The Fighting Meerkats'

GAME BOY®

Originally by

Nintendo®

Elon Bauer, Joseph Carlos, Alice Tsai

Table of Contents 2

Statement of Use and Contact Information

The members of this team, Joseph Carlos, Elon Bauer, and Alice Tsai, hereby give permission for anyone to use the code they produced for this project in an academic, educational, or otherwise non-profit-generating manner as long as the original authors (the members of this team) are given credit for their work.

If you have questions about the project, you can email us at:

Joseph Carlos: jdcarlos1@gmail.com
Elon Bauer: eob@andrew.cmu.edu
Alice Tsai: alicet@andrew.cmu.edu

The source code can (hopefully) be found at https://github.com/nightslide7/Gameboy.

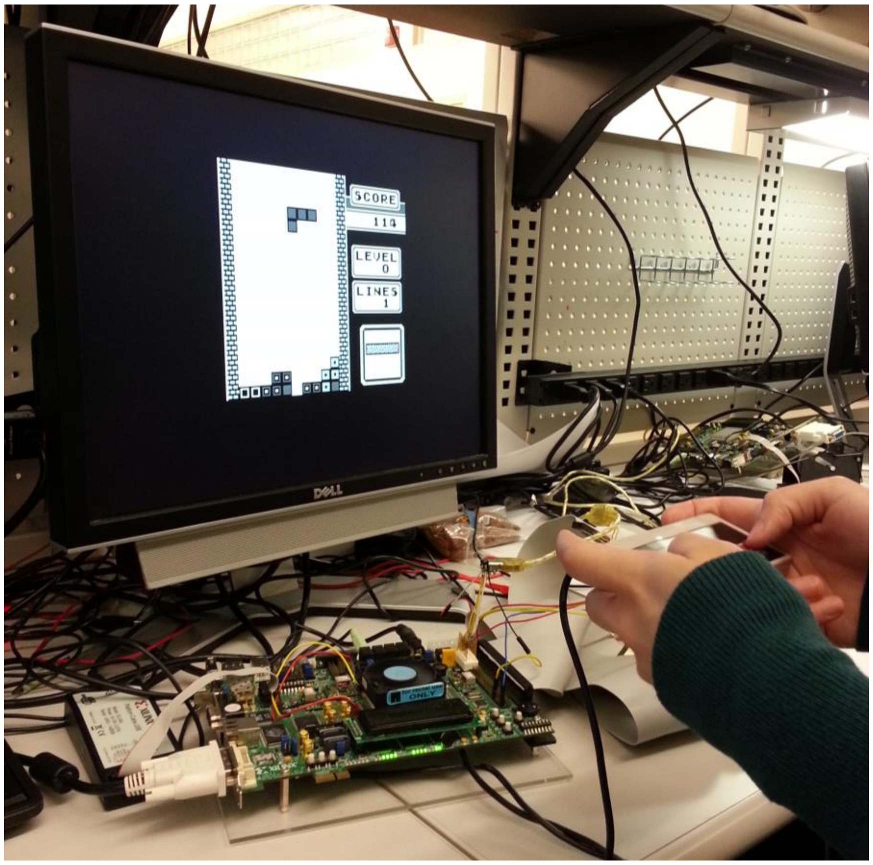
Project Description

What we set out to do

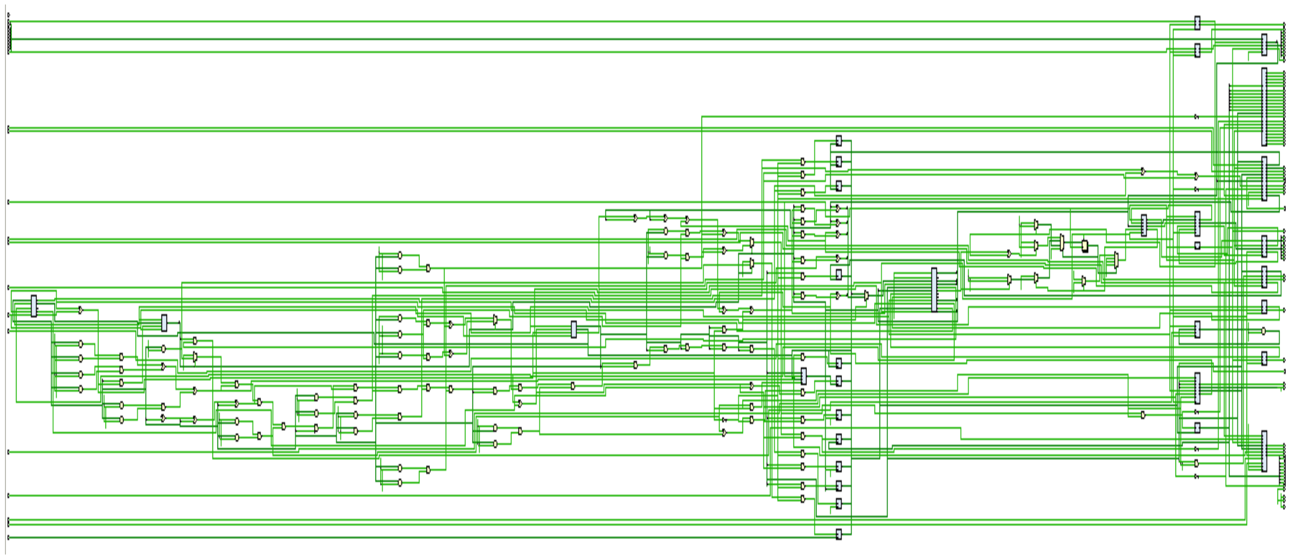
We set out to make a fully functional original Game Boy on an FPGA. The Game Boy supports any original Game Boy cartridge, grayscale graphics, user input through buttons, four-channel stereo sound, and a serial link cable for multiplayer capability. There are also other functions like an IR communication device that were not considered.

What we actually achieved

What we actually created was a mostly functional Game Boy, which supports Tetris with sound, full control, and a link cable. We created a working CPU, which interfaces correctly with the GPU, cartridge connector, audio interface, NES controller and link cable. The GPU does not support sprite manipulations but otherwise works. The cartridge connector should support all games, but only Tetris works properly. Three of the four audio channels were implemented and work for all intents and purposes. The NES controller interface is perfect, and the link cable basically works sometimes. Here is a shot of our working design playing Tetris:



And here’s a picture of our elaborated design in Vivado:



Development Tools Overview

Board

We used a Xilinx Virtex-5 LX 110T FPGA on a Xilinx development board. The documentation for the board is equivalent to the documentation for the ML505 development board.

Xilinx Tools

We used Xilinx's ISE 14.3 IDE to compile and synthesize our code, as well as its built-in CORE Generator tool for generating block RAM and Chipscope modules. We used ChipScope extensively to debug the board.

Operating Systems and Workstations

The lab machines given to us had Red Hat Enterprise Linux (RHEL) installed on them. RHEL does not play nice with Xilinx tools. We lost cable drivers three times during the semester, halting all development for two members of the group, as they had no backup machines. Do not use Xilinx tools with RHEL if you know what's good for you. Joseph luckily ditched these workstations early on and installed ISE on his own laptop running Windows 7. He found that this operating system runs the design tools with no problems.

Version Control

We created a Git repository, hosted on GitHub, early on in the project and used it to manage our code throughout the semester.

Hardware Overview

Our Game Boy's hardware consists of a custom CPU, a GPU for tile mapping, 8Kb of work RAM, 8Kb of video RAM, a sound chip, a cartridge connector, a link cable driver, a controller interface, a set of timers, and a block transfer module.

The CPU executes code read in from the cartridge, and sets memory-mapped registers in the other modules to control them. The modules also present various status registers for the CPU to access. Four of the modules generate interrupts that cause the CPU to start execution of interrupt handlers: the GPU, the controller, the timers, and the link cable.

CPU

This section will attempt to outline our design as well as call attention to things that were unclear from existing documentation.

Overview

The Game Boy CPU is an 8-bit Z80/8080 hybrid with some custom functionality added by Nintendo. There are approximately 48 different instruction groups, such as bit shifting, 8-bit arithmetic, 16-bit loads, etc.

Description

The most complete document describing the CPU is the Game Boy Programming Manual, Nintendo's reference for people making games for the Game Boy. A copy of this can be found at http://www.romhacking.net/documents/544/. A summary version of this manual with some additional information, the Game Boy CPU manual, can be found at http://marc.rawer.de/Gameboy/Docs/GBCPUman.pdf. And finally, an HTML version of the CPU manual called the PAN docs can be found at http://nocash.emubase.de/pandocs.htm.

Timing

Game Boy documentation refers to "machine cycles" and "T cycles." This was initially confusing. A T cycle is an actual clock cycle and a machine cycle is 4 T cycles. Machine cycles are just a shorthand that Nintendo engineers used to calculate timing. For the purposes of a Verilog Game Boy emulator, machine cycles are irrelevant. When this document refers to a cycle, it refers to a T cycle.

The original CPU runs at 2^{22} Hz, which is 4.194304 MHz. Our CPU, however, runs at 4.125 MHz due to the limited clocking functionality of the development board.

Each instruction executes in 4, 8, 12, 16, 20, or 24 cycles. Conditional instructions have a variable number of cycles (fewer when the condition is not true). The instruction timing is extremely important and must be preserved in any Game Boy emulation system, hardware or otherwise. The programmers of the Game Boy expected the cycle values to be constant across Game Boy systems. Unfortunately the official Game Boy Programmer's Manual incorrectly specifies a number of these timings. A list of the correct timings can be found at http://pastraiser.com/cpu/gameboy/gameboy_opcodes.html.

