |+- Grayscale (0: normal color; 1: produce a monochrome display)

||| | ||| +- 1: Show background in leftmost 8 pixels of screen; 0: Hide

||| | +- 1: Show sprites in leftmost 8 pixels of screen; 0: Hide

||| +- 1: Show background

|| +- 1: Show sprites

| +- Intensify reds (and darken other colors)

| +- Intensify greens (and darken other colors)

+ - Intensify blues (and darken other colors)

Register \$2002

7654 3210

||| | |||

|| | +++++- Least significant bits previously written into a PPU register
(due to register not being updated for this address)| +----- Sprite overflow. The intent was for this flag to be set
| whenever more than eight sprites appear on a scanline, but a
| hardware bug causes the actual behavior to be more complicated
| and generate false positives as well as false negatives; see
| PPU sprite evaluation. This flag is set during sprite
| evaluation and cleared at dot 1 (the second dot) of the
| pre-render line.| +----- Sprite 0 Hit. Set when a nonzero pixel of sprite 0 overlaps
| a nonzero background pixel; cleared at dot 1 of the pre-render
| line. Used for raster timing.+----- Vertical blank has started (0: not in VBLANK; 1: in VBLANK).
Set at dot 1 of line 241 (the line *after* the post-render
line); cleared after reading \$2002 and at dot 1 of the
pre-render line.

Register \$2003

This is the OAMADDR register where the CPU tells the PPU what address the sprite data will be written to.

Register \$2004

This is the where the CPU will write the sprite data. The PPU grabs the data from register \$2004 and stores it at the location specified by register \$2003.

Register \$2005

This is the scrolling register. This registers tells the PPU what pixel to write from the nametable on the top left corner

Register \$2006

This register writes the address of where the nametable data is going to be stored.

Register \$2007

This register tells the PPU the data that is going to be stored at the address specified by register \$2006

Timing

Timing was one of the hardest principles of the PPU to implement. A simple mistake grabbing data or grabbing extra data on the specific cycle could potentially affect the output. The PPU renders 262 scan lines per frame, where 240 scanlines correspond to outputting to the frame since we are dealing with 256x240 screen resolution. The remaining 20 scan lines are called Vertical Blank (Vblank). During the Vertical Blank scan lines, the CPU begins writing to the IO Registers and the PPU begins filling up the nametables, attribute tables and OAM for the next upcoming frame. This is the only time the CPU is able to write to the PPU via the IO Registers because otherwise it could affect the current frame being render at the moment. The PPU is only allowed to write one pixel for every PPU cycle. Therefore, it takes about 341 PPU cycles per scanline for grabbing data and rendering the 256 pixels per scanline.

During scanline 240 the PPU begins the vertical Blanking mode for the next 20 scan lines. During this time, the CPU begins updating all the information that will be rendered on the following frame.

NES APU - The pseudo-Audio Processing Unit

The APU generates the sounds from the NES. The APU registers are mapped to locations \$4000-\$4013, \$4015, and \$4017 in CPU memory. We plan on using GPIO pins to communicate the audio from a game with an attached speaker. The following figure shows which channel is mapped to a particular set of registers.

Registers	Channel	Units
\$4000-\$4003	Pulse 1	Timer, length counter, envelope, sweep
\$4004-\$4007	Pulse 2	Timer, length counter, envelope, sweep
\$4008-\$400B	Triangle	Timer, length counter, linear counter
\$400C-\$400F	Noise	Timer, length counter, envelope, linear feedback shift register
\$4010-\$4013	DMC	Timer, memory reader, sample buffer, output unit
\$4015	All	Channel enable and length counter status
\$4017	All	Frame counter

SD Card interfacing

For our NES Emulator we used the SD Card module built onto the DE2i-150 board to receive files from a SD card. The SD card holds the game roms we will be using to test and play with the NES system. The SD Card module reads the data stored in the game rom and sends it to its proper locations in memory. The game ROM have data that is stored on the CPU memory. They also have data belonging to the PPU memory. After the data has been sent to the components, the SD Card is no longer used.

VGA Interfacing

The Terasic DE2i-150 Board has a built in VGA port that we use to render pixels to the screen. Our NES Emulator supports a resolution of 256x240, which is the original resolution from the original Nintendo. To render the pixels onto the screen, we utilize the NIOS II processor built in the VGA C library. The VGA module only supports RGB signals format. We then modified our NES system to produce output in RGB instead of the original output of the NES, which was in NTSC format.

One important thing to note is that the Nios VGA Controller uses a frame buffer. In order to keep reading and writing times fast enough to now slow down frame rendering, we used SRAM stored in the board to work as a frame buffer. This is how the Nios library for VGA rendering is able to write to certain pixels at different times.

Controllers

Utilizing the GPIO pins on the DE2i-150 board, we created a Verilog module for accessing button inputs from either an original NES controller or SNES controller. Button inputs will be sent to the corresponding I/O registers through 8 bits from the NES controller and 12 bits from the SNES controller.

Construction of a Prototype

Outline

- NIOS II SOPC Interface
- CPU
- PPU
- SD Card Interface
- VGA
- Controller

NIOS II SOPC Interface

The first stage of our prototype starts on designing the layout of what board components we want to use using the software called Qsys. This software allows us to tell the board what components we are going to be using. Some examples of the components we use are the NIOS II processor, a clock, Sram memory, VGA and the Altera SD Card.

Qsys - nios_system.qsys (C:\Users\Hector\Documents\Switzerland\CS179\NES_FPGA-previous\NES_FPGA-master\source\NES_FPGA\nios_system.qsys)

