SNES on an FPGA

Team Bubba Gump Shrimp Co.

Rachael Harding
Dustin Musselman
Grant Newsome
Lincoln Roop

Overview

Our capstone design project was to build a Super Nintendo Entertainment System on an FPGA.

Game Description

The SNES was a popular game console in the early 1990's. Over 40 million units were sold worldwide. Popular games available on the console include Super Mario World and The Legend of Zelda: A Link to the Past .

The Super Nintendo Entertainment System was a 16-bit video game console designed by the Nintendo corporation that was a successor to their original and highly successful Nintendo Entertainment System. It contains four chips that run simultaneously to produce the video and sound while interfacing to the two controller inputs used by the user of the system and a cartridge providing the game data. Two chips control the sound system, a Sony SPC700 processor and a custom designed Digital Signal Processing (DSP) chip, which work in tandem with each other and a digital to audio converters to generate the stereo sound for the system. These each share a 64kB of SRAM that are interfaced by the different clock times associated with each chip.

In parallel to the sound system we have the video cores, which are two nearly identical processors that work almost seamlessly (called one unit: the PPU) to produce the video. The PPU has 64kB of SRAM itself, along with some extra memory specifically for storing sprite data for it's video output. The PPU also features several different video Modes (0-7) which can lead to various visual effects dependent on design. These units are controlled by a single main CPU which is a Ricoh 5A22 processor, which is based on a 16-bit 65c816 core and has added DMA/HDMA, parallel I/O processing, and hardware multipliers and dividers. It has 128kB of SRAM for it's memory.

The main operation of the SNES comes from the data passed in on an input data bus provided by the cartridge inserted into the top of the system. These cartridges contain a variety of chips, and when powered interact with the main CPU to provide information to the audio and video units of the system. The main CPU is also where the controllers interface to the game, providing a 'central hub' to the system that controls both input and processing. Input pins vary from the Japanese PAL design compared to the North American design (which we are basing our project off of.)

Hardware Description

The SNES hardware consists of a cartridge, controllers, and four main cores: the CPU, DSP, SPC700, and PPU.

Cartridge & Controllers

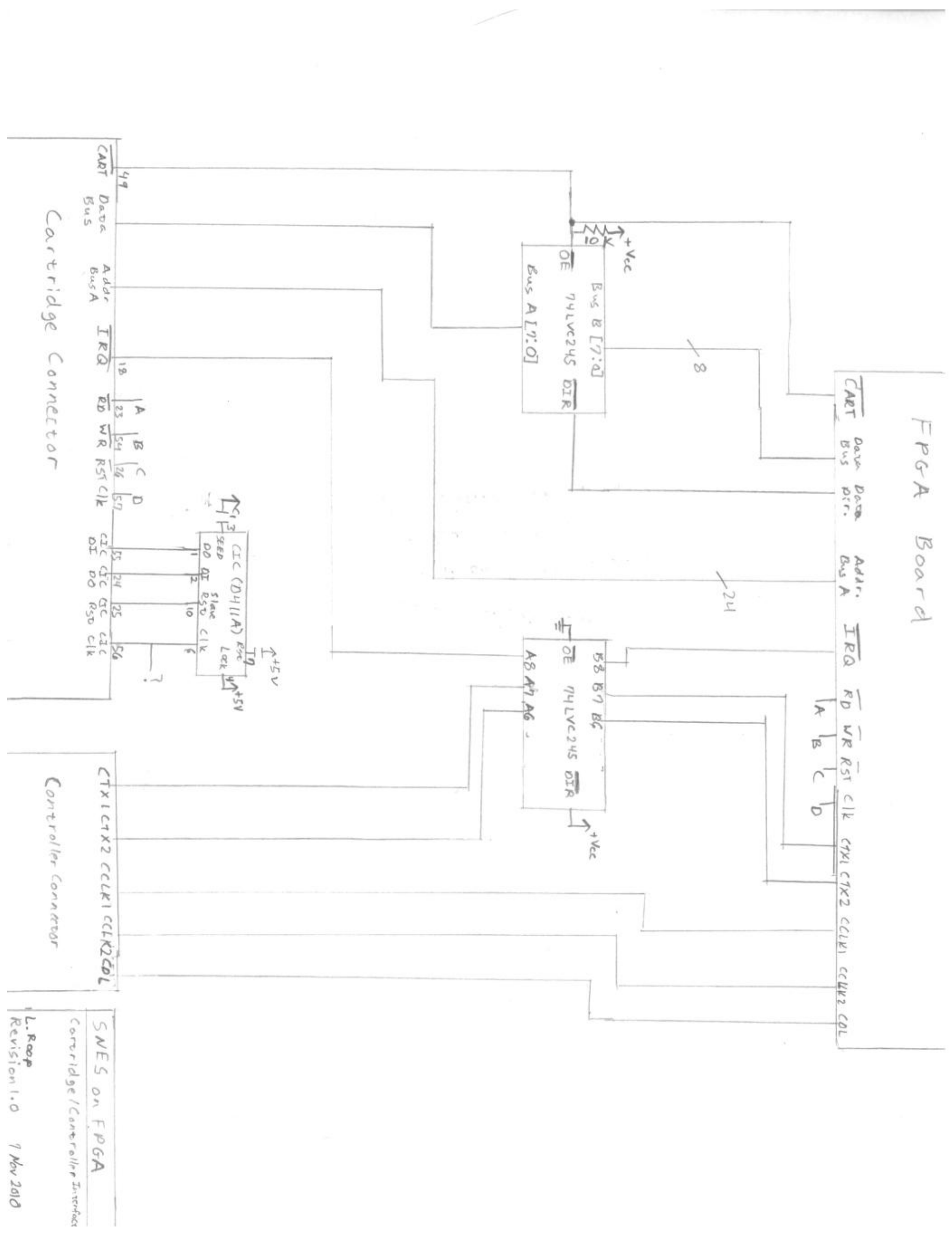
As we wanted to use actual SNES cartridges and controllers in our project, we had to deal with interfacing these devices to the FPGA board, which presented us with three problems:

