Game Boy Color

System Design

David Campbell, Jonathan Leung, Bailey Forrest

Fall 2014

Table of Contents

Forward
1 Design Overview
    1.1 What was built
    1.2 Credit
    1.3 Detailed Software Description
    1.4 System Block Diagram and Description
    1.5 Our Hardware Implementation
2 CPU
    2.1 Summary
    2.2 Additions & Omissions from Z80
    2.3 Pinout Diagram
    2.4 Testing
    2.5 Our modifications
3 Memory
    3.1 Summary
    3.2 Implementation
    3.3 Memory Busses
    3.4 IO Register Bus
    3.5 IO Bus Parser
    3.6 Memory Router
    3.7 Memory Banking
    3.8 Bus Interface to BRAM Interface
4 DMA
    4.1 OAM DMA
    4.2 General DMA
    4.3 Horizontal Blanking DMA
5 Cartridge Interface
    5.1 Cartridge Hardware Interface
    5.2 Cartridge Simulator
    5.3 Game Switching
6 Video Module
    5.1 Description
    5.2 Our Implementation
    5.3 Implementation process
7 Controller
8 Link Cable
    8.1 Summary
    8.2 Timing
    8.3 Diagram

8.4 Issues
9 Audio Interface
9.1 Description
9.2 AC97 Audio Codec
9.2 BRAM Lookup Tables
9.3 Square Waveform Generator
9.4 Random Waveform Generator
9.5 Sound Channel 1
9.6 Sound Channel 2
9.7 Sound Channel 3
9.8 Sound Channel 4
9.9 Top Level Sound Module
9.10 Performance Enhancements
9.11 Sound Testing
10 Miscellaneous Modules
10.1 Timer Module
10.2 Interrupt module
10.3 Clock Module
10.4 Infrared Communication
10.5 Reset Module
10.6 Undocumented Gameboy Color Registers
11 Utilities
11.1 Self Generated Scripts
11.2 BGB Boy Emulator
11.3 Research
11.4 Test ROMs
12 Tested ROMS
12.1 Game Functionality Score
12.2 Demo
13 Schedule/Management Decisions
13.1 The First Schedule
13.2 The Second Schedule
14 How We Built it
14.1 Approach
14.2 Design partitioning
14.3 Tools and design methodology
14.4 Testing and verification methodology
14.5 Status and future work
15 What We Learned
15.1 What we wish we had known at the beginning of the project
15.2 Particularly good/bad decisions you made
15.3 Words of wisdom for future generations
16 Personal Statements

16.1 Jonathan Leung
16.2 David Campbell
16.3 Bailey Forrest

Forward

We Bailey Forrest, David Campbell, and Jonathan Leung give the 18545 staff permission to freely use this report and our code in anyway they see fit.

Our repository is located at the following URL:
https://bitbucket.org/bcforres/18545

1 Design Overview

1.1 What was built

What we set out to build:
We set out to build a fully functional Game Boy Color video game console. Specifically, our goal was to be able to play the game Pokemon Crystal.

What we accomplished:
We completed a mostly complete implementation of the Game Boy Color console, which could successfully play most Game Boy Color and Game Boy games, with some games having minor bugs. Pokemon Crystal ran almost flawlessly.

1.2 Credit

F13 Gameboy Team (https://github.com/nightslide7/Gameboy)
From their repository we obtained the following components:

- CPU
- PPU Video converter
- DMG PPU (Initially from FPGABoy)
- DVI Module (originally from F10 Virtex Squared)
- AC97 Module (originally from F10 Virtex Squared)
- MCS generation script

Pandocs (http://problemkaputt.de/pandocs.htm)
This is the site that we got most of our implementation details from, as well as a lot of the details in this document.

1.3 Detailed Software Description

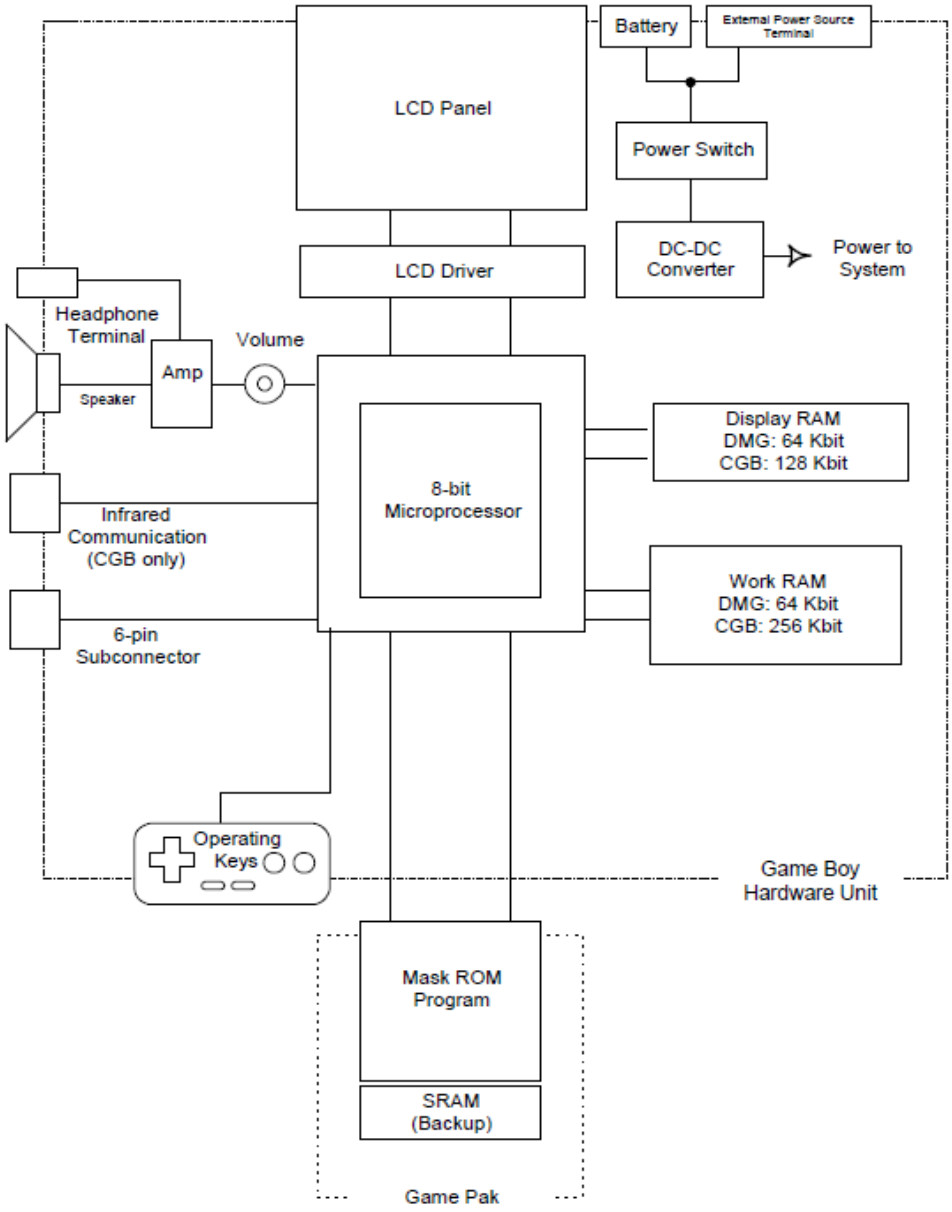
For our project, there really wasn't any software that needed to be implemented with the exception of some self written unit tests. Most of the software we ran on the system was obtained through external sources. Thus, the main body of this document will be used to outline the implementation of the various hardware components.

1.4 System Block Diagram and Description

The Game Boy Color's architecture is almost identical to the Original Game Boy's except for some memory size difference and clock differences, as well as the addition of several modules.

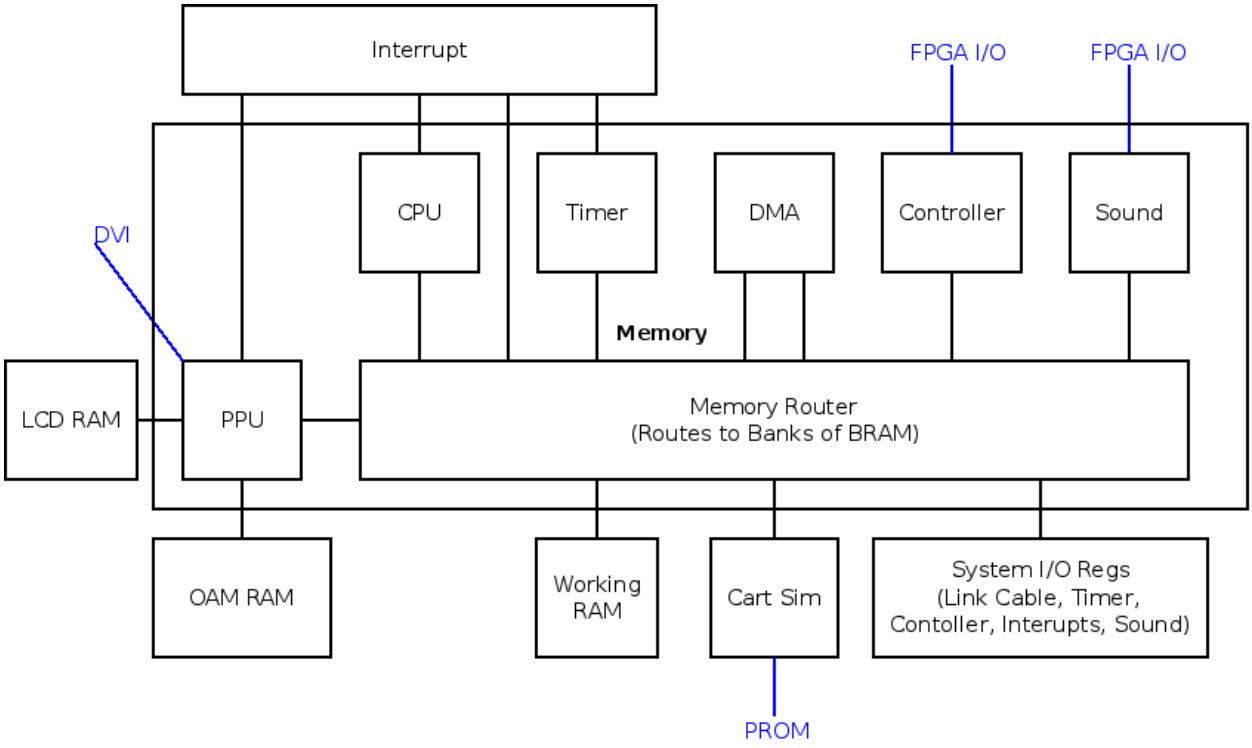
Below is the block diagram of the system taken from the Game Boy Programming Manual. DMG notes hardware specs of the Original Game Boy, while CGB notes the hardware specs of the Game Boy Color.

1.2 GAME BOY BLOCK DIAGRAM



1.5 Our Hardware Implementation

Below is the block diagram of the system we implemented:



Due to devices on the system being accessed through memory mapped IO, the system is all centered around the Memory Router and there are several different components that see data that goes through the memory router. First there are the master interfaces such as the CPU and DMA that will actually do the address generation to read and write from the slave devices. The other components are mostly comprised of BRAM or memory mapped IO Registers that service the data requests from the master interfaces. There is a simple bus protocol defined in the memory connections seen above that are defined later in the memory section. This allows for system integrity and consistency that will improve robustness of the overall design. Keep in mind that each of these components are much more detailed, and their descriptions can be seen in later sections.

One exception is the PPU, which has its own RAM modules self contained rather than going through the memory router. This is because during the rendering process, the PPU must have its own high speed access to these memory regions. When devices attempt to read or write from the video memory or PPU control registers, the memory router instead routes the address and data lines to the PPU itself, which handles memory requests.

2 CPU

2.1 Summary

The CPU is the Sharp LR35902 which is a modified Zilog Z80 chip. Nintendo added and removed a couple features from the Z80 to make the custom chip. The chip runs optionally at 8.4 Mhz for GBC games and at 4.2 Mhz for GB games.

Drawing from our experience taking 18-447 (Computer Architecture) last semester, we decided to use the CPU code from the F13 Gameboy project group as a starting point. We concluded that writing a CPU from scratch is too time consuming, and our goal is to get to play a game not write a CPU.