Special CB Instructions

The CPU has support for multi-byte instructions in the form of so-called "CB instructions." These are simply instructions prefixed with the byte 0xCB. When 0xCB is read as the next instruction, the CPU reads the next byte as a CB instruction rather than a normal instruction. For example, the instruction 0x80 normally corresponds to ADD A, B. However, if the CPU reads 0xCB and then 0x80, the instruction corresponds to RES 0, B (reset bit 0 of register B).

Architecture

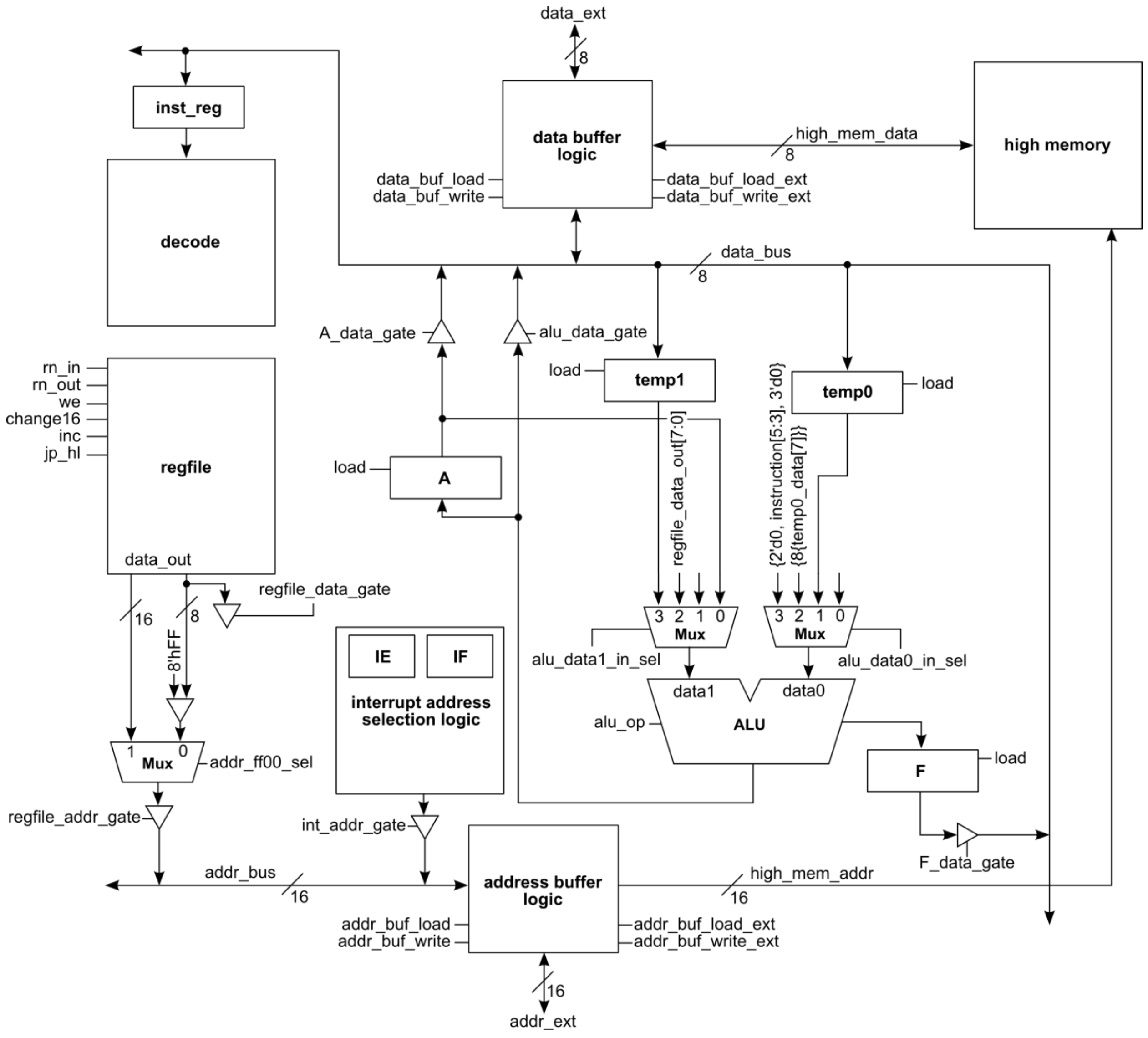
The architecture of our CPU deviates as little as possible from the original architecture of the Z80/8080, at least as far as we could find documentation for it. A good resource on the hardware design of the Z80 can be found at http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm.

Our design is a multi-cycle microcoded non-pipelined in-order design. The basic components are the register file, ALU, internal and external data buses, input and output buffers, and the all-important decode module.

The CPU also includes a block of "high memory" separate from the rest of work RAM. This module is implemented as a Verilog array and can be accessed by the CPU even if the GPU or DMA module is accessing memory elsewhere on the address and data buses. When the CPU reads or writes to this section of memory, it does not output anything on the address or data buses, or the read and write enable signals.

The CPU interfaces with the rest of the system through the interrupt register inputs, a single signal to disable CPU memory access (needed for DMA transfers), and most importantly the shared address and data buses. The CPU assumes that its bus interface to memory is asynchronous read, single-cycle write, since every memory component of the Game Boy is an SRAM chip.

Below is a diagram of the CPU's core modules. This diagram does not include all the signals in the design, just the control signals from the decode module and a few extra useful labels (such as bus labels). The reason for this is that we used this diagram to write microcode, so we wanted a simple overview of the system with just the control signals we needed. Be aware that there are additional signals in the decode module that simplify some things, such as incrementing the PC specifically.



Decode

The decode module of the CPU contains logic that generates control signals for all the hardware in the CPU, including the timing logic and interrupt handling control flow. It is essentially a few thousand lines of microcode in a giant case statement on the current instruction, then smaller case statements - per instruction category - on the cycle number.

Interrupts

Interrupts are handled by examining the contents of the IF and IE registers on each 0 cycle. If an interrupt occurs and is enabled, the CPU resets the IF bit for that interrupt and proceeds to handle it. In order to interface with the IF register, external logic is required. This logic needs to load the IF with its current contents plus the new interrupt bit of the interrupting module. The CPU will then do the right thing with it.

The ALU

The ALU is a basic design with its own set of opcodes, two inputs, and one output. It also outputs the result flags as well as taking the current flags as an input. The flags are Zero, Negative, Half-carry, and Carry (Z, N, H, C). The flags are set differently for each instruction. For example, RLCA (rotate A left through the carry flag) sets Z, N, and H to 0 and sets C to A[7]. However, RLC r (rotate register r left through the carry flag) sets N and H to 0 and C to r[7], but sets Z to 1 if the result is 0, and Z to 0 otherwise.

The DAA Instruction

The worst part of the ALU was the DAA instruction, which adjusts the result of a BCD operation that occurred in A. For example, if A contains 0x19, and then you add 0x19 to A, A will contain 0x32. After a DAA instruction, as long as F hasn't changed, A will be adjusted to 0x38, the BCD result of 0x19 + 0x19. We adapted code from the Internet that correctly calculates DAA on every possible combination of A and the flags. That code is found at http://forums.nesdev.com/viewtopic.php?t=9088 in a forum post by user DParrott.

Since the DAA instruction has so many edge cases, is essentially magic, and is almost impossible to implement from scratch, we're going to include the Verilog description here for anyone in the future who might want to use it. The signals are labeled with their types above the case. `` `ALU_DAA: begin `` is a case in a switch statement on the ALU opcode. This switch statement is inside an `always @(*)` block. During the DAA instruction, `alu_data1_in` is connected to the output of A and `alu_data_out` is connected to the input of A.

```
output reg [7:0] alu_data_out;
output reg [3:0] alu_flags_out;
input [7:0]      alu_data0_in, alu_data1_in;
input [3:0]      alu_flags_in;

parameter
   F_Z = 3, F_N = 2, F_H = 1, F_C = 0;

reg [8:0]        intermediate_result1, intermediate_result2;

...
`ALU_DAA: begin
   if (~alu_flags_in[F_N]) begin
      if (alu_flags_in[F_H] |
          ((alu_data1_in & 8'h0f) > 8'h9)) begin
         intermediate_result1 = {1'b0, alu_data1_in} + 9'h6;
      end
      else begin
         intermediate_result1 = {1'b0, alu_data1_in};
      end
      if (alu_flags_in[F_C] | (intermediate_result1 > 9'h9f)) begin
         intermediate_result2 = intermediate_result1 + 9'h60;
      end
      else begin
         intermediate_result2 = intermediate_result1;
      end
   end
   else begin
      if (alu_flags_in[F_H]) begin
         intermediate_result1 = {1'b0, (alu_data1_in - 8'h6)};
      end
      else begin
         intermediate_result1 = {1'b0, alu_data1_in};
      end
      if (alu_flags_in[F_C]) begin
         intermediate_result2 = intermediate_result1 - 9'h60;
      end
      else begin
         intermediate_result2 = intermediate_result1;
      end
   end // else: !if(alu_flags_in[F_N])

   alu_data_out = intermediate_result2[7:0];

   alu_flags_out[F_N] = alu_flags_in[F_N];
   alu_flags_out[F_H] = 1'b0;
   alu_flags_out[F_C] = intermediate_result2[8] ? 1'b1 :
                        alu_flags_in[F_C];
   alu_flags_out[F_Z] = (intermediate_result2[7:0] == 8'd0) ?
                        1'b1 : 1'b0;
end
...
```

Process

We read the project reports for the previous teams that attempted the Game Boy and noticed that they both attempted to use a buggy design they found online, failed to do so, and ended up implementing their own CPU. We decided based on these reports to just implement our own CPU from the beginning and not waste any time with other peoples' non-working code.

In order to do this, we planned to have a few phases of CPU development.