- The connectors used aren't just standard parts that can be purchased from an electronics supplier.
- The SNES cartridges and controllers use 5V TTL signaling versus the 3.3V TTL used by our FPGA board.
- The SNES uses a cartridge lock-out system designed to make pirating cartridges or consoles more difficult. Since our project is essentially a pirated console, this will have to be dealt with.

The physical connectors were scavenged from a used Super Nintendo console. These were carefully desoldered from the mainboard of the SNES, and then soldered to a piece of perfboard (A type of premade PCB with a grid of plated holes on it). For the lock-out system, we scavenged a lockout chip from an old SNES cartridge, as the chips found in consoles and cartridges are identical in function, but the console one is in a surface-mount package as opposed to the dual inline package used in most cartridges.

To deal with the difference in voltage levels, we used Texas Instruments SN74LVC245AN 8-bit bidirectional level shifter ICs. These ICs can accept inputs of up to 6.5 volts and output the appropriate logic level, but limited by the voltage supply to the IC. Here, we used the 3.3V DC rail from the FPGA board to power our level shifters, and the logic high output was around 3.2V, which will work fine with the FPGA board's inputs. These level shifters were used to shift down the voltage of any inputs from the cartridge and controllers to the FPGA board (Cartridge data bus and a few control signals, controller data inputs, see schematic.), level shifting in the other direction was not necessary, as the 3.3V logic high output of the FPGA board is high enough to register as a logic high on the older 5V logic.

This circuitry was wired, and connected to the FPGA board using the user I/O expansion headers on the board. The connectors on the board are standard 0.1" spaced inline pin headers, we chose to use the same connectors on our interface circuit. During assembly, we checked each new solder joint with a multimeter to ensure that the correct connection had been made, and no incorrect connections were made. After the circuit was built, we injected test voltages with a standard bench power supply to ensure the proper operation of the level shifter ICs. Finally, we were able to confirm that this circuit works properly, as we were able to read valid data from game cartridges and controllers, all without damaging the FPGA board, game cartridges, or controllers.



CPU

The CPU in the SNES is a modified Ricoh 5A22. We were fortunate to get a synthesizable Verilog description of the 65C816 from Western Design Center (WDC), which was used as the core in the SNES CPU. We added several peripherals to the CPU.

- Multiplier: An 8-bit multiplier
- Divider: A fixed-point divider with 16-bit dividend and 8-bit divider
- Interrupt Module: Gathered interrupts from other devices (e.g., cartridge) to feed into the core

- Work RAM (WRAM): A 128KB byte-addressable RAM accessible by the CPU (see “CPU Memory Map” below)
- DMA/HDMA: A module that quickly transfers data from the Cartridge ROM to WRAM, ROM to VRAM, WRAM to VRAM, or vice versa.
- Memory Map registers: A series of registers mapped to locations in memory (see “CPU Memory Map” and “Hardware Registers” below)
- Timers: Timers for vertical and horizontal blanking periods

DMA/HDMA

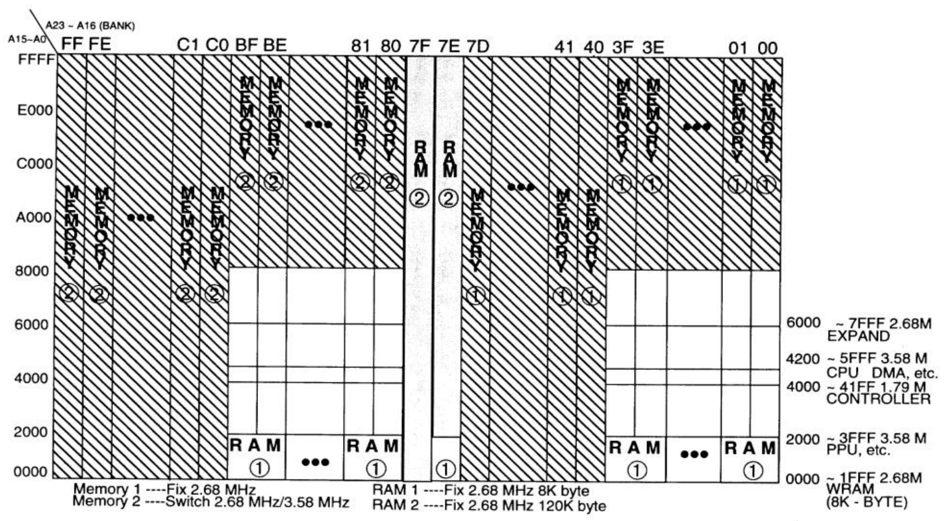
The DMA/HDMA has 8 channels which can hold information for DMA/HDMA transfers. HDMA is prioritized over DMA, and lower channels are prioritized over higher channels. A DMA transfer begins when the CPU enables a channel by writing a 1 to the DMA enable register bit corresponding to that channel. Execution is suspended in the CPU until the DMA completes. HDMA only transfers 1-4 bits on every horizontal blank, which enables certain special effects on the screen.

