Apple IIGS

Willis Chang

Brian Osbun

Robert Walzer

Table of Contents

1 Introduction....4
2 Our Project Results....4
3 Hardware Architecture....5
 a System Diagram....5
 b 65C816 Processor....6
 c Memory Bus....7
 d Graphics Subsystem....9
 e Audio Subsystem....10
 f Input Devices....11
 g Disk Interface....12
4 Design Process....13
 a Processor....13
 b Memory Management Unit....14
 c Graphics....15
 d Sound....16
 e Disk Drive....16
 f Keyboard Inputs....17
5 Testing Process....18
 a CPU and MMU....18
 b Video....19
 c Synthesis Issues....19
6 Advice....20
7 Individual Pages....21
 a Willis Chang....21
 b Brian Osbun....22
 c Rob Walzer....23
8 Some Resources....25

Introduction

The Apple IIGS was the final model in the Apple II line of personal computers, and was released in 1986. It maintained full backward compatibility with the previous Apple II, II+, IIe, and IIc computers, while providing considerable upgrades in *graphics* and *sound* (source of the GS name). These improvements come from an improved processor, the WDC 65C816. This is a 16-bit CPU that can emulate the 8-bit 6502 used in previous models. In addition, the IIGS provides new, higher-resolution graphics modes and multi-channel digital sound capability. The functionality of earlier Apple II models is generated by the Mega II, which is basically an Apple-IIe on a chip.

Our Project Results

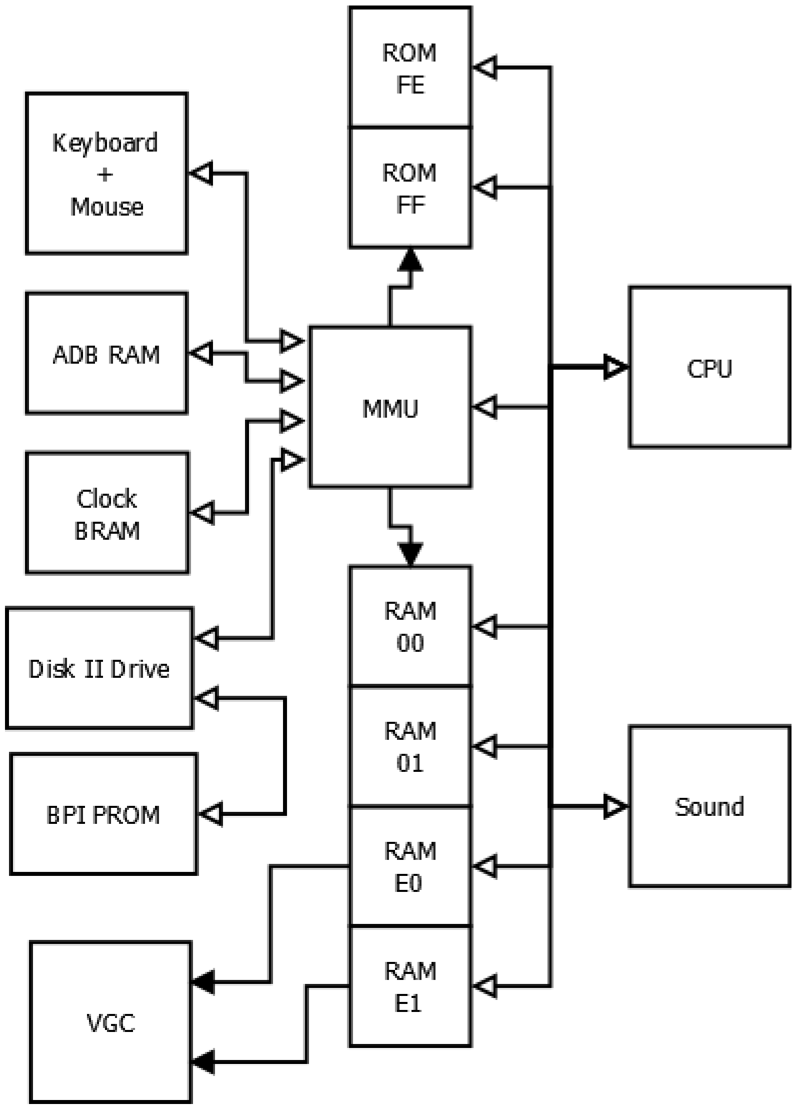
We were able to get a considerable amount of functionality working for the public demo. At the end of the semester, we were able to boot into the IIGS System Monitor, which is a text input/output interface that allows the user to view memory values and the CPU register contents. From the System Monitor, we could also enter the BASIC interpreter and enter commands. PRINT statements worked almost all of the time, but most other commands were flaky. We also used the emulator to dump some Super Hi-Res video memory images, and ran them through our graphics system to show the IIGS capabilities.