1. Design the CPU at a block-diagram level.

2. Implement the CPU hardware and test one instruction in simulation.
3. Microcode a few instructions and run a simple program, such as calculating a Fibonacci number, on the FPGA.
4. Adapt an open-source emulator's CPU to use in a testing harness.
5. Microcode all the instructions.
6. Write unit tests for all the instructions and test them using the testing harness in simulation.
7. Implement the DMA, timers, and interrupt handling.
8. Use a standard Verilog testbench with compiled Game Boy assembly to test the DMA, timers, and interrupts in simulation as thoroughly as possible.
9. Run the bootstrap ROM in simulation using modified versions of the GPU scroll registers.
10. Create a breakpoint module using the FPGA's LCD, switches, and buttons to print register state, step through instructions, and break on addresses.
11. Integrate the other modules and run the bootstrap and the Tetris ROM on the board.

Most of these phases were simple to complete using standard techniques and strategies learned in previous courses. The testing harness using the emulator's CPU was not.

Testing Harness

We decided that we needed some kind of quick way of testing the CPU. We were inspired by the testbench from 18-447 to create a similar testing harness. The testing harness we wrote for the CPU involved three major elements: our CPU simulation, assembled assembly files, and a working emulator's CPU code hacked to behave in a nice way. These components came together to create an automated testing harness that, when finished, allowed us to type a single command with an assembly file as input that compared the correct register values of the emulator against our register values at the end of the program.

This testing harness was absolutely invaluable throughout the course of the project. It allowed us to run each test and find the bugs in the tested components in minutes rather than hours. It also allowed us to reproduce bugs in simulation that we found in synthesis. This allowed us to see all the CPU's wires rather than ChipScope's limited subset. It also allowed us to make changes to the CPU without having to resynthesize to test.

CPU Simulation

We decided to automate this using batch files on Windows with Xilinx's ISim tool. The batch files call the Xilinx tools to compile our design and create a simulation executable. We use the $memreadh system call to read bytecode from a file into a Verilog memory module connected to the CPU, then set the CPU's PC to 0 and let it work it's magic. When it encounters a HALT instruction, it saves the register values to a file on disk, which we can compare with the correct values from the emulator's CPU, which we changed to do the same thing.

Emulator

We used the CPU from DMGBoy, found here: http://code.google.com/p/dmgboy/. We downloaded the source, imported the relevant files into Visual Studio, and changed the CPU to write the register values to disk when it encounters a HALT instruction. The emulator code could only load ROM files, so we output the machine code to a 32 kB file padded at the end with zeroes.

Assembled Assembly

We acquired a Game Boy assembler from http://gbdk.sourceforge.net/. The instructions for installing this are found either there, or more succinctly at http://www.loirak.com/gameboy/gbprog.php. When the assembler (as-gbz80) is invoked with a -l option, it outputs a list file, which is basically just machine code with comments. We wrote a Perl script that takes these list files and translates them into actual machine code. We had two formats for the machine code. The first was the Verilog-readable memory data file consisting of each byte followed by a newline. The second was the emulator-readable ROM file consisting of a string of binary data.

Putting it Together

We wrote a batch script that takes the assembly file and some utilities (such as the Perl scripts mentioned above, and the tcl file for Xilinx's compilation tool), copies them into a new directory with the same name as the assembly file, compiles the Verilog, assembles the program, runs the simulation and emulator, and invokes a Perl script that compares the output files. If everything is correct, we're golden. If not, we can descend into this directory and run the simulation executable ourselves and look at the waveform.

Breakpoint Module

The breakpoint module is a simple module that allows the user to set a breakpoint with the switches on the FPGA board. The user selects either the top or bottom bits of the 16-bit address with a button, changes the 8 switches to a number, and then presses a button to save that number to the corresponding half of the address. The decode module then interprets this and stops execution when the PC is equal to the address. At this point the user can step through the code or continue using buttons on the board. We found this to be extremely helpful.

On-Board Debugging

Even though the testing harness managed to catch most of our bugs, we still encountered bugs while running Tetris and the test ROMs. The process for finding bugs here was to find where the code was different from the emulator's output and trace back using ChipScope and the emulator to where exactly the difference began. For example, we had a bug in the JR Z, e (jump relative if zero) instruction that caused us to jump to an invalid location in code. We could see on the LCD display that the PC was invalid, so we used ChipScope to go further and further back in time to the valid JR instruction that caused the jump, and were able to determine why it was incorrect. Another problem we had was that we couldn't get into the Tetris start screen from the credits, so we stepped through the emulator code to see where exactly this happened and figured out from there why our design wasn't executing the code in the same way (it turns out we weren't allowing writes to the IF and IE registers correctly).

Final Results

We got Tetris to run on the board with no single-player CPU-related bugs, which indicates that the CPU is working quite well. At the end of the project, we found some very extensive CPU tests on the Internet called Blargg's test ROMs: http://blargg.8bitalley.com/parodius/gb-tests/cpu_instrs.zip. We managed to get the CPU to pass all of the functional tests found there. We believe the CPU is essentially correct and complete.

Video Interface

Overview

We adapted Dragonforce (VirtexSquared)'s framebuffer and FPGABoy's video_module and video_converter to work with our system. The video interface uses the Chrontel CH7301c video chip to output the DVI signals to the monitor. The video interface is made up of a controller FSM, which takes in data input to be written to the video control registers, which controls the state of the controller, and hence it's outputs. The VRAM and OAM portions of the memory are memory mapped inside of the video interface. If the CPU needs to access the VRAM, OAM, or any of the memory-mapped registers, the address of those requests are mapped to the video interface which outputs the necessary information.

Description

Video Module

The Video Module contained the memory for the VRAM and OAM, the memory mapped registers, and the FSM used to control the reading, writing and data output. The functions of the 10 memory mapped registers in the video module include controlling the display data and memory accessed, the horizontal and vertical scrolling, the mode of the video controller, the background and object palettes, as well as the window positions. The Video Module contains two scanline buffers, which contain the upper and lower bits of each pixel. One line is stored at a time. First, it is determined whether the background or window tile is displayed, and then the appropriate pixel data is fetched and stored in the scanline. Next each sprite in the OAM is checked to see whether or not it intersects with the current line being displayed. If it does, the sprite data is pulled out and examined to determine the final pixel data in the scanline. Depending on the color and attributes of the sprite and background pixel data, the correct pixel data is then masked and written into the scanline buffers. Once the pixel data for a line has been finalized, it can be outputted to the Video Converter one pixel at a time, reading the upper bit from one scanline and the lower bit from the other scanline.

Video Converter

The Video Converter generates the horizontal and vertical sync signals, which are then passed on to the DVI Module. In addition, it takes the pixel data from the video module and stores it in a framebuffer which stores one Game Boy LCD screen frame at a time. Note that this is not the frame of the monitor. There are two framebuffers so that as one is reading, the other can be outputting pixel data to the DVI Module. The pixels are stored as 2-bit data inside the framebuffer and converted to a 24-bit RGB value before being outputted to the DVI Module. The final main task of the Video Converter is to offset the Game Boy LCD screen frame pixel data so that it can be centered on the monitor and outputs the color black when the pixel is not in the area occupied by the frame.

DVI Module

To output pixel data to the monitor, the Chrontel chip's I^2C needs to be set up and horizontal and vertical sync signals are outputted with the 12-bit data. The DVI Module outputs data on both edges of the clock so that in one period, the entire 24-bit RGB value will have been sent.

Process

We started with trying to output simple color to the monitor.
To set up the I^2C and output DVI signals, we took code from Dragonforce's framebuffer and modified it to only do the setting up of the I^2C and generate the sync signals. There were some Xilinx modules (such as IO_DELAY) that were used that required other Xilinx modules to function and at first, we commented it out. However, after realizing that those modules were required for proper timing, we added them back in. After reading documentation, we were able to recreate the required missing modules.

After the initial video was functioning properly, we looked into FPGABoy's video code. It was not easy to test FPGABoy's video code without hooking it up to the CPU since all of its output depends on input which comes from the instructions read by the CPU so after combining the DVI code with the GPU code, we hooked it up to the CPU and it failed. Upon simulation, we found that one of the signals had been hooked up incorrectly. After correcting that, it worked and the Nintendo logo showed up on the screen. Further testing was done in simulation by comparing waveforms with what was in the registers in the BGB emulator. These waveforms assured us that the GPU was functioning correctly. Unfortunately, later on, we found that sprite reflection and transparency did not work properly. The rest appeared to function correctly though.

Audio Interface

Overview

The audio interface on the Virtex5 uses the AC'97 codec to output PWM audio levels to the headphone jack. These are literally just "volume" values. The Game Boy has four sound channels. Channels 1 and 2 are square waves, channel 3 is a waveform player, and channel 4 plays white noise. All audio functions come from memory-mapped registers, which are written to by the CPU. Functionality includes frequency sweeps, volume envelopes, and length controls.

Description

AC'97 codec

In order to play sounds from the Virtex5, you first need to setup the AC'97 codec. This process is complicated and not completely understood by anyone other than team Dragonforce aka. VirtexSquared. The Dragonforce code properly sets up the codec and plays sounds from flash starting at address 0x0000. In order to program the flash, first you need to generate a .mcs file. This can be done using the Xilinx "promgen" command on a hex file. The hex file can be generated from a raw audio file using the "xxd" command. Detailed commands are found in VirtexSquared's Makefile. Once you have a .mcs file, in Impact assign a PROM to the device and select the .mcs file. Select the BPI PROM 28F256P30 and click OK. In the window that pops up, select the attached flash on the left and change the pull down menu from "automatically load FPGA with Flash contents <default>" to "automatically load FPGA with currently assigned bitstream." Then right click the flash and hit program. It should automatically load the bitfile onto the FPGA. If this doesn't work, power cycle the device and try again.

