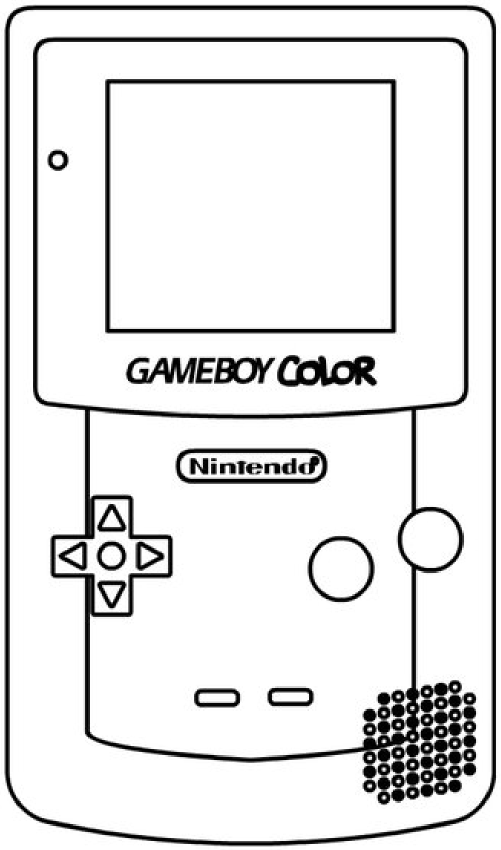
18-545

# Team FPGBC

## Design Review

**Mike Gardner, Costas Akrivoulis, Arjuna Hayes**

**12/11/2012**

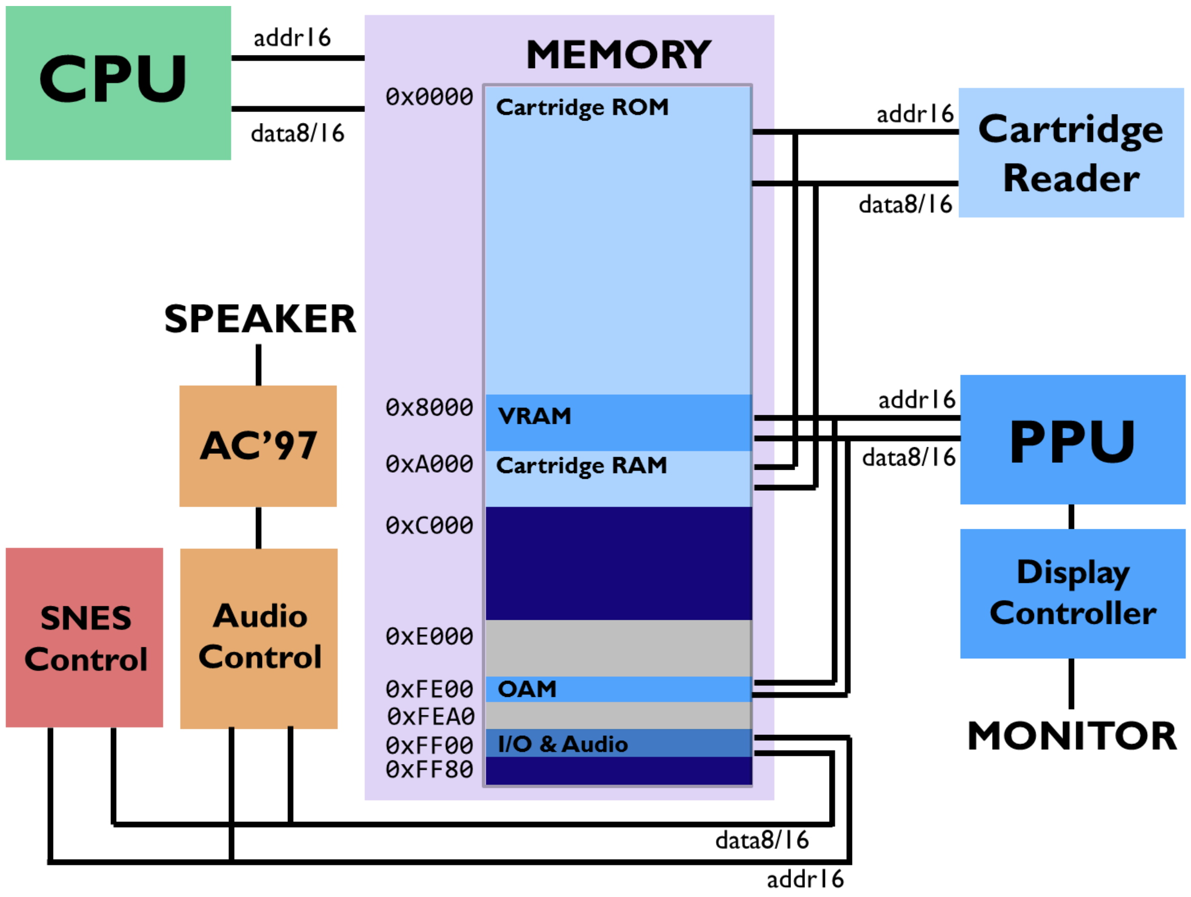


## Overview

Nintendo's 1998 Game Boy Color was a handheld console that ran on an 8-bit microprocesspr, up to 8MB of ROM, 32kB of RAM, and 16kB of VRAM. It was also the first handheld console to be backwards compatible with older Game Boy consoles. With our hardware implementation we aimed to play arbitrary cartridges with a SNES controller. What we actually accomplished by demonstration day can be found in the last section of this paper.

## Memory

Memory is the central interface through which all the components in the Game Boy Color system communicate. The majority of control across subsystems occurs through manipulating memory-mapped registers. The Game Boy Color has an address space of size 32KB, more than half of which is dedicated as a view into the attached cartridge. The following shows how the address space is divided across various subsystems within the GBC:



Here is a detailed breakdown of the address space:

## General Memory Map

```
  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
```

Our memory module simulates many of the listed segments of the address space, but there is also a significant portion of the address space which maps to the cartridge. In these cases, our memory interface will act as an address multiplexer, directly feeding address ranges to the appropriate subsystem (either the cartridge reader, our RAM, or our ROM).

We used the Xilinx Core Generator to create the video and work RAM. We chose this option over simply making a large array because the size of the memories made synthesis time unreasonable. We faced two main issues when designing the RAM: bank switching and access ports.