Name	Description	Export	Clock	Base	End	IRQ	Tags	Opcode Name
clk_0	Clock Source	clk_0_clk_in						
clk_in_reset	Reset Input	clk_0_clk_in_reset						
clk	Clock Output	clk_0	clk_0					
clk_reset	Reset Output	clk_0_reset						
Clock_Signals	Clock Signals for DE-series Board Part...	Double-click to export						
clk_in_primary	Clock Input	Double-click to export	clk_0					
clk_in_primary_reset	Reset Input	Double-click to export	clk_0					
sys_clk	Clock Output	Double-click to export	sys_clk					
sys_clk_reset	Reset Output	Double-click to export	sys_clk					
sdram_clk	Clock Output	clock_signals_sdram_clk	clock_signals_sdram_clk					
vga_clk	Clock Output	Double-click to export	vga_clk					
CPU	Nios II Processor	Double-click to export	sys_clk					
clk	Clock Input	Double-click to export	sys_clk					
reset_in	Reset Input	Double-click to export	sys_clk					
data_master	Avalon Memory Mapped Master	Double-click to export	sys_clk					
instruction_master	Avalon Memory Mapped Master	Double-click to export	sys_clk					
flag_debug_module_in	Reset Output	Double-click to export	sys_clk					
flag_debug_module_in	Avalon Memory Mapped Slave	Double-click to export	sys_clk					
custom_instruction_in	Custom Instruction Master	Double-click to export	sys_clk					
Pixel_Buffer	Pixel Buffer DMA Controller	Double-click to export	sys_clk					
clock_reset	Clock Input	Double-click to export	sys_clk					
clock_reset_reset	Reset Input	Double-click to export	sys_clk					
external_interface	Conduit	Pixel_Buffer_external_interface	clock_reset					
avalon_sram_slave	Avalon Memory Mapped Slave	Double-click to export	clock_reset					
Pixel_Buffer_DMA	Pixel Buffer DMA Controller	Double-click to export	clock_reset					
clock_reset	Clock Input	Double-click to export	clock_reset					
reset_reset	Reset Input	Double-click to export	clock_reset					
avalon_ctrl_slave	Avalon Memory Mapped Master	Double-click to export	clock_reset					
avalon_ctrl_slave	Avalon Memory Mapped Slave	Double-click to export	clock_reset					
avalon_stream_source	Avalon Streaming Source	Double-click to export	clock_reset					
Pixel_Buffer_PPG	Pixel Buffer PPG	Double-click to export	sys_clk					
clock_stream_in	Clock Input	Double-click to export	sys_clk					
clock_stream_in_reset	Reset Input	Double-click to export	sys_clk					
clock_stream_out	Clock Output	Double-click to export	vga_clk					
clock_stream_out_reset	Reset Input	Double-click to export	clock_stream_out					
avalon_dc_buffer_sink	Avalon Streaming Sink	Double-click to export	clock_stream_out					
avalon_dc_buffer_src	Avalon Streaming Source	Double-click to export	clock_stream_out					
VGA_Controller	VGA Controller	Double-click to export	vga_clk					
clock_reset	Clock Input	Double-click to export	vga_clk					
clock_reset_reset	Reset Input	Double-click to export	vga_clk					

Messages

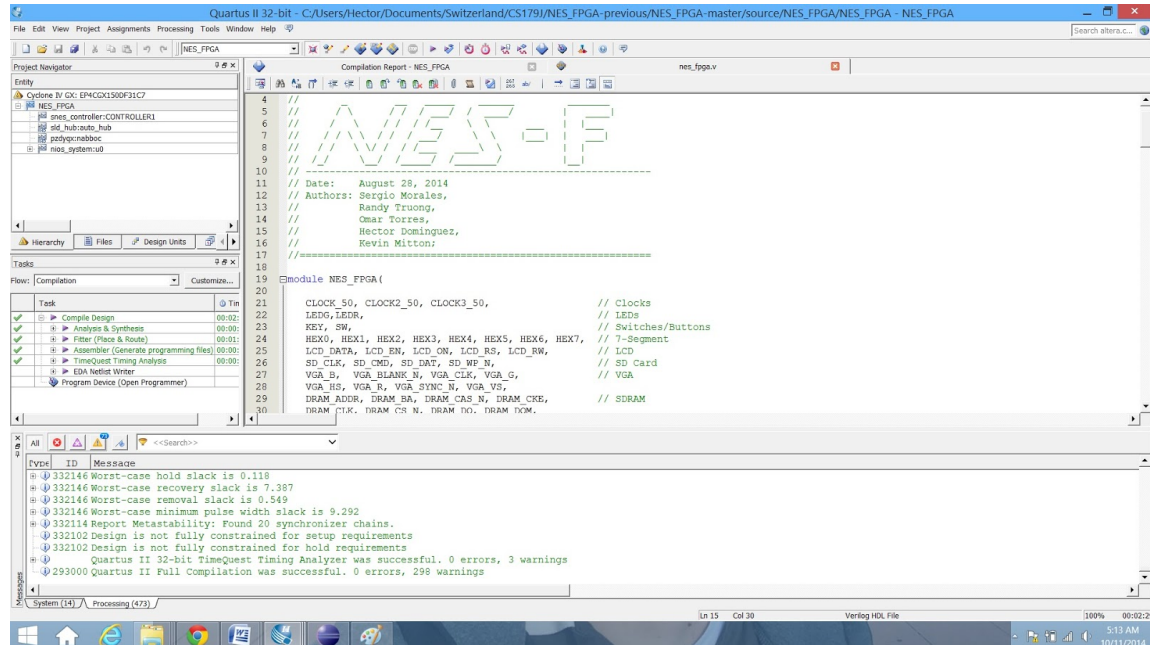
4 Info Messages

- Video Output Stream Format: 640 x 480 with Color: 10 (bits) x 3 (planes)
- CPUID control register value is 0. Please manually assign CPUID if creating multiple Nios II system
- System ID is not assigned automatically. Edit the System ID parameter to provide a unique ID
- Time stamp will be automatically updated when this component is generated.

0 Errors, 0 Warnings

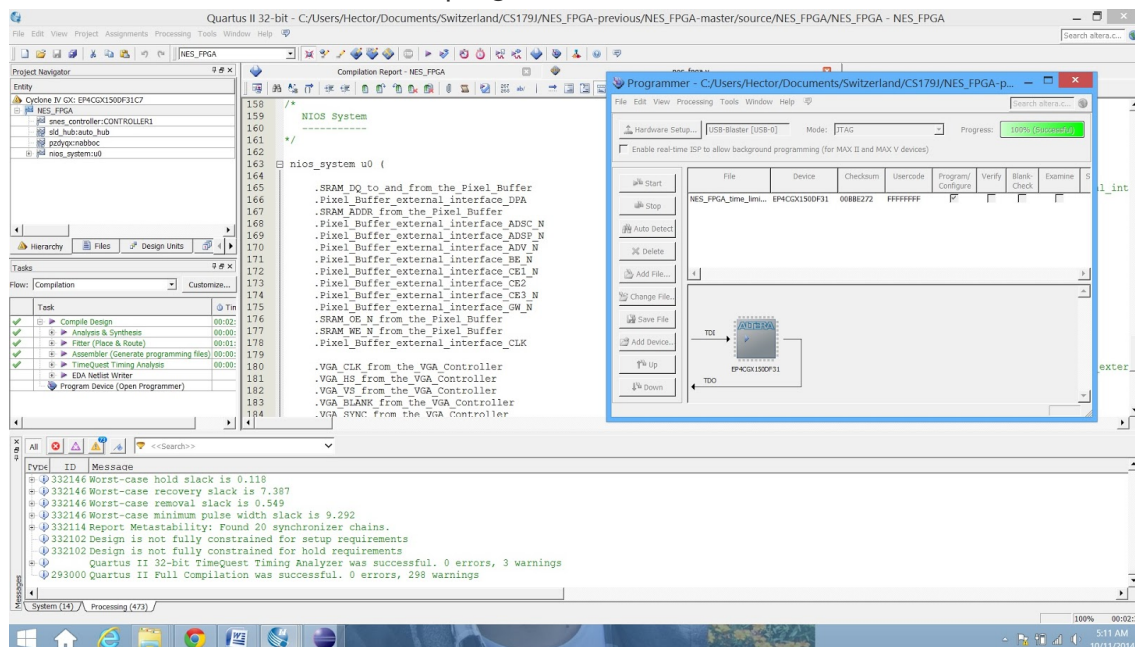
Building our components using Qsys

After we finish telling the board what component we are using, Qsys creates a verilog code that helps us link these components to the NIOS processor. This helps us access the components whenever we write C code in NIOS. We can view the Verilog code in a software called Quartus II. This software programs the components using the Verilog programming language to prepare them for the heavy work that the NIOS II processor processes.