We are happy that our final product demonstrated the CPU, bus, memories, graphics, and keyboard input all working (relatively) nicely together. It would have been nice to have some actual games running in real time, but we got very close and cleaned up most of the real bugs. The biggest impediment to achieving the last success was the complexity of the disk drive, which we were unable to finish or avoid. Overall, we completed a very good portion of the system by the deadline.

Hardware Architecture

System Diagram

The Apple IIGS is composed of a few major subsystems, which we have decided to split up into the design as shown in our architecture diagram.



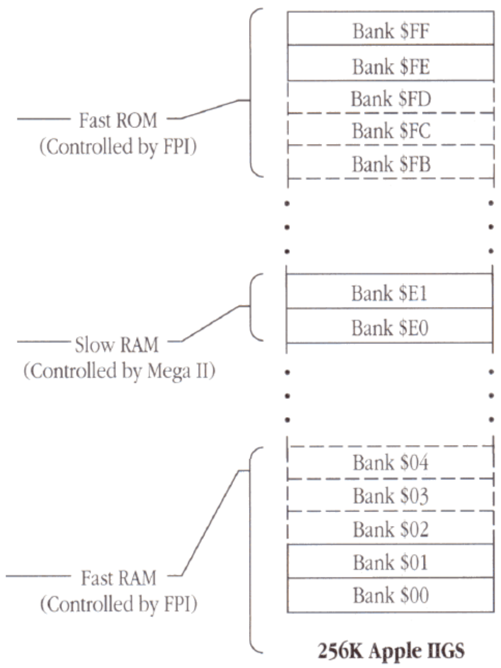
65C816 Processor

The CPU in our system is a 16-bit processor with a 24-bit address space. The processor operates on a single clock, but performs actions on both the negative and positive clock edges. It has a 16-bit address bus and an 8-bit data bus that pulls double-duty as a bank-enable bus as well.

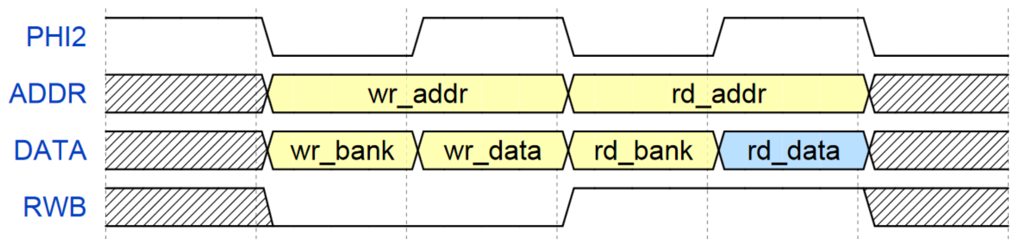
Internally, the CPU has an 8/16 bit accumulator, two 8/16 bit index registers, a stack pointer, and a direct page register. Almost all of these registers can operate in 8-bit mode to emulate the 6502, operate in a 16-bit native mode, or operate natively in an 8-bit limited mode. The 65C816 rounds out the 256 opcodes of the 65xx series, and adds some new addressing modes as well.