*Bank Switching*

Both the VRAM and the WRAM could switch memory banks by writing to certain memory mapped IO registers:

## FF4F - VBK - CGB Mode Only - VRAM Bank

This 1bit register selects the current Video Memory (VRAM) Bank.

```
  Bit 0 - VRAM Bank (0-1)
```

## FF70 - SVBK - CGB Mode Only - WRAM Bank

In CGB Mode 32 KBytes internal RAM are available. This memory is divided into 8 banks of 4 KBytes each. Bank 0 is always available in memory at C000-CFFF, Bank 1-7 can be selected into the address space at D000-DFFF.

```
  Bit 0-2  Select WRAM Bank (Read/Write)
```

To implement this, we made each of the RAMs a contiguous memory and used the upper bits of the address to specify the bank. This reduced the number of memories we had to create without adding complexity to the addressing.

*Access Ports*

The largest drawback to the CoreGen memory is that each has only two read/write ports. This was a problem because we had more than two devices that could potentially access each memory: the CPU, HDMA controller, OAM DMA controller, and the PPU. Thankfully, due to restrictions the original Game Boy Color imposed, we were able to work around the problem. Firstly, when the HDMA controller is active, it pauses CPU execution, which means neither the CPU nor an OAM DMA transfer will access RAM. When an OAM DMA is running, the GBC spec specifies that the CPU can only access high ram (which we implemented as a separate block of memory). Thus, we were able to make it so that three of the devices could share one port per, letting the other port be used exclusively for the PPU.

What was left of the memory (OAM, memory mapped I/O, and high RAM) was implemented as a simple array. They only account for 512 bytes total and it would have been very difficult to restrict access to this memory (mostly because of the many IO interactions which exist). Since it did not prove to be a significant problem for synthesis, we decided to just keep it as an array.

There is a significant amount of miscellaneous logic in the ROM and RAM modules. In ROM, there is logic for “hijacking” execution when an interrupt occurs so that the CPU jumps to the correct interrupt vector. In the RAM, there are multiple handlers for different memory mapped I/O. Enabling and disabling interrupts, triggering DMAs, setting screen parameters, and controller input is all handled in RAM.

## CPU

*Design*

The Game Boy Color has a modified version of the Zilog Z80 chip. The Z80 is a popular chip found in a variety of embedded systems, from arcade cabinets and the original Game Boy to fax machines and TI calculators. The chip found in the Game Boy Color has been modified in a multitude of ways. There are no block commands, no IX- or IY- registers, and no exchange instructions. Most of the 16 bit memory accesses and arithmetic have been removed. Additionally, the Z80 in/out instructions have been removed. Instead, I/O is done via the load instruction to memory mapped registers. Finally, the sign, parity, and overflow flags have all been removed (the Zero and Carry flags remain).

Multiple instructions have been added. Among them are additional addressing modes for loads and the return and stop commands. Many of the now-unused instruction opcodes have been repurposed for these new instructions.

## Moved, Removed, and Added Opcodes

| Opcode | Z80 | GMB |
|---|---|---|
| 08 | EX AF,AF | LD (nn),SP |
| 10 | DJNZ PC+dd | STOP |
| 22 | LD (nn),HL | LDI (HL),A |
| 2A | LD HL,(nn) | LDI A,(HL) |
| 32 | LD (nn),A | LDD (HL),A |
| 3A | LD A,(nn) | LDD A,(HL) |
| D3 | OUT (n),A | - |
| D9 | EXX | RETI |
| DB | IN A,(n) | - |
| DD | <IX> | - |
| E0 | RET PO | LD (FF00+n),A |
| E2 | JP PO,nn | LD (FF00+C),A |
| E3 | EX (SP),HL | - |
| E4 | CALL P0,nn | - |
| E8 | RET PE | ADD SP,dd |
| EA | JP PE,nn | LD (nn),A |
| EB | EX DE,HL | - |
| EC | CALL PE,nn | - |
| ED | <pref> | - |
| F0 | RET P | LD A,(FF00+n) |
| F2 | JP P,nn | LD A,(FF00+C) |
| F4 | CALL P,nn | - |
| F8 | RET M | LD HL,SP+dd |
| FA | JP M,nn | LD A,(nn) |
| FC | CALL M,nn | - |
| FD | <IY> | - |
| CB3X | SLL r/(HL) | SWAP r/(HL) |

The CPU has a 16bit stack pointer and program counter, as well as eight 8bit registers, including an accumulator and a flag register. The registers can be referenced as individual registers or paired up to make four 16bit registers. It has a clock frequency of about 4.19 MHz, but can also run at 8.4MHz in double speed mode.

*Implementation*

We are using an open source Z80 in our Game Boy Color implementation. Originally, a VHDL project (the T80) was created to model the Z80 with nearly identical cycle timing. This code was ported to Verilog and renamed as the TV80, and is what we will use as our CPU. The processor has a Game Boy mode which changes all the appropriate instructions. Originally, it was thought that the TV80 was nearly complete (as far as the Game Boy mode was concerned). However, after thorough testing, we determined that there were many bugs with the open core implementation of the Z80. Multiple instructions did not work, some signed numbers were incorrectly converted to unsigned numbers, interrupts would need to be handled in a separate module because the TV80 (in GBC mode) could not support them, and treatment of condition codes did not work at all. Thus, our main task involving the CPU is integrating it into the rest of our project.