The source address (usually ROM or WRAM) can either increment or decrement. The destination address (usually a PPU register) can be accessed in a pattern determined by the 3-bit bus mode:

DMA:
000, 010 : B B B B ....
001 : B B+1 B B+1 B ....
011 : B B B+1 B+1 B ....
100 : B B+1 B+2 B+3 B B+1 ....

HDMA: (per line)
000 : B
001 : B B+1
010 : B B
011 : B B B+1 B+1
100 : B B+1 B+2 B+3

Figure 2-21-1 Super NES CPU Memory Map


A23 ~ A16 (BANK)
A15~A0
FF FE C1 C0 BF BE 81 80 7F 7E 7D 41 40 3F 3E 01 00
FFFF
E000
C000
A000
8000
6000
4000
2000
0000
MEMORY ②
MEMORY ①
RAM ②
RAM ①
6000 ~ 7FFF 2.68M EXPAND
4200 ~ 5FFF 3.58 M CPU DMA, etc.
4000 ~ 41FF 1.79 M CONTROLLER
2000 ~ 3FFF 3.58 M PPU, etc.
0000 ~ 1FFF 2.68M WRAM (8K - BYTE)
Memory 1 ----Fix 2.68 MHz
Memory 2 ----Switch 2.68 MHz/3.58 MHz
RAM 1 ----Fix 2.68 MHz 8K byte
RAM 2 ----Fix 2.68 MHz 120K byte

SNES DEVELOPMENT MANUAL

2-21-2

(NCL PG 37)

Hardware Registers

Address	Register name	Comment
0x2100	Screen Display Register	a000bbbb a: 0=screen on 1=screen off, b = brightness
0x2101	OAM Size and Data Area Designation	aaabbccc a = Size, b = Name Selection, c = Base Selection
0x2102	Address for Accessing OAM	
0x2104	OAM Data Write	
0x2105	BG Mode and Tile Size Setting	abcdefff abcd = BG tile size (4321): 0 = 8x8 1 = 16x16, e = BG 3 High Priority, f = BG Mode
0x2106	Mosaic Size and BG Enable	aaaabbbb a = Mosaic Size b = Mosaic BG Enable
0x2107	BG 1 Address and Size	aaaaaabb a = Screen Base Address (Upper 6-bit), b = Screen Size
0x2108	BG 2 Address and Size	aaaaaabb a = Screen Base Address (Upper 6-bit), b = Screen Size
0x2109	BG 3 Address and Size	aaaaaabb a = Screen Base Address (Upper 6-bit), b = Screen Size
0x210A	BG 4 Address and Size	aaaaaabb a = Screen Base Address (Upper 6-bit), b = Screen Size
0x210b	BG 1 & 2 Tile Data Designation	aaaabbbb a = BG 2 Tile Base Address, b = BG 1

		Tile Base Address
0x210c	BG 3 & 4 Tile Data Designation	aaaabbbb a = BG 4 Tile Base Address, b = BG 3 Tile Base Address
0x210d	BG 1 Horizontal Scroll Offset	Scroll offset registers are all 16 bits wide.
0x210e	BG 1 Vertical Scroll Offset	
0x210f	BG 2 Horizontal Scroll Offset	
0x2110	BG 2 Vertical Scroll Offset	
0x2111	BG 3 Horizontal Scroll Offset	
0x2112	BG 3 Vertical Scroll Offset	
0x2113	BG 4 Horizontal Scroll Offset	
0x2114	BG 4 Vertical Scroll Offset	
0x2115	VRAM Address Increment Value	
0x2116	Address for VRAM Read/Write (Low Byte)	
0x2117	Address for VRAM Read/Write (High Byte)	
0x2118	Data for VRAM Write (Low Byte)	
0x2119	Data for VRAM Write (High Byte)	
0x211a	Initial Setting for Mode 7	aa0000bc a = Screen Over b = Vertical Flip c = Horizontal Flip
0x211b	Mode 7 Matrix Parameter A	Registers 211b through 2120 are 16 bits wide. 0x211B is also used as the 16-bit multiplicand for registers 0x2134-6 (write twice) 0x211C is also used as the 8-bit multiplier for registers 0x2134-6
0x211c	Mode 7 Matrix Parameter B	
0x211d	Mode 7 Matrix Parameter C	
0x211e	Mode 7 Matrix Parameter D	
0x211f	Mode 7 Center Position X	
0x2120	Mode 7 Center Position Y	
0x2121	Address for CG-RAM Write	
0x2122	Data for CG-RAM Write	
0x2123	BG 1 and 2 Window Mask Settings	aaaabbbb a = BG 2 Window Settings b = BG 1 Window Settings
0x2124	BG 3 and 4 Window Mask Settings	aaaabbbb a = BG 4 Window Settings b = BG 3 Window Settings
0x2125	OBJ and Color Window Settings	aaaabbbb a = Color Window Settings b = OBJ Window Settings
0x2126	Window 1 Left Position Designation	
0x2127	Window 1 Right Position Designation	
0x2128	Window 2 Left Postion Designation	
0x2129	Window 2 Right Postion Designation	