Memory Bus

There is one main bus in our architecture, with all of the components having some connection to it. For the most part, the Apple IIGS is completely memory mapped. The video buffer, sound pins, keyboard registers, and various memory modules are all simply accessed as memory locations. Addresses are usually provided by the processor to activate a certain component and read or write data, but the bus can also be controlled by the video system through direct memory access.

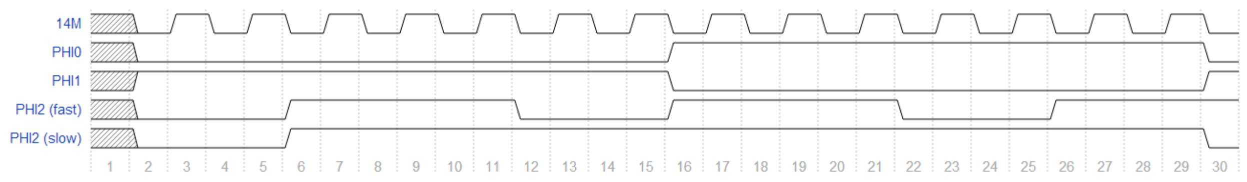


The physical memory of the IIGS is broken up into several banks, as shown in the above memory map. The CPU is a 16-bit processor with a 16-bits address line, so a bank system is necessary to address the full 24-bit space. Rather than using bank switching as in some other computers, the processor multiplexes the 8-bit data line with an additional 8-bit address. Therefore, the data line drives the bank address and the address line drives the location within that 64KB bank. This multiplexing occurs during a single clock cycle as shown in the simple timing diagram below.



The Memory Management Unit is primarily used to decode the address on the bus and enable the proper memory bank or mapped device. In the original design, it has additional duties such as RAM refresh. Since we are using the Block RAM directly on the FPGA rather than interfacing with an external chip through a memory controller, functionality like this is not necessary for our design.

It is also important to note the clocking of the Apple IIGS in order to handle backward compatibility. The original 6502-based versions run on a 1MHz clock which is generated from a base of 14MHz. The 65C816 typically runs on a 2.8MHz clock which consumes 5 cycles of the fast clock. However, when it needs to access slow RAM or certain I/O on the Mega II side of the system, it stretches to a 1MHz period. The different clock periods are shown in the following diagram. PHI0 and PHI1 are clocks of the 6502, while PHI2 is the varying clock of the 65C816. PHI2 is only one signal, so the waveforms shown represent different possibilities in time (“fast” cycle and “slow” cycle).



Graphics Subsystem

The Apple IIGS supported all the original video modes (text, lo-res, and hi-res) of previous Apple II in addition to new super hi-res graphics modes. Each of these video modes has a predefined region in memory where the display data is stored. In previous models of the Apple II, the current video mode was controlled by many soft switches. The GS added a new soft register, called the new video register, which controlled the super hi-res graphics mode. The display soft switches were still used for the emulation video modes. On the GS, all of the emulation video mode operations were performed on the Mega II, while super hi-res video modes were controlled from the new Video Graphics Controller.

In text modes, the display data for each line is stored contiguously in memory as ASCII values. The display lines themselves are not stored linearly: the 0th, 8th, and 16th lines are stored in the first 128 bytes, followed by the 1st, 9th, and 17th. This would was done to save memory given the technical constraints of the time. The ASCII value was then used as input into a font ROM, which would output the corresponding pixel values.

The super hi-res graphics mode added a true graphical video mode to Apple II series, allowing it utilize the graphical features seen in GS/OS. Super hi-res graphics mode supported either 640 or 320 column video with 200 rows. The user could create up to 16 color palettes, each with 16 12-bit colors. The color palette for each line was then controlled by a scan line control byte, which also determined whether the line had 320 or 640 columns. In 320 mode, 4 bits were used to represent each pixel which would then be used to index into the color palette for that line. In 640 mode, 2 bits were used to represent each pixel. Each pixel within a byte was given a separate group of four colors within the color palette to choose from. The scan line control byte also determined whether color fill mode was enabled. When color fill mode and 320 column mode were both enabled, if a pixel had the stored value of 0, the output color would be the same as the previous pixel. This decreased the number of colors available, but increased the speed at which video data could be generated.