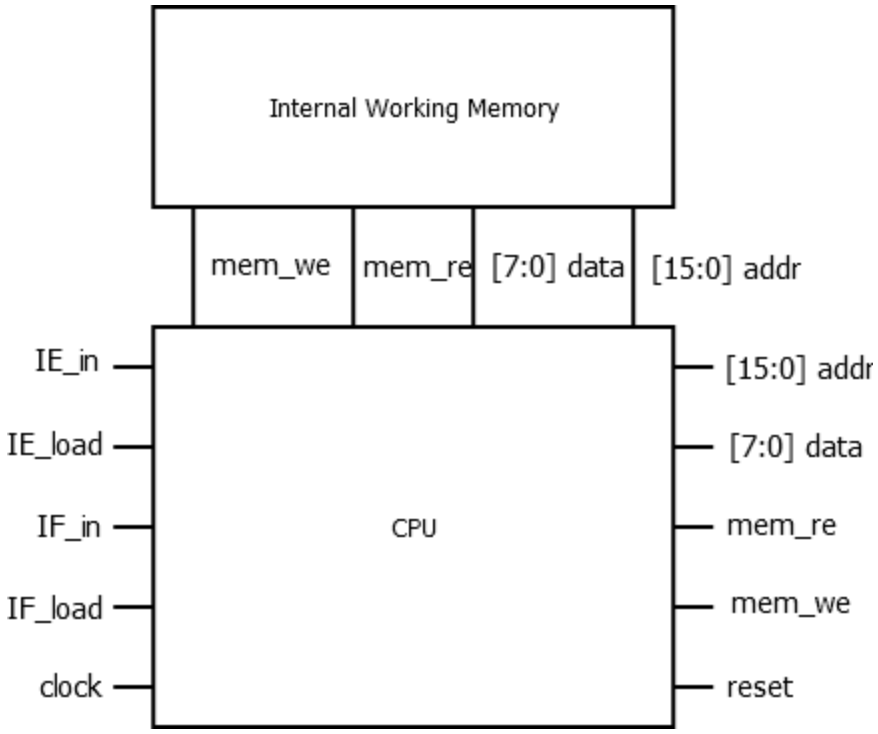
2.2 Additions & Omissions from Z80

The goal of the modifications is to help optimize load and store memory instructions, as everything is memory mapped, and the I/O instructions are useless now.

Additions:
~ 16 Extra instructions accessed through second stage of decode
Used for bit operations that were not in original Z80

Omissions:
IX, IY registers
~ 26 Z80 instructions

2.3 Pinout Diagram



There is not much information on the timing of the custom Z80 chip. The various Z80 docs and one waveform that claims to be measured from a real Game Boy have indicated that the CPU expects memory to have a 1 clock cycle delay. The F13 Gameboy team's CPU, however expects memory to be immediate access.

2.4 Testing

We did not start formally testing the CPU until we were done integrating the CPU and memory system. We wrote some small test programs in assembly and made sure basic functionality worked and timing was okay before moving on to more rigorous methods.

As we learned from F13 Game Boy team, there is a set of test ROMs called Blargg's test ROMs to test the functionality of Game Boy systems. The CPU was able to pass all instruction tests.

2.5 Our modifications

There were several modifications that we made to the F13 Game Boy team's CPU.

We changed I/O pins to be active low, as the top level signals are active low. No modification was needed to make the CPU operate correctly at the faster clock speed.

As explained earlier memory access should take 1 clock cycle, but we decided to keep immediate memory accesses. This is because we didn't want to modify the CPU too much and introduce bugs.

While testing instruction timing using one of the Blargg test ROMs, we found one instruction timing bug, but it doesn't seem to affect the system much.

Halting from a external halt signal was not implemented, so we had to add that to support GBC games. This is because the GBC DMA units halt the CPU while copying.

3 Memory

3.1 Summary

The Game Boy Color has a memory mapped IO system such that there are different memory locations for working ram, IO registers, cartridge program data, LCD ram and a special area of memory called the OAM (Object Attribute Memory). The components of the system are accessible from 2 different main entities in the system: the CPU and the DMA engine. The memory system enables communication and synchronization for each of these components through the specific memory mapped IO registers.

Below is the memory map of the GBC (credit: gameboy pandocs):

```
0000-3FFF   16KB ROM Bank 00     (in cartridge, fixed at bank 00)
4000-7FFF   16KB ROM Bank 01..NN (in cartridge, switchable bank number)
8000-9FFF   8KB Video RAM (VRAM) (switchable bank 0-1 in CGB Mode)
A000-BFFF   8KB External RAM     (in cartridge, switchable bank, if any)
C000-CFFF   4KB Work RAM Bank 0 (WRAM)
D000-DFFF   4KB Work RAM Bank 1 (WRAM)  (switchable bank 1-7 in CGB Mode)
E000-FDFF   Same as C000-DDFF (ECHO)    (typically not used)
FE00-FE9F   Sprite Attribute Table (OAM)
FEA0-FEFF   Not Usable
FF00-FF7F   I/O Ports
FF80-FFFE   High RAM (HRAM)
FFFF        Interrupt Enable Register
```

Below is a description of the different components of the memory system:

Program Data - (cartridge) - this is composed of 32 kB of memory comprised of the address space from 0x0000 - 0x7FFF. The first 16kB are always mapped to the 0th bank of the cartridge, and the second 16kB are multiplexed between the rest of the cartridge banks.

Video RAM - this is composed of a total of 16 kB each separated into 2 different memory banks. Both memory banks share the same address space of 0x8000-0x9FFF. Which bank is mapped to the address space is determined by the VBK register at address location 0xFF4F. Within the register there is a single bit that indicates which bank is being used.

Expansion RAM - this is composed of 8kB of external RAM on the cartridge that can be accessed through the cartridge interface. This feature is optional and it depends on the game being played whether it is used or not. It contains the address space defined by 0xA000 - 0xBFFF.

Work RAM - in the GBC there is a total of 32 kB of working RAM for the CPU to use, however it is divided into 8 different memory banks of 4kB each. The first memory bank is always accessible through the address range of 0xC000 - 0xCFFF. The next seven banks share the address space of 0xD000 - 0xDFFF, and they are switched via the SVBK register. A value of 0 or 1 in the register specifies bank 1, and then bank 2 is specified by 2 in the register, bank3 is specified by 3 in the register, and so on.

OAM - this small memory space is composed of 160 bytes from the address space of 0xFE00 - 0xFE9F. This memory stores information about the sprites which are displayed.

IO Registers - this is the memory space where all the io registers can be accessed and they are utilized throughout different components in the system. Its address range is composed of 0xFF00 - 0xFF7F and 0xFFFF.

Additional Working Ram - the additional working ram is a very small segment of memory in the range of 0xFF80 - 0xFFFE. This segment of memory is only accessible to the CPU.

Finally, take a moment to consider the timing associated with memory. Specified by the manual, memory reads should return the data 1 cycle after the requests, and memory writes to memory should be executed in one cycle. As discussed above, our implementation slightly differs, where memory accesses return the same cycle.

3.2 Implementation

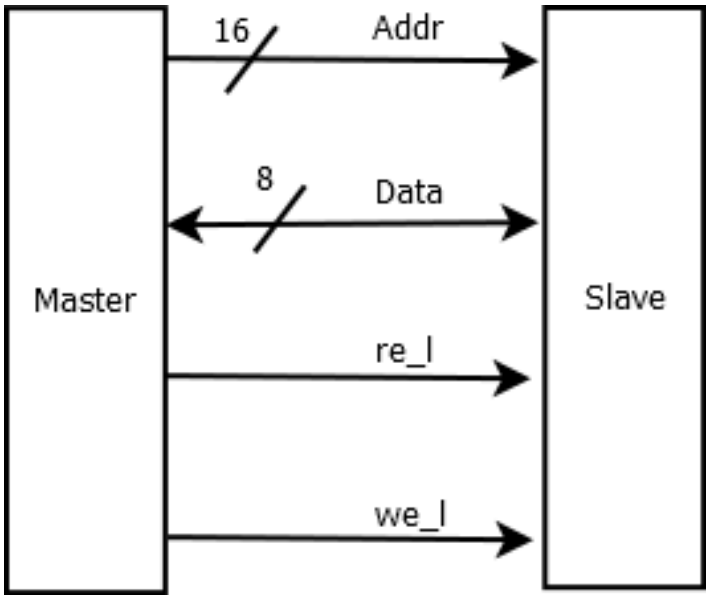
Transferring the memory system to the Virtex FPGA, by design is relatively simple. Since there is on board BRAM, there weren't really be any difficulties in getting a simple memory system together. As it turns out, each BRAM block has 36 kB of memory, which is larger than any sub-memory space. Therefore, this made the implementation pretty easy because each memory space can be written to a specific BRAM. Also, on the Virtex 5 there is a proliferous amount of BRAM blocks for us to use (compared to how many we need) so being "wasteful" of memory space is not a big issue. For example, the OAM only has 160 bytes of memory, but since there is a large amount of BRAM, to keep the design simple it can have its own BRAM block. However, since there are different sections of memory in these BRAM blocks a memory router is needed to route the memory requests to the right location and return the data back to the correct entity (CPU and DMA).

Following is a the design of each memory component in the system and why they were chosen to be used.

3.3 Memory Busses

For all memory components in the system, the protocol defined by the GBC CPU was used, even though there were some discrepancies from other components in the system (such as BRAM). A memory bus is composed of a 16 bit address, an 8 bit data bus, an active low read signal, and an active low write signal. The interface is pretty simple. A write operation will push the write signal to 0 with the data on the data bus, and the address it wants to write to on the address bus. A memory read operation will simply put the address on the address bus, and then 0 on the read enable signal. The bus operates by expecting the read data to be available on the same cycle of the read request. Note that the data bus is “logically” tri-state (though on the FPGA it will most likely be synthesized to another hardware structure).

The figure below shows the components of a memory bus in the system:



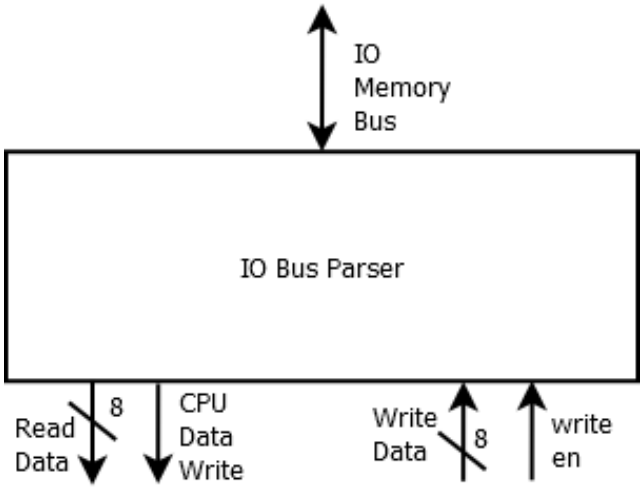
3.4 IO Register Bus

In particular, an IO register bus was used because it made reading and writing from the IO registers much easier. Since all IO registers share the same address space, it was determined that they all could share access to the data going in and out, and it was up the the logic surround a specific register whether to read or write from the bus or not.

3.5 IO Bus Parser

Since we decided to implement the IO registers via a shared memory bus, for convenience of other parts in the system, an IO bus parser was made. This component is very simple because it monitors the IO Register Bus and if the address is the one that pertains to the register, the data will be returned on the bus, or it will be written to the register. Therefore, this allows other entities in the system to easily duplicate many IO registers, without having to worry about the interface with memory. To make ease of access, a separate user interface was given that simply forwards the register data, or allows the user to write to the register with a simple active high write enable signal. Furthermore, a signal was given to the user so that it knows when the IO bus is writing to the register.

The figure below shows the two interfaces for the IO Bus Parser:

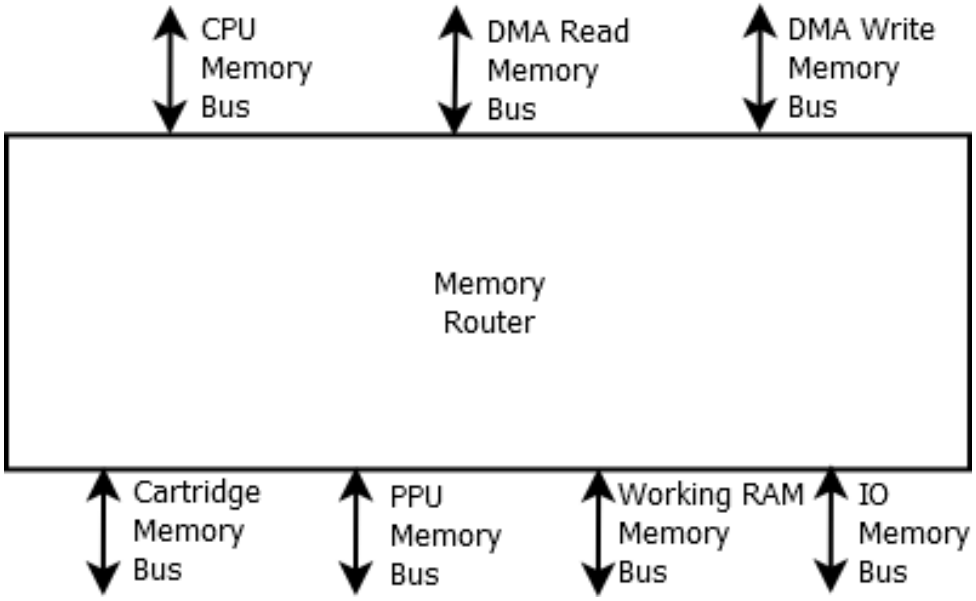


3.6 Memory Router

In order to properly make sure that all the addresses get to the right locations, a memory router was implemented to connect each memory component with each memory accessing entity in the system. The memory router has 3 master interfaces: the CPU, the DMA Writer, and the DMA Reader. The memory router then connects them to 4 other components of memory: the cartridge, the working RAM, the PPU, and the IO Register Bus.