The Audio Registers

The Game Boy uses 18 memory-mapped registers to allow the CPU to communicate with the sound module. Basically there are five registers per channel and three master control registers. These registers can be written to at any time while producing sound. They define various properties of the sound currently being played. The labels in the Pan Docs are of the form NRXY, where X refers to the sound channel (5 refers to the master control registers) and Y refers to the sub register of that channel.

Channels 1&2

For the square waves, the registers are as follows:

Y=0 is the frequency sweep register (channel 1 only). When the sweep time is not zero, the sound's frequency is either increased or decreased by the current frequency right shifted by "Number of sweep shift." Note that "Number of sweep shift" is not the number of sweeps that should be performed. It is simply a value (n) in the function X(t) = X(t-1) +/- X(t-1)/2^n. Note that the sweep time is implemented using a 128Hz clock

Y=1 is the length and duty cycle register. Of note, the wave pattern duty is the percentage of time that the wave is low, not what one would expect. The length of the sound is given by the function (64-t1)*(1/256) seconds, where t1 is bits 5-0 of the Y=1 register. This timing is implemented using a 256Hz clock.

Y=2 is the volume register. The initial volume and direction of the volume envelope is defined, as well as a variable called "Number of envelope sweep." Note that like with the frequency sweep function, Number of envelope sweep is not the number of sweeps to perform. It is simply the value n in the function Length of 1 step = n*(1/64) seconds. This length is implemented using a 64Hz clock.

Y=3 is simply the lower 8 bits of the 11 bit frequency data. The true frequency is given by the formula F = 131072/(2048-x)Hz where x is the 11 bit frequency data. This was implemented with a 131072Hz clock.

Y=4 looks simple but was probably the register that caused the most trouble and was hardest to understand. It defines whether or not to use the length value previously defined, and it contains the "Initial" bit. The initial bit restarts the sound on that channel when a value of 1 is written to it. This is REGARDLESS OF THE PREVIOUS VALUE.

Channel 3

This channel plays waveform files stored from address FF30 to address FF3F. It plays them one byte at a time, most significant four bits first. I believe they are played from address FF3F to address FF30, but that has not been confirmed. Our waveform playback module is not perfect. However I will still describe what I know about the registers.

Y=0 is simply an enable flag. If it's not set, don't play sound.

Y=1 is the length data. The actual length is given by the formula (256-t1)*(1/256) seconds. This is implemented with a 256Hz clock.

Y=2 just includes the output level which is an amount to shift right by.

Y=3 is the upper 8 bits of the frequency.

Y=4 includes the flag determining whether or not to use the length data, the higher bits of frequency, and the initial flag. Note again, sound is reset when the initial flag is written with a 1, REGARDLESS OF THE PREVIOUS VALUE. The frequency of this channel is given by the formula F = 65536/(2048-x)Hz where x is the frequency data. This is implemented with a 65536Hz clock.

Channel 4

Channel 4 produces white noise of various types. This seems to be mostly used to produce sounds similar to drum beats to intersperse in Game Boy music. It seems to be non-trivially difficult, including a polynomial counter, which essentially functions as a random number generator.

Control Registers

Y=0 is basically a “master volume” register. It has additional flags for outputting a Vin signal to the cartridge, but this is almost certainly superfluous for most games. The volume values should be used in conjunction with each channel’s volume in a mixer sort of fashion. However, note that no other mixing needs to be implemented and the channel’s values can simply be added together to produce the final mixed sound.

Y=1 is a set of enable flags for the right and left stereo channels for each of the four sound channels.

Y=2 is a master sound enable flag which stops all sound functionality if not set. Additionally there are read only flags, which say whether or not a sound is currently playing. We did not implement these last flags.

Our Design

In our design all of the sound files are located in the sound_src folder. The AC97.v file sets up the AC’97 codec and “mixes” the audio channels based on their enable signals. The sound registers are coded in sound_registers.v. Each register is simply a reg variable and each of the variables that is set inside of the registers is output from the module to make them easier to access and give them names. In sound_functions.v you will see the waveform player and square wave player. They have a few similarities, but have mostly different functions. Finally everything is hooked together in audio_top.v and the sound register values are given to the sound function generators, which use those values to output sound levels to the AC97. The .ucf file in this folder is specific to sound and no other .ucf files are allowed to clobber it.

Process

AC’97

Starting from lab 2 we realized that we needed to use Dragonforce’s code to set up the AC’97 codec because of the complexity involved in doing it ourselves. When we compiled their code and programmed our board with the bit file, Dragonforce began to emanate from our speakers. We weren’t entirely sure why. We later learned how to program the flash. This same process was also needed later to program the flash with the bootstrap ROM.

Square Waves

The first step we took towards producing Game Boy sound was to use the Dragonforce SquareWave module to produce a square wave of various frequencies. The tricky part about that was to get the duty cycle correct. Without a proper 50% duty cycle, the measured frequency would be slightly off and wouldn’t follow a linear function as would otherwise be expected. But otherwise it was an easy process. The AC’97 simply outputs unsigned voltage levels when you assign the slots to a different value. In this way, sound is created by changing the output level. So to make a square wave, we simply needed to oscillate between a value of 0 and F at a specific frequency. We determined that the best way to control the frequency, because of the way it was defined in the Pan Docs, was to count cycles of a specific frequency clock, and only change the output value after a certain number of cycles. By doing this we were able to use “reverse division” to output whatever frequency we wanted.

Volume Control

Once we got a square wave that could be assigned to a frequency accurate to within 1Hz, we worked to implement a volume controller. We decided to use the rotary controller on the FPGA dev board to act like the actual volume knob on the Game Boy. It was simple to implement with a 7 state FSM. Essentially there is a signal for going “up” and a signal for going “down” and both are asserted at every tick of the wheel. It seemed to work just fine at first but actually we had a lot of problems where the volume would start all the way turned down and we would think that sound wasn’t working because of it. So later I changed it to only go between the values E and 9, since below 9 the sound wasn’t audible anyway. Since the AC97 outputs 20 bit volume levels, the 4-bit volume level output by the Game Boy had to be shifted over significantly. As it turns out a shift of 9 is the minimum where it is audible, but a shift of F or higher made the sound clip.

Memory-mapped Registers

The next step we decided to take was to code the audio registers. At first there was some confusion about how memory-mapped registers actually work, but eventually we determined that we could just make them reg variables and everything would work out fine. We just used the CPU’s memory write_enable signal to determine when to load the reg’s with new values, and used the address to determine which reg to load. Then each of the variables in those registers would be updated automatically and sent out to the sound function generator, which in turn sent levels to the AC’97 codec. It was all very simple in the end…

Sound Functions

Once the audio registers were hooked up in such a way that they could be assigned values, we got to work on the sound functions that would be used to test them. We started with the waveform generator but worked on the square waves in parallel, which were often easier to understand. We set the registers to specific test values and filled the waveform registers with a test waveform. At first we were unclear about the fact that each byte is played most significant four bits first, and simply played the waveform four bytes at a time from bottom to top. It was extremely unclear if it was working properly because without the CPU actually changing the register values in real time, it would never sound like actual music. But we could still test some functionality.

Length

The first functionality to test was the length function. We set the length register to 0, which meant that sound should play for ¼ of a second for square waves. This was driven by a 256Hz clock and the “reverse division” method mentioned earlier. Essential we took the number of 256Hz clock ticks were necessary before reaching the desired length, then counted that many 256Hz clock edges and then stopped output. This method was used for frequency sweeps and volume envelopes as well, and for anything that required a frequency.

Frequency

After that we made sure that the waveform and square waves played back at the correct frequency as determined by the audio registers. This was mostly tested using a music tuner from a smartphone. For the waveform player, the waveform was output to the LEDs and looked at to see how fast it changed. We had some trouble getting a perfectly accurate clock with which to generate different frequency signals, but it seemed to work out pretty well.

Frequency Sweep

Next came the frequency sweep function for square waves. Essentially this changes the frequency of the signal by a certain amount at every clock cycle until it either becomes zero or maxes out to 2048Hz. The sound was already playing at a frequency specified by a variable, so all we had to do was change the value of that variable over time. One strange thing that was never properly resolved was that when we added to the frequency, the output frequency would decrease, and when we subtracted, the output frequency would increase. We decided that this was strange but probably not incorrect, just flipped. So we flipped the wires and went on with our lives. The output appears to match the sound test ROM output, so it's reasonable to assume that this is at least mostly correct.

Duty Cycle

After that came the duty cycle. This was simply a modification to the frequency function, which changed the square wave from low to high based on a right or left-shifted version of the original frequency counter. We believe this is how the Game Boy actually implemented it, because the possible duty cycles lend themselves perfectly to this. One bit of confusion was that the Pan Docs specified that the % duty cycle was the percentage of time the signal was held low, which is backwards from most electrical engineering courses. However again, the measured output matches the output from the sound test ROM, so presumably it's correct.

Volume Envelope

The volume envelope function was the last piece of the square waves that needed to be implemented. Essentially this function just adds or subtracts 1 from the output volume after a certain number of ticks of a 64Hz clock. There was one really sticky point about the volume envelope and that was the variable referenced in the Pan Docs as "Number of envelope sweep." This variable is NOT the number of envelope sweeps to perform. Instead, the Game Boy performs as many envelope sweeps as it needs to until it either gets to a volume of 0 or a maximum volume of F. The "Number of envelope sweep" variable is simply a number that determines the length of time to wait between each step in volume. After that was fixed, the function worked fine.