The main Verilog module for our project

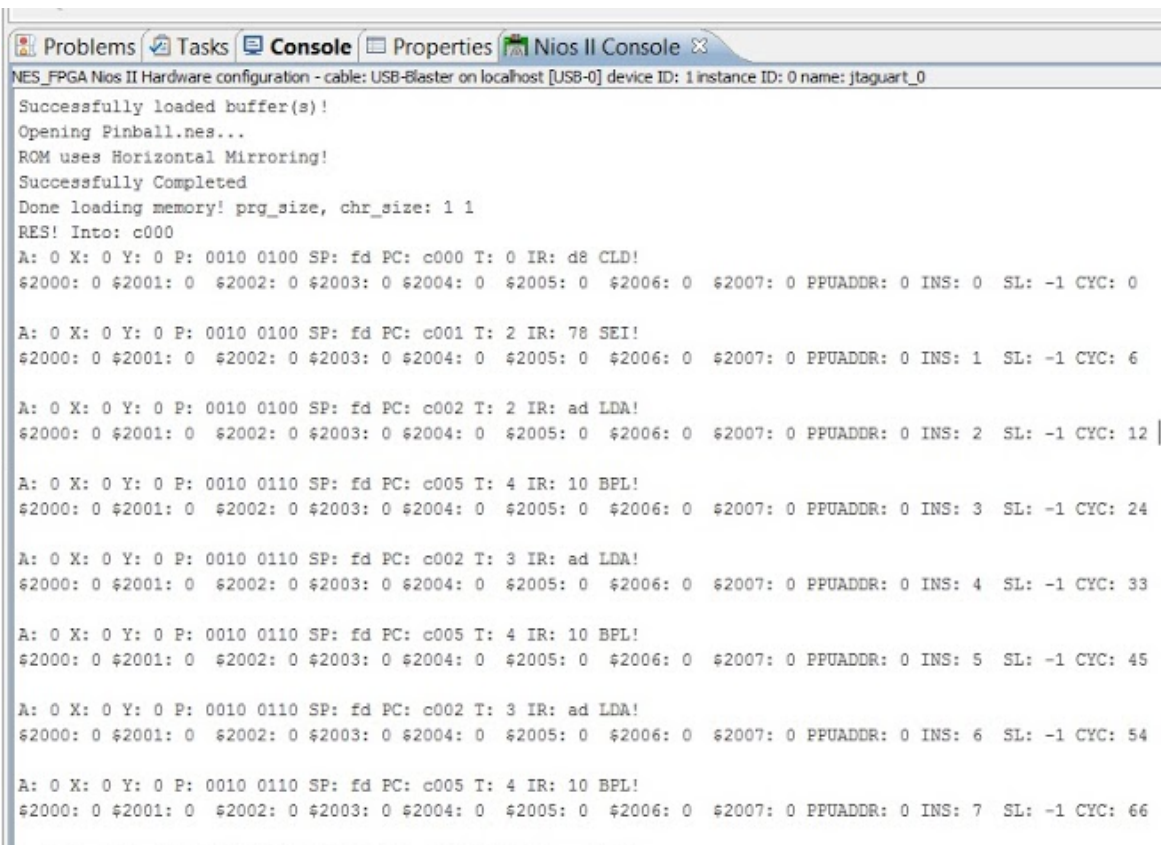
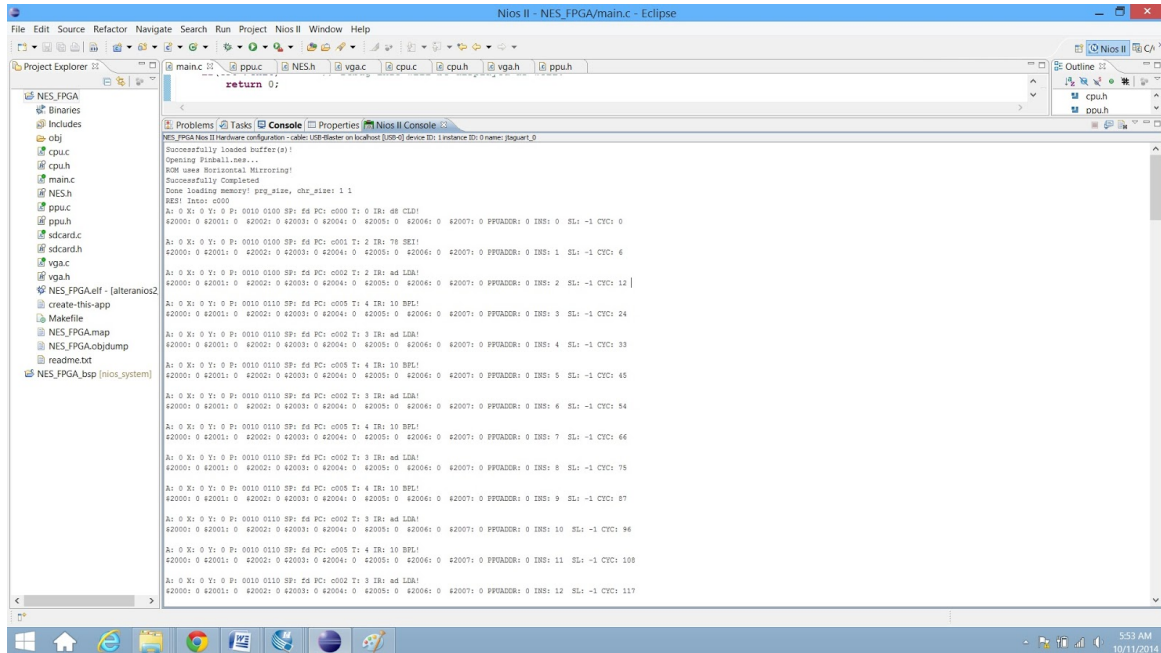
Once we are ready to program the FPGA, we simply use the programmer within Quartus and program the board.