The use of the Memory Router allowed for system simplicity and robustness. It always guaranteed that the data would be accessible only to its proper destination, and it also guaranteed that only the requester of the data would have access to that word, and it forced only one master interface to have access to memory locations at a time. Thus, the isolation of the different system resources allowed for easier debugging as well as assurance of correctness.

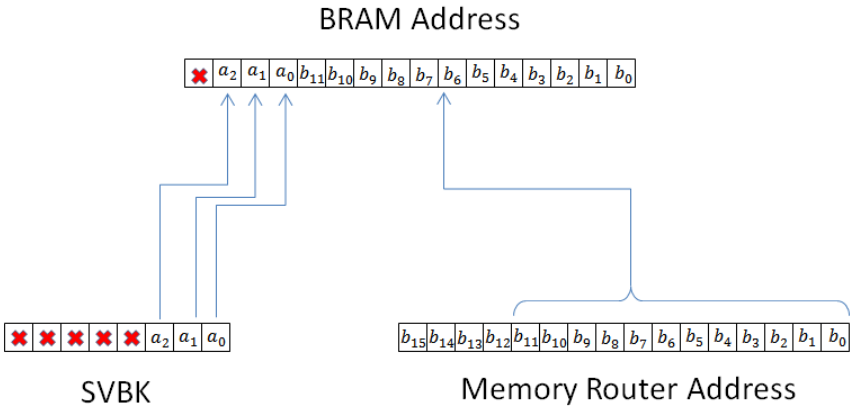
The figure below shows each of the ports in the memory router. Note that each port contains a memory bus as defined above in the memory bus section.



3.7 Memory Banking

Since there are two segments of memory that implement memory banking, there needs to be a special translation to make sure that the address request gets to the right bank. Furthermore, since the total memory size for all components is less than 1 BRAM block, this allows us to just chunk up the banks across different segments of the BRAM block. An address translation is therefore utilized such that it takes the address coming from the memory router and then removes the high bits, replacing them with the bank specified by the VBK register and SVBK register for LCD RAM and working RAM, respectively.

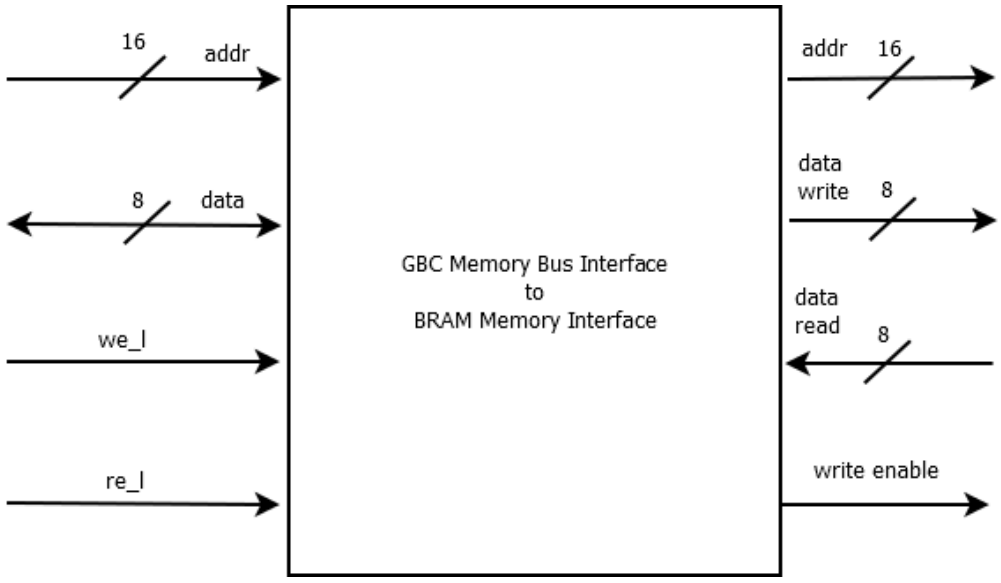
The figure below shows that address translation to the BRAM block such that it will bank the memory by offsetting it. In particular, this is the address translation for the working RAM memory:



3.8 Bus Interface to BRAM Interface

Since reading and writing from BRAM follows a different interface than the memory bus architecture described before, a simple translation component is needed across the interfaces. In particular, BRAM read accesses return the data a clock cycle later, and we expect them in the same clock cycle. The solution was to double clock the BRAM so that it would return the data in time, and the BRAM interface converter handled this situation. It was also responsible for generating the write enable signal on a write transaction, as well as acting as an arbiter for the tri-state bus for the read and write data.

The figure below shows the interface that converts the system memory interface to the BRAM memory interface:



4 DMA

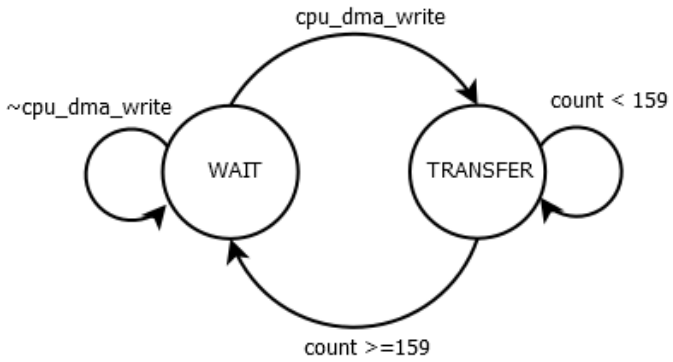
There are actually 3 different types of DMA in the Game Boy Color: OAM DMA, General DMA (GDMA) and Horizontal Blanking DMA (HDMA). Within our implementation, all of these different controllers were implemented in the same module, and they all shared the same DMA reader and DMA write access ports to the memory router because the architecture of the system required only 1 DMA controller to be active at a time. The DMA units are controlled by memory mapped IO registers, where the destinations and source addresses of the memory copying are specified. The DMA within the Game Boy Color is actually quite complicated, and it is essential for the system to be complete. Many of the games use the DMA to do quick data movement into VRAM. In terms of our design, getting a fully functional DMA was largest step in our system integration and it produced the largest results. We would not have gotten a functional Gameboy Color without it.

4.1 OAM DMA

This DMA is specific for moving data quickly into the OAM RAM section of memory within the PPU. It enables the CPU to quickly change the sprites that are active on the screen at a given point. The OAM DMA is allowed to work on both the DMG and GBC architectures. Since the DMG does not support CPU halting (from the DMA) the CPU must copy code into internal high RAM, where it spins in a loop until the DMA is finished. Consequently, this makes the timing of the OAM very important because it must be synchronized with how long the CPU executes this code. An OAM DMA transaction goes as follows:

1. The CPU writes the upper 8 bits of the source address to the “DMA” register. (the lower 8 bits of the address must be 16 byte word aligned making them always 0).
2. This write to the “DMA” register activates a waiting FSM that will begin copying the data from the source address to the destination address starting at 0xFE00.
3. The CPU executes the copied code within the high RAM, while the DMA unit increments a count, and uses this count as an offset to the read address and write address to copy each byte over.
4. Exactly **160** cycles later, the DMA is finished and the CPU exits the loop.

The state machine for the OAM DMA is as follows:



- **WAIT** - the DMA controller waits for a write to the “DMA” register to go to the TRANSFER state
- **TRANSFER** - the count is incremented and the write and read enable signals are activated. When the count is greater than 159, the DMA goes back to the WAIT state

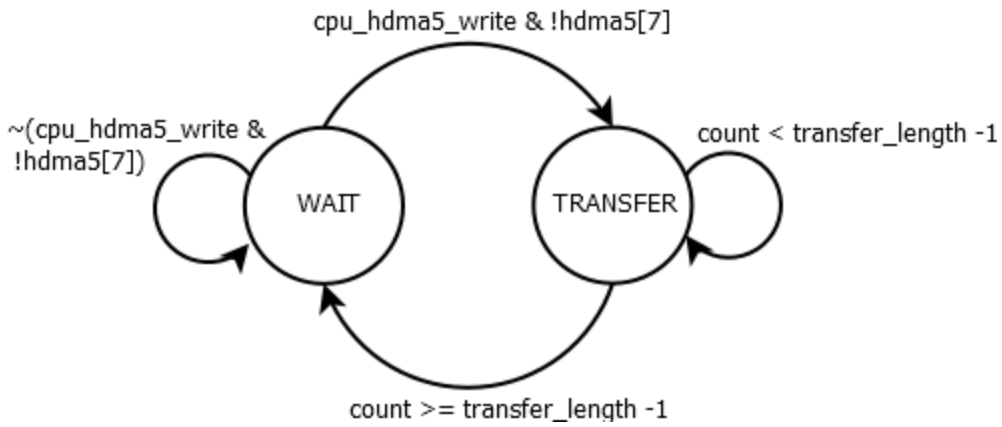
4.2 General DMA

The general DMA allows the transfer of memory from either cartridge, expansion RAM, or working RAM into VRAM. It can copy from 16 to 2048 bytes of data at a time, however *execution of the CPU must be halted.* The control registers are as follows:

- **HDMA1** - source address, higher byte
- **HDMA2** - source address, lower bytes. Lower 4 bits are ignored
- **HDMA3 -** destination address, higher byte. Upper 3 bits are ignored since value must be 0x8xxx to 0x9xxx
- **HDMA4** - destination address, lower byte. Lower 4 bits are ignored.
- **HDMA5** - indicates whether a HDMA or GDMA is active on read. On write, indicates to start a transaction or cancel a transaction, and also it specifies the amount of bytes to transfer.

For more information on the HDMA control registers, please visit the pandocs.

The state machine for the General DMA is as follows.

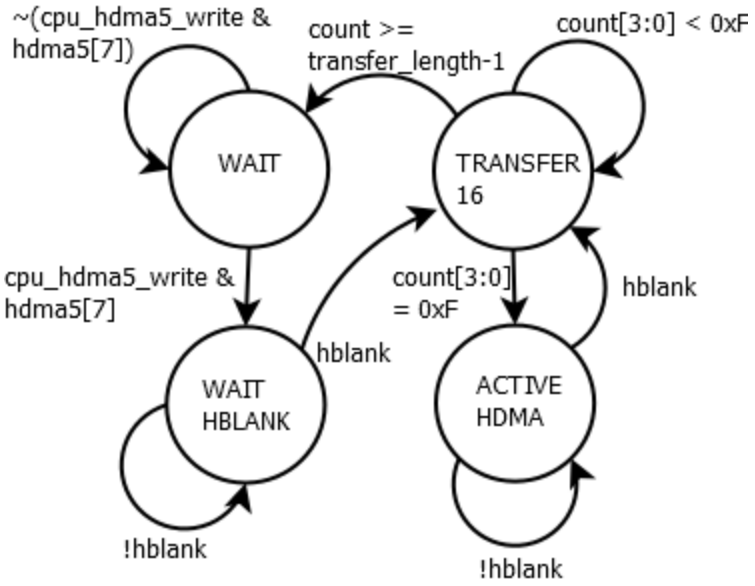


- **WAIT** - the controller waits for a specification activation by a write to the HDMA5 register
- **TRANSFER** - the controller counts the amount of bytes sent. The read and write enable signals are activated and the CPU is halted. When the count is greater than the transfer length from the HDMA5 register, the CPU is resumed and the controller goes back to the wait state.

4.3 Horizontal Blanking DMA

The HDMA is very similar to the GDMA, however it is only allowed to transfer 16 byte spurts of data during the hblank period of the PPU. Consequently, this made the HDMA much more complicated than the GDMA. They both share the same control registers as seen above, but which controller to use is determined by bit 7 in the HDMA5 register. During an HDMA, the CPU is halted, but then execution of the program is continued immediately after the 16 bytes transfer of the HDMA transfer. To halt the CPU, the DMA controller simply outputs an active high halt_cpu signal to indicate that the CPU should suspend program execution.