Frame Sequencer

The sound functions use a "frame sequencer" which generates various clocks from a 512Hz timer. We implemented this by just creating a 512Hz clock and then using our clock divider. Here's a diagram of the frame sequencer:

Step	Length Ctr	Vol Env	Sweep
0	Clock	-	-
1	-	-	-
2	Clock	-	Clock
3	-	-	-
4	Clock	-	-
5	-	-	-
6	Clock	-	Clock
7	-	Clock	-
Rate	256 Hz	64 Hz	128 Hz

The Initial Flag

The final and possibly biggest hurdle that we had to overcome to get sound working was the initial flag specified in the last register of every sound channel. At first we implemented it as a negatively asserted reset. This worked sometimes but produced sound only a small percentage of the time and was really only enough to get the ping noise working. Then we tried making it reset everything only on its positive edge, which was much harder to implement. This worked better, but was still wrong. Sound wasn't working properly and we honestly had very little idea why, but since this flag was so poorly understood, it seemed the likely culprit. Finally we determined that a sound channel is reset and started over whenever a value of 1 is written to this flag, regardless of its previous value. Apparently this is how a lot of embedded systems work, but that was not something that anyone on the team knew. Once this was fixed, sound worked great.

Results

Channels 1 and 2 work almost perfectly and play Tetris music accurately to the extent that the human ear can perceive it. According to the sound test ROM, the frequency sweep function might still be slightly incorrect, but everything else has been tested and appears working.
Channel 3 mostly works. It plays back the waveform from memory, but it doesn't sound quite right. It's a fairly simple module, so it's very unclear what's wrong.
Channel 4 was not implemented. This means sounds like drum beats are left out of the final audio. This is mostly imperceptible unless you know it's missing, but it does add something when it's implemented. It likely wouldn't be much more work to implement, we just ran out of time.

Cartridge Connector

Overview

The cartridge connector was desoldered from an original Game Boy PCB and hooked up to a protoboard. Wires were then run from the protoboard into the GPIO pins on the FPGA dev board. Initial configuration involved only reading from the ROM, but later additions allowed writing to the SRAM on the cartridge as well.

Description

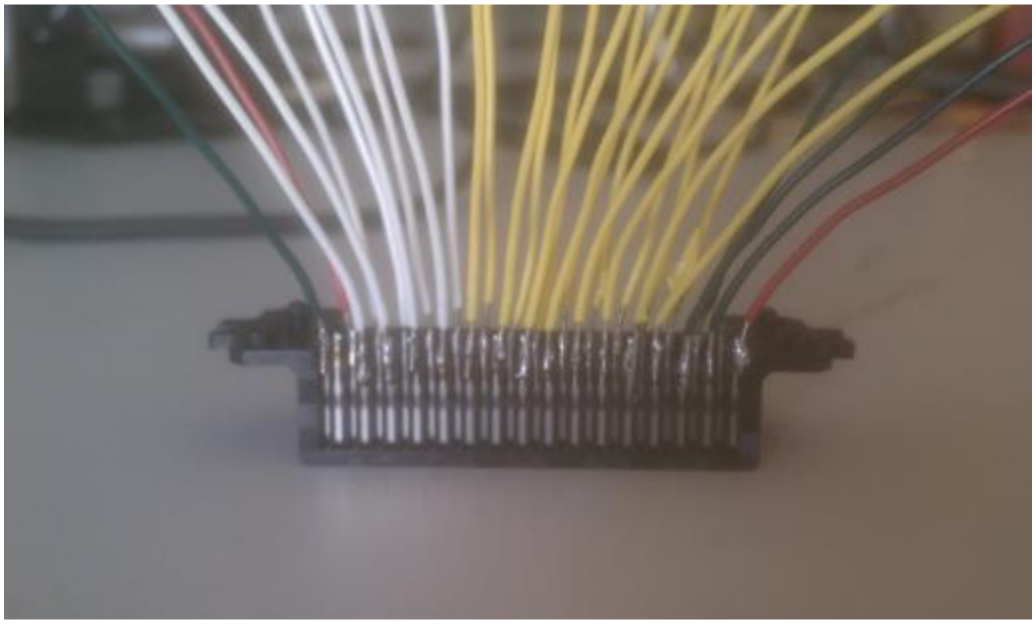
Every Game Boy cartridge has the same pinout. This includes Vdd, ground, 16 address pins, 8 data pins, a read enable, write enable, and a chip select, as well as a reset and two other pins that are probably never used. There is no clock that is necessary to read from the cartridge. As long as it gets power, while reading, the data on the lines will be the data stored in ROM at the address given by the address lines. When writing, the chip select must be asserted. Writing only happens for save files.

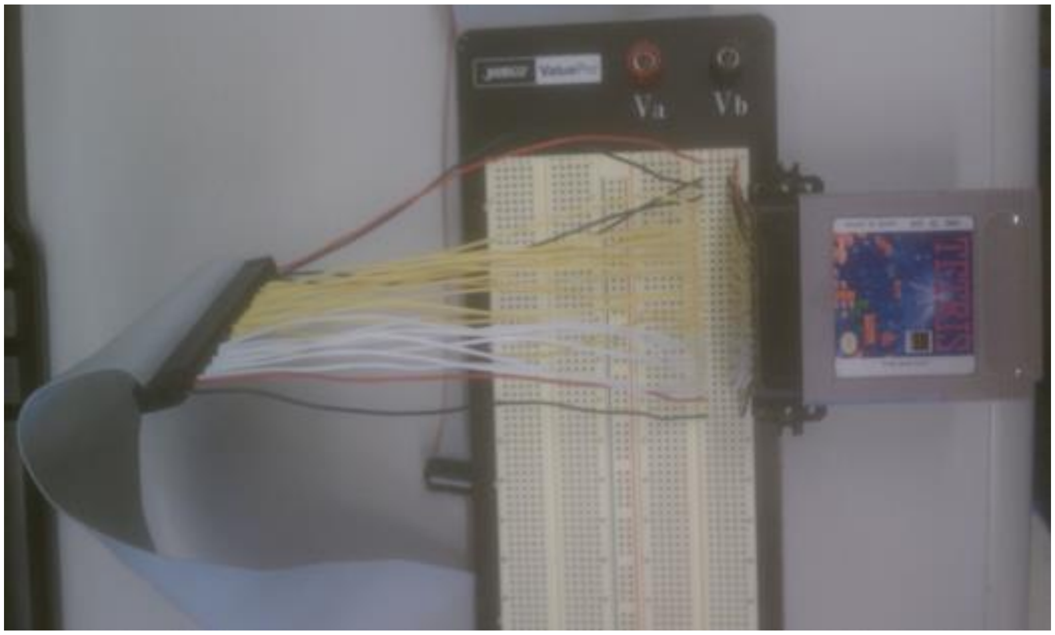
Process

We ordered a replacement Game Boy Advance cartridge connector from the Internet with the intention of using that to plug in our games once we got rid of the mechanical stops that disallow original Game Boy cartridges from being inserted. However it took a very long time to arrive in the mail, so we decided to remove the connector from the original Game Boy that we had purchased. This proved to be a splendid idea. The bulk of this connector meant it was easy to hold on to, as opposed to the Game Boy Advance connector, which was very small. This was important because of the delicate nature of the solder connections. The cartridge connector had offset and oddly spaced pins, so it could not be inserted directly into a protoboard. In order to get around this, it was decided that the best course of action would be to solder wires to each of the individual 32 pins and then get those wires to the GPIO pins somehow.

The original plan was to take female-female GPIO connector cables and cut them in half and solder the cut ends to the pins so we could plug directly into the GPIO pins. This might have been a better idea in some ways because we believe we may have had timing issues due to the length of our final cables, but there weren't enough female-female GPIO cables and we didn't feel like ordering more. Also the GPIO cables did not have a solid wire inside but rather a bundle of smaller wires, which is much more difficult to solder properly. It was wise not to complicate an already difficult soldering job.

We decided to solder protoboard wire to each of the pins and plug them into a protoboard rather than directly into the GPIO pins because of modularity and the ability to probe the signals this way if we encountered issues. This meant that each of the wires had to fan out to be almost exactly the same length or the connector wouldn't sit properly. This was no mean task. Here are some pictures of the final result:





In order to go from the protoboard to the GPIO pins, we had a few options. One would be to take a removable header and solder wires to that which would then plug into the protoboard.

But Elon was tired of soldering so we went with a ribbon cable to break out the pins into female connections instead. The shortest ribbon cable we could find was very long, which again might have contributed to timing issues in the final product. However, the cartridge connector worked fine for Tetris.

We had to write a simple interface, which sent out the address and received data, but it was about the simplest interface in the entire design.

Results

The cartridge connector works for Tetris without any problems. Other cartridges don't work, and we're not entirely sure why. We can read their Game Boy logo but it either comes in corrupted, or comes in correctly but the game is unable to continue. This could be a problem in any number of modules, but is likely a problem with the length of the wires connecting the cartridge connector to the board.

Timers/DMA

Overview

Both of these modules were implemented with little to no changes from the descriptions in the Game Boy Programming Manual. The timer registers are:

0xFF04 DIV - upper 8 bits of a counter on a CPU clock, cleared when written to
0xFF05 TIMA - counter that increments based on a frequency selected by TAC, generates an interrupt on overflow
0xFF06 TMA - value to load into TIMA when TIMA overflows
0xFF07 - timer control, selects the frequency for incrementing TIMA and enables or disables TIMA

The DMA module is a block transfer module that copies sprite data from the work RAM to sprite RAM (referred to as OAM in the documentation). It copies 160 bytes from 0x8000-0xDFFF to 0xFE00-0xFE9F. Whenever the user writes to 0xFF46 (the DMA register), the DMA module disables the CPU's ability to read and write from non-CPU-internal memory and begins transferring data. At this point the CPU should be executing code in high memory, or else it will read Z's into the instruction register and promptly begin to destroy everything.

The main reason that CPU instruction timing is important is that the DMA transfer code in most games relies on instruction timing. This code lives in high memory and consists of writing to the DMA register then entering a tight loop calculated to take exactly the same number of cycles as the DMA transfer before exiting the DMA transfer function. If the instruction timings are too fast, or the DMA transfer is too slow, the CPU will exit high memory while the DMA module is using the data and address buses. This will lead to random data being put into sprite RAM and possibly random instructions being executed by the CPU.