Configuring the FPGA using Quartus II

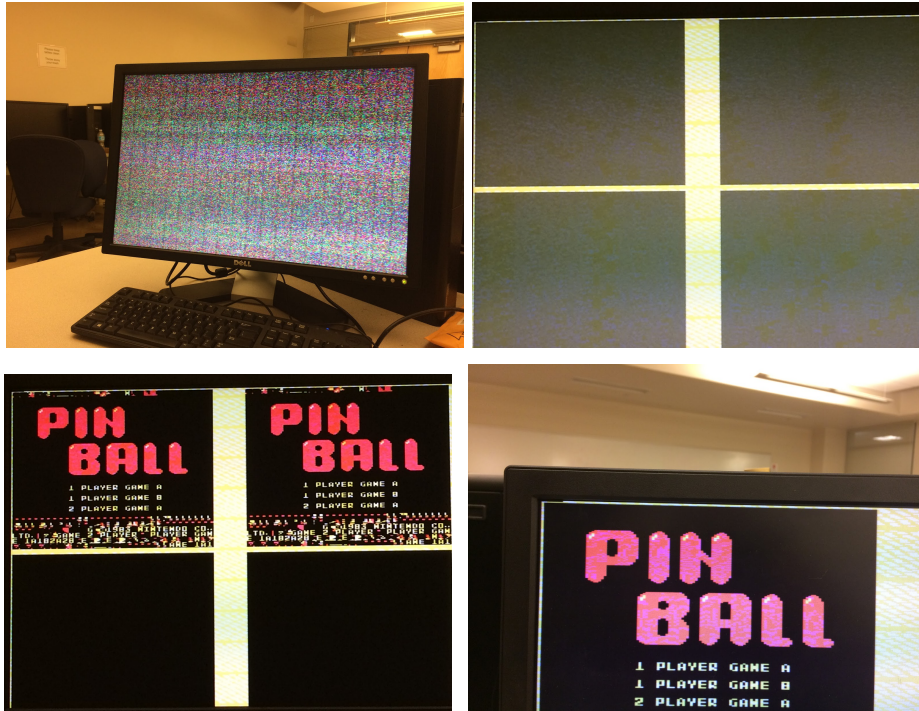
CPU

On our working prototype we managed to create a working CPU that is able to properly execute all of the instructions that the original NES could run. The CPU is able to execute over 150 instructions.



PPU + VGA

Throughout the process of constructing our prototype, we had several issues with displaying a correct output. We first began displaying random pixels as shown on the image below, then we were able to display a solid color. As we kept researching on the functionality of the PPU and its colors, our output to the screen was improving. Now, we are able to display full frames with the correct colors and tiles with some minor bugs.



PPU and VGA process throughout project

SD Card Interface

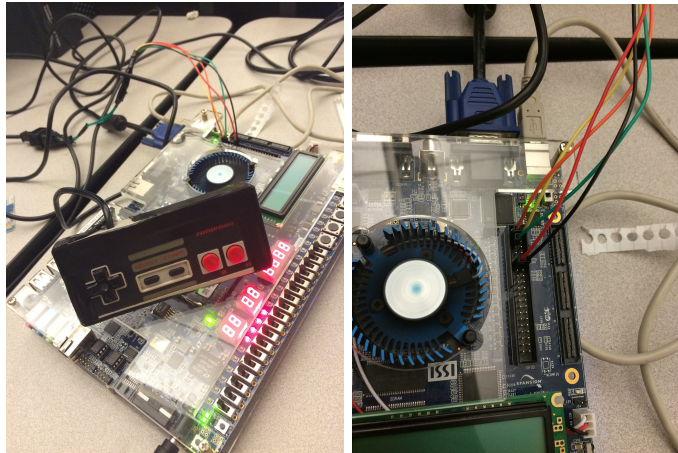
The SD Card was one of the main components that needed to be constructed in order to test the CPU and PPU. Our project uses an SD card with standard capacity (SDSC) to read data. In order for the SD card to be compatible with the DE2 FPGA board it needs to be formatted to FAT16.



SDSC FAT 16 Card

Controller

We have the NES controller functioning properly with the boards GPIO pins. See implementation for more details.



NES controller Hooked up to the DE2i-150 Board

Implementation

Our implementation of an NES FPGA Emulator revolved around working with Quartus II Web Edition for Verilog development, and Eclipse

Outline

- Implementation of the NES CPU
- Implementation of the NES PPU
- pseudo-Audio Implementation
- NES Controllers

Implementation of the NES CPU

Outline

- Registers and Memory
- Parsing Through Data
- Addressing Modes
- Decoding Instructions
- Execution
- Interrupts
- Mirroring

Registers and Memory

To ease data manipulation, two main data types are defined in our Nios C project, a byte, and a word. This ensures we deal solely with 8-bit and 16-bit numbers for addressing and transferring data. For any registers related to the CPU, we implemented them by creating a struct that encapsulates all registers, A, X, Y, PC, P, and S, as well as memory (which is implemented as an array of bytes) and any necessary flags for interrupts. These are all bytes, with the exception of PC, since this is 16-bits wide. Also, to ease us in working on certain regions in memory, we made a set of pre-processor defines.

Parsing through Data

Since programming the 6502 CPU core was done in C, we utilized structs and pointers. After using the SD Card Interface in Nios II to read in the NES ROM, the CPU starts parsing through data in terms of bytes. It's important to note that the CPU merely reads from the PRG ROM previously mentioned in the Program Design (see 'CPU Memory Map' figure.) This PRG ROM, thanks to our use of SD Card interfacing functions from Altera, has already been read into one big array used within our CPU struct instance.

Addressing Modes

We implemented addressing modes by making all opcodes go through a set of functions that grabbed the operand based on the addressing mode. This saved space and possibly resulted in better performance, because implementing all opcodes without having specific functions for each addressing mode would definitely have redundancy. For example, there would be over 20 instructions that need their operand to be fetched through Zero Page. Instead of writing the code of fetching it each time, for each instructions, we simply call the corresponding function for Zero Page. Certain addressing modes required careful manipulation of reads and writes. Zero Page, X, for example, needed the calculated address to wrap around in the case of going over the page boundary. In this, I implemented wrapping by basically performing a bitwise AND by 0xFF, since the calculated address would have to wrap back to 0x00 in case the calculated address went over.

Decoding Instructions

We implemented instruction decoding by setting up the CPU to parse the very first byte as the opcode, which will always be an 8-bit number. From here, we essentially use a switch statement that goes to certain cases based off of the instructions to be executed. This is essentially a jump table under certain optimizations in a compiler. From here, we take the next step in executing instructions by deciding what type of addressing mode we're in, which is decided based on what opcode we have.

Execution

The general flow for executing an instruction in our implementation of the NES CPU is to:

- Grab the operand by calling the corresponding addressing mode function.
- Access memory, if need be.
- Perform operation. This is usually moving data, arithmetic, or manipulating the flags.
- Increment the PC every time we read another byte. This is done in C extensively by calling 'CPU->PC++' (return PC and increment it after)
- Keep track of number of clock cycles needed to execute last instruction.

To avoid redundancy, we made specific functions for fundamental actions in the CPU. This is essentially memory reads, memory writes, and one unique function per type of instruction, NOT per opcode. This is because we may have 4+ versions of the same instruction, such as LDA. Therefore, we split up instructions in terms of addressing mode and type of instruction.

In an attempt to save on runtime overhead, we tried to inline our functions as much as possible, even though the compiler may already be doing so.