The state machine for the HDMA is as follows:



- **WAIT** - wait for the CPU to activate an HDMA in the HDMA5 register
- **WAIT_HBLANK** - wait for the next H-blank period to start the DMA transaction. The CPU is NOT halted, but the DMA controller indicates that a DMA is active to the cpu via a read transaction from the HDMA5 register. When a hblank occurs, go to TRANSFER16 to start the transfer, unless the CPU indicates to cancel the transaction.
- **TRANSFER16 -** move 16 bytes of data by offsetting the base and destination addresses from a count. The read and write enable signals are activated. When 16 bytes have been transfered go to the ACTIVE_HDMA state, unless all bytes have been transferred where then it would go to the WAIT state, indicating the transfer is complete. The CPU is halted during this state.
- **ACTIVE_HDMA** - not all the bytes of data have been transferred yet, but the HDMA is still active. Go to TRANSFER16 on the next H-blank period, unless the CPU indicates to cancel the DMA transaction. The CPU is not halted in this state.

5 Cartridge Interface

The Game Boy cartridge contains the game ROM and is mapped to the lowest 32kB of the address space. The lowest 16kB of memory is always mapped to the same location in the cartridge, and the rest of the cartridge's memory is accessed with the 2nd 16kB of the address space using a bank switcher build into the cartridge (called the Memory Bank Controller).

5.1 Cartridge Hardware Interface

Here is the pin layout of the Game Boy cartridge:

```
  Pin   Name    Expl.
   1     VDD     Power Supply+5V DC
   2     PHI     System Clock
   3     /WR     Write
   4     /RD     Read
   5     /CS     Chip Select
   6-21  A0-A15  Address Lines
   22-29 D0-D7   Data Lines
   30    /RES    Reset signal
   31    VIN     External Sound
 Input
   32    GND     Ground
```

Cartridge Memory Mapping:

```
  0000-3FFF   16KB ROM Bank 00     (in cartridge, fixed at bank 00)
  4000-7FFF   16KB ROM Bank 01..NN (in cartridge, switchable bank number)
  A000-BFFF   8KB External RAM     (in cartridge, switchable bank, if any)
```

Source: http://problemkaputt.de/pandocs.htm#externalconnectors

For our cartridge system, we had originally planned to use a cartridge reader for an existing Game Boy Color and connect it to the Virtex 5 using the GPIO pins. Unfortunately we were not able to accomplish this due to two problems. First the FPGA operates on 3.5V and the cartridge at 5V, the level shifters we obtained to fix this problem would only work up to around 50 KHz, when we needed 4MHz. We could've ordered better level-shifters but that didn't matter as we realized that the GPIO pins on our board were also not fast enough to operate at 4 MHz. At 4 MHz the digital signals directly coming out of the GPIO pins looked like triangle

waves instead of square waves. Consequently, we decided instead to implement the cartridge in hardware on the FPGA and implement the ROM data in flash memory.

5.2 Cartridge Simulator

Since we cannot read from a real cartridge we need to simulate a cartridge on our board. There are various types of cartridges, and we decided to use the Pokémon Crystal type, which is an MBC3 controller + battery backed RAM + timer. There are actually 5 different cartridge types for the GBC/DMG systems, but we decided to implement MBC3 because this is the one that Pokemon crystal uses. Fortunately, however, this type of cartridge was a mostly superset of the others so most games worked on our cartridge hardware anyways.

The MBC3 cartridge has a built in controller that controls which RAM bank is used, which ROM bank is being read from, and it also had a timer interface. Though most games have a backup battery to save data in the cartridge RAM, our system could not save games when powering down the FPGA, due to the BRAM contents being lost, but it could save between soft resets.

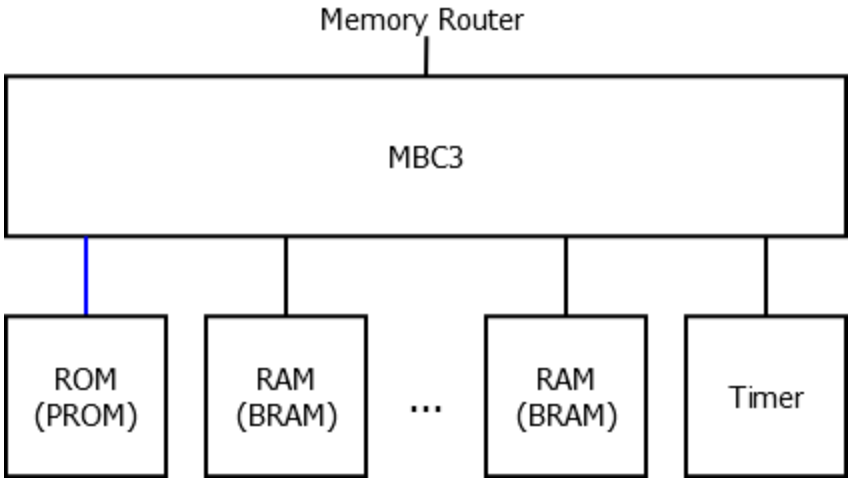
The cartridge controller is set as follows:

When **writing** to any of the following address ranges, the functionality of the cartridge will be as follows:

- 0x0000 - 0x1FFF - enables access to the expansion RAM and the cartridge timer
- 0x2000 - 0x3FFF - writes the ROM bank number to a register
- 0x4000 - 0x5FFF - determines which RAM or timer register to be read from
    - 0-3 specifies the RAM bank number.
    - 8 - read from seconds register
    - 9 - read from minutes register
    - A - read from hours register
    - B - read from days register
    - C - halt timer, includes higher bit of number of days
- 0x6000 - 0x7FFF - latch the clock data into the registers to be read from by the CPU

When **reading** from the cartridge, the functionality is as follows.

- 0x0000-0x3FFF - ROM bank 0
- 0x4000-0x7FFF - ROM bank specified by the rom bank register
- 0xA000-0xBFFF - read from RAM bank if bank is < 4, else read from the corresponding timer register.



5.3 Game Switching

To be able to play more than one game per flash program, we implemented game switching. The cartridge sim module takes a selection input that comes from the FPGA DIP switches and based on the number adds an offset to the base_catridge_address. Also the selection determines which BRAM the games will save to, so that saves will not be corrupted in between game switching. For now though we only protect the saves of Pokémon games. This enables us to load multiple games into ROM and have the ability to save games.

6 Video Module

5.1 Description

The Pixel Processing Unit (PPU) of the Game Boy Color works by keeping backgrounds stored in memory, and rendering sprites on top of or underneath the backgrounds. It uses many control registers to perform operations such as controlling the offset of the background that should be displayed on the screen, and how sprites should be displayed (e.g. flipped).

Below are the relevant portions of the memory map for the PPU:

```
8000-9FFF   8KB Video RAM (VRAM) (switchable bank 0-1 in CGB Mode)
FE00-FE9F   Sprite Attribute Table (OAM)
FF00-FF7F   I/O Ports
```

Video Ram Contents:

- 8000h-97FFh - (2 banks for GBC) (Tile Data) - Bitmaps for 8x8 tiles (up to 192). Each pixel is encoded in 2 bits, so a tile can have up to 4 distinct colors, chosen by the current color palette.
- 9800h-9BFFh and 9C00h-9FFFh (Tile backgrounds). Contain 8 bit indexes into the tile data. Can be used to display “normal” scrolling backgrounds, or “window” backgrounds, which don’t scroll.
    - The same address bytes in the second bank of the video RAM contains information about the color palette, which bank the tile is in, and sprite flipping.

Sprite Attribute Table (OAM) - Contains 40 4 byte entries:

- Byte 0 - Y position on screen minus 16
- Byte 1 - X position on screen minus 8
- Byte 2 - Tile/Pattern number - offset into the range 8000h-8FFFh
- Byte 3 - Attributes - e.g. overlap priority, flipping, palette number

The CPU can only access the VRAM and OAM during the V-Blank and H-Blank periods of the LCD.

The video module is controlled with a set of memory mapped control registers.
The registers control many aspects of the display, such as the the offsets of the rendered backgrounds, color palettes.

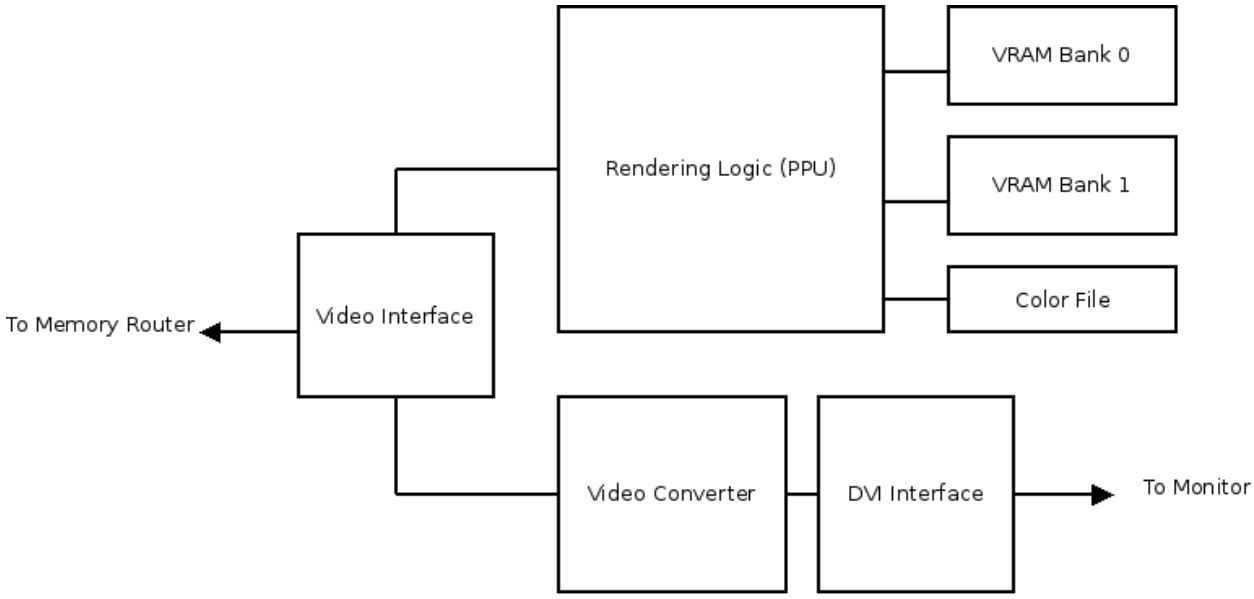
5.2 Our Implementation

We decided to adopt our video system from the F13 Game Boy team. Their video system uses team dragonforce's DVI module and FPGAboy's Gameboy PPU (https://github.com/trun/fpgaboy). There are some significant changes between the Gameboy and Gameboy color PPU however, so these changes need to be implemented.

Here are the main differences in the Game Boy and GBC PPU:

- LCD Color palette and associated control registers
- Secondary Video RAM memory bank
- Additional background map attributes (on second VRAM bank)
- Minor differences in behavior in control register settings

Below is a high level diagram of how the video modules are laid out:



Note that the video memory regions and control registers are stored in the PPU, so they can be read with a higher clock rate during the rendering process.

The PPU uses scanline based rendering, where it renders one line of pixels at a time. In between rendering each line, there is a horizontal blank (H-Blank), which is a short period of time where the PPU is not using the video memory. After rendering all of the lines on the screen, there is a vertical blank (V-Blank) which lasts much longer, and also during which the video memory can be accessed.

Below is a diagram of this process (credit: pandocs):

```
Mode 0: The LCD controller is in the H-Blank period and
        the CPU can access both the display RAM (8000h-9FFFh)
```

```
          and OAM (FE00h-FE9Fh)

  Mode 1: The LCD contoller is in the V-Blank period (or the
          display is disabled) and the CPU can access both the
          display RAM (8000h-9FFFh) and OAM (FE00h-FE9Fh)

  Mode 2: The LCD controller is reading from OAM memory.
          The CPU <cannot> access OAM memory (FE00h-FE9Fh)
          during this period.

  Mode 3: The LCD controller is reading from both OAM and VRAM,
          The CPU <cannot> access OAM and VRAM during this period.
          CGB Mode: Cannot access Palette Data (FF69,FF6B) either.

The following are typical when the display is enabled:
  Mode 2  2_____2_____2_____2_____2_____2___________________2____
  Mode 3  _33____33____33____33____33____33__________________3___
  Mode 0  ___000___000___000___000___000___000________________000
  Mode 1  ____________________________________11111111111111_____
```

Our implementation of the PPU runs at 33 MHz, 8x the clock speed for regular speed mode. There is a giant FSM that handles rendering onto the screen. Here is the general control flow of rendering one scanline on the display:

1. Wait for Video Lock mode, where the PPU has exclusive access to the video memory regions
2. For each of the 32 background or window tiles on the current scanline (based on the SCX, SCY - scrolling registers), check if any parts of these tiles are on the current scanline. If so, put the tile data into the scanline, along with the color palette data.
3. For each sprite of the 40 sprites, check if the sprite fits on the current scanline, based on the sprite's position stored in the OAM. For each of the sprite's 2 bit pixels which are not 00 (which denotes transparent), store the sprite's index, and its color palette in the current scanline, along with marking that it is a sprite.
4. Render the scanline on the display. The 16 bit pixels are sent out one by one to the video converter module. The color is determined by whether or not the pixel was part of a sprite or a background, and choosing using the color index. In determining the color, the color index is used to lookup into the color file.
5. The video converter module uses two frame buffers which it switches between while rendering frames. The video modules renders the currently selected frame buffer to the center of the screen of the DVI output.