Process

The DMA module and timers were both implemented after the CPU had been thoroughly tested, so testing them was rather simple. We added them to a version of our automated testbench and ran simple programs on them that relied on timer interrupts and generated DMA transfers, and examined the waveforms to ensure everything was working properly. Shockingly and amazingly, the documentation for these modules is accurate and complete, so there were really no surprises.

Results

The DMA module works perfectly, as we can see the Tetris blocks on-screen with no graphical corruption. The timer module is not exercised by Tetris as far as we know, however it is used by one of the test ROMs that we ran, and that test passed.

Memory and System Integration Strategy

Overview

Most of the memory in our system is related to the GPU (scanline memory, frame buffers) and is described in that section. The non-GPU-related memory exists in the work RAM and high memory, which is described in the CPU section.

The work RAM is 8192 bytes in a block RAM.

The components are all connected to the same address and data buses using tristate modules. When the address bus is within a certain component's address range, the data bus is routed to the input of the component or the output of the component is routed to the data bus, depending on whether the read or write signal is asserted. This applies to memory-mapped registers as well as the various RAM blocks.

Echo RAM is a special section of RAM that cannot be written to, but produces the same information as the work RAM when read from. We're not sure if that's what the actual Game Boy does, but that's how we did it in our system.

The interrupts from each module are routed into the CPU's IF input. On an interrupt, the IF register is loaded with that interrupt bit and the other interrupt bits present in IF are preserved. The CPU handles clearing interrupt bits internally.

Here is the memory map of our implementation, which may or may not differ from the Game Boy standard:

Address Range	**Description**	
0xFFFF	IE memory-mapped register	
0xFF80-0xFFFE	High memory (CPU-internal)	
0xFF00-0xFF7F	Memory-mapped registers	
0xFEA0-0xFEFF	Unusable (reading here gives 0x00)	
0xFE00-0xFE9F	OAM (sprite RAM)	
0xE000-0xFDFF	Echo RAM (can't write here; reads give same information as 0xC000-DDFF)	
0xC000-0xDFFF	Internal work RAM	
0xA000-0xBFFF	Cartridge RAM (writes and reads go to cartridge lines)	
0x8000-0x9FFF	Video RAM of various flavors	
0x0104-0x7FFF	Cartridge ROM	
0x0000-0x0103	**0xFF50 == 1**: Cartridge ROM	**0xFF50 == 0**: Flash ROM with bootstrap code

All of the chips in the Game Boy are SRAMs with asynchronous read and single-cycle write. Therefore the CPU was designed to interface with such a memory block. However, our memory modules are mostly Xilinx-generated block RAMs. These block RAMs are single-cycle read and write. In order to allow these RAMs to interface with our system using asynchronous reads, we simply input a much faster (8 times faster) clock to the block RAMs than the CPU.

Bootstrap ROM

The one special component of the high-level system integration is the bootstrap ROM. The bootstrap ROM was found online at http://gbdev.gg8.se/wiki/articles/Gameboy_Bootstrap_ROM, adapted to our assembler syntax, assembled, and converted to a Xilinx flash file before being loaded onto the flash chip on the FPGA board.

Initially, the bootstrap ROM stands in for addresses 0x0000-0x0103, which are normally in the cartridge ROM space. On reset, the system begins executing the bootstrap ROM, which clears VRAM to white (zeroes) and loads the Nintendo logo from the cartridge. If the Nintendo logo and an additional checksum do not match what the bootstrap expects, it halts execution. If these things do check out, the bootstrap ROM includes an instruction that writes a 1 to a special memory-mapped register, 0xFF50. When this happens, the bootstrap ROM is disabled and the addresses 0x0000-0x103 are mapped back to the cartridge. The system then begins executing the game code.

Process

We basically had no well-defined process for integration. We just started hooking things together in the top module and didn’t bother to do any real design. Surprisingly, this strategy yielded results. We had a few bugs, as one might expect with this strategy, but not really enough to cause real problems. We were also able to find most of our bugs in the integration logic by just looking at the code.

Our first step was to run the bootstrap ROM with just the CPU to see if it got to the address 0x0100, and then we added and debugged the sound module so that we could hear a ping noise. Some time after that, we added the GPU to the design so we could see the Nintendo logo scroll. After that, we started adding everything else.

Results

The integration worked pretty well. We really should have gone with a more consistent design for it, however.

User Input

Overview

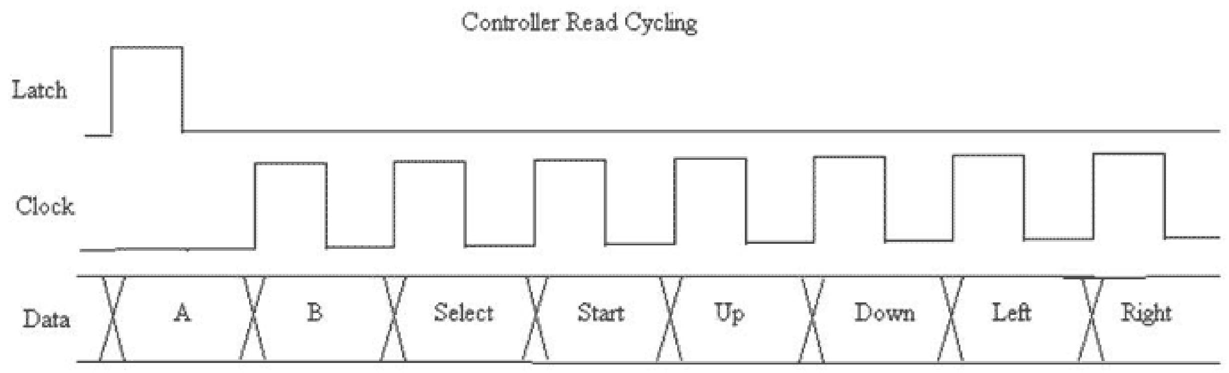
In order to get button data to the CPU, we used an NES controller hooked up to GPIO pins and an FSM to poll the button data using the NES protocol.

Description

The NES controller has 5 pins of value. One is Vdd, one is ground, and the rest are strobe, data, and clock. The interface is a 17 state FSM that polls the controller for all of its button data. Every 60Hz the Game Boy sends a strobe to the controller, which sets off the FSM. The FSM sends a clock signal to the controller, and on every positive clock edge, the controller sends back the status of one of it's buttons, in order, giving A, B, Select, Start, Up, Down, Left, and Right. Note, the buttons are asserted LOW. This is important because if all buttons are asserted, most Game Boy games reset, so it will look like nothing is happening.

Process

It wasn't hard to set up the controller. We found a timing diagram:



The protocol is obviously simple and was not hard to implement. We simply send a strobe every 60Hz, as determined by a clock divider, and then send out an approximately 3MHz clock by using a 6Mhz clock as our state clock. On every positive edge of this clock we send, we expect a new button value to be on the line. The frequency of this clock took some tuning to get right, but simply outputting the received button values to the LEDs was enough to show whether or not it was working properly. Then we simply needed to hook up those wires to the CPU memory-mapped register, which required an extra control signal because the Game Boy is a little ridiculous sometimes, but that wasn't too hard either. Basically the CPU selects either direction keys or buttons to be output on its 4 data slots in the memory-mapped register (address FF00) and then it’s up to the game designer to poll the controller properly. Usually they will poll it a bunch of times in a row to take care of any debouncing issues that might occur.

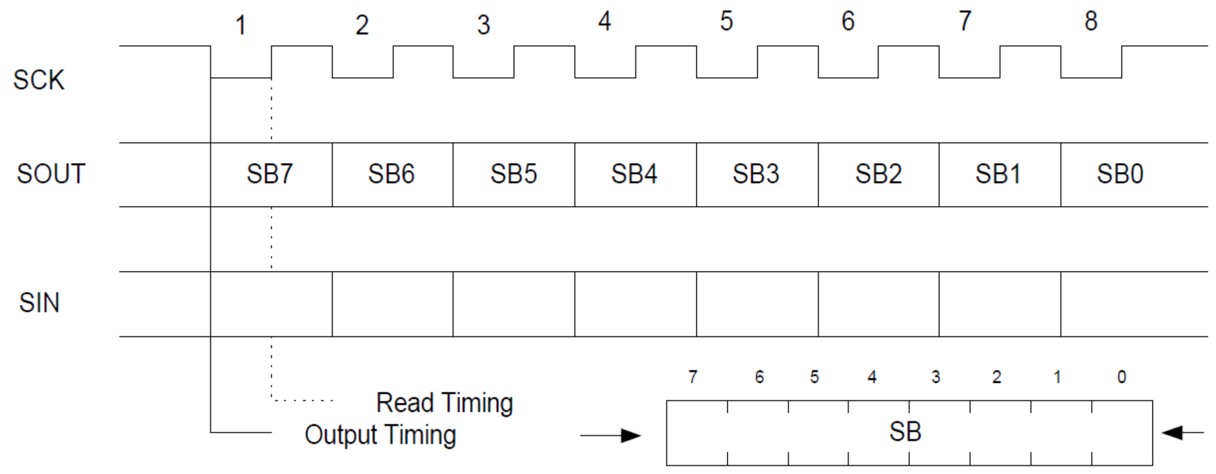
Link Cable

Overview

The link cable is a 6-pin serial interface. It has two data lines, a shared (!) clock line, ground, 5V VDD, and an unused pin. A Game Boy writes data bits onto its data out line (the other Game Boy's data in) on the falling edge of whichever clock is selected by the system. The other Game Boy reads data on the rising edge of the clock. The clock itself is generated by one of the Game Boys. The other Game Boy uses the clock, which is output over the clock line.