Audio Subsystem

The Apple IIGS supported the one bit sound of previous Apple IIs while also adding an advanced Ensoniq 5503 digital synthesizer. One bit sound was controlled by either reading or writing to a specified softswitch. The faster this was performed determined the frequency of the output sound. One bit sound was mainly used by legacy programs.

The Ensoniq 5503 contained a 64k by 8-bit wavetable and 32 independent oscillators, the interface to which was provided by the Sound GLU (general logic unit). The 5503 allowed up to 8 audio channels, though the GS used only 2. The GLU provided 4 soft registers to the system, a sound control register, data register, address low register, and address high register. The sound control register determined whether a write/read went to the Ensoniq registers or wavetable, whether wavetable operations auto-incremented, and the master volume.

Each of the 32 oscillators were controlled by a pointer to the beginning of its wavetable, a frequency register, and a size/resolution register. Anywhere from 1-32 oscillators could be enable at any time. In order to determine the wavetable address, the frequency value was accumulated in a 24-bit accumulator at each update to the oscillator. Depending on the size and resolution of the wavetable, the value stored in the accumulator was selected and concatenated with the wavetable pointer register to form a 16 bit address. Sequential oscillators could also be paired so that when one started its wavetable, the paired oscillator also started or so that when one oscillator finished its wavetable, the paired oscillator started its wavetable.

The output from the enabled oscillators was time domain multiplexed and sent to a DAC. The stereo card of the GS would then demultiplex this stream and output audio on the selected channel.

Input Devices

The Apple IIGS used the Apple Desktop Bus (ADB) to maintain all the input devices (including the keyboard and mouse). It composed of the ADB GLU, ADB microcontroller, and Apple Desktop Bus cables. The ADB microcontroller was primarily responsible for interfacing the actual devices and the bus. Keyboard inputs were converted to one of 128 ASCII codes and saved in the $C000 register. Whenever the $C010 register was accessed by the CPU, the strobe bit in $C000 would be cleared.

The ADB GLU (General Logic Unit) consisted of the ADB Command/Data register (used to keep track of input devices), Keyboard Data register, Modifier Key register (records which special keys were pressed), Mouse Data register, ADB Status register (keeps mouse and keyboard information).

Disk Interface

The disk-port connector is compatible with both 3.5-inch and most 5.25-inch Apple II disk drives. The actual disk-port interface was maintained by the Integrated Woz Machine (IWM), consisting of the mode register, status register, handshake register, and the data register. Sixteen soft switches at addresses $C0E0 to $C0EF were used to keep track of the stepper motor phases, drive selected, and what types of reads and writes were requested. Programs, stored on the disks, actually had to access the softswitches in order for the disk drive to keep track of the data on the disk being accessed.

The Apple IIGS also supported “intelligent drives”, which rather than being controlled at the floppy disk track level, were controlled at the block level. A firmware interface to this was known as Smartport. Unfortunately, while the firmware and external bus for this were well documented, the internals of this operation were not. The emulator seemed to get around this by simply detecting when the program jumped to the firmware routine.

Design Process

Processor

Although this processor is considerably more complicated than the 6502 used for previous Apple II computers, we were fortunate to find a soft core available from the Western Design Center. After going through some formalities with Non-Disclosure Agreements and making lots of signatures, we obtained a synthesizable Verilog description of the CPU. We made a couple of minor modifications to get it compiling successfully, and then worked on the output signals to integrate properly with our bus system. Or so we thought.

It turned out that the core had several more apparent errors. Unfortunately, we were not expecting to require debugging the CPU because it was obtained directly from the original manufacturer. When we started seeing errors very deep into the project, it took us very off guard. For example, the Rotate Right (ROR) instruction gave incorrect answers in several cases. A couple of instructions did not set the condition codes when they were supposed to, or had timing problems such that they received the condition codes from a different instruction. There were major bugs with the JSR instruction in the absolute indexed indirect addressing mode. It saved the wrong address on the stack, and then jumped to the wrong location.