0x212a	BG 1, 2, 3 and 4 Window Logic Settings	aabbccdd a = BG 4 b = BG 3 c = BG 2 d = BG 1
0x212b	Color and OBJ Window Logic Settings	0000aabb a = Color Window b = OBJ Window
0x212c	Background and Object Enable (Main Screen)	000abcde a = Object b = BG 4 c = BG 3 d = BG 2 e = BG 1
0x212d	Background and Object Enable (Sub Screen)	000abcde a = Object b = BG 4 c = BG 3 d = BG 2 e = BG 1
0x212e	Window Mask Designation for Main Screen	000abcde a = Object b = BG 4 c = BG 3 d = BG 2 e = BG 1
0x212f	Window Mask Designation for Sub Screen	000abcde a = Object b = BG 4 c = BG 3 d = BG 2 e = BG 1
0x2130	Initial Settings for Color Addition	aabb00cd a = Main Color Window On/Off, b = Sub Color Window On/Off, c = Fixed Color Add/Subtract Enable, d = Direct Select
0x2131	Add/Subtract Select and Enable	abcdefgh a = 0 for Addition, 1 for Subtraction, b = 1/2 Enable c = Back Enable, d = Object Enable, efgh = Enable BG 4, 3, 2, 1
0x2132	Fixed Color Data	abcddddd a = Blue b = Green c = Red ddddd = Color Data
0x2133	Screen Initial Settings	ab00cdef a = External Sync, b = ExtBG Mode, c = Pseudo 512 Mode, d = Vertical Size, e = Object-V Select, f = Interlace
0x2134	Multiplication Result (Low Byte)	
0x2135	Multiplication Result (Mid Byte)	
0x2136	Multiplication Result (High Byte)	
0x2137	Software Latch for H/V Counter	
0x2138	Read Data from OAM (Low-High)	
0x2139	Read Data from VRAM (Low)	
0x213a	Read Data from VRAM (High)	
0x213b	Read Data from CG-RAM (Low-High)	
0x213c	H-Counter Data	
0x213d	V-Counter Data	
0x213e 0x213f	PPU Status Flag	
0x2140 0x2141 0x2142 0x2143	APU I/O Port	

0x4200	NMI, V/H Count, and Joypad Enable	a0bc000d a = NMI b = V-Count c = H-Count d = Joypad
0x4201	Programmable I/O Port Output	
0x4202	Multiplicand A	
0x4203	Multplier B	
0x4204	Dividend (Low Byte)	
0x4205	Dividend (High-Byte)	
0x4206	Divisor B	
0x4207	H-Count Timer (Upper 8 Bits)	
0x4208	H-Count Timer MSB (Bit 0)	
0x4209	V-Count Timer (Upper 8 Bits)	
0x420a	V-Count Timer MSB (Bit 0)	
0x420b	Regular DMA Channel Enable	abcdefgh a = Channel 7...h = Channel 0: 1 = Enable 0 = Disable
0x420c	H-DMA Channel Enable	abcdefgh a = Channel 7 .. h = Channel 0: 1 = Enable 0 = Disable
0x420d	Cycle Speed Designation	0000000a a: 0 = 2.68 MHz, 1 = 3.58 MHz
0x4210	NMI Enable	
0x4211	IRQ Flag By H/V Count Timer	
0x4212	H/V Blank Flags and Joypad Status	
0x4213	Programmable I/O Port Input	
0x4214	Quotient of Divide Result (Low Byte)	
0x4215	Quotient of Divide Result (High Byte)	
0x4216	Product/Remainder Result (Low Byte)	
0x4217	Product/Remainder Result (High Byte)	
0x4218	Joypad 1 Data (Low Byte)	abcd0000 a = Button A b = X c = L d = R
0x421a	Joypad 2 Data (Low Byte)	
0x421c	Joypad 3 Data (Low Byte)	
0x421e	Joypad 4 Data (Low Byte)	
0x4219	Joypad 1 Data (High Byte)	abcdefgh a = B b = Y c = Select d = Start efgh = Up/Dn/Lt/Rt
0x421b	Joypad 2 Data (High Byte)	
0x421d	Joypad 3 Data (High Byte)	
0x421f	Joypad 4 Data (High Byte)	

DMA Registers

'X' being from 0 to 7:

Address	Register name	Comment
0x43X0	Parameters for DMA Transfer	*ab0cdeee a = Direction b = Type c = Inc/Dec d = Auto/Fixed e = Word Size Select*
0x43X1	B Address	
0x43X2	A Address (Low Byte)	
0x43X3	A Address (High Byte)	
0x43X4	A Address Bank	
0x43X5	Number Bytes to Transfer (Low Byte) (DMA)	
0x43X6	Number Bytes to Transfer (High Byte) (DMA)	
0x43X7	Data Bank (H-DMA)	
0x43X8	A2 Table Address (Low Byte)	
0x43X9	A2 Table Address (High Byte)	
0x43Xa	Number of Lines to Transfer (H-DMA)	

DSP

When compiling data on the DSP, we had been fortunate to come to the realization that people have, in the past, modeled the DSP on FPGA's before. Long after the time of the SNES, many independent technophiles came up with a format for the sound binary file associated with the cartridges called an .SPC file. This, when pushed into a software-designed parser, can play the sounds associated with whichever game the .SPC file was created from. There were some ambitious graduate students at various universities that wrote the actual DSP hardware to generate the sound on an FPGA already, and while it was done in VHDL, it allowed us to have a fairly good idea on how the DSP should be designed. This code, along with several documents explaining how the different parts of the DSP worked, allowed us to design it properly.

In brief, the DSP is a highly customized and advanced 8-voice audio processor that outputs 16-bit stereo sound at 32 KHz. It decompresses and processes audio samples as opposed to actually generating audio signals with oscillators. These samples are stored in the SPC700/DSP shared memory, and the SPC700 sets various control registers inside the DSP to point it to the correct source of samples for each voice.