Description

First, here's a timing diagram of the link cable's data transfer protocol:



SCK is the shared clock, SOUT is the data out line, SIN is the data in line, and SB is the shift register.

The programmer's interface to the link cable consists of two memory-mapped registers.

0xFF01 SB - The shift register containing data to send or data received.
0xFF02 SC - The serial transfer control register.

SC[0] selects whether the transfer is using the internal clock or other Game Boy's clock. This bit is asserted low. SC[7] is a 1 when a serial transfer is in progress. The user can write a 1 here to initiate an outbound transfer or check the bit to see if an inbound transfer is occurring. The other bits of SC are unused. When a serial transfer completes, an interrupt is generated (whenever SC[7] goes from high to low).

SB is shifted out to the left.

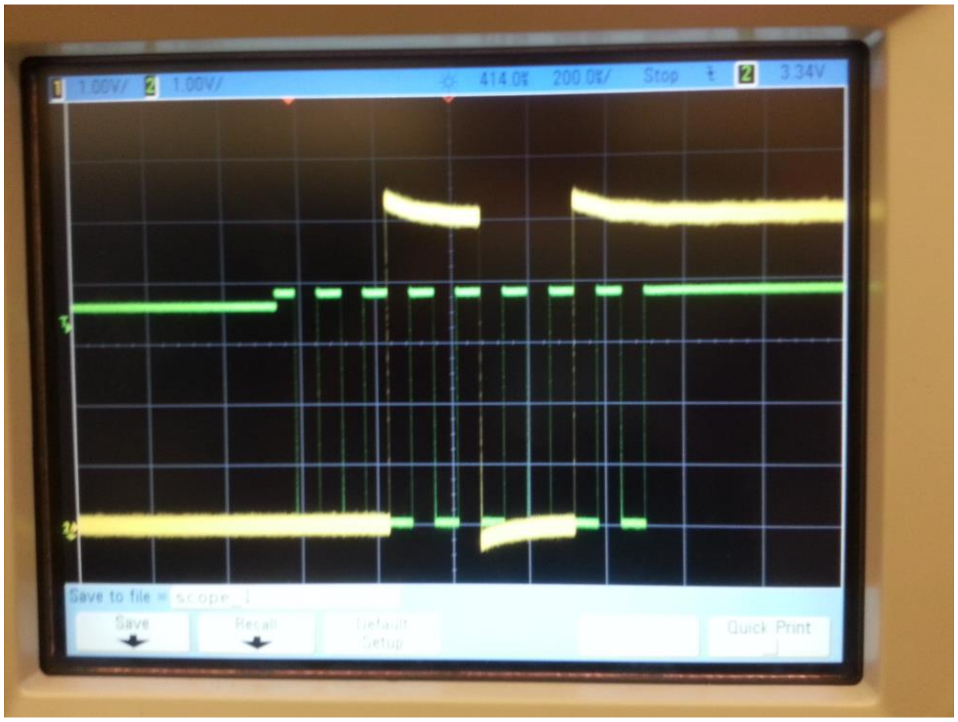
Our design uses the CPU's clock, running at 4.125 MHz, to run an FSM that detects whether the selected clock is high or low and outputs an edge signal whenever it goes from low to high or high to low. This edge detector is used in conjunction with a 3-bit counter to output the shift register on the falling edge of the selected clock, then read the incoming data and shift the shift register on the rising edge.

Additionally, the FSM and counter generate an interrupt when the transfer is complete (on the final falling edge).

Process

The link cable was the last thing we implemented. We began work on the link cable at 11 PM the night before the public demo and finished around 6 AM. We tested it by writing a simple testbench and a couple corner cases that we could think of. We simulated it for a while then just hooked it up to the board and tested it with a real Game Boy.

We also verified the timing of the transfers by attaching an oscilloscope to an actual Game Boy's link cable:



The green signal is the clock and the yellow signal is the data.

Results

We were able to connect the Game Boys and get multiplayer Tetris mostly working. We discovered that there was some graphical corruption, some synchronization issues, and some packets were getting dropped, but it was possible to play something resembling a two-player game of Tetris using our design and a real Game Boy.

Group Thoughts

What we wish we had known

We wish we'd known that the Virtex7 is a terrible board for this class. We struggled for a while trying to get it working but it really wasn't worth it. Other than that we knew most of what we needed because two other teams implemented the Game Boy in the past.

Good/bad decisions

Good

1. The best decision that two of our members made was to start working immediately. There is no reason and no reward (except pain) for procrastinating in this class. The third member of our team discovered this at the end of September at which point a whole month had already been wasted.

2. Another good decision we made was to ditch the Virtex-7 board and use the Virtex-5. We made this decision early on and avoided wasting time trying to get an HDMI driver written in Verilog.

3. We used Git. Version control is absolutely necessary for a project of this scale, and even more so when working on a team.

4. Not attempting the Game Boy Advance was obviously a fantastic decision in hindsight. It's so much better to attempt a project that might be slightly less difficult and actually have a working project at the end of the semester than to attempt something way over your head and have nothing to show for all your hard work in the end.

Bad

1. The worst decision we made was to go with a project that required us to implement our own CPU. Implementing a CPU is an especially difficult task, simply because CPUs have so much functionality. With 48 separate instructions, an interrupt procedure, 27 ALU operations, and around 3500 lines of code, there will inevitably be bugs. In our case, we got it working. But also in our case, we got it 100% working only 9 hours before the in-lab demo. We were able to get the CPU to run because we had access to two amazingly well written emulators, the Game Boy documentation is very complete, and we started with the knowledge from the past Game Boy teams. And because we were very, very lucky.

2. Another terrible decision we made was to keep Red Hat Enterprise Linux on the workstations even after we had difficulty getting ISE to install and run. We should have immediately ditched RHEL for a more stable (Xilinx-wise, at least) solution such as Windows 7. This cost us at least a few days of work just trying to get the code onto the board when the drivers failed.

3. Not reusing previous team's code earlier. While the best thing you can do for your project and yourself is to write all of the code in your project from scratch, giving you the greatest possible understanding of its internal structure, at some point it's necessary to realize that you can't do it all in the amount of time you have. At that point you should steal as much code from other sources as possible. And do this early so that you have a chance of actually understanding it enough to integrate it into your design.

Advice

1. USE CHIPSCOPE TO DEBUG ON THE BOARD. This is the most important component of the class. We were initially afraid of ChipScope. This is misguided. It is the best thing on the planet for debugging your design after it's been synthesized. It is basically the simulation waveform viewer but for synthesized designs. Learn it and use it. IT IS AMAZING (and kind of terrible) BUT MOSTLY AMAZING.

2. Start immediately. Put in at least 12 hours a week in the first half of the semester. By the second half of the course, you won't need to self-motivate. Your non-working project will be motivation enough. Your reward for procrastinating is pain and perhaps a failed project, so don't do it. You will never have enough time. However painful it is to start coding at week 3, it is much more painful to be unable to sleep because you're coding 24/7 at week 10.

3. Do not implement a CPU from scratch unless you know exactly what you are getting into. It is not easy, and if you've taken 18-447, the sort-of-MIPS core you implement there is nothing compared to implementing a fully working design. It is possible to do if your CPU is really simple and you have loads of documentation, really good emulators, and a way of testing your design in simulation that is at least as good as what we had. If you are thinking about doing this, read the CPU process section above and let the ridiculous amount of work you will be doing sink in. It's possible, but painful.

4. Plan everything. Design everything. We didn't think about how we were going to integrate, and it kind of bit us in the ass. It could have been worse.

5. Use simulation as much as possible. Changing your code and simulating it is usually a one-minute process. Changing your code and synthesizing is a 25-minute process. When we got to the bootstrap ROM, which takes a few seconds at 4,125,000 cycles per second, simulation stopped being a viable system-level testing option, but we could still debug our components in simulation on directed test cases.

5. Focus. Don't waste time on something that isn't going to work. Obviously it's hard to know what things are going to work or not beforehand, but it's possible to avoid certain pitfalls. We knew from reading previous reports not to use the OpenCores Game Boy CPU, so we avoided that trap.

6. Make your top-level pins have the same names as the .ucf file and rename them internally. You will save yourself a lot of trouble later. Also use emacs Verilog-autos to connect all your modules and follow a consistent naming convention. Seriously.

7. Do not use RHEL and Xilinx tools. Xilinx tools work best (as of the time of writing) on a Windows 7 system. However much you hate Windows, just get Windows and be thankful you can actually load programs onto the board.

8. Xilinx sucks. Sorry. You can't really avoid them unless you "borrow" one of the 18-240 Altera boards, and they don't have very much block RAM.

Sources

This section contains a list of useful documents and links we found throughout our adventure, sorted by section and in decreasing order of usefulness.