It certainly looked like the errors that we found were coming from inside the CPU and not from some interaction with our system. We expect that the WDC would have informed us if the core was incomplete or untested. Also, there was a previous group (the SNES) which used the same core but did not mention any errors in their report.

Memory Management Unit

The MMU was one of the central modules in the design. The main function was to arbitrate the reads and writes among the CPU and the various memory banks in the system. To do this, we put enables on all the data outputs of the memories that would default to tri-state drivers, like a distributed multiplexor. Based on the MMU address decoding and whether the clock signal was high or low, different banks could drive the lines for a given 16-bit address.

Since the boards had over 600KB of Block RAM, we were able to use this for all of our needs rather than interfacing with the mounted DRAM or other resources. We mainly built 64KB “banks” of RAM for our modules. Four were used for the general purpose RAM, two were used for the IIGS board ROMs, and a couple were used by the sound, graphics, and other modules.

The MMU also had the responsibility of handling soft switches. These are memory-mapped registers that either store configuration data or toggle between configuration options when they are accessed by the CPU. The IIGS memory space between $C000 and $C080 is mostly all composed of these switches. Register-type soft switches were simply hooked up to the main data bus and enabled at their respective address, so they could capture writes from the CPU and respond to reads with the stored value. Toggle-type switches were only connected to the address bus, and would flip-flop on a memory access. Both types were connected to internal signals in the MMU to configure things like memory shadowing and bank switching.

Graphics

The Virtex 5 LX has a Chrontel 7301C chip in order to output DVI video. WIth a DVI to VGA converter, we are able to output VGA. In order to communicate with the Chrontel chip, we adapted Team Dragonforce’s video code. Video was output at 640x480 pixels. This left a border around the actual display which was colored according to the color border soft register.

Since nearly all GS specific software utilized the super hi-res graphics mode, we chose to only implement the 40 column text and super hi-res graphics modes. The 40 column text mode was needed for the BASIC interpreter, system monitor, and boot up screens.

While the “Apple IIGS Hardware Reference Manual” did at times insinuate that the video controller took control of video memory at times, we did not consider this a necessary complication. Instead, we implemented dual-ported RAMs which allowed the Video Graphics Controller to read from memory whenever necessary. This also meant the VGC could read at a faster clock than the system clock. Our video display buffer consisted of two line-buffers; while one line was being generated, the previous line was displayed twice. This did not pose a problem since the maximum lines the VGC ever generates per frame is 200.

For text mode, the ASCII value from memory is combined with the three least significant bits of the line currently being generated to index into the font ROM. This returns the current pixel line, which is shifted out to the line buffer. To generate our font rom, we adapted the KEGS font file to the required format for a Xilinx .coe file.

When generating display data in super hi-res graphics mode, the scan line control byte is read, followed by the specified color palette, and then the pixel data. Each line being individually specified as 320 or 640 columns did not pose a problem, since in 320 mode each pixel was just displayed twice.

Sound

We implemented the GLU and Ensoniq 5503 as specified. Simulation results verified that they performed to this specification, but we were never able to test with actual GS program sounds since we could not figure out how to dump the wavetable values from the KEGS emulator. Rather than time-domain multiplex the oscillator output, we planned to sum the individual oscillator samples which would then be sent to the AC97 output.

Disk Drive

We emulated the original Disk II specs as closely as we could. The sixteen disk-port soft switches were implemented along with the data register that would store data to be read from or written to the disk. We had no idea how to maintain the track and position of the head of the physical disk until we happened across the book “Understanding the Apple IIe” and Stephen Edwards’s implementation of the Apple II+. We also needed to store the 256KB Apple II Boot ROM at address $C600. The boot ROM is run when a disk has been detected and it will load track 0, sector 0 of the disk.

Since we did not have a physical disk drive to store games and not enough block RAMs on the FPGA, we decided to use parallel NOR flash memory, specifically 32MB of BPI PROMs.