*Interrupts*

The interrupt vectors are stored in the first 256 bytes in memory:

| NAME | LOCATION | |
|---|---|---|
| V-Blank Interrupt | 0040 | The V-Blank Interrupt is to occur while the PPU is experiencing a V-Blank. When this event occurs, the LY register (FF44) is between the range of 0x144 and 0x153. |
| LCDC Status Interrupt | 0048 | This interrupt watches the STAT register (FF41), which defines what the interrupt does. It can watch for an H-Blank, V-Blank, OAM search, and/or VRAM search. |
| Timer Interrupt | 0050 | The Timer Interrupt is to occur when the TIMA register (FF05) overflows, that is, when it becomes FF. |
| Serial Interrupt | 0058 | Used for Link Cable Connection. Not supported in our GBC implementation. |
| Joypad Interrupt | 0060 | Used for determining button pushes. Most (all test games in our case) use loops to check for button pushes and ignore Joypad Interrupts entirely. Not supported in our GBC implementation. |

As mentioned in the previous section, the TV80 could not handle interrupts without assistance from an outside module. When an interrupt is called, we send an NMI (Non Maskable Interrupt) to the CPU. The

NMI automatically jumps to address 0x0066. When the NMI is executed, we also send a bit to the CPU to tell it to execute our custom instructions instead of the instructions in the ROM. Our custom instruction is a jump to a register value, which we set to the correct value based on the type of interrupt. For example, if a V-Blank occurs:

```
1) 40 in register (interrupt vector location for V-Blank)
2) Set interrupt bit (to tell CPU to run our instructions at 0x0066)
3) Send CPU an NMI instruction
4) Go to address 0x0066
5) Run our instruction of jump to 00nn (where nn in this case is 40)
6) Jump to correct interrupt vector
```

These steps ensure correct handling of interrupts. We have also verified this in simulation testing and by matching operation of ROMs with an emulator.

## Pixel Processing Unit (PPU)

*Background:*

The Game Boy Color has a Pixel Processing Unit (PPU) which is responsible for using data stored in video memory to render the image which is displayed on the Game Boy Color LCD screen. The PPU performs multiple, specific lookups in various graphics tables in order to determine which color should be displayed at each pixel, for the entire screen. The PPU manages these graphics objects:

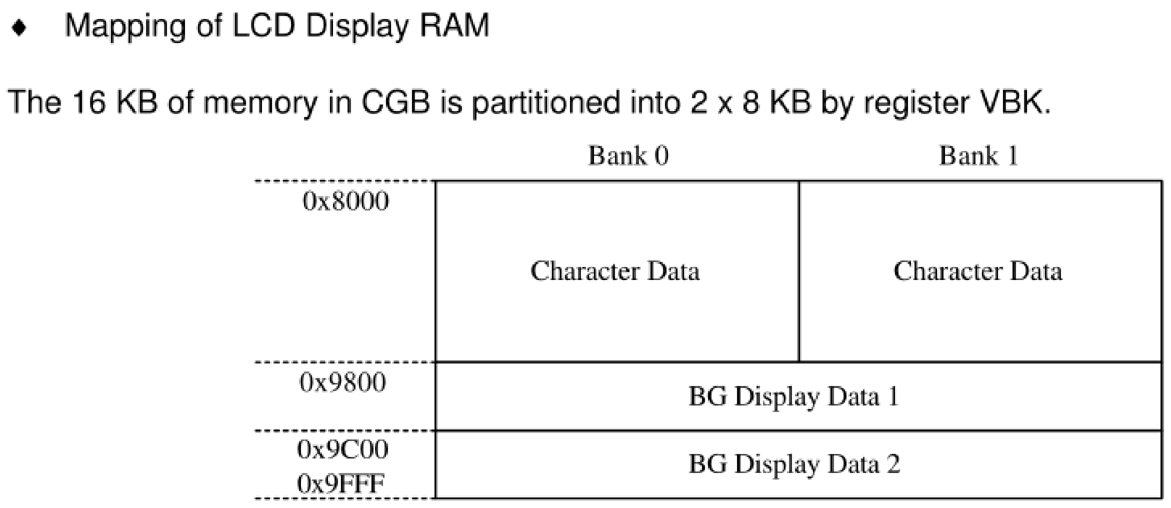
1. Tile (character) map
2. Background map
3. Sprite attribute table (OAM)
4. Color palettes
5. Graphics-related memory-mapped registers

The memory region from 0x8000 to 0xA000 is used as video RAM (VRAM) and serves as the operating memory region for the PPU. In addition, there is a sprite attribute table, also known as the Object Attribute Memory (OAM), which resides from 0xFE00 to 0xFEA0. There are also a few control registers which the PPU needs to monitor and update, so that the CPU can read an accurate value from them; these are located in higher memory, lumped together with other I/O ports.

```
  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
```

The PPU is a fixed-function graphics pipeline, which reads the data in different parts of the VRAM + OAM space, in order to perform the various lookups for each graphics object, to determine pixel color. The CPU, either through the DMA controllers or directly, will place the correct information in VRAM & OAM periodically so that the screen can update. The figure below shows how the VRAM is partitioned into two main groups (1 and 2 from the list above):


Mapping of LCD Display RAM
The 16 KB of memory in CGB is partitioned into 2 x 8 KB by register VBK.
Bank 0
Bank 1
0x8000
Character Data
Character Data
0x9800
BG Display Data 1
0x9C00
0x9FFF
BG Display Data 2

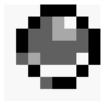
*Tiles*

In order to reduce the amount of memory required to store graphics objects, the Game Boy Color uses tiles to describe graphical objects and uses look-up tables to determine which tiles to use (instead of maintaining a copy of the screen in memory, i.e. a framebuffer, with a few bytes per pixel to hold precise color information). This allows for a more space-efficient, malleable representation of what should be shown on screen, and a much more sane way for GBC games to manage their graphical objects. Because of this, tiles are the PPU's atomic unit, not individual pixels.

The Game Boy Color has two types of tiles, the more common 8x8 tile and the less common 8x16 tile. Although it is possible to switch between and use both types of tiles in a single game, we will focus on the 8 pixel by 8 pixel tiles in this documentation to make explanation simpler.

*Tile (character) data*

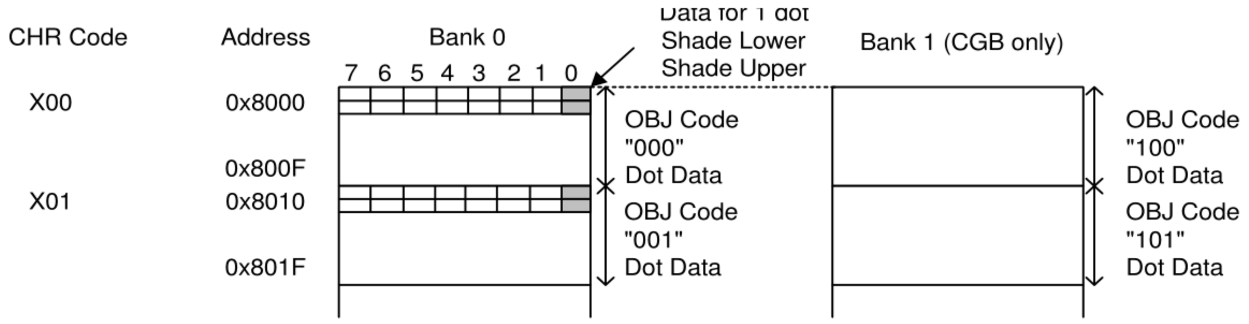
A large portion of VRAM, as shown in the figure above, is dedicated to storing the tile information that is used when drawing the background or sprites. Each tile consists of 8x8 pixels (or 8x16 for large tiles), and each pixel can be one of four colors. Note however, the actual colors that are used for a particular tile are not stored; only the relative color between pixels within the same tile is stored. This tile information is stored in the tile (character) map, and is referenced later by other tables.