What we noticed about the DSP, as opposed to the SPC700 was that it was much more modular a design. There are several major subsystems to the DSP, many of which we were able to implement and test as individual Verilog modules:

- *Sample decoder: Converts compressed audio samples to raw digital audio*
- *Pitch Modulator: Adjusts the pitch of each voice by a selectable value, and for voices 1 through 7 by the output value of the previous voice.*
- *Noise Generator: Generates a pseudorandom noise signal that can be substituted for the output of each of the 8 voices by setting a bit in a control register.*
- *Envelope Generator: Generates a volume envelope for each channel. Supports ADSR (Attack, Decay, Sustain, Release), simple direct gain, and simple variable gain.*
- *Echo system: Produces an echo effect with selectable delay.*
- *Per-channel and master volume control.*

The DSP follows a very rigid 32-step process to generate each stereo output sample. Within each 32-step block, an output sample is generated for each voice that is keyed on. Then, these samples are combined along with the sample generated by the echo system (if enabled), the master volume is applied, and the samples are output. In the original SNES, two 16-bit samples, left and right, are output to a serial DAC; our implementation behaves similarly, but outputs to the AC'97 control logic. Because of this rigid behavior, our DSP's control logic is a simple hard-coded finite state machine that has a maximum of 24 states for each of the 32 steps.

SPC700

The SPC700 is an 8-bit CISC CPU core that was originally manufactured by Sony. Upon initial review we were skeptical that all 256 op-codes in the chip were actually used by the games that were played on the console, but according to the documentation that we eventually read all of the codes could have been used by the cartridges. o avoid having to return to the chip and spend more time implementing ops at the end, we decided to include all of the opcodes into our design. The SPC700 is large in scope, but following basic processor design priciples we were taught in 18-447 we were able to model the core with a standard data path and FSM.

As we had no layout diagram of the core itself, we reversed engineered the core from several ISA models we had found that were being used by emulator designers to provide the sound emulation for their various SNES ROM emulators. Our design consists of a decode stage which upon the collection for the opcode form memory decides the further actions taken by the chip for that operation cycle. In all of the ISA diagrams we were given the amount of cycles each operation needed to take, so ha gave us a fairly solid base to work from when designing the interactions needed in the decode stage of our chip.

We knew here was no easy way to get around the 256 ops needed by the chip, so we designed the decode around a large case-stamen that sets the various control signals throughout the code. there were some similarities in ops that shortened our codebase, but a lot of the operations required very unique control throughout the chip, leading to our decoder being very large (in terms of lines of Verilog.) Our ALU is fairly straight forward, taking in two line inputs and producing an output based on a signal generated from the decoder.

Once the decoder and ALU were finished, depending on a given op, there may be aneed for memory access. As at this point we were unaware as to whether or not our memory coudl be designed as synchronous or asynchronous, we decided to play it safe and model the system as a synchronous read/write hat will return or finalize writing on the next clock cycle. This made our cycle conditions fairly tricky, as before we were without the need to wait for memory and were able to perform all ops in one cycle (decoder -> alu -> register/memory all in one cycle was feasible by our design.) With reading from memory being a one cycle operation, it brought our minimum cycles per op to two, while giving us added cycles to any op that required memory operations. After reviewing this we decided we had sufficient cycles, even with the cycles for reads and writes, to have our design work.

PPU

We started building the PPU about a month and a half into the project. On the actual system, the PPU is comprised of two closely integrated ICs. In our design, however, we decided to make it just one system. It works by monitoring a number of registers and then translating that data into the video. The registers are address locations 0x002100 to 0x00213F. These are memory mapped from the CPU and are stored in the PPU itself. The 8-bit address bus "Bus B" controls which register is updated or read from by two additional control lines, read and write. All of these are handled by the CPU wrapper.

The registers' purposes are listed above and many of them seem self explanatory. It is important to note that it doesnt not matter how the data is getting to these registers, whether the CPU is using the DMA channels or just updating a single register at a time, it is all viewed the same by the PPU. The main data transfer occurs in registers 2116, 2117, 2118, and 2119. Everytime the address is updated {2117,2116} the PPU reads the data from {2119,2118} and writes it to the appropriate location and then increments by the value in register 2115. The CPU then writes new data and this continues, copying data into the PPUs VRAM, which is 64kB in size. Color data is stored in a 256x15 bit separate RAM and the sprite data is stored into yet another 544 bytes, called the OAM. These locations are written to much the same way as VRAM, just in different registers. Any writing to the PPU memory occurs only on vertical blanks.

After data is loaded into memory, it is time to put a picture to the screen. This uses almost all of the other registers to create the picture. A few of the important registers are the BG Mode register which sets the possible different sprite sizes and background resolutions are to be used. Modes 0-6 are pretty much the same, but Mode 7 uses a lot of complicated Matrix multiplication that we never got to. This was only used in later games. At any given time, there are four backgrounds that can be overlapped on the screen. The sizes of them in memory are actually much larger than the actual screen resolution itself and can be scrolled all over the screen. This also depends on the mode and a few other settings. As I stated earlier, most of the register purposes are self explanatory such as the window masks and window positions.

Again, we never got to fully test the PPU. We were able to load information through a various number of tests and then produce backgrounds and scroll them with buttons. We were confident that if we loaded the correct information, we could create a picture close to what was created by the original system pending some debugging. But seeing as the CPU never got out of its loading stage, we could never get enough information from the cartridge to test it with actual SNES video.

Approach

We knew from the beginning that this was a large project, and we wanted to make as few errors as possible in the design. Therefore we began with thorough research in available documentation and write-ups about the SNES. Most of the documentation we found was programmer-side interface detail written by members of the large SNES fan base. We were fortunate enough to find a manual from Nintendo that detailed the CPU and PPU components. Unfortunately our search did not bring very many documents about the sound system, however. We also tracked down Western Design Center (WDC), the company that designed the actual CPU core for Nintendo, and managed to get in contact with them about potentially using their product in our SNES emulation. They agreed to send us their Verilog after we signed a confidentiality agreement.