The parts marked in red denote the changes that had to be made to support color features. In order to support color, below are the changes that were required:

- Addition of a color file, and its associated memory mapped interface
- 16 bit rather than 2 bit frame buffer so that the color information could be stored. To support these larger frame buffers, the frame buffers were changed from verilog arrays to BRAM.
- Extra scanline bits to store whether or not a pixel is a background pixel or a sprite pixel.
- Extra scanline bits to store the color file index of the pixel
- Updating and reading the newly added scanlines at the right places in the rendering FSM to properly display color.
- Passing a 16 bit color value rather than a 2 bit color code to the converter module
- Conversion of the 16 bit color value to a 24 bit color value in the converter module

In addition to the changes necessary to render color on the PPU, there were many bugs found in the existing PPU.

Below are some of the major game breaking bugs:

- The condition of whether to render the window or the background was not correct (based on the WX, WY registers)
- Tiles from the second tile map 8800-97FF, where the tile index is supposed to be treated as a signed number, was not handled correctly.
- Backgrounds did not show up correctly behind sprites which were between two background tiles.

Overall, the PPU module was very difficult to work with, and in retrospect we wish we had written it from scratch. The in code documentation is very poor, and there were many bugs.

5.3 Implementation process

Because of complicated rendering FSM, it was difficult to unit test parts of the PPU, because everything needed to be connected for it to work. To test the PPU during development, we took memory dumps from the BGB emulator and placed them in the video module's BRAM. That way, while adding features to the PPU, it was automatically regression tested. This strategy overall worked well, but each testing iteration was slow because it required synthesis.

7 Controller

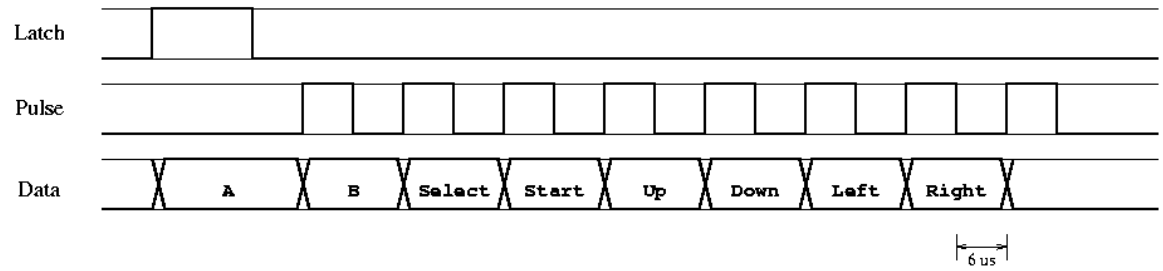
We decided that we would like to use an NES controller for our Game Boy because it has the same buttons. Below is the pinout for the controller connector:

```
          +----> Power  (white)
          |
5 +---------+  7
  | x  x  o   \
  | o  o  o  o |
4 +------------+ 1
    |  |  |  |
    |  |  |  +-> Ground (brown)
    |  |  +----> Pulse  (red)
    |  +-------> Latch  (orange)
    +----------> Data   (yellow)
```

The controller is used with the following protocol:

1. Every 60hz, a latch signal is sent.
2. The state of each button is latched
3. A pulse signal is sent 8 times, and on each cycle of the pulse signal, each of the 8 buttons are sampled

Diagram of this process:



We have simple FSM that implements this protocol. The Virtex 5 GPIO pins are used to communicate with the controller.

The register FF00 - P1/JOYP - Joypad (R/W) contains the controller data. The CPU needs to write to the register to select whether the buttons or directional pad are pressed. We implement a bit vector register that always maintains the most up to date state of each button, and the 0xFF00 address will map to some control logic connected to the bit vector.

8 Link Cable

8.1 Summary

Serial link cable allows communication between Gameboy systems and also select Nintendo Consoles. This is a very simple protocol, there aren't any start and stop bits, so it is import to have the timing correct to be able to communicate properly. Therefore we concluded that talking directly to a Game Boy would be impossible with our imprecise clock and decided to instead implement link cable between two FPGA boards.

FF01 - Control Register
Bit 7 - Transfer Start Flag (0=No Transfer, 1=Start)
Bit 1 - Clock Speed (0=Normal, 1=Fast)
Bit 0 - Shift Clock (0=External Clock, 1=Internal Clock)

8.2 Timing

The clock signal specifies the rate at which the eight data bits are transferred. When the Game Boy is communicating with another system then either one must supply internal clock, and the other one must use external clock.

Normal speed is the input clock divided by 2^9, fast speed is input clock divided by 2^4 hence:

8192Hz - 1KB/s - Bit 1 cleared, Normal

262144Hz - 32KB/s - Bit 1 set, Normal

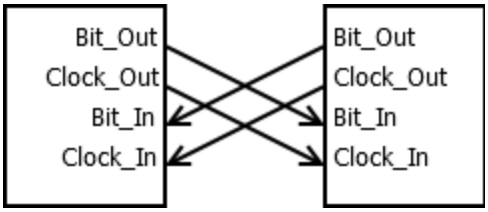
16384Hz - 2KB/s - Bit 1 cleared, Double Speed Mode

524288Hz - 64KB/s - Bit 1 set, Double Speed Mode

Transmitting and receiving serial data is done simultaneously. A Byte is shifted in as a byte is shifted out, rate is determined by the internal or external clock. Transfer is initiated by setting the Transfer Start Flag. This bit is automatically set to 0 at the end of Transfer.

8.3 Diagram

Our implementation uses 4 wires between two boards: bit_out, bit_in, clock_out, and clock_in.



8.4 Issues

We were able to get Tetris two player mode to start and keep playing, but they were not synced. Pokémon trading/battle can initiate, but the slave board will get out of sync and receive garbage data. We think these problems are due to timing issues.

9 Audio Interface

9.1 Description

The Game Boy Color's audio system has a stereo output with four different sound channels. Below is a description of the four channels:

1. Quadrangular wave patterns with a frequency sweep and envelope functions.
2. Quadrangular wave patterns with envelope functions.
3. Voluntary wave patterns from wave RAM.
4. White noise with an envelope function.

The sweep functionality on the 1st channel steadily increases or decreases the frequency of the sound depending on the settings in the control registers. The envelope functionality sweeps the volume of the output.

The Quadrangular wave patterns are square waves with variable duty cycles. The wave pattern contains a user specified wave pattern located in memory. The white noise function randomly switches between a high and low at a fixed frequency.

9.2 AC97 Audio Codec

For the actual sound output, we used team Dragonforce's AC97 module. We decided to create the system which gives signals to the AC97 system from scratch. To implement this, we had a separate waveform generator for each of the 4 channels. Each were independently controlled by its own control registers. For the actual signal sent to the AC97 module, the four channels were mixed together. This AC97 module made it fairly simple to generate some waveforms. Using the strobe signal, we simply fed the codec a 20 bit signed value to generate the sound.

9.2 BRAM Lookup Tables

To implement sound, the CPU can specify up to 2^11 different tones. However, given by the pandocs there was a specific formula that was needed to convert the binary value specified by the CPU to the actual frequency that needs to be played. For example, for sound channel 1 the equation was

$$f = 131072/(2048-x)$$

where x is the 11 bit specification. Since there was division involved here, and division is somewhat complicated to deal with in hardware, we decided to use a lookup table in BRAM of

2^11 values todo this computation. Furthermore, since we needed to handle frequencies using a series of counts based of a given clock frequencies, we added one more computation to this to determine the amount of clock cycles for the period of a waveform of the specified frequency. Thus, the clock cycles were computed as

$$clocks\_in\_period = clock\_rate/f$$

making the total computation as

$$clocks\_in\_period = clock\_rate*(2048-x)/131072$$

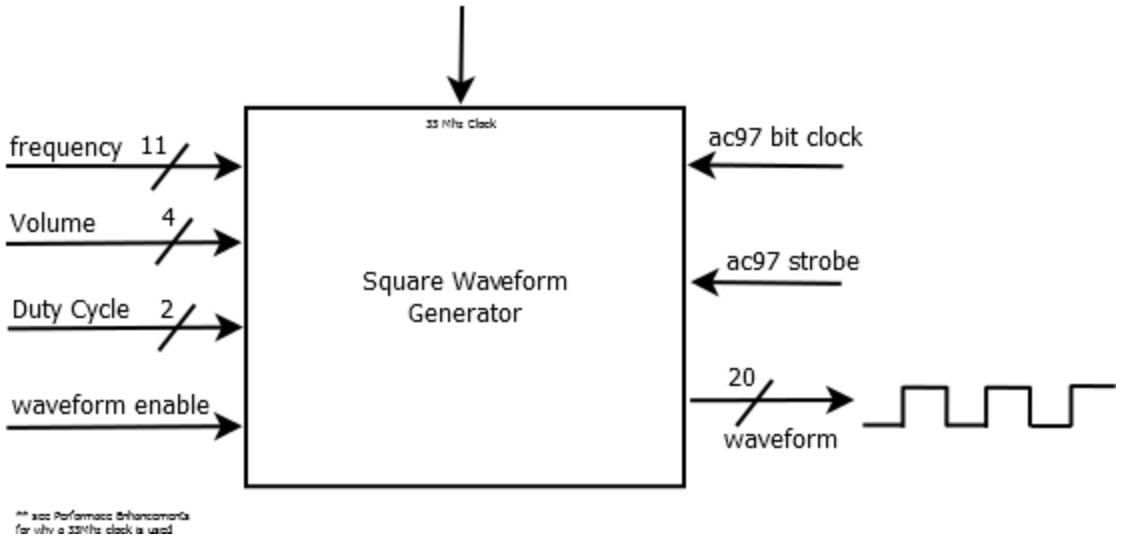
Consequently what resulted from the BRAM lookup table was an input of a 11 bit frequency specification from the CPU to a 32 bit unsigned values of the clocks in a period of the waveform to generate that frequency. Again, though this calculation could have been done in hardware, we felt that this was the best option due to limited time (to figure out the divider logic) and since we had an abundant amount of BRAM readily for use.

To generate the BRAM .coe files, a python script was used to simply generate the lookup tables using the equations above.

9.3 Square Waveform Generator

The first two sound channels used a square waveform, so it seemed necessary to build a submodule that generated a square wave given a few specifications. Going into the module were the following specifications:

1. *Frequency* - 11 bit value specified by the CPU (pre-lookup table).
2. *Volume* - four bit value to specify how loud the waveform should be.
3. *Duty Cycle* - two bit value where b00 is 12.5%, b01 is 25%. b10 is 50% and 11 is 75%.
4. *Enable* - allow waveform output (active high)



The output of the waveform was a *20 bit signed integer* since this is the sample type that the AC97 Codec uses. Thus, the idea for the Square Wave Generator was to create an easy translation between the CPU specification and the AC97 codec interface. The interface specifications could be easily changed to make it easier to handle frequency and volume enveloping.

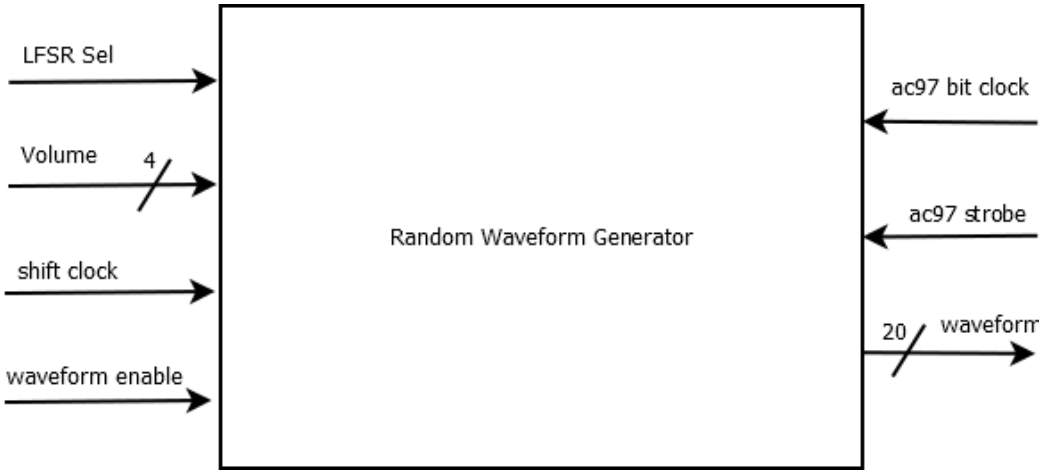
Within the square waveform generator, a BRAM lookup table was used to convert to find the amount of clocks in a period of the square waveform. From that value, simple bit shifting was used to find the amount of clock cycles that the waveform would be high to produce the proper duty cycle. Given the volume specification, another simple translation table was used to map the 4 bit unsigned volume specification to a 20 bit signed “sample”, meaning the value the would be sampled into the codec on the strobe signal. Note however that this translation was a range from 0 to 2^19-1 because we simply negated the positive sample on the low part of the duty cycle. Finally using this computed data, a counter was simply used to make the squarewave. If the counter was less than the amount of clocks high for the duty cycle, the positive sample was fed into the codec. If the counter was greater than the amount of clocks high, but less than the clocks in the period, the negative value was fed into the codec, thus generating a square waveform.