For example, let's examine a single tile (the pokeball in the Pokemon title screen): 

Notice that this pokeball only has 4 distinct colors (black, dark gray, light gray and white).

Because there are only 4 colors per tile, each pixel requires 2 bits to represent the color data. In memory, each row requires 2 bytes; the first byte is the bottom half of the row color, and the second byte is the top half. For example, the *second* row is represented as: (fifth pixel shown in bold)

| | | |
|---|---|---|
| 0010_**1**010 | (bottom) | [ white, white, black, dark gray, **light gray**, white, black, white] |
| 0011_**0**010 | (top) | [ 0, 0, 3, 2, **1**, 0, 1, 0] |

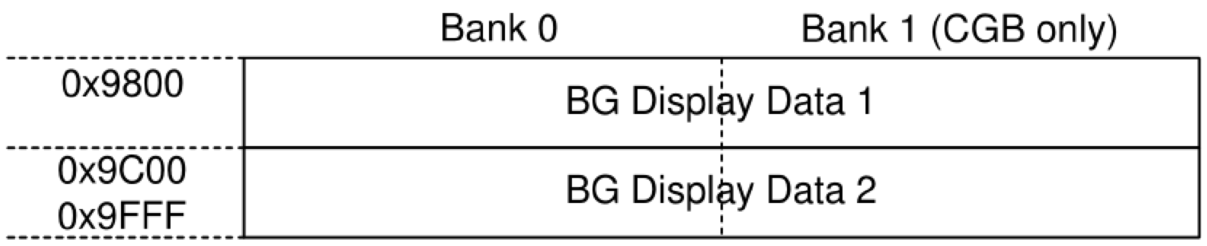


In this case, these numbers happen to correspond to those colors, but this can be changed by changing the color palette, which we will see later. As shown in the figure above, two banks of memory are dedicated to providing character (tile) data and both are available for reference from the other tables.

*Background Map*

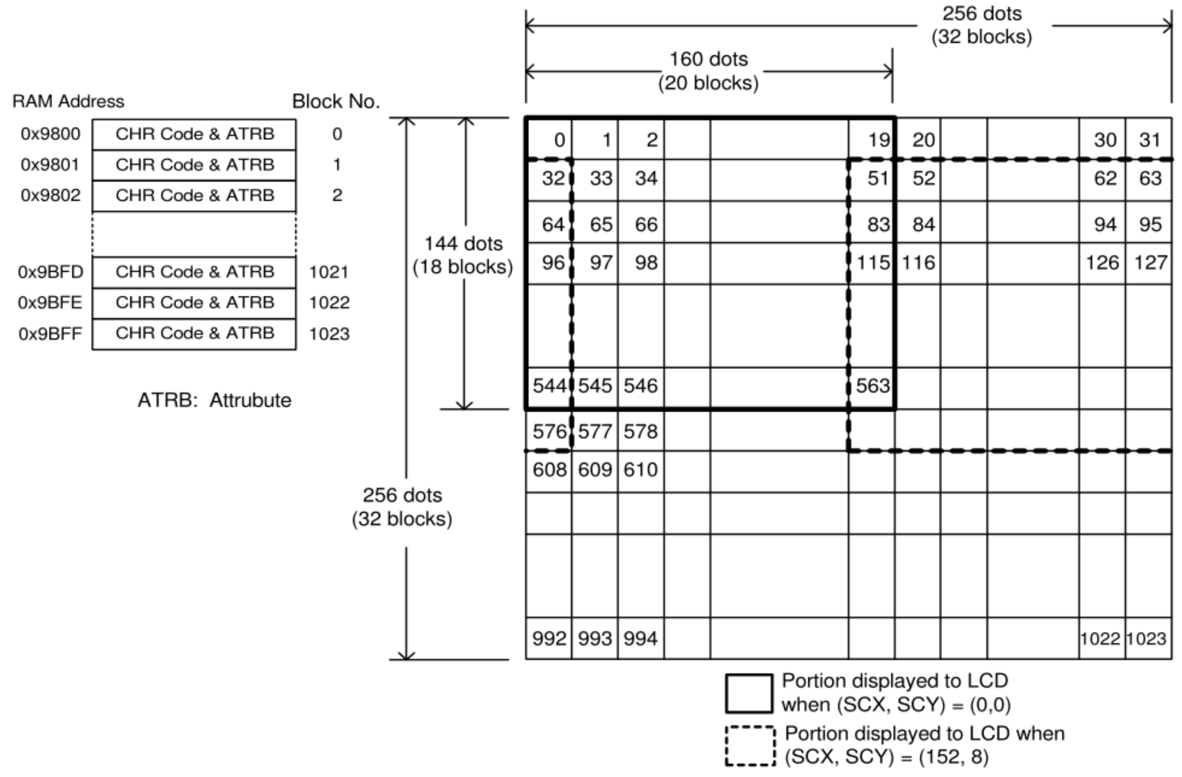
The Game Boy Color is designed to hold a large background map in memory, only part of which is actually displayed on the screen. There are also scroll registers (SCX and SCY) which allow the LCD screen to display different parts of the background map. This mechanic allows games to place a large, static map in memory and just scroll to different parts of it.