Due to the large size of the sound system (2 cores) and the strong push this year to get sound in the games, we focused our first efforts on the SPC700 and the DSP. Due to the limited documentation and the complexity of the chips, the SPC700 took until after mid-semester to complete, with the DSP an on-going effort throughout the semester.

At mid-semester, our team grew, which allowed us to tackle more pieces of the design at once. We finally obtained a synthesizable core from WDC, which allowed us to work on the CPU. We also started building the physical system, cartridge and controllers from a real SNES.

After completing the SPC700, we moved on to the PPU. With two weeks to go until demo, we began integrating the CPU and PPU with the working cartridge.

Design Partitioning

As alluded to previously, we partitioned the design into the four cores (CPU, PPU, SPC700 and DSP) and the physical hardware (cartridge and controllers). We divided the work between the team members in the following way:

Rachael: CPU (joined mid-semester)
Lincoln: DSP, cartridge, controllers
Dustin, Grant: SPC700, PPU

Tools and design methodology

We used the Xilinx toolset for our work. Our hardware description language of choice was Verilog.

We designed the hardware based on the documentation we found from fans and Nintendo.

Testing and verification methodology

We used a combination of simulation via ModelSim and synthesis in Xilinx ISE to test the cores and verify correct operation.

Status and future work

As of demo day, we have the cores completely implemented in Verilog (minus some instructions in the DSP and SPC700) and the physical hardware built and operational. We have a partially integrated CPU/PPU/Cartridge which is not quite operational.

Future work would need to integrate the different cores fully and insert the missing instructions in the DSP and SPC700. In the future additional modes could be added to the PPU which could enable a greater variety of cartridges.

What you learned

- How to organize (and not organize) a large-scale project
- How to search for relevant information online

Hindsight

We wish we had a better comprehension of the scope of the project we were taking on. If we had known the sound system was going to take so long, we may have organized the project differently.

Design Decisions

We made several major design decisions that helped, or in some cases hurt, our project

Real Cartridges

Our early decision to build a cartridge reader from real SNES parts, as opposed to reading a ROM off flash, turned out to be a great idea. The reasoning behind this decision was to enable the use of a variety of cartridges (and in a working system to play a variety of games). Allowing the player to choose their game made the project much more appealing.

Although the system did not work as a whole, the cartridge reader worked on the first try. It was an accomplishment that gave us not only something to show for our work, but let us use a variety of cartridges for debugging purposes. Many people who came to our demo were impressed by the cartridge reader we built.

Timing

We decided to work on the sound system first because it had the least documentation, there was a push in class to get sound working on games this year, and it looked easy. These were all the reasons why we should have worked on it last (or, at least, not first).

First, the limited documentation meant that we were trying to figure out what went in a black box. After feeling around for awhile, we found that the black box was more complicated than we initially thought. By that point, weeks had passed.

Second, although pushing sound was good in that most groups end up running out of time for it, there is a reason why groups leave it until the end. Sound is not critical to game play. If the sound is not working, the game can still be played (“It's just muted!”). Without the CPU or video decoder, however, game play is impossible. We learned this the hard way when, instead of running out of time on sound, we ran out of time on system integration, and ended up losing everything.

Third, a good lesson we learned is that if something looks easy, it probably isn't, especially if there is no documentation.

Protocol

We chose to follow the documentation we had to the letter. This led to unnecessary complications during implementation and integration that we could have avoided by figuring out what Nintendo was trying to accomplish and making it simpler. For example, timing in the DMA controller was unnecessarily complex, due to the latency between ROM accesses and chip responses. Since we were building our design on customizable hardware, we probably did not have to incorporate the timing switches.

We also found in integration that the CPU-SPC700 communication protocol is very complex, and could have possibly been simplified.

Advice to Future Groups

To future 545-ers, we suggest the following:

- Get started on implementation early! (Corollary: Get started early!) We spent several weeks on research before starting to implement our project. There is a sweet spot to the amount of research you do (more research to prevent bugs, less research to get more time to debug at the end).
- Be here for Thanksgiving. Unless you have your project done.

To future groups that attempt the SNES:

- Read this document carefully to avoid our pitfalls.

Individual Pages

Rachael

I joined this team at mid-semester after my first team lost a member and dissolved as a result. So I'd like to start with some short advice on choosing your team: Take skill set into account, but also take commitment to the class. I was lucky that my teammate dropped early enough that my team could disperse into other groups (other teams were not so lucky), but joining a group halfway through the class was also a challenge. I had to quickly familiarize myself with their project and group dynamics.

When I joined the SNES team, I began work on the CPU peripherals: the multiplier, divider, and DMA (plus some other small units to control interrupts and other signals). The multiplier and divider took about a day to complete and debug. The DMA, however, was much more complex. The available documentation was written from a programmer's perspective, so it did not include too much information about the underlying hardware. There was also the HDMA, which only transferred a few bytes of data on horizontal blanks (between scan lines). This mode was even more complex. Over time I found that the DMA could have had a much simpler design than implied by the documentation and fan observations. Since the DMA's purpose was to transfer data from ROM/RAM to the PPU, the timing, for example, could have been done much simpler.

I also did significant integration work at the end. I spent a solid 60 hours in lab over Thanksgiving break working on the Cartridge-CPU interface and getting a response from the controllers. I spent the last week of classes in lab debugging the CPU-PPU interface, which did not end up being enough time to complete the integration.