9.4 Random Waveform Generator

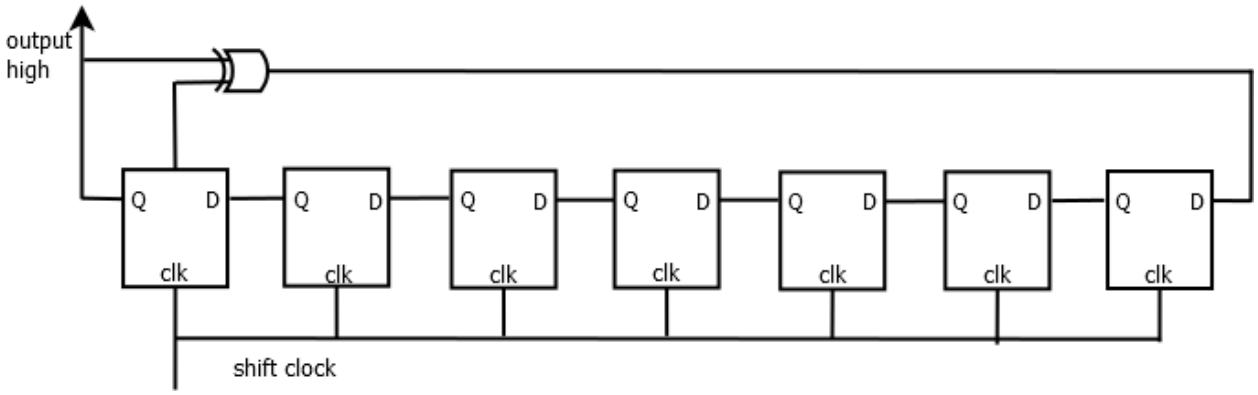
The fourth sound channel needed to produce white noise, so a random waveform generator was created. The specifications for the Random Waveform Generator were:

1. *LFSR sel* - a signal to switch between using a 15 bit LFSR (Linear Feedback Shift Register) or a 7 bit Linear Feedback Shift register
2. *Shift Clock* - the rate at which the LFSR operates

3. *Volume* - 4 bit unsigned value that indicates the volume of the output.
4. *enable* - indicates to output the waveform



The random waveform generator worked very similar to the square wave generator. The volume specified was converted to a 20 bit signed value from 0 to 2^19-1. When the bit width specification was 15, if the lowest bit in the LFSR was high, the positive sample was output, else the negative sample was output. When the bit width specification was 7, the same logic was used, however, it simply used the lowed bit of the 7 bit LFSR to determine if the sample was high or low. The 7 bit LFSR is shown below. The 15 bit LFSR is the same except there is just more registers.



9.5 Sound Channel 1

Sound channel 1 used the square waveform generator module to simple generate the waveform based off the CPU specification, however, this sound channel supported frequency and volume enveloping, so further control logic was needed to generate this. There are a lot of specifications that the CPU could do to this sound channel, so for more information visit the

pandocs. In summary, the CPU can control this sound channel across 5 different memory mapped IO registers: NR10, NR11, NR12, NR13, and NR14. For ease of use, the IO register bus parser was used to quickly extract the data in these registers (see memory section). Given a certain volume envelope time and frequency envelope time, different counts were used to keep track of when to increment (or decrement) the volume and frequency to generate the sweep. Thus, the current volume and frequency were stored in a register and fed into the square waveform generator, while the counting logic would update the value when it needed to do so.

9.6 Sound Channel 2

Sound Channel 2 was almost an exact copy of Sound Channel 1 except that it did not support frequency enveloping, so this feature was removed. Finally, the second sound channel interfaced with the CPU over the 4 memory mapped registers: NR21, NR22, NR23, and NR24. For more information on these registers, please visit the pandocs.

9.7 Sound Channel 3

Sound Channel 3 was probably the most challenging sound channel because it implemented waveform ram. This gave the CPU the ability to generate a waveform of its own based off 32 4 bit signed samples that would be played once per sample. The waveform RAM was stored in the addresses 0xFF30 to 0xFF3F, giving 16 8 bit values where the high bits were the first sample, and the lower 4 bits were the second sample. The 32 samples would then be evenly spread across the period of the waveform based of the specified frequency. Since the frequency encoding in sound channel 3 was different than channels 1 and 2, another separate lookup table was used specifically for sound channel 3. Using this lookup table, the amount of clocks in a period was divided by 32 (shifted left by 5) to determine the amount of clocks in a sample. Every time a timer expired beyond the clocks in a sample, a 5 bit pointer to waveform ram was incremented. Thus, since the pointer easily rolls over back to zero, this easily allowed for playback of the signal. Using the pointer, the 4 bit unsigned sample was converted to a 20 bit unsigned value ranging from 0 to 2^19-1.

The CPU specifications for the sound channel 3 were stored the the control registers NR30, NR31, NR32, NR33, and NR24. For more information on these registers, please visit the pandocs.

9.8 Sound Channel 4

With the use of the random waveform generator, the 4th sound channel was fairly simple. The fourth sound channel was controlled by the control registers NR41, NR42, NR43, and NR44. For more information on these registers, please visit the pandocs.

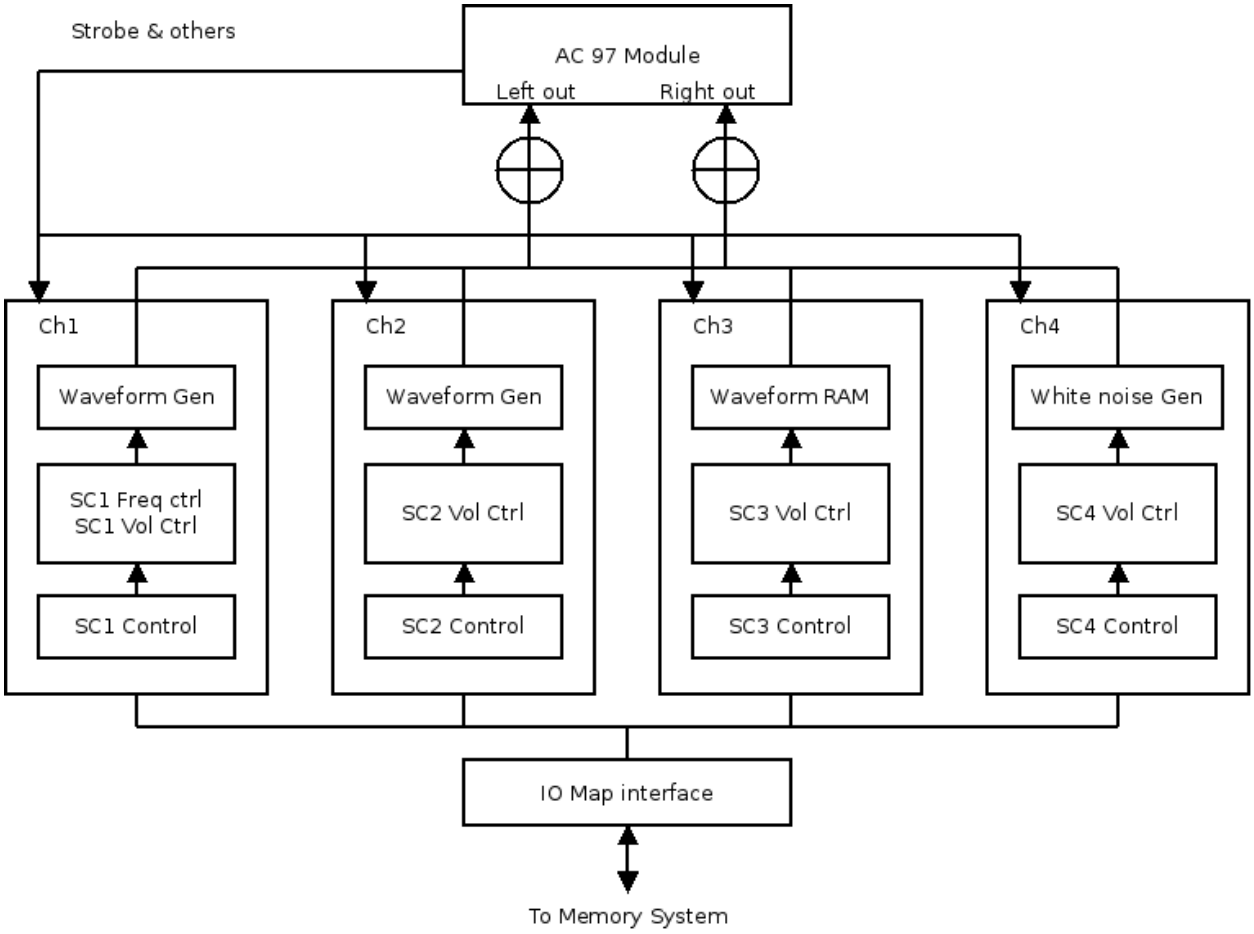
Note that volume enveloping was also implemented in this module, so further logic was needed surrounding the random waveform generator.

This sound channel also had to generate the shift clock for the LFSR in the random waveform generator, and the CPU could specify it to be randomly changed. This was simply done using a counter to generate a clock signal based off the CPU specification.

9.9 Top Level Sound Module

Finally, the top level sound module had 3 control registers NR50, NR51, and NR52. These registers could control whether to output each sound channel across two different stereo output channels. Furthermore, they could also specify the volume of each stereo output channel, however this functionality in our implementation was ignored.

To generate the final output signals to each waveform, a simple sum was taken of the samples from each channel at each strobe signal from the AC97 codec for each stereo output channel. Thus, this mixed the four sound channels together. Below is a diagram of how the entire sound system comes together.



9.10 Performance Enhancements

We originally were generating our waveform on the 48kHz strobe signal the the AC97 codec uses, however this led to poor sound quality. Often in the high frequency sounds, they would be slightly flat, making the listener somewhat cringe to the noise. To fix this, we generated the waveform on a faster 33 Mhz clock to given more precision from the lookup tables because on the slower clock, the flooring of the values from the Python scripts in the higher frequencies had a much larger effect than anticipated. Thus, running at a higher frequency allowed more precision on the amount of clocks in a period at these higher frequencies. To generate the waveform, we used the AC97 strobe to simply take a snapshot of the faster clocked waveform.

This, however, led to more complications because it introduced a “crackling” sound, kind of like the ones you would hear on an old record player. We figured out this was because crossing the clock domains on the signal was sometimes reading a metastable value. To fix this, we simply added one register in between the clock domains to stabilize it, and it served

as an interesting lesson in crossing clock domains (from what we always discussed in earlier course work but never really got hands on experience with).

9.11 Sound Testing

Sound testing in the early stages was quite difficult because it would often unexplainably not have any output at all. There was no true way to test it in simulation since it produced real time results. Consequently, we started out by using a working square waveform generator that we found. Modeling after this design, we slowly added functionality to our square waveform generator including the volume enveloping and frequency enveloping. Once this functionality was created for the first sound channel, the interface for the AC97 codec was very understood allowing the other sound channels to be up and running fairly shortly. After the sound channels were independently working, they were then integrated with the CPU and a simple assembly program was written to produce sounds that would mix together from the different channels. Finally, once we had some games up and running, we compared the sound to the BGB emulator as well as the gameboy itself, to further verify the correctness of the sound, where several more bugs were fixed.

10 Miscellaneous Modules

10.1 Timer Module

The timer module for the GBC is overall fairly simple. There are a few control registers, and it periodically sends timer interrupts. Note that this is completely separate from the timer specified in MBC3 cartridge.

Below is the specification of the timer registers (courtesy of the pandocs).

FF04 - DIV - Divider Register (R/W) - Register which is incremented at 1/256 the CPU clock speed, writing to it zeros the register.

FF05 - TIMA - Timer counter (R/W) - Register which is incremented at rate specified in TAC register, when it overflows an interrupt is asserted and it is reset to the value in TMA

FF06 - TMA - Timer Modulo (R/W) - When the TIMA overflows, this value will be loaded.

FF07 - TAC - Timer Control (R/W) - Bit 2 controls whether or not the timer is enabled. Bits 0-1 control the rate TIMA is incremented at.