We were able to find many different game ROMs for the Apple IIGS online. However, the .2mg format is only compatible with emulators. This format was easily converted with the help of xxd and promgen to the .mcs format, which could then be stored on the board. Another challenge with incorporating the disk drive was the Apple II GS's high degree of compatibility. We decided to stick with playing one game ROM, in the 3.5inch disk format as opposed to also supporting multiple disk drives and 5.25inch disks.

Ultimately, we did not have time to fully load a game from flash memory and work out integration with the rest of our system. However, we were able to store data on the PROMs and see the correct data being read via ChipScope.

Keyboard Inputs

Instead of emulating the original and outdated input interfaces used in the Apple IIGS to communicate with the keyboard and mouse, we have decided to use the PS/2 controllers already on the Virtex 5 board. The PS/2 controller interface consists of the PS/2 clock and PS/2 data inputs as well as two 8-bit data ports for reading and writing to and from the attached devices.

The expected incoming data for the keyboard and mouse differ per the specs of the PS/2 protocol. The scan codes for the keys are not in a form the Apple IIGS system understands, so these codes have to be converted into ASCII values. This is accomplished through a large lookup ASCII table. This data is then saved at a specified memory location ($C000), used specifically for keyboard inputs.

For the mouse, data is sent periodically and serially in a similar manner. However data is sent in packages of three bytes: a status byte consisting of button clicks and directions of mouse movement, a byte consisting of X movement, and a byte consisting of Y movement information. This data has to be then converted and saved at specific memory locations in order for the Apple IIGS to interpret. We did not, however, have time to integrate the mouse components into our system.

The PS2 controllers for the keyboard and mouse only use the bus to write the respective information to the expected memory locations.

Testing Process

Most of our high-level verification was based on the official Apple IIGS documentation, such as the Hardware Reference and Firmware Reference manuals. We designed our modules based on these specifications and then verified that the visible behavior was consistent. Most of our project was tested in simulation and then examined in ChipScope on the board.

CPU and MMU

In terms of testing the main operation of the system in simulation, our reference emulator was incredibly useful. We used the open-source KEGS emulator, which was written in easily modifiable C code. The most effective test, once the basic functionality was working, was to print the emulator printf() output and Verilog $display() output into exactly the same format. These files could then be compared in a program like vimdiff to see where our implementation was going off track. The outputs included information like the program counter, accumulator, and other registers, so it was easy to see when there was an incorrect branch or the wrong data was getting loaded. From that point, we could load up the simulation in the DVE waveform viewer and see the details of what might be going wrong. In some cases, the emulator was actually doing more than we wanted to implement, so we had to edit its source to simplify it down to our level. This process actually became very efficient, and should have been used earlier in the semester.

Video

In order to validate that video worked as specified, we generated ROMs out of the memory dumps from the specific video memory regions in the emulator. The video hardware output could then be compared against the emulator output to debug any video issues. Since doing this on the board was a very time consuming process given the synthesis times, we eventually made a script to create an image from the verilog simulation.

Synthesis Issues

At some point our simulation results became pretty clean, and we made the jump to the board. That transition was not quite as simple as we hoped. We were able to view the bus waveforms in ChipScope and compare them to the DVE waveforms, but we were experiencing some seemingly random errors. Registers didn’t end up with proper values for their input functions, and sometimes there was no data at all going on the bus lines. We are still not sure about the exact cause of these problems, but it looks like it helped to use the proper clock buffers and DCMs from the Xilinx Coregen. Home-brewed clock dividers and uncontrolled fanout may be unreliable and manifest in unexpected places.

Advice

- Be careful with clock signals in synthesis.
- Try to get some integrated modules on the board as soon as possible.
- The semester is shorter than you think.
- Getting things to work on the board always takes longer than expected.

Individual Pages

Willis Chang

Of course I hoped we could have accomplished what we set out to do--mainly to play Apple IIGS games with the system's enhanced graphics and sound. Even still, I am content with what we have done. On demo day, we were able to demonstrate the Apple IIGS running both the System Monitor and a Basic Interpreter (though with extremely limited functionality). Various high resolution images from sample game ROMs were also displayed to showcase Apple IIGS graphics.