Lincoln

I'm Lincoln Roop, and I worked on the following things for our group:

Obtaining 65c816 source from Western Design Center - 4 hours (Time spent includes
estimated time it took to communicate with WDC and prepare/scan necessary paperwork,
not time spent waiting on e-mail replies.)

Cartridge & Controller Interface circuit
- Research (Finding pinouts, parts, etc.) - 14 hours
- Circuit Design - 4 Hours
- Assembly/Test - 14 hours

This work was completed over the course of 2 1/2 weeks

Audio DSP
- Research - 20 hours
- Coding/Debugging - More than 40 hours

Other work:

Debugging/helping find documentation for the rest of the project.

My impression of the class:

This class is a nice opportunity to work on something approximating a self-directed real world project, but it could use a bit of improvement. Most importantly, more feedback would be nice. While the in-class feedback from group status reports is useful, I think it would be good for students to sit down (maybe once every other week) and meet with the professor or a TA to discuss progress, current issues, and such.

In hindsight, I feel that my group would have chosen a different project if we knew about some of the issues we ran into later on. This isn't a fault in the class, but it might help to encourage students to do more research into the feasibility of their chosen projects in the early stages when a change is still possible.

Grant

For the SNES on an FPGA project I, along with Dustin Musselman, worked mainly on the SPC700 and PPU chips, as well as doing major testing and debugging on all other chips and their interconnects. The first month and a half of work was mainly devoted to building the SPC700 chip. As mentioned, the chip was mainly built by working with ISA's that had been assembled by various emulator developers who worked on creating SNES emulators based on ROM files (the data files pulled straight off the cartridges.) What we thought was going to be a simple design turned out to be a nasty 256 opcode processor with a degree of complexity far more advanced than anything we had attempted in the past. After much struggle with overall design, including several re-writes to conform to the monstrosity that is Xilinx development tools, Dustin and I created a working and tested version of the chip with several opcodes. We had all of the other opcodes together and ready to be placed in our final design file, yet our development towards the end lent to other things being more important.

The other major component I worked on, for roughly 3-4 weeks, was the PPU. A large amount of the processor's workings were mainly unknown to us, so we had to start from scratch developing FSMs based on what the software development handbook issued by Nintendo had said. This was moderately difficult, as without any working knowledge of the hardware it was difficult to establish what indeed had to be done to get the processor working. I specifically worked on the sprite pixel manipulation module, which when prompted for a specific x and y coordinate would generate a proper color address for the CGRAM and a priority so that the over-arching PPU module could decide whether or not it was to be displayed over the four different background modules that were to be instantiated. I also wrote major components for input manipulation from the main CPU, as there were specific registers that needed various outputs to be changed based on whatever input sent to it by the main CPU. With these, as well as the background logic and other data compiled by Dustin, we had what we were confident was a working PPU, as the testing we did showed accurate object and background data manipulation based on false input we provided.

Towards the conclusion of the project the majority of my time was spent debugging the main CPU wrapper and the DMA to try and get data moving from cartridge interface -> main CPU -> PPU -> monitor. Unfortunately we were unsuccessful with getting this to work however we were fairly confident all of our chips were working. We think if we had an extra month of work we would have gotten it working, but sadly we ran out of time.

For the class, there were several areas I enjoyed while I feel that there are some things that could use a bit of work. I enjoyed the freedom and the “go do something” attitude that the course staff took towards our projects, and the lab and technology provided to us was great. Mandatory lab I feel is necessary, but I feel that more instructor and staff involvement would be useful for feedback on progress the group is making. In the same sense, I feel a meeting once or twice a year with the professor as groups privately would be beneficial for future semesters, as a more direct contact from the professor to the groups on how the pace for the groups is going I feel is vital for the project to be a success. Reflecting on the course, I feel our project was a bit beyond the intended scope of the class, as there is a large leap from the NES hardware and the SNES hardware, which we thought we could manage successfully. Feedback early on about this difference could have led to different choices being made on project ideas; however we were pretty adamant about doing it so the staff likely couldn’t have convinced us otherwise anyway.

Overall I enjoyed the class, and obviously I wish our project could have been more complete by demo time, but I put in a lot of long nights over the course of the semester and I am pleased we got as far as we did given the complexity of the system we were dealing with. We hope that a group in a future semester can take our work and creating a working version of the SNES on an FPGA. Hopefully this report will provide enough documentation so that they further our work and make this project a success.

Dustin

Although our group spent a lot of time working together on this project, as individuals we spent a lot of time working on the different sub-components of the overall system. After deciding that we were going to build the SNES, we first spent a lot of time researching what all was going to be required to build it. We spent a week getting a rough idea of what we were going to need.

We made a decision to first start working on the sound module, which included a DSP and an SPC700 processor. Lincoln started the DSP, while Grant and I worked on the SPC700 and how it communicated with the outside world (memory, CPU, data registers, etc) We spent a week or so researching and then 3-4 weeks coding it. Because of the lack of documentation, there were many things we found out as we went and needed to go back and change. In the end, however, we did get a working SPC700. I ran a quite a few tests that tested the majority of the op codes to make sure they worked how we believed they worked base on documentation. But, as with everything in this project, it could not be completely tested until then entire system was running together.