The background map was 256 x 256 pixels large, meaning 32x32 tiles. By contrast, the actual LCD screen could only display a 160x144 pixel image, or 20x18 tiles. In addition, the GBC had two full background maps that a game could switch between, stored in memory as follows:



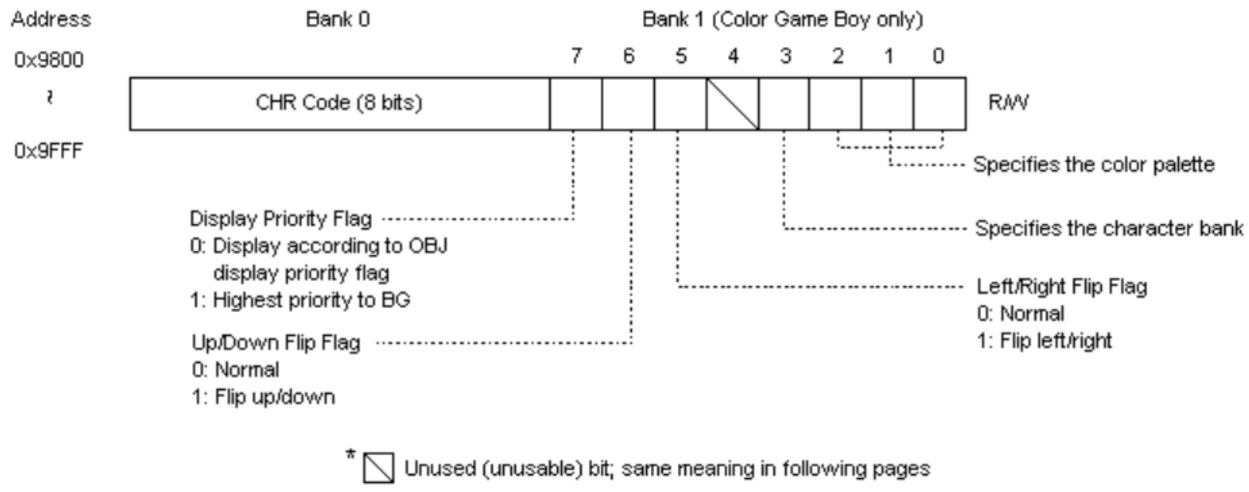
Each background map is organized in this fashion:

1) With BG display data allocated to 0x9800-0x9BFF:



Note: Attributes specified only with CGB

For each tile, there is one byte (in bank 0) which is an index into the character map that tells the PPU which tile to use, and another byte (in bank 1) which that tile's attributes. Note that the same tile can be used multiple times in the background simply by referencing the same tile index in multiple blocks. This also allows the ability to make larger, symmetric objects efficiently by using fewer tiles, but modifying the tile attributes. The different attribute parameters for a tile are shown in the figure below:

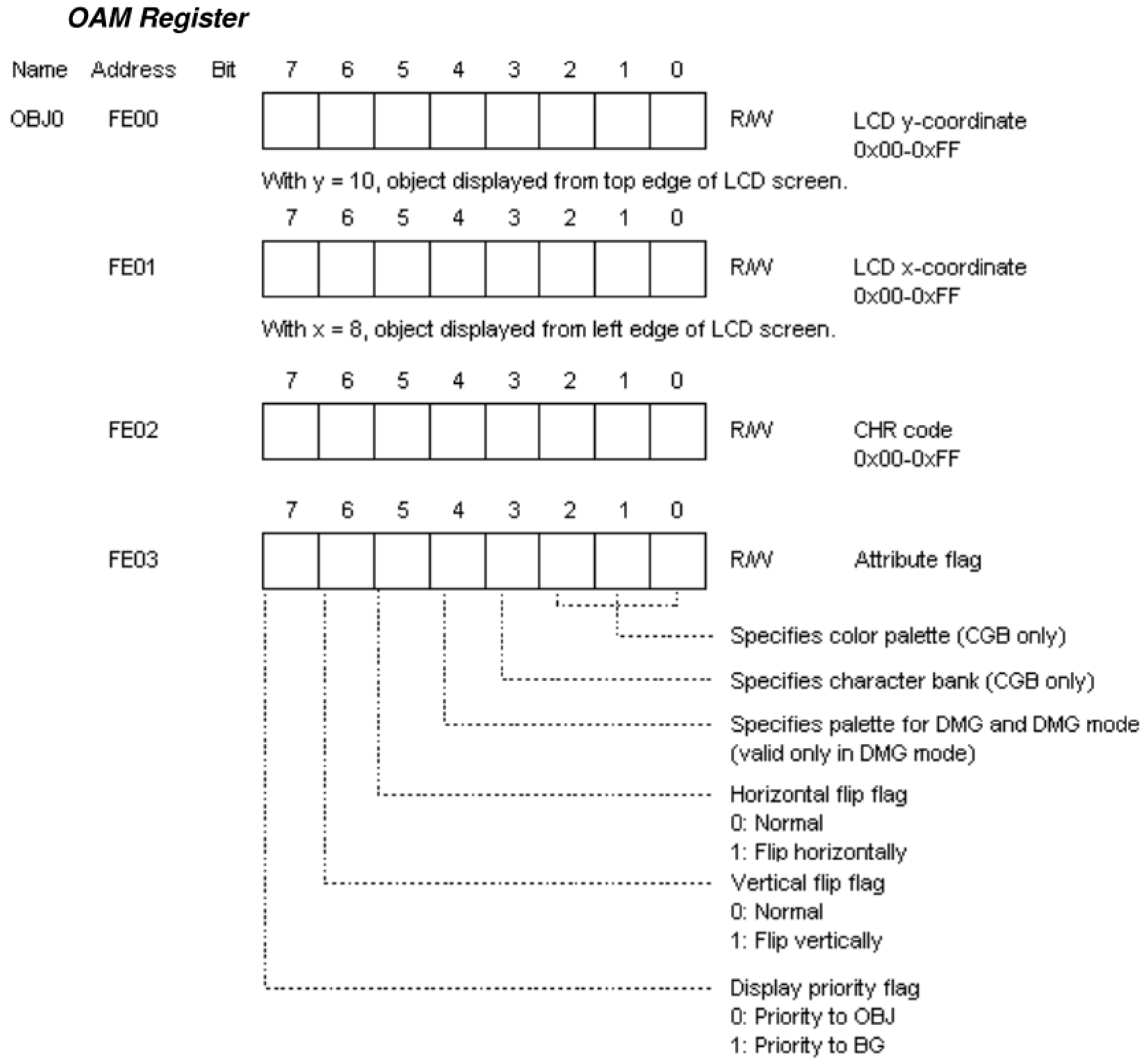

Address
0x9800
~
0x9FFF
Bank 0
CHR Code (8 bits)
Bank 1 (Color Game Boy only)
7 6 5 4 3 2 1 0
R/W
Specifies the color palette
Specifies the character bank
Left/Right Flip Flag
0: Normal
1: Flip left/right
Display Priority Flag
0: Display according to OBJ display priority flag
1: Highest priority to BG
Up/Down Flip Flag
0: Normal
1: Flip up/down
* Unused (unusable) bit; same meaning in following pages

Interestingly, the GBC is able to store in its character (tile) map 256 tiles dedicated for use by the background map, 256 tiles dedicated for use with sprites, and another 256 tiles that can be used for either.