We have one module that has all of these registers, and implementations of the specified behavior. It is connected to the rest of the system through the memory router, and its interrupt line goes to the interrupt module.

Despite being a relatively simple module, our implementation initially had some bugs, and fixing this module was critical to getting properly working games.

10.2 Interrupt module

The interrupt module manages the interrupt registers, and receives all of the input lines.

Below are the interrupt registers in the GBC:

IME - Interrupt Master Enable Flag (Write Only) - Global interrupt enable and disable, set by the CPU instructions EI, DI, RETI

FFFF - IE - Interrupt Enable (R/W) - Bit mask for interrupts being enabled.

FF0F - IF - Interrupt Flag (R/W) - Bit mask for asserted interrupts. Can write to the register to manually assert interrupts.

The IME and IE registers were implemented in the CPU, so this module essentially was responsible for managing the IF register. Essentially, it detected rising edges on the interrupt lines and stored them in the IF register, which was then fed into the CPU.

10.3 Clock Module

The clock module handled all the clocking logic for the system using clock dividers. It output a 4Mhz, 8Mhz, and 16Mhz signal. A register was used stored whether the system was in double speed mode or not, which can be controlled by the KEY1 register in the CPU memory mapped interface.

Though the Gameboy Color supports dynamic speed switching, it appeared that it would all be setup during the boot procedure of the device. Since we ignored the boot procedure, we simply used a dip switch upon reset to determine whether to double clock the CPU or not. Nevertheless, the CPU dynamic clock switching was implemented, using a count down to let the CPU state settle (at a STOP instruction), when the CPU clock would be changed. We do not know if this would work or not because it was never tested due to our fix using the dip switches.

10.4 Infrared Communication

The Gameboy Color also has an infrared LED that can be used to communicate with other devices, but this was omitted in our design. However, the RP control register was still in the design in case the CPU ever did anything with it.

10.5 Reset Module

To assure that the system was reset properly, a reset module was used that took an indefinite push button signal, and converted it to a countdown to zero where the synchronous reset signal was held high. This was a very simple module, and it just kept consistency across our resets.

10.6 Undocumented Gameboy Color Registers

According to the pandocs there were some areas of memory that the CPU could access and do things with, but the functionality was not defined anywhere in documentation. Each of

these registers were still implemented in our design with an IO Bus Parser component. What is known about these registers is as follows, taken from the pandocs:

FF6C - Undocumented (FEh) - Bit 0 (Read/Write) - CGB Mode Only
FF72 - Undocumented (00h) - Bit 0-7 (Read/Write)
FF73 - Undocumented (00h) - Bit 0-7 (Read/Write)
FF74 - Undocumented (00h) - Bit 0-7 (Read/Write) - CGB Mode Only
FF75 - Undocumented (8Fh) - Bit 4-6 (Read/Write)
FF76 - Undocumented (00h) - Always 00h (Read Only)
FF77 - Undocumented (00h) - Always 00h (Read Only)

Note the hexadecimal value in parentheses are the reset value.

11 Utilities

11.1 Self Generated Scripts

Our scripts for generating memory initialization (coe, mcs, dat) files:

- `gen_coe.sh` – Shell script to convert binary mem file to BRAM initialization file
- `lookup_table_gen.py` - Python file to generate BRAM lookup table
- `gen_dat.sh` – Shell script to convert binary mem file to simulation memory file
- `asm.sh` – Shell script to compile assembly and generate a coe and dat file
- `mcs_gen.bat` – Batch script to convert binary ROM file to PROM flashing file

11.2 BGB Boy Emulator

This emulator was very useful in debugging our system, especially when system integration came around. It allowed us to break at problem spots in the code where our system was failing, and analyze exactly what the CPU was trying to accomplish at the point in time. It allowed us to isolate different bugs, determining the origin of certain errors and brainstorm solutions on how to fix them. The link to the emulator download is below:

http://bgb.bircd.org/

11.3 Research

Most of our research can be found in the Gameboy Color Pan Docs. It had a very well rounded description of the system that was detailed, but brief.
http://problemkaputt.de/pandocs.ht

If the data in the pan docs were not good enough, we also referred to the Gameboy Programming Manual
http://www.chrisantonellis.com/files/gameboy/gb-programming-manual.pdf

11.4 Test ROMs

To aid in our integration testing, we utilized the Blargg test ROMs. Using these ROMs made testing much easier because we had access to the test source code, so knew what they were trying to do.
http://gbdev.gg8.se/wiki/articles/Test_ROMs

12 Tested ROMS

When we started getting a working system, we put together a list of the functionality of our system. Most of the Game Boy games were mostly functional with a few graphical errors, and in almost all games sound worked (and for those that didn't we suspect it is due to the bootload process being not implemented in our design). Our system we feel was successful, but it isn't perfect. Hopefully below can give a good idea of what we built is capable of.

12.1 Game Functionality Score

Legend:
5 - At most 1 small bug
4 - One glaring bug or couple small bugs
3 - Couple glaring bugs
2 - Barely Playable
1 - Does not boot

GB Games

Game	Score	Notes
Pokémon Blue	5	
Zelda Links Awakening	4	Need better color palette, white noise generator hangs
Tetris	5	
Tetris 2	5	
Super Mario Land	4	No Sound
Super Mario 4	4	
Kirby 2	1	
Metroid II	5	Need better color palette
Megaman V	5	Need better color palette

GBC Games

Game	Score	Notes
Pokémon Crystal	4	Window corruption when moving to new locations
Super Mario Bro. Deluxe	4	Semi-invisible character

Wario Land 3	3	Semi-invisible character, text window rendering issues
Army Men	3	Semi-invisible character, HUD rendering issue
1942	5	
Konami Collection	1	Controller doesn't work
Zelda Ages/Seasons	3	Background rendering issues
Zelda DX	4	White noise generator hangs
Pokémon Pinball	5	
Pokémon Yellow	4	Color rendering issues

12.2 Demo

To demo, we used two different concatenated rom files that worked with game switching within the cartridge. Below shows the contents of each ROM file on the FPGA that was used to demo.

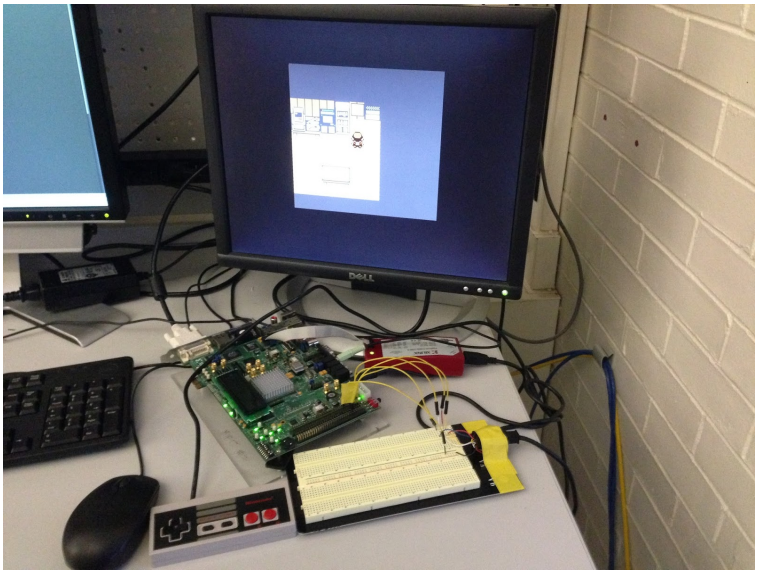
Combined_v1

- Pokémon Crystal
- Zelda - Links Awakening DX
- Tetris

Combine_v2

- Pokemon Blue
- Pokemon Pinball
- 1942
- Megaman V
- Metroid II
- Tetris 2
- Tetris

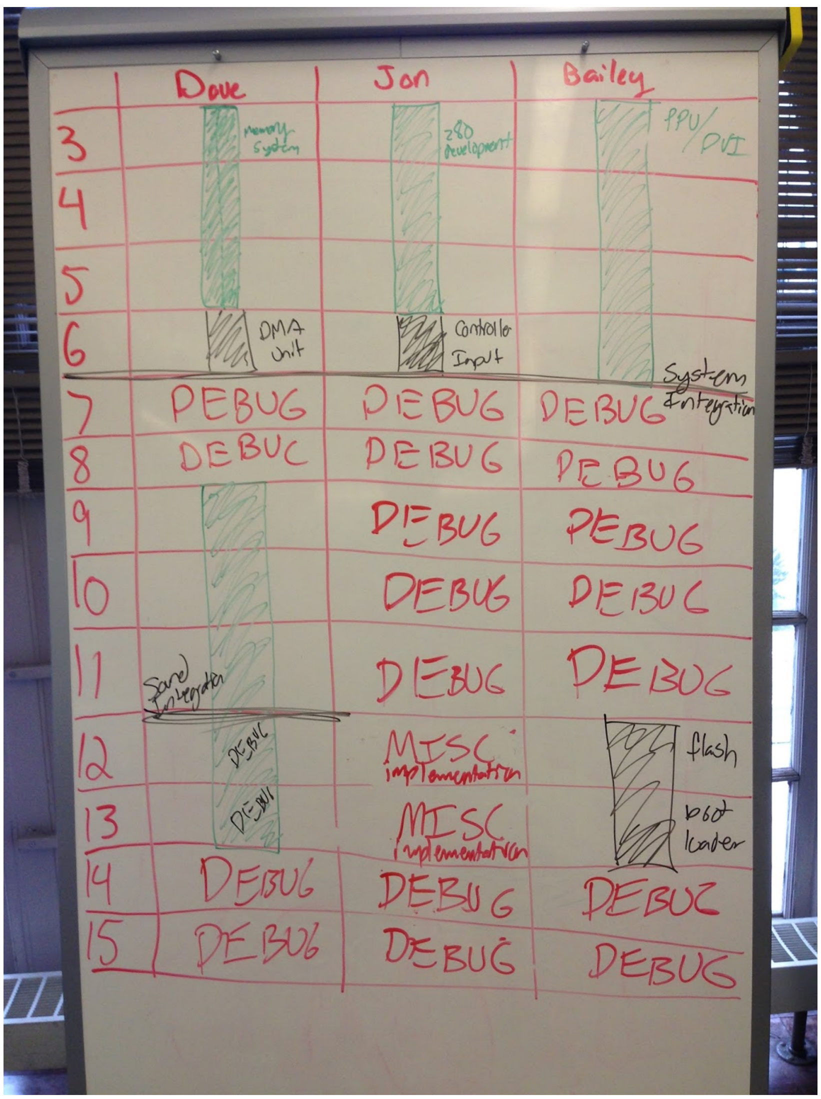
12.3 Screenshots



13 Schedule/Management Decisions

13.1 The First Schedule

For the first few weeks of the project, we did not accomplish as much as we would have liked. We believe that this is due to insufficient planning, and failure to break up tasks into smaller pieces. In the beginning of the semester, we were not very descriptive in what we needed to do, and it resulted in disappointment at mid-semester. We were very naive to think we could get to system integration in 6 weeks (we did not reach it until Week 13). Below is our first proposed schedule. When it came time for mid-semester status reports, we had nothing to show except for a simulation of CPU and memory.



13.2 The Second Schedule

We learned from our mistakes, and for our revised schedule, we broke up the tasks into smaller, more tangible chunks. We regrouped at mid-semester, and made a better schedule. For the most part, we stuck to this schedule, though we got off task a little bit with the PPU. We determined it was necessary for some members to take on a larger role due to other members coursework. In the end, most of our time was spent in the system integration, in countless long nights over Thanksgiving to get it done. Advice to future teams would be to give yourself buffer room for getting behind in your work and unexpected bugs. Future teams should also expect to give many hours to the project in the final weeks.

Below is the revised schedule. Though again we did not follow it in certain situations, it kept us on track and alarmed us if we were getting too far off-schedule. The red is where we fell behind and the green are comments after the fact about the schedule.