We then moved to the PPU to begin getting video on the screen. Once again, due to spotty documentation, our time researching at the beginning was basically useless as we rediscovered a lot of things working differently then we initially thought. It was around this time when I started putting in 25-30 hours a week in lab. I worked mainly to get background data working. The tests I made successfully read in information from the registers and then I was able to create a background complete with scrolling. Again, I was not able to completely finish this because testing is impossible without seeing the actual data coming from the CPU. So after understanding this module almost perfectly, I was again stuck and couldn't move forward to clean up the last few things until I was getting actual data from the CPU.

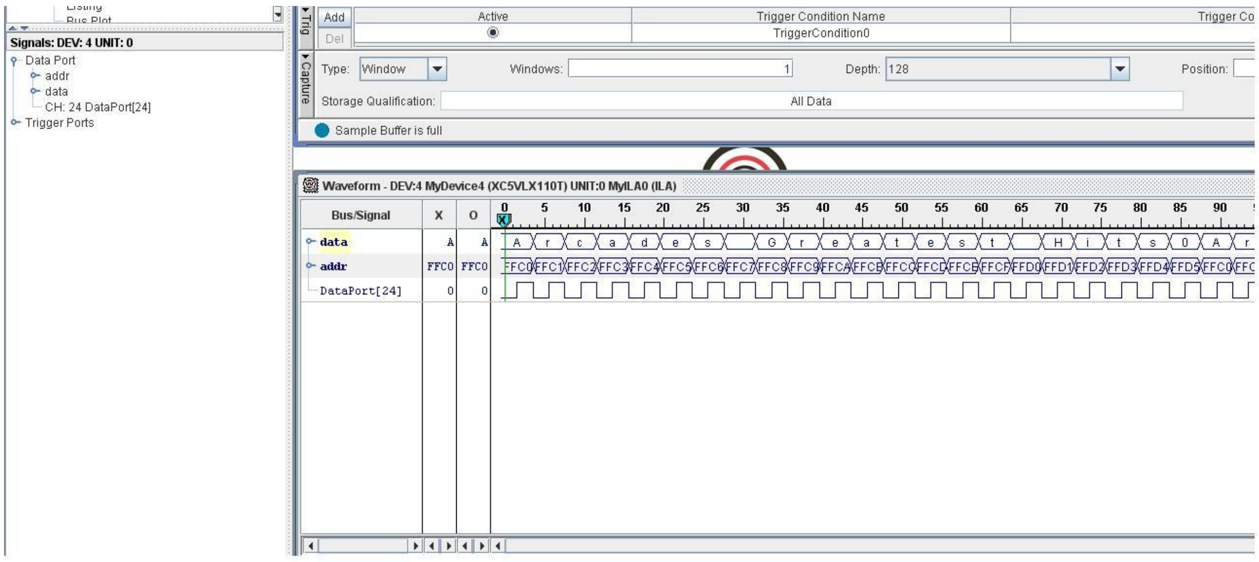
In the final few weeks, I switched over to help with the CPU. After Lincoln had created the working cartridge reader, we were faced with the problem of getting the cartridge to correctly talk to the CPU, which turned out to be much more difficult then we initially thought. We had many timing issues and were putting in at least 40 or 50 hours per week trying to get the CPU from infinite looping incorrectly.

My overall impression from our project was this. The way we had decided to implement the cartridge reader and actual controllers was an awesome idea, but ended up delaying the testing of the system by at least a few weeks. And even after we had the thing built, we spent even more time getting the timing to work. Since each component of the system was completely customized and not completely documented, we had no way to test anything other than “the big bang theory” of throwing everything together and seeing what happened. An idea that is sometimes necessary, but when it is, it's nice to get to this stage prior to only having a week or two left to debug the entire system. As far as going back and doing it again, we definitely should have concentrated first on the main CPU and cartridge interface, but this was pretty much impossible since we didn't get the code for the CPU until after 3 or 4 weeks, at this point we were in the middle of building other things so even then we didn’t start working on it right away. Also, I felt like I was constrained more than I initially thought since instead of just “making it work” we were confined to “making it work” as long as the CPU and cartridge timing agreed, which put a lot of limits on what we could do.

This was really the first open ended project class I had ever taken. Going in, I was excited to have such a wide variety of options and a minimal amount of restrictions. I never particularly enjoyed the strict set of boundaries that needed to be followed in projects in other courses, as I'm sure no one does, and was eager to have complete control over the SNES. The class itself was a great idea but I do have a few complaints. The additional reading and labs that we had to do with 0 feedback seemed like a waste of time, whether they were or not. Also, the status updates were never responded to. It seems like a short reply each week to the group as a whole would be helpful. Also, I felt like we had a lot of problems with Xilinx that no one really knew how to fix and that led to a lot of wasted time in the beginning of the project.

Accomplishments

At the demo we showed off our working cartridge and the DSP. Although we accomplished much more than that, and have 5000+ lines of Verilog that works in simulation to show for that, it was not flashy enough to demo.



Type: Window
Windows: 1
Depth: 128
Position:
Storage Qualification: All Data
Sample Buffer is full

Bus/Signal	X	O
data	A	A
addr	FFC0	FFC0
DataPort[24]	0	0

Overall Class Impressions

We enjoyed the flexibility of the capstone, in that we could choose something interesting to work on for the semester, whether it be a game or a research project. The project we chose was challenging, and we liked how the project played to our strengths (all our members had a computer architecture background).

Our only reservations about 545 were the lack of preparation to work with the toolset and the lack of feedback. Xilinx has a complicated toolset with cryptic documentation at best. Also, none of us had touched an FPGA since 18-240, and the Spartan-3's capabilities pale in comparison with the Virtex-5 we used in 545. An early hands-on workshop on the FPGA, the toolset, and how to use them would have been useful for this class. In addition, although we had status reports