*Sprites + Sprite Attribute Table (OAM)*

In the Game Boy Color, sprites are graphics objects that are used to display tiles at arbitrary locations on the screen. Sprites are commonly used to display tiles that move around frequently, such as the protagonist of a game, enemies, and objects that need to move precisely, like bullets. The GBC allows for a maximum of 40 sprites that can be stored and displayed on the screen at once, but due to PPU limitations, only a maximum of 10 of those sprites can appear in any given scanline (screen row). These 40 sprites are stored in an area of memory called the OAM (object attribute memory) from addresses 0xFE00 to 0xFEA0, and consist of 40 sets of 4 bytes blocks. The figure below shows the parameters that correspond with a single sprite.

Sprites have very similar parameters to background tiles. Sprites consist of the same two parameters that define a background tile (tile index and attributes), but sprites include an X and Y position, which is an absolute position in the large background map to place the sprite. This means that sprites will also move on the LCD screen according to the scroll registers that shift which part of the background map is displayed.



With the background map and sprites, the Game Boy Color has only two different graphics layers. Typically the background is behind the sprites, though sometimes the sprites can be placed behind the background to hide them, or off the LCD screen (but somewhere else on the background map) to achieve the same effect. In both the sprite and background attributes, each object specifies if it thinks it should be on top. A table summarizing the different modes and which object remains visible is shown below:

| ***Display Priority Flag*** | | Dot Data | | Screen Display | |
|---|---|---|---|---|---|
| ***BG*** | ***OBJ*** | OBJ | ***BG*** | ***Palette*** | Data |
| 0: Use OBJ priority | 0: Priority to OBJ | 00 | 00 | BG | 00 |
| | | 00 | bg | BG | bg |
| | | obj | 00 | OBJ | obj |
| | | obj | bg | OBJ | obj |
| | 1: Priority to BG | 00 | 00 | BG | 00 |
| | | 00 | bg | BG | bg |
| | | obj | 00 | OBJ | obj |
| | | obj | bg | BG | bg |
| 1: Highest priority to BG (by character) | 0 1 | 00 | 00 | BG | 00 |
| | | 00 | bg | BG | bg |
| | | obj | 00 | OBJ | obj |
| | | obj | bg | BG | bg |

* obj and bg represent dot data (01, 10, 11) for OBJ and BG, respectively.

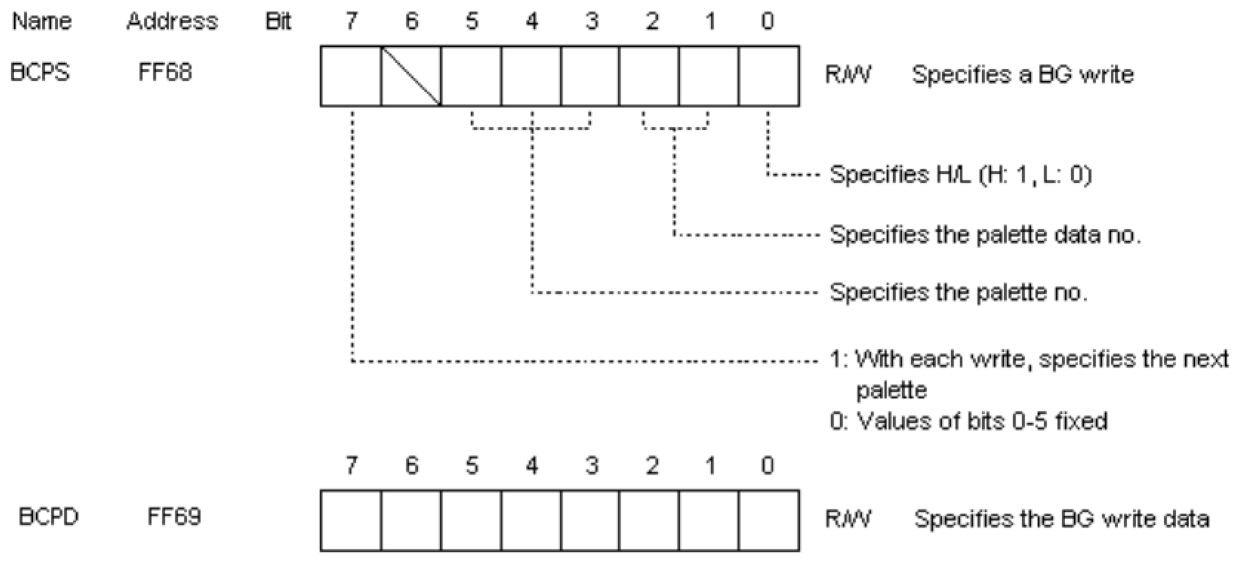
Notice that if the distinct color of either the sprite or the background tile is 0, the GBC treats that pixel as transparent and shows the other object instead.

Finally, in cases where multiple sprites overlap in the same pixel, the GBC resolves conflicts by choosing the sprite with the lowest OAM index to be the candidate for comparison against the background tile.

*Color Palette*

The Game Boy Color LCD screen is capable of showing 15-bit color, with 5 bits dedicated to each channel (R, G, B) for a total of 32,768 colors available to choose from. However, only a small subset of these colors can appear on the screen at any given time. The GBC has a color palette for both the sprites and background tiles. The color palette consists of 8 palettes, each with 4 colors available. As mentioned before, the tiles already specify in their data which of the 4 colors to use within a palette, and the attribute (for either the sprite or background tile) specifies which color palette to look in.