I was primarily responsible for figuring out how to interface the peripherals. This included both working with the PS/2 interface with keyboard and mouse and Disk II. This entailed working with the NOR flash memory (BPI PROMs) as well.

Though we had weekly Sunday meetings to push forward on the project, they did not prove particularly effective. It wasn't until about two weeks before demo day that we committed ourselves fully to the project and blocks of Verilog code were realized and integrated on the board. At that point each of us spent at least 12hrs a day in the lab. Aside from time constraints, we were often held back by the lack of documentation. Unlike the Apple IIe, documentation for the Apple IIGS were much harder to locate. It also did not help that the Apple IIGS was vastly more complicated than the Apple IIe.

I actually do believe this was a worthwhile experience and only wished I allotted more time for the project. This might have happened had I not been taking three other project courses.

Brian Osbun

I originally worked with the Western Design Center to transfer the soft-core RTL model. They were very responsive to our project, but required some e-mail reminders to keep the transaction going. Once we received that IP, I mostly worked on getting the CPU integrated as well as the MMU and bus protocol. I probably underestimated the amount of work necessary for this, given that we had a CPU provided and only a couple of chips working together. The worst part was the long simulation times just to reach the point where bugs occurred, and most of them were impossible to diagnose at the ChipScope level.

As basically all the groups from past years have said, this capstone is one of the most time-consuming things to do at CMU. For myself, balancing two graduate ECE classes on top of this one led to a lot of priority conflicts with getting stuff done. When I got to a point where I could devote full-time effort to the course, it really became a lot more fun as well as becoming more effective. So I would suggest that if you have an awesome project, try to make it the centerpiece of your semester rather than just another workload.

I don’t think we had any major group problems during the semester. All members contributed to their parts of the project, kept up communication, and put in hours when necessary, especially during the final week before Demo. We simply ran out of time to fully complete the massive complexity of the Apple IIGS. As a group, we probably should have pushed harder early in the semester to get some confidence and have concrete checkpoints. Still, we got most of the modules integrated and running in demonstration. I was really impressed how everything came together fairly well right at the end.

Robert Walzer

I spent the majority of my time working on the output functionality, sound and video. This was initially a slow process, as it took significantly longer than expected to get a proper interface to the board's onboard video hardware. Once I had this working, I was able to start on the actual video generation. While the GS's spotty documentation certainly slowed me down in this area, I eventually found that if you read the documentation repeatedly, it starts to make sense.

Once I finished the video generation hardware, I was able to move much quicker through the sound hardware. While this was initially even more unclear than video due to Ensoniq's deliberately non-existent documentation, the lessons I learned writing the video hardware were very helpful in this (i.e. read the GS hardware reference manual over and over again). Once I finished as much sound and video as I could, I spent the remainder of my time until the demo working on the emulated floppy interface.

Although I was disappointed that the majority of the components I wrote did not get used during our final demo, I was still impressed with how far we got given the complexity of the GS. While we planned our project from the beginning of the semester, I don't think any of us truly appreciated the complexity of the GS, so our planning did not reflect that complexity.

I think our group dynamics played a big role in us getting anything working by the final demo. While members of other groups were getting audibly annoyed with each other in the final weeks when things didn’t work out as planned, we all were able to keep pushing forward despite nothing really working on the board at that point.

Some Resources

Official IIGS Manuals: http://www.apple-iigs.info/doc/dociigs.htm

65C816 ISA: http://www.westerndesigncenter.com/wdc/datasheets/Programmanual.pdf

Kent’s Emulated GS: http://www.emaculation.com/doku.php/kegs

RESET information: http://www.macgui.com/usenet/?group=2&id=6463

2MG Info: http://apple2.org.za/gswv/a2zine/Docs/DiskImage_2MG_Info.txt

BPI PROM: http://www.spansion.com/Support/Datasheets/S29GL-P_00.pdf

PS/2: http://www.eecg.toronto.edu/~jayar/ece241_08F/AudioVideoCores/ps2/ps2.html