Game Boy Documentation

1. The Game Boy Programming Manual. This is the document released to Game Boy developers by Nintendo. It is the most complete reference for the Game Boy, much more so than the PAN docs/GBCPUman. It's kind of disorganized, but don't let that throw you. Everything you could possibly need to emulate the Game Boy is in here.
http://www.romhacking.net/documents/544/

2. The "Game Boy CPU manual." This is a PDF version of the PAN docs and is slightly less technical than the programming manual. It essentially summarizes the content of the programming manual. Good for a quick info lookup, but for real implementation reference information consult 1. http://marc.rawer.de/Gameboy/Docs/GBCPUman.pdf

3. PAN Docs. The HTML version of 2. http,//nocash.emubase.de/pandocs.htm

4. Meatfighter. This is a copy of the Pan Docs, but I like the way it looks a lot better. Although the links don't work so you should probably use the Pan Docs link above.
http://meatfighter.com/gameboy/TheNintendoGameboy.pdf

5. Game Boy opcode list. Contains the correct instruction timings and short descriptions for all the opcodes. Useful for translating machine code into assembly on-the-fly as well.
http://pastraiser.com/cpu/gameboy/gameboy_opcodes.html

6. Z80 architecture overview. Useful for understanding the hardware of the CPU.
http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm

7. Loirak's Game Boy programming tutorials. The page here is a great quick-start for GBDK, a good Game Boy compiler/assembler. http://www.loirak.com/gameboy/gbprog.php

8. Gbdev Wiki sound hardware. In general this wiki is extremely useful, but the sound page especially offers a different view than the Pan Docs on how sound is meant to function.
http://gbdev.gg8.se/wiki/articles/Gameboy_sound_hardware

9. Memory Map. This is the best memory map I could find. Remember that ROM is actually in the cartridge. http://gameboy.mongenel.com/dmg/asmmemmap.html

10. Gbdev Memory Map. Another page from the Gbdev wiki. This one talks about the cartridge header, which will be important if you ever get to bank switching, or attempt the Game Boy Color. http://gbdev.gg8.se/wiki/articles/Memory_Map

11. Arduino-based Gameboy Cart Reader. This doc was useful as this person is basically doing the same thing to the cartridge as we were trying to do. It lays out fairly well how to dump the ROM. http://www.insidegadgets.com/2011/03/19/gbcartread-arduino-based-gameboy-cart-reader-—-part-1-read-the-rom/

12. Sound text file. Proceed with extreme caution. This document is useful as yet another look at sound, but is obviously very sketchy. http://www.devrs.com/gb/files/hosted/GBSOUND.txt

Emulators

1. BGB. This is a great emulator with a debugger and RAM viewer. The debugger can do all the neat debugger things you might want, including breakpoints and watches on memory locations. Without this emulator, we would not have completed our project. http://bgb.bircd.org/

2. DMGBoy. This is another great emulator. It doesn't have a debugger, but the source code is extremely well written. This was an invaluable resource, as we could read the source code to figure out certain behaviors that weren't clear to us. It is also the basis for the automatic testbench. http://code.google.com/p/dmgboy/

Special Files

1. Blargg's instruction test ROMs. These are directed tests for the CPU. They require a working graphics system, but they helped immensely in debugging the CPU. They even print out the opcodes of incorrect instructions when they fail. http://blargg.8bitalley.com/parodius/gb-tests/cpu_instrs.zip

2. Bootstrap ROM. This thing is just awesome.
http://gbdev.gg8.se/wiki/articles/Gameboy_Bootstrap_ROM

3. Sound test ROM. Run this in an emulator to test your sound functionality. I'd suggest BGB although it might be slightly incorrect. Be sure to default your waveform RAM to something

useful and set the waveform RAM areas of memory to the same thing in BGB before testing channel 3.
http://www.romnation.net/srv/roms/14092/gb/Sound-Test-PD.html

Other Teams' Work

1. The FPGABoy project. This was a project at MIT that was basically exactly the same as ours, except in the beginning of the course we didn't know how legit it was. Our doubts were mostly based on the fact that the code in their Git repository is inconsistent from the code in their final lab report. Apparently, the code in the document was super legit. This is where we completely copied the GPU code from. However the code in the actual repo is nonfunctional.
https://github.com/trun/fpgaboy

2. Team Dragonforce (AKA VirtexSquared). This was an old 18-545 team that did an ARM SoC. We used their AC97 and DVI drivers. https://github.com/teamdragonforce

Individual pages

Joseph Carlos

What I Did
I was solely responsible for implementing the CPU, DMA module, timers, and ChipScope interface. I worked on getting most of the system integration done and worked with Elon on the Link Cable controller. I also spent some time during the semester helping to debug sound and doing the labs.

Class Impressions
I can say that in this class, I took a few empty Verilog files and wrestled them into a working Game Boy CPU. That required a lot of patience, planning, luck, and crazy ideas, but I found it to be really fun.

I enjoyed the chance to sit in lab with some very smart, motivated, and at times equally crazy classmates doing pretty wild things with Verilog.

One of the most rewarding parts of the class was the ability (and necessity) to plan ahead of time for problems, and to develop my own schedule and testing apparatus. Doing this well allowed me to complete my parts of the design in time for the demo and was a very cool experience.

Some of the things I could think of that I didn’t particularly agree with or enjoy:
1. The labs were either really easy (lab 1, lab 3) or had no guidance at all (lab 2), or both (lab 3). I understand the idea is for us to interact with the outside world (of Xilinx) on our own. However, in my opinion at least, giving us a lab that says, “get some selectable sound playing on the board, good luck” is not really the right way to approach this.

2. RHEL + Xilinx = some people’s own personal hell. Xilinx alone is bad enough.

3. The VC707 board is just not suited to the class. Sorry for beating a dead horse - err… FPGA.

4. The oscilloscopes in lab are hilariously terrible. We realize most people don’t use them, so one or two good ones would suffice. Unfortunately the ones in the other labs are locked to the benches, so we couldn’t borrow any.

5. The lab is not well stocked with things like parallel cable crimpers and connectors, headers, SMA connectors, whiteboards, etc.

Despite the things mentioned above, I thought this was a really good class. There was a good emphasis on process, which was actually very helpful, and it was generally a lot of fun.

How Much Time I Spent

I spent somewhere between 150-200 hours on the class, maybe more. I didn't log my hours so that is a very rough estimate.

Alice Tsai

What I did

My role was solely the GPU and DVI portion. I read a lot of documentation in order to debug the DVI. Since we got the iic_init and sync_gen from Dragonforce's code, I needed to understand their original framebuffer to see how they used it in order to hook it up so that it would work for our purposes. Elon also looked into their code to try and help me out with the debugging as well. Dragonforce used some of Xilinx's built-in models so I had to familiarize myself with those to make modules that could be used to adapt their code to our system. And in the end, it was reading the documentation on how I^2C is initialized that fixed my last bug with the DVI portion.

With regards to the GPU portion, I first read through the video portion of some very helpful PAN docs. Next I read through all FPGABoy's video code because once again, it is not a good idea to simply stick someone else's code into your own design without knowing what it is doing. Also, initially, I was supposed to change something in their design that would have a big impact on the Video Module's FSM. And any time one needs to change something in a FSM, I think they need to thoroughly understand what is going on in the FSM itself. Unfortunately, that time was wasted because we were short on time and an executive decision was made to not make that change after all. But on the positive side, I got to know the innards of the GPU portion pretty well. I did not know how to use CoreGen to make the VRAM and OAM block rams but luckily Elon already familiarized himself with that tool and created the block rams for me to use. Elon helped me with hooking the GPU to the CPU as well. My last main contribution was debugging why video did not work after everything was hooked up. Through simulation, I found the culprit: an incorrectly connected wire. Once that was changed, the next time we tried, it worked!

After integration, we found that we couldn't get farther than the Nintendo Logo so I simulated the GPU on the side and confirmed that it worked as expected. Then I tried to help with the debugging of the CPU but wasn't able to contribute any findings of bugs, not for lack of effort.

I worked on the poster for our public demo but because I took too long, Elon made a nice poster that we ended up using instead. I ended up finishing it in time for our demo but felt that it would be weird to have two posters so just kept my poster rolled up.

Overall, I learned how helpful documentation can be. But at the same time, I learned (from Joseph) that at some point, one has to stop reading documentation and just start coding and from there, when you encounter problems, go back to the documentation.

Class Impressions

Overall, this was a great course. It was helpful in providing my first experience with a team project that did not involve much "hand-holding." I think I learned a lot of how to go look online for information. It was also really cool to work on an actual complete project that did something, rather than just an exercise that demonstrates ability to code something in Verilog.

Things I Did Not Enjoy

I wish that I had spent more time more wisely. It ended up causing a lot of stress and friction for my team so if there was one thing I'd change, it would be spending more time earlier.

How Much Time I Spent

I logged most of my hours and they came to a total of > 160 hours. I only started logging them partway through the semester though so for the earlier part, I only counted class time.

Elon Bauer

What I Did

I was solely responsible for sound, the cartridge connector, and the NES controller interface. Basically anything that required soldering, wiring, drilling, or dremelling was my turf, since I was the only one who had taken 18-320. I also helped Joe with the link cable the night before the final demo, I helped catch the last two bugs in the CPU, and I hooked up the GPU to the CPU and helped figure out why DVI wasn't working. Joe and I traded off being team leaders. For things like labs, presentations, and general deadlines, I usually had to remind the others that they were happening. But Joe was good about generally keeping us on track and reminding us to have goals and stick to them.

Class Impressions

This class was a lot of fun. I would think that regardless of whether or not our final project worked. It was great having a well-defined goal to shoot for and knowing exactly what we had to implement it and when to implement it by, but also having complete freedom about how we went about implementing it. And while this class basically killed any free time I could have possibly had and pulled me into lab with a force not unlike that in Star Wars, it all seemed worth in it the end. That part might have had something to do with the free iPad…

I liked the fact that the instructors didn't butt into the projects too much. We were mostly left to our own devices, which means we can either succeed or fail and it will be completely our own doing.

I'm not sure what to say about the labs. They were useful in a way, but having to do write-ups about them really detracted from time that could have been spent working on our project. I think the labs would be a lot more useful if we didn't have to do write-ups for them. If we fall on our faces because we didn't actually do the labs, then that's our own fault and we'll fail the final project anyway. So I guess going even more hands-off is what I'm suggesting.

Things I Did Not Enjoy

I didn't like struggling with the tools. It would have been nice if we didn't have to waste time not coding while struggling with the tools. Also I would have liked more hardware like GPIO wires and breakout pins to have been in lab. Better oscilloscopes would be nice too.

How Much Time I Spent

My total time spent on the project was probably close to 200 hours, plus or minus 50. For the first two weeks I probably spent about 4 hours outside of class on average. For the next 9 or so I probably averaged about 2 hours a day, and in the final crunch weeks I averaged a little more.