Interrupts

Implementing interrupts was simple. Before running the next CPU instruction tick function, we had a check, one for each type of interrupt. Since reset is the highest priority, this was checked first. Since our CPU struct had registers reserved for each interrupt, we would check to see if these registers were '1', and if so, we would run the corresponding subroutine for them. It is up to the NES game program itself to set these registers, so all we need to do to maintain this system of interrupt checking is to make sure we implement the interrupt setting correctly. For NMI, this is done by the PPU during the the start of Vblank. Reset is implemented by setting the register to '1' if the reset button is pressed on the FPGA board itself. And for IRQ, since this is essentially a BRK instruction, the program should be responsible for setting this.

Mirroring

Mirroring was also pretty simple to implement. We have each mirrored region in memory hardcoded into our memory access functions. This means, for example, that when writing to \$2008, would produce a write to \$2000. This is done by checking if we are trying to read or write to a mirroring region, and if so, we would simple read or write to the location '&' mirror_mask. In this case, mirror_mask would be '7', since in this region, we mirror every 8 bytes. So fully implementing mirroring was just a matter of hardcoding these address manipulations to write to the actual memory location.

Implementation of the NES PPU

Outline

- Memory Map
- Pattern Tables
- Nametables
- Attribute Tables
- Color Palettes
- OAM
- Rendering/Timing

Memory Map

Initially, we implemented the PPU memory map using a struct which only encapsulating the main memory, which is created by an array of bytes. However, as our design increased in size, we added the ability to store the sprite tables, a sprite buffer for rendering, variables to keep track of the current scanline and also variables that are used to read off the CPU registers. Lastly, we added a special array of structs. The array contains specific information for every frame tile.

Pattern Tables

As explained in the Program Design section, Pattern tables are tables that contain background tiles and sprite tiles. The memory map struct that we created encapsulates an array of bytes called MEM. This array is 0xFFFF long and contains all the data that will be used by the picture processing unit. The actual data that goes inside the Pattern table addresses (0x0000-0x2000) come from the CHR portion of the game ROM, and that comes when we read the ROM file using the SD Card. This data will never be modified after is written on these address. Also, data can only be read.

Nametables

NameTables contain the address of the tiles first byte from the pattern tables. There a total of four nametables inside the MEM array within our PPU struct where two of those nametables serve as mirrors of the first two nametables. There isn't a complete implementation for the nametables. They are just addresses in memory where the I/O registers write to those memory locations.

Attribute Tables

The functionality of the attribute table were already explained in previous sections. Attribute tables are part of memory that also get written by the IO registers. However, we

created a separate space of memory that is 960 bytes in size. This is the special array that we mentioned earlier. This array of structs hold all the possible information that needs to be known for every tile on the visible screen. As mentioned earlier, every attribute byte modifies 4x4 tiles. Thereafter, the 4x4 tiles are split into four 2x2 tile sections. Each of those sections share the same bits from the attribute byte. For more clarity of how it works, see the Program Design section. We can conclude that every tile will have an specific bit location to grab from the attribute byte.

This special array of structs called `ATTRIBUTE_TABLE_INFO`, contains which bits need to be access from the attribute byte for every tile on the screen. The array also contains if the tile is on the top left, top right, bottom left or bottom right of the 4x4 tile set. For debugging purposes, we also added a scanline as a member of the struct to be able to identify on which scanline the tile resides.

Color Palettes

The color palettes also reside within the MEM array in our PPU struct. The implementation for this is quite straight forward. The I/O registers write data onto these memory addresses. However, the data stored are bytes that are used as offsets. We created a global dynamic array of ints that contain the original NES colors in hex in the RGB format. These values are 30 bits long and are used with the VGA controller to output the correct pixels. The data stored on the color palettes are offsets to this global array of ints.

OAM

The OAM works the same as how the nametables work. The OAM is an extra allocated memory in the PPU struct outside the MEM array. It was implemented using an array of bytes. The I/O registers write to these memory locations and updates it for every frame. When it is time to render to the screen, the PPU grabs the information of the sprites on this table and outputs it onto the screen.

Rendering

Overview

The main task of the PPU is to have the memory allocated for the nametables, attribute tables, pattern tables, sprite tables and color palettes. The PPU then renders the appropriate data onto the screen after the I/O registers write to these memory locations.

Rendering Process

The PPU begins rendering from Scanline 0 to scanline 240. For every Scanline the PPU process 340 PPU Cycles to fetch the scanline data and to render. During these Scanlines, the CPU is not supposed to write to the PPU via I/O registers because they can ruin the output. However, they are some rare special cases were writing to the PPU is required during the rendering cycles.

When rendering the background tiles, the PPU first fetches the first byte in the nametable. This byte is the address offset from the pattern tables for the first address of the

tile. After that, the PPU combines the following 16 bytes into a set of 8. This gives us the first 2-bits for each individual pixel. Thereafter, the PPU locates the special array of structs mentioned earlier called `ATTRIBUTE_TABLE_INFO`. We obtain the bits that we need access from the attribute byte and we do a logical AND. This will give us the upper 2 bits for the color for every individual pixel. Now, we combine the upper and lower 2 bits that we obtained and will serve as an offset the color palette. When accessing the color palette, the data stored will be another offset. This will be the offset of the global array of ints. The data obtained here will be the true RGB color that will displayed for that specific pixel. We repeat the rendering process for each pixel of all of the tiles that will be rendered to the screen.

Vertical Blanking

After the PPU is done rendering, the PPU runs twenty non-visible scanline. During this scanlines, the CPU writes to the I/O registers and these registers write to the PPU memory locations. This updates the information to be displayed in the following frame.

APU -Implementation

- **We did not have enough time to implement Audio in our NES Emulator design.**

Controllers

We used Verilog for the implementation of the NES/SNES controllers.

The controller has 2 input pins: Latch and Pulse, while having one output pin Data. The datasheet of the NES and SNES controllers are very similar as they first start off taking in a Latch. When the Latch receives an off signal, the pulse would oscillate on and off. As the pulse oscillates, the data would return the state of the button that the pulse is on. If the button is pressed for that specific pulse, data would return a 0 (active low).

We check for every pulse corresponding to each button. NES Controllers have a total of 7 pulses with 1 latch, making 8 buttons. The SNES controller has 16 pulses as well as a latch, with only 12 buttons. The last 4 pulses of the SNES controller return nothing. We build a

state machine to run through every pulse and button to build the bits to send to the rest of the NES/SNES.

State Machine: The state machines are shown are the following diagrams and table. The SNES is not shown as it is very similar to the NES Controller state, however we do have a diagram of the timings of the SNES controller.