Week # and date)	Dave	Bailey	Jon
7 (10/6)	Design review work (slides, practice presentation) Memory banking Memory Duplicating? Memory CPU Integration	Design review work. PPU GBC work.	Design review work Memory CPU integration Memory Wrapper
8 (10/13)	Memory & CPU Integration Memory and CPU Testing Add PPU to memory router	Finish GPU GBC modifications	Memory & CPU Integration Memory simulation Memory and CPU Testing
9 (10/20)	DMA Controller Implementation and Integration	Integrate PPU with rest of system and DMA integration	Interrupt module and connection to CPU

		Figure out emulator logistics to prepare for cartridge interface	Timer Module Implementation
10 (10/27)	Cartridge Interface Cartridge Interface Testing We fell behind the cartridge interface because level shifters were a week late.	Cartridge Interface Hardware. Cartridge Interface Testing Continued integration and debug with rest of system. Link cable planning.	Push Button Basic Controller Module (make in adaptable for NES controller later)
11 (11/3) Decided to put games on flash not cartridge	General Integration Begin sound implementation	General integration Link cable implementation Tested level shifters and GPIO ports	General integration Controller integration. adapt for NES controller I/O Registers
12 (11/10)	Sound implementation Decision about sound integration Happily, we chose to do so because we were ahead of schedule on sound at this point.	Sound support Sound Testing	Sound Support Sound Testing
13 (11/17)	Full Integration and Debug Sound integration Implemented cartridge on flash and FPGA due to level shifter failure	Full Integration and Debug PPU integration didn't really get completed until here (3 weeks late)	Full Integration and Debug

14 (11/24) (Thanksgiving) most of debugging and integration happened here decided to add game switching	Full Integration and Debug Unforeseen Design Corrections	Full Integration and Debug Unforeseen Design Corrections	Full Integration and Debug Unforeseen Design Corrections
15 (12/1)	Full Integration and Debug Finishing touches	Full Integration and Debug Finishing touches	Full Integration and Debug added game switching added link cable Finishing touches

14 How We Built it

14.1 Approach

The main way we really approached the project was to break up the system into its different components. After that, we thought about how we would handle each component, such as whether or not we would reuse existing code, or how we planned to implement it and how it would integrate with the rest of the system. During this step, we thought carefully about how the different pieces would fit together. Different components were unit tested when possible. Lastly, in the system integration step, all of the components were put together and tested as a whole by running GBC ROMs on the system.

14.2 Design partitioning

We tried to break up modules into units with similar functionality. Sometimes this was very easy, other times it wasn't as clear. For example it was very clear that we should have a CPU module, a PPU module, and a Timer module. Other components such as the interrupt module and working RAM modules were less clear. On sufficiently large modules like the PPU and sound systems, appropriate sub modules were implemented by similar principle. Overall, we think that our design partitioning was successful, and the partitions were our basic units of work that were distributed between team members. Most of the major submodules of the design are described in the implementation section.

14.3 Tools and design methodology

We developed our project on the Virtex 5 board, using the Xilinx ISE IDE. We chose to use Windows over Red Hat because we felt the tools were slower and not as stable under Red Hat. Because we used ISE, we made extensive use of Chipscope for debugging.

Other tools we used included the BGB emulator and the Blargg Game Boy unit tests which are described above.

Nintendo is very secretive about the different chips actually used in the implementation of the GBC. As a result, rather than implementing the actual chips in the GBC, we focused on producing the same behavior divided into modules that we felt to be appropriate divisions. This means that the general methodology involved viewing the pandocs and the Game Boy Programming manual for the specifications of the component we were trying to produce. After that, we implemented the hardware that we believed to be necessary to produce the desired behavior.

14.4 Testing and verification methodology

There were a few parts we tested in simulation. In retrospect, we should have made greater use of simulation. Below is a list of the parts we tested in simulation:

- CPU instruction test
- CPU memory integration test
- OAM DMA test

The PPU was tested in a unique way, where memory dumps were placed into the video ram and examining the screen output. The assumption was that if the PPU could successfully display an image with the right bytes in memory, then it could display the right thing when the memory was being dynamically updated, and this mostly held true. However, the sample memory dumps used were not ideal, because they did not fully test the features of the PPU, leading to some bugs not being found until integration. As work was done on the PPU, the screen contents could be viewed for the expected results as well as used for regression testing.

The sound was tested in a similar way as the PPU, where we started from a square wave generator and gradually more of the Game Boy features were added onto it. For more information, see the section on sound testing.

The remaining modules were primarily tested in system integration. Some components we wish we had tested with unit tests, such as the timer and interrupt modules. The bugs were much more difficult to find in integration because we had to narrow down the possibilities of the bugs. Fortunately, we did not have that many major bugs, so were still able to have a successful project at the end of the semester.

To give some direction to our integration testing, we used a set of Game Boy test ROMs known as the Blargg test roms, which is a set of unit tests for the Game Boy and GBC.

14.5 Status and future work

Our end result was a mostly complete implementation of the Game Boy Color system. We could run our target game, Pokemon Crystal, with only one noticeable bug. We could also run most DMG games without bugs, and most GBC games worked with no issues or minimal issues.

Future work would mainly be refining the current system, such as tracking down the remaining bugs, completing implementation of the link cable, and the infrared port.

15 What We Learned

15.1 What we wish we had known at the beginning of the project

First of all, we wish we knew how long this project would take. We did not anticipant the shear number of hours that it would require. This was one of the reasons why our schedule slipped quite a bit. On a related note, we wish we had known the importance of managing and optimizing our time to avoid our scheduling issues.

Secondly, we learned that reusing existing components is not always a good idea. It worked out very well with the CPU, because it was very well written, had very few bugs, and we could readily get support from Joe. The same cannot be said about the PPU, with its poor documentation, many bugs, and no support. We should have realized these red flags sooner, and rewritten our own from scratch.

Lastly, we wish that we had better skills with FGPAs. We did take a lot of Verilog based classes, such as 18447, 18340, and 18341. However, the last time any of us had touched real FPGAs was in 18240. This means we did not handle the differences between simulation versus synthesis well, and in general did not know how to debug the hardware effectively coming into the course.

15.2 Particularly good/bad decisions you made

Good Decisions

- Reusing the CPU from the F13 Game Boy team - If we had not reused the CPU from last year's team we likely would not have finished the project.
- Using Virtex 5 - Overall, we didn't have any issues with the virtex 5 board, and it was what the course staff was most familiar with.
- Switching to windows - We quickly realized in the beginning of the semester that the Xilinx tools very very slow and unreliable on the Red Hat installation on the lab computers. We speculated that the Xilinx tools would run better on Windows, and this turned out to be true
- Utilizing our resources - In particular, we made extensive use of the block memory in place of more complicated solutions. Since we had the extra BRAM available, we used it.
- Choosing to do the GBC - We originally wanted to do the Nintendo 64, but after reading the specs, we quickly realized that it would probably be too much for us. Our next option was the SNES, but the lack of good documentation put us off. Finally, we settled for the GBC because it was a good balance of complexity and documentation.

Bad Decisions:

- Reusing PPU - We must sound like a broken record at this point, but reusing the PPU was very painful. We should have identified its issues and decided to make one from scratch, possibly using the old one as reference.
- Assigning tasks that are too large and vague - Our initial schedule was to work on the “CPU”, the “PPU”, and the “memory system” for 5 weeks separately. Because these tasks are so large and vague, it was easy for the schedules to slip.
- Didn’t do enough research - The main example of this is wasting time on the physical cartridge and level shifters.
- Lack of unit tests - Many of our modules didn’t have proper test modules written, so sometimes simple bugs slipped by and weren’t found until system integration, wasting valuable time

15.3 Words of wisdom for future generations

These words of wisdom mostly echo the above two sections.

First of all, these projects take a lot of time, so be ready for that.

Secondly, be practical in what you can do. Don’t set too ambitious of goals, such as our planning to be at system integration by week 6. By setting goals so ambitious, we subconsciously knew that we couldn’t actually meet them, and so our schedule slipped.

Lastly, know your tools well. Learning how to use chipscope (the logic analyzer) successfully was essential to getting our project finished on time.

16 Personal Statements

16.1 Jonathan Leung

What I did
In the beginning I was assigned the task of studying, testing, and integrating the CPU from the F13 Game Boy team. After that I implemented the timer, interrupt, and serial link modules. I was in charge of system integration and maintained the top level module, debugging modules as we integrated them. I was also in charge of tools which include: GB assembler, ISIM, BRAM generation, mcs/coe/bin memory dump generation, rom generation, and PROM flashing. Finally I did a fair share of debugging.

In terms of hours, we did not do any logging so I will do some estimates. On the lighter weeks I would generally spend 6 hours a week on the class, on the heavier weeks around 10, and during crunch week it went up to 50 hours. With these estimates I would imagine I spent around 150 hours on the project.

Class Impressions
I enjoyed this class a lot, as I was very lucky to have two great friends who are also great teammates. We are usually on the same page, and our personalities balance out. If our team dynamic wasn't as good then I would imagine the class to be far less enjoyable.

I felt that we received the right amount of help from the TAs, though it seems like we were the only group really asking them questions. Though, I think that we were more inclined to do so because one of the TAs was on the F13 Game Boy team.

As expected the class is fairly demanding, though we could've handled our time more wisely. Though there really wasn't much we can do to improve our time management. Like many other seniors during the fall semester, much time is devoted to finding jobs and flying out to interviews. We probably had at least one person missing more than half the time. Though this experience definitely taught me how to tackle planning and executing a real-world project.

I found that it would be great if we can get more help with the labs, as it was pretty time consuming trying to figure out everything on our own. This led to us barely working on our project until the labs were over.

16.2 David Campbell

What I did

When first starting out the semester, I was assigned to getting the memory system together, which would eventually lead to CPU-Memory integration for the mid-semester design review. By mid-semester I mostly had all of the components of the memory up and running which include the memory router, the IO bus parser, the working memory, and some test memory locations for the VRAM and OAM that we needed for the CPU memory integration, but couldn't actually implemented until the PPU was integrated.

Once we had CPU-Memory Integration mostly complete, I began to work on the DMA engines, and I developed some assembly test procedures that would test the OAM DMA. I was not truly capable of verifying the GDMA and HDMA until the PPU integration because of the timing sensitivity.

Since this had to go on hold, I then started to get an early start on sound. Over the course of about 3 weeks I had developed the implementation of the sound, which ended up being really successful. Further down the road, I started integrating the CPU and sound to generate multiple sound channel mixing. I feel very proud about the sound implementation because it was a lot of fun playing with the sounds, and it also added a nice touch to our project.

When my team also decided to not do a cartridge interface, I also stepped up and implemented the cartridge sim module, which implemented the MBC3 cartridge on the FPGA and then fetched from flash for the ROM data. This also turned out to be very successful because not only did the internal time of day register work, but also game saving worked as well.

Once the majority of the components I created were integrated into the system, the majority of my time was assisting in the system integration in the last 3 weeks. Whereas Bailey was much more involved with the chipscope, I played a more of a role in hypothesizing why certain bugs might be happening and then searching for evidence to back my claims (and Bailey and Jon did this too). Overall, I think that this strategy really helped because it forced us to literally write down all the possible reasons for a bug, and we ended up working well together.

Overall, I probably spent about 18 hours a week for this class on average. Some weeks were more and some weeks were less. In particular probably around Week 13 and Week 14, I spent about 60 hours a week in lab.

Class Impressions

I actually really enjoyed the class. A class like this was the reason that I wanted to become an engineer, and consequently, I'm afraid I don't have many complaints or suggestions for improvement. Whereas other classes give good technical background, I found that this class gives the opportunity to learn and practice project management skills, goal setting, teamwork, and system level debugging techniques. I felt that these areas were a strength of mine, and other classes do not give the opportunity to utilize them as much.

16.3 Bailey Forrest

What I did

At the beginning of the semester, I was assigned to work on the PPU (Pixel Processing Unit) of the PPU. The main objectives were to add support for the GBC features to last year's DMG PPU. This included storing of the color information in memory, exposing the color palette interface to the memory system, and handling of the color information in the PPU. In addition, because the PPU from last year's group was obtained from an online source (FPGAboy), the general quality was very poor, and there were many bugs that needed to be fixed. In retrospect, I regret making the decision to reuse the PPU, and wish that I would have written it from scratch.

In addition to working on the PPU, I worked on the interface with the NES controller and the cartridge interface hardware. Unfortunately, due to the poor performance of the Virtex 5 GPIO pins at the required 4 MHz, the cartridge connector that I had setup was not used. In retrospect, it would have been better to test the GPIO pins and the level shifters before creating the hardware.

Towards the last few weeks of the project, I did a lot of system integration and debug work. I probably ended up doing most of the debugging. I also did all of the work for the team poster.

The amount of time I spent on the class greatly varied. Unfortunately due to other classes and having to fly out for interviews almost every week, I did not get as much done it as I would have liked for the beginning of the semester. As we did not keep track of hours, I would say that my hours spent per week in the class varied between 8 to 90 (for the thanksgiving week). Probably on average, I spent around 14 hrs per week on the class.

Class Impressions

Overall, I felt that this class was a great experience. Not only did I learn technical skills, but also learned in terms of soft skills such as team and schedule management. The long compile debug test cycles meant that while debugging, I had to really understand the system I was looking at and make careful decisions of which signals to put into chipscope. This experience has made me a much better debugger at software system.

I feel like not enough classes in the curriculum emphasize the importance of working with people with self imposed deadlines. I could complain about the course staff not prodding us enough, but I think that in the grand scheme, it makes for a better learning experience for us to manage ourselves.

I can't really think of any real complaints for the class. Maybe the labs took us a while, so it delayed in our project work. However, labs became integral parts of our project later on (chipscope and sound), so I can't really complain about this.