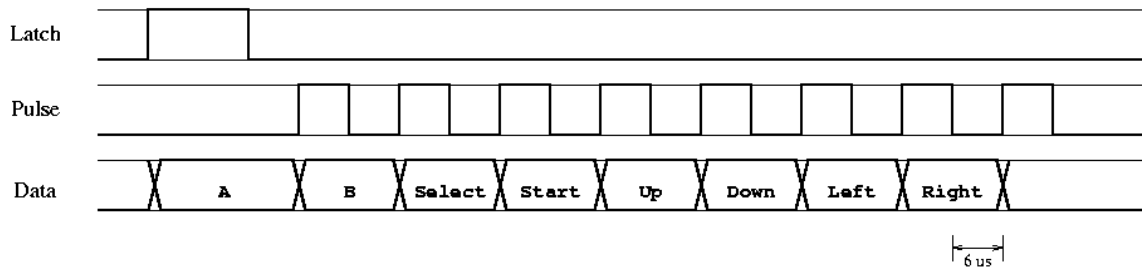
NES Controller States

#	STATES (Transitions)	Data (Actions)
1	Latch On 1	
2	Latch On 2	
3	Latch Off	buttons[0] = A state
4	(PULSE STARTS) B On	
5	B Off	buttons[1] = B state
6	Select On	
7	Select Off	buttons[2] = Select state
8	Start On	
9	Start Off	buttons[3] = Start state
10	Up On	
11	Up Off	buttons[4] = Up state
12	Down On	
13	Down Off	buttons[5] = Down state
14	Left On	
15	Left Off	buttons[6] = Left state
16	Right On	
17	Right Off	buttons[7] = Right state

buttons[0] = A
buttons[1] = B
buttons[2] = Select
buttons[3] = Start
buttons[4] = Up
buttons[5] = Down
buttons[6] = Left
buttons[7] = Right

Example: 01010011 for NES controller have buttons: A, B, Up, and Left pressed.

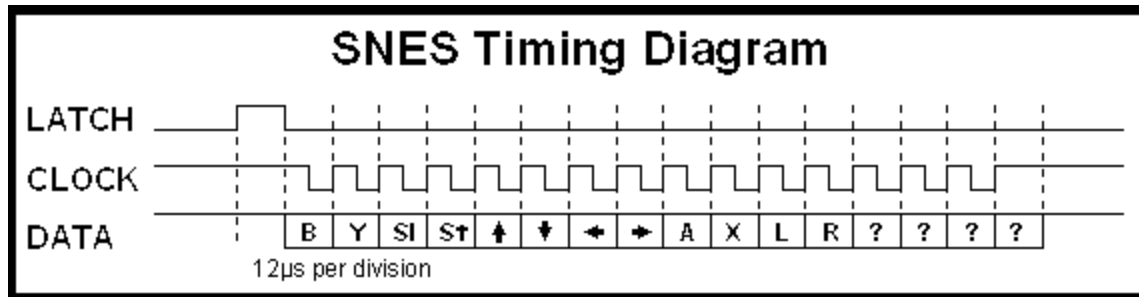
NES Controller Diagram



Datasheet of NES Controller

SNES

SNES Controller Diagram:



Datasheet of SNES Controller

buttons[0] = B
buttons[1] = Y
buttons[2] = Select
buttons[3] = Start
buttons[4] = Up
buttons[5] = Down
buttons[6] = Left
buttons[7] = Right
buttons[8] = A
buttons[9] = X
buttons[10] = Trigger Left
buttons[11] = Trigger Right

Example: 12 bits: 101100100101 for the SNES controller have buttons: Trigger right, X, A, Down, Select, and B are press

We switched the bits as 1 being buttons pressed and 0 for not pressed. We send the entire 8 bits of the NES controller and 12 bits of the SNES controller to a button register for the CPU, PPU, and etc. to use..

Testing

We will go over three types of testing that we have done on our NES FPGA emulator:

- Unit testing
- Integration testing
- Acceptance testing

Unit testing:

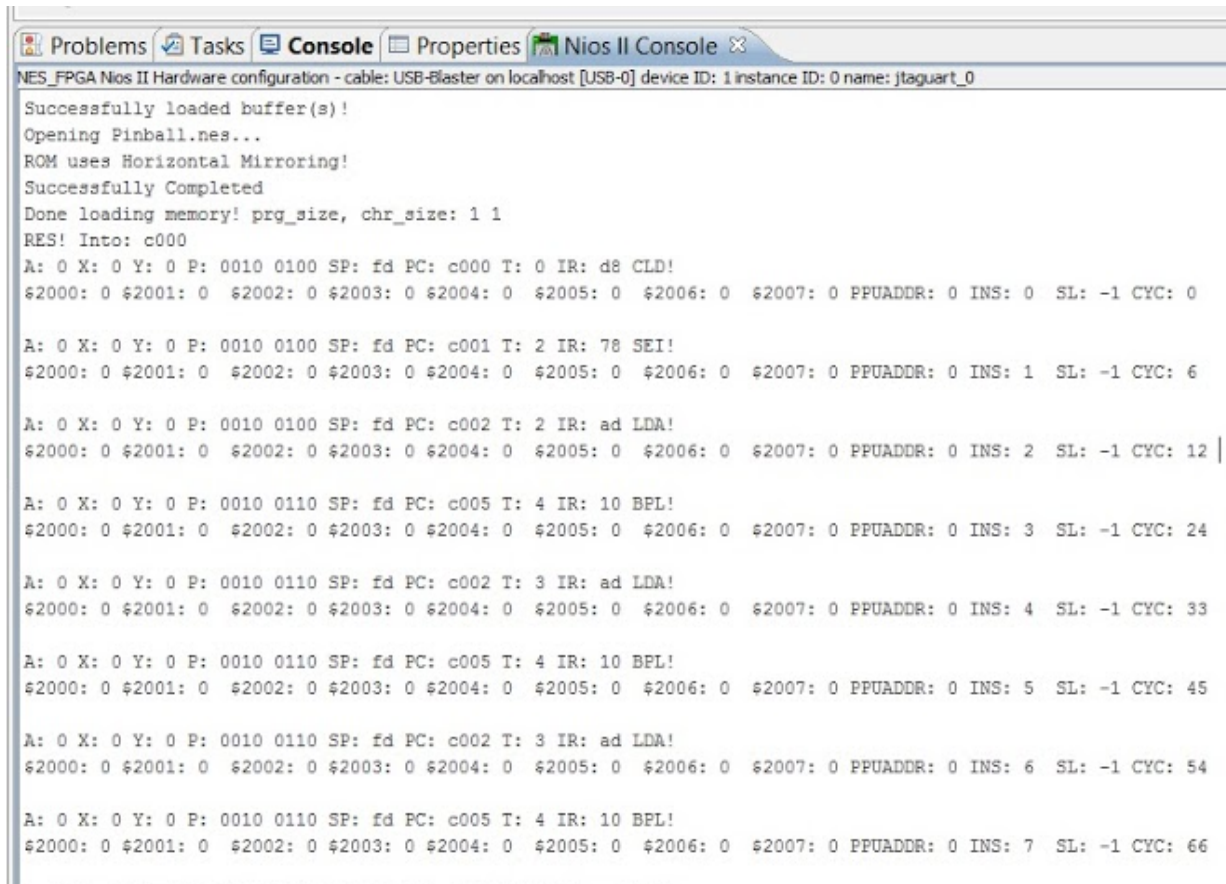
Unit testing outline

- CPU
- PPU
- SD Card Interface
- VGA
- Controller

We deal with testing with a lot of outputs to either the terminal through the LED lights on the FPGA board. The variety of methods provides versatility when it comes testing. The LED lights definitely let us know what parts of the code do and do not work. The testing process helped us accurately know if everything up and running correctly.

CPU:

We test the CPU by outputting every instruction on every cycle. While each instruction executes, we get to see the IO registers and their values in every one of these cycles. We were also able to use some NES test roms, which essentially test out each instruction individually and lets us know if a certain instruction is implemented properly or not. Note that we don't need the PPU implemented to be able to test out the CPU as well.



```

NES_FPGA Nios II Hardware configuration - cable: USB-Blaster on localhost [USB-0] device ID: 1 instance ID: 0 name: jtaguart_0
Successfully loaded buffer(s)!
Opening Pinball.nes...
ROM uses Horizontal Mirroring!
Successfully Completed
Done loading memory! prg_size, chr_size: 1 1
RES! Into: c000
A: 0 X: 0 Y: 0 P: 0010 0100 SP: fd PC: c000 T: 0 IR: d8 CLD!
$2000: 0 $2001: 0 $2002: 0 $2003: 0 $2004: 0 $2005: 0 $2006: 0 $2007: 0 PPUADDR: 0 INS: 0 SL: -1 CYC: 0

A: 0 X: 0 Y: 0 P: 0010 0100 SP: fd PC: c001 T: 2 IR: 78 SEI!
$2000: 0 $2001: 0 $2002: 0 $2003: 0 $2004: 0 $2005: 0 $2006: 0 $2007: 0 PPUADDR: 0 INS: 1 SL: -1 CYC: 6

A: 0 X: 0 Y: 0 P: 0010 0100 SP: fd PC: c002 T: 2 IR: ad LDA!
$2000: 0 $2001: 0 $2002: 0 $2003: 0 $2004: 0 $2005: 0 $2006: 0 $2007: 0 PPUADDR: 0 INS: 2 SL: -1 CYC: 12

A: 0 X: 0 Y: 0 P: 0010 0110 SP: fd PC: c005 T: 4 IR: 10 BPL!
$2000: 0 $2001: 0 $2002: 0 $2003: 0 $2004: 0 $2005: 0 $2006: 0 $2007: 0 PPUADDR: 0 INS: 3 SL: -1 CYC: 24

A: 0 X: 0 Y: 0 P: 0010 0110 SP: fd PC: c002 T: 3 IR: ad LDA!
$2000: 0 $2001: 0 $2002: 0 $2003: 0 $2004: 0 $2005: 0 $2006: 0 $2007: 0 PPUADDR: 0 INS: 4 SL: -1 CYC: 33

A: 0 X: 0 Y: 0 P: 0010 0110 SP: fd PC: c005 T: 4 IR: 10 BPL!
$2000: 0 $2001: 0 $2002: 0 $2003: 0 $2004: 0 $2005: 0 $2006: 0 $2007: 0 PPUADDR: 0 INS: 5 SL: -1 CYC: 45

A: 0 X: 0 Y: 0 P: 0010 0110 SP: fd PC: c002 T: 3 IR: ad LDA!
$2000: 0 $2001: 0 $2002: 0 $2003: 0 $2004: 0 $2005: 0 $2006: 0 $2007: 0 PPUADDR: 0 INS: 6 SL: -1 CYC: 54

A: 0 X: 0 Y: 0 P: 0010 0110 SP: fd PC: c005 T: 4 IR: 10 BPL!
$2000: 0 $2001: 0 $2002: 0 $2003: 0 $2004: 0 $2005: 0 $2006: 0 $2007: 0 PPUADDR: 0 INS: 7 SL: -1 CYC: 66

```

Output of every instruction by the CPU.

PPU:

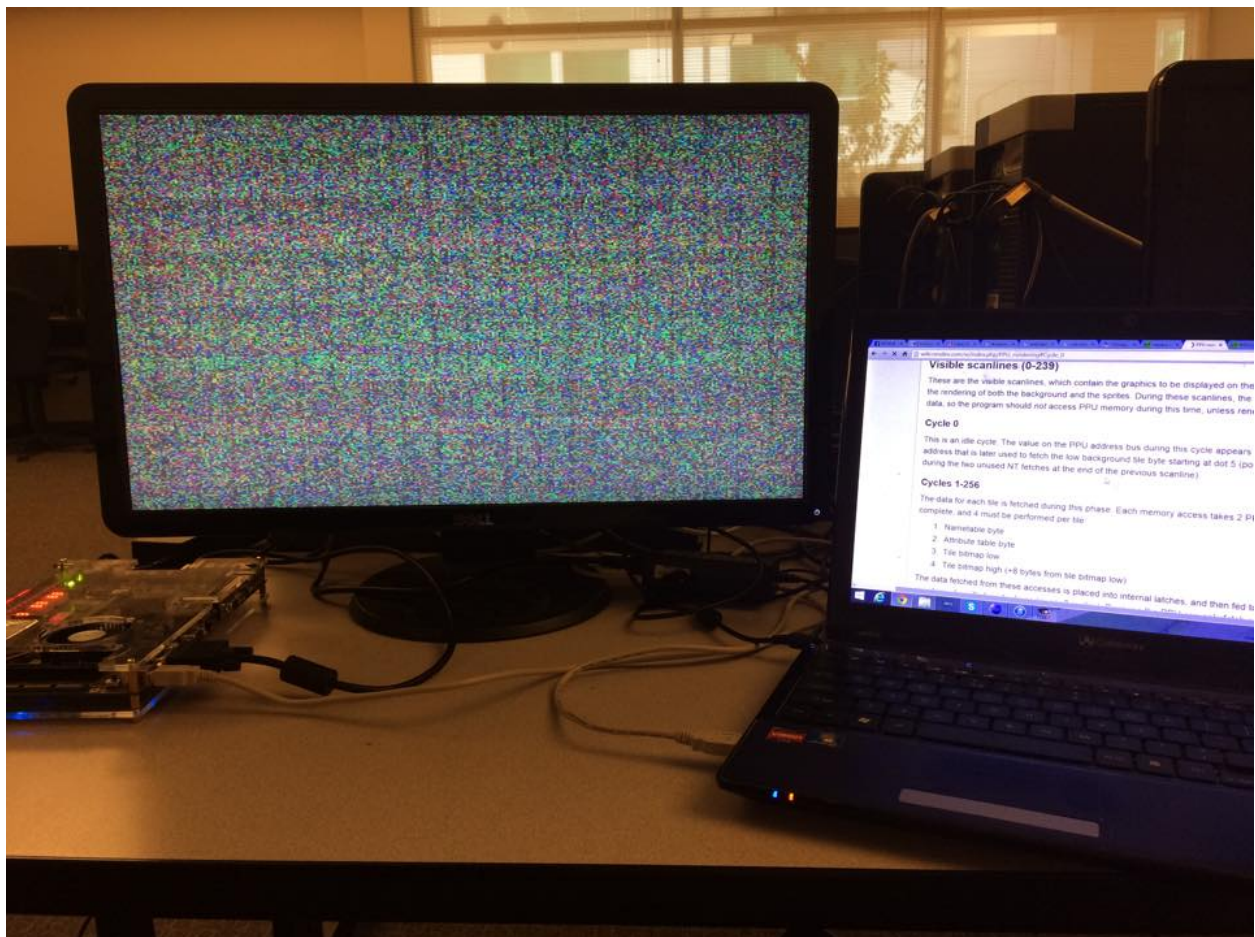
Rendering was tested on the PPU by outputting the contents of the nametables. We used an emulator and compared the tiles to be outputted to what our emulator was displaying in order to confirm that it was correct.

SD Card Interface:

We tested the SD Card interface by writing our modules for reading data from the SD card. We first preloaded the card with appropriate roms, confirmed that the interface was able to load the rom, and compare bytes being loaded with bytes being shown on a Hex editor program.

VGA:

The VGA is easily tested by checking what is output to the screen. We used different VGA outputs including projectors and monitors to show a variety of outputs. Each screen contains different resolutions as well requiring different usages of numbers for the porches and timing. At first each screen showed no inputs detected. Eventually we were able to change the screens into purely black screens receiving no RGB signals. Finally by extensive changes to the timing signals for the VGA, we were able to output colors to the screens. After finding the appropriate clock, we tested each color with a variety of different scenarios to test on each pixel of the screen. The VGA worked out perfectly in the Verilog state. We also converted it for use through the NIOS II Processor which allowed us to run the VGA portion under C. However this proves slow to load compared to the Verilog portion of the VGA. This was one example of printing out different colors to each pixel of the screen.



VGA Controller reading random data from the SRAM frame buffer (initial startup)

Controllers:

The LED lights are used extensively for the NES Controller as we get to see what buttons are received by the FPGA board. At first, only some buttons were showing up on the LED lights, but as we progressed we eventually found all lights lighting up accordingly to each button press. Each LED lights represents the bits of each button that we send throughout the NES Emulator. Receiving the button input turns on the LED light. There are 8 lights corresponding the NES controller and 12 lights for the SNES controller.

Integration testing:

The great thing about the FPGA board is that it deals with the connections of each component separately. For example, we can take inputs from the NES Controller, send the inputs to a register that would send information to the CPU, APU, and PPU. The CPU would send information to the APU and PPU, and the PPU would send information to the VGA screen. All of these things can be tested separately first of all to see if each one is functioning correctly. We would slowly integrate each feature together to test if small incrementals of collaborating components would still successfully work. We continue on with this method to eventually achieve a fully working NES Emulator on a FPGA board.

Acceptance testing:

The most important part of the NES Emulator is getting the game to output on the screen where we can send inputs to the game from the controller. That would be the main acceptance testing for the game as it shows that the NES Emulator actually works the way it is suppose to.

Maintenance Plan

Our plan for maintaining the project due to future environmental changes involves working with newer FPGAs. Luckily, our main backbone for our emulator wouldn't have to be altered, since all we would have to change to accommodate for a new FPGA would be dealing with drivers for peripherals. Since our project does rely on the RISC Nios processor, we would most likely have to keep up with Altera-only models of newer FPGAs. Since we weren't able to parallelize some PPU and CPU, improving performance would simply rely on the updated platform to be faster.

Conclusions

In conclusion, our NES FPGA Emulator involved a large amount of collaboration, and being able to design an entire system, component by component, and piecing it all together. This was a big step from our regular courses, as it required our work to be presentable in a manner that was understandable and clearly documented for our partners to be able to digest and work with.

Personal Growth

We personally learned a lot about being able to work in a team on a semi-large scale project. This wasn't with just being able to share and connect our work together, but about communication. Being able to clearly outline what needs to be done between each partner, and laying down the ground rules for well and timely a task needed to be done was crucial to getting a working prototype.

What Surprised Us

What surprised us most about the project was the sheer amount of research required to truly be able to understand the ins and outs of the NES. We can say with confidence that nearly half of the time spent working on the project, was spent going through dozens of documentations, some from possibly unreliable or outdated sources. Something that also surprised us was the lack of aid we received from the community at Altera, specifically with the DE2i-150 boards we had. Nearly all documentation we could find about a certain peripheral or driver related to our board was actually referring to older versions of our board. Not to mention, the lack of support for the many errors or bugs we encountered using the software, mainly Quartus II and the Eclipse Development Tool for Nios II.

Things done differently

One thing we would definitely do differently if we had another chance, would be to make sure everyone learns how to use the tools prior to actually starting the project. Also, if we had a better sense of communication, there would have been a better ordering of the different tasks that needed to be done. There were many times that not everyone would be on the same page in terms of understanding everything, which would have been easily remedied if we approached each other in a better manner.

To conclude

In short, we were able to implement controls, a CPU core, and the video rendering, all to a certain degree to which we were able to get a decently working prototype. The only component we would be missing would be the audio processing. We used Verilog and C, thanks to the NIOS II processor we configured to be embedded on the FPGA.

References

Corporation, Altera. *Nios II Hardware Development Tutorial* (n.d.): n. pag. Web.

Online at http://www.altera.com/literature/tt/tt_nios2_hardware_tutorial.pdf

"6502 Programmers Reference" *Ntlworld*. N.p., n.d. Web. 14 Aug. 2014.

Online at http://homepage.ntlworld.com/cyborgsystems/CS_Main/6502/6502.htm

"Instruction Reference" *Obelisk*. N.p., n.d. Web 18 Aug. 2014.

Online at <http://www.obelisk.demon.co.uk/6502/reference.html>

"Fpga4fun.com - Pong Game." *Fpga4fun.com - Pong Game*. N.p., n.d. Web. 16 Aug. 2014.

Online at <http://www.fpga4fun.com/PongGame.html>

"The NES Controller Handler." *MIT*. N.p., n.d. Web. 10 Aug. 2014.

Online at <http://www.mit.edu/~tarvizo/nesc-controller.html>

"Nesdev Wiki." *Nesdev Wiki*. N.p., n.d. Web. 03 Aug. 2014.

Online at http://wiki.nesdev.com/w/index.php/Nesdev_Wiki

"Site Index." *NES Info, Programs, and Demos*. N.p., n.d. Web. 03 Aug. 2014.

Online at <http://nesdev.com/>

Acknowledgments

We would like to thank Dr. Philip Brisk for letting us use the FPGA boards and for advising us throughout the program. We would also like to take this time to thank Elizabeth Claassen Thrush for all of her help on setting up the Summer Abroad program to Switzerland. And, finally, we'd like to thank UCR's staff and faculty for giving us the opportunity to go on this trip. The program would not be possible without any of you. Thank you.