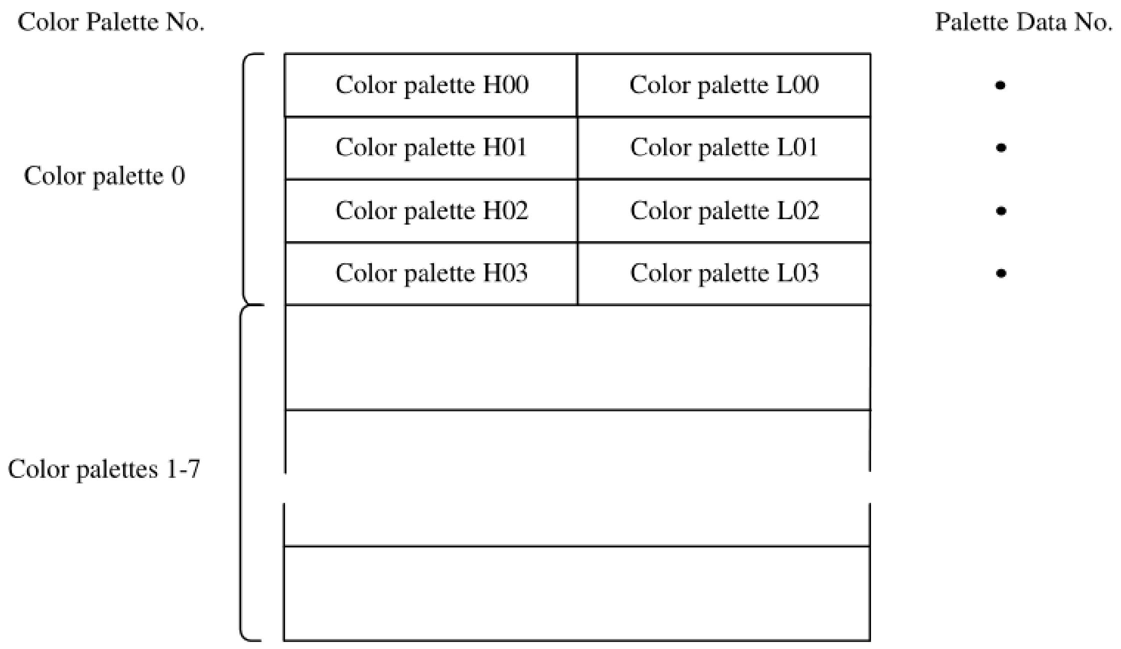
The color palettes are interestingly stored in a location where they are not directly accessible through memory-mapped I/O. Only an interface use to read and write to the palettes is exposed as memory-mapped registers: a specification register, which explains what palette location to access, and a data register which can be read from or written to. In addition to direct writing, the color palettes also have an auto-increment specification register mode, allowing for faster palette writing.  The details of the two registers (spec. and data) for the background palette are shown below, though the object (sprite) color palette operates in an identical fashion, just accessed using different memory addresses.



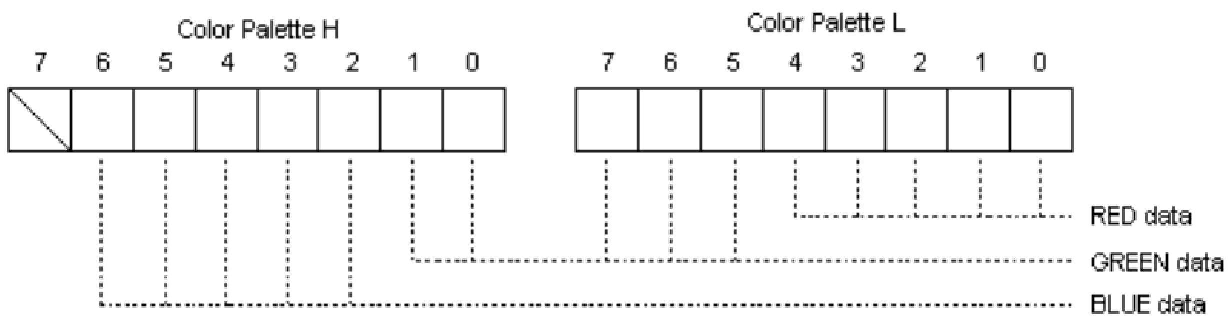
Name
Address
Bit
7 6 5 4 3 2 1 0
BCPS
FF68
R/W
Specifies a BG write
Specifies H/L (H: 1, L: 0)
Specifies the palette data no.
Specifies the palette no.
1: With each write, specifies the next palette
0: Values of bits 0-5 fixed
7 6 5 4 3 2 1 0
BCPD
FF69
R/W
Specifies the BG write data

Below is what the color palettes look like, taking into account the H and L registers.  A single entry in the palette table is composed of a concatenation of the {H, L} values.

## 1. BG Color Palettes



Finally, the colors are mapped onto the palette data word as follows, using 5 bits per color channel:



*Graphics-related Memory-mapped Registers*

Finally, the Game Boy Color's PPU maintains and modifies a few memory-mapped registers in order to inform the CPU of the PPU's activity. The most important of these are the LCDC register, and the STAT register.

The LCDC register controls various parameters, such as enabling sprites, which map area to select from for background, and more as summarized below:

## FF40 - LCDC - LCD Control (R/W)

```
  Bit 7 - LCD Display Enable             (0=Off, 1=On)
  Bit 6 - Window Tile Map Display Select (0=9800-9BFF, 1=9C00-9FFF)
  Bit 5 - Window Display Enable          (0=Off, 1=On)
  Bit 4 - BG & Window Tile Data Select   (0=8800-97FF, 1=8000-8FFF)
  Bit 3 - BG Tile Map Display Select     (0=9800-9BFF, 1=9C00-9FFF)
  Bit 2 - OBJ (Sprite) Size              (0=8x8, 1=8x16)
  Bit 1 - OBJ (Sprite) Display Enable    (0=Off, 1=On)
  Bit 0 - BG Display (for CGB see below) (0=Off, 1=On)
```

The STAT register allows the CPU to request interrupts when the video hardware is performing certain tasks, including when the LCD is in the vertical or horizontal blanking period, when the OAM is being searched, or when a particular scanline has been drawn. The register is shown in more detail below:

**FF41 - STAT - LCDC Status (R/W)**

```
  Bit 6 - LYC=LY Coincidence Interrupt (1=Enable) (Read/Write)
  Bit 5 - Mode 2 OAM Interrupt         (1=Enable) (Read/Write)
  Bit 4 - Mode 1 V-Blank Interrupt     (1=Enable) (Read/Write)
  Bit 3 - Mode 0 H-Blank Interrupt     (1=Enable) (Read/Write)
  Bit 2 - Coincidence Flag  (0:LYC<>LY, 1:LYC=LY) (Read Only)
  Bit 1-0 - Mode Flag       (Mode 0-3, see below) (Read Only)
            0: During H-Blank
            1: During V-Blank
            2: During Searching OAM-RAM
            3: During Transfering Data to LCD Driver
```

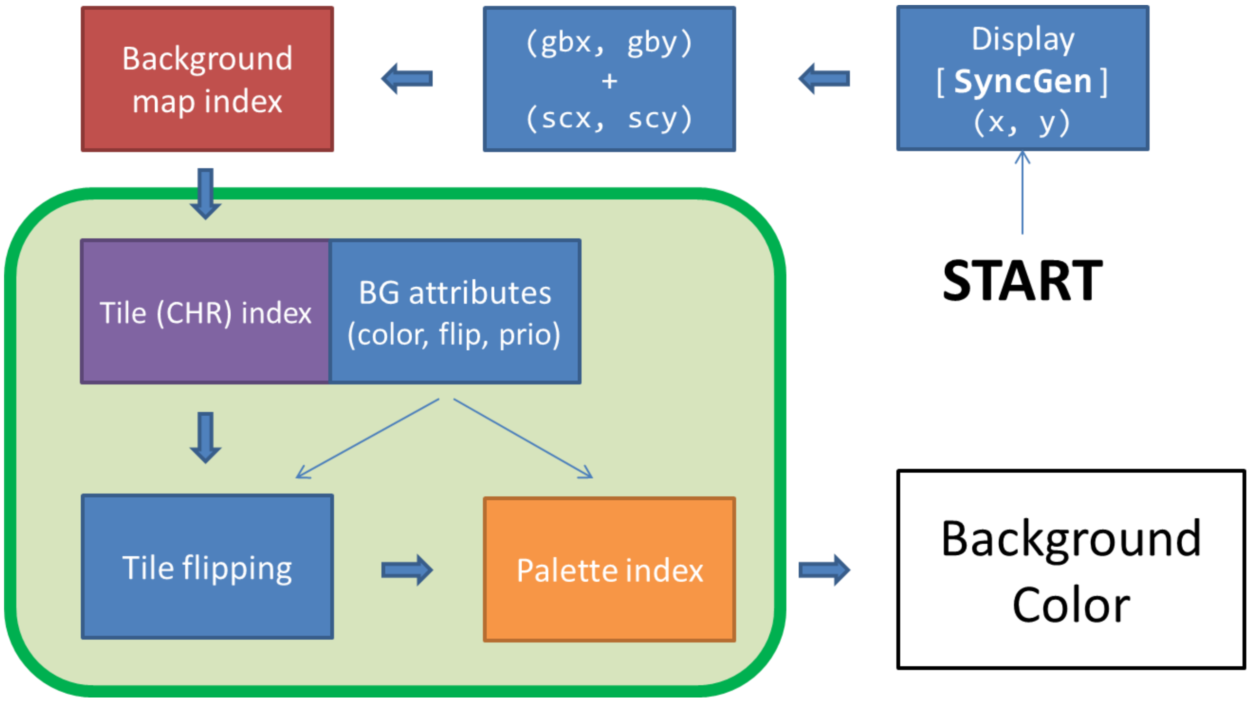
```
The two lower STAT bits show the current status of the LCD controller.
  Mode 0: The LCD controller is in the H-Blank period and
          the CPU can access both the display RAM (8000h-9FFFh)
          and OAM (FE00h-FE9Fh)

  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_____
```

*Implementation*

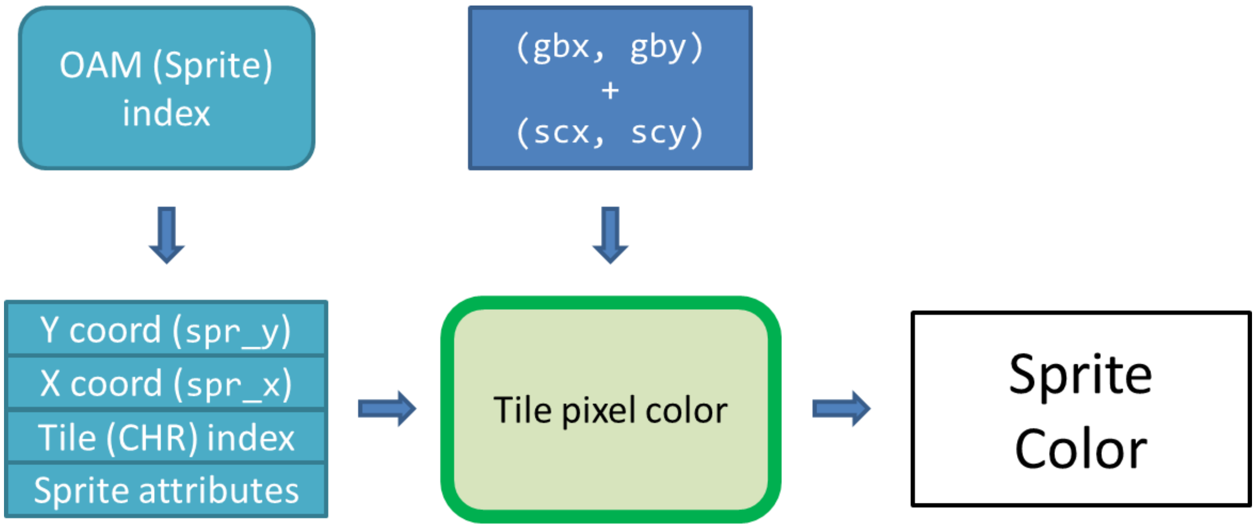
In our implementation of the PPU, we are able to display both backgrounds and sprites on the screen, using the appropriate colors from the color palette. We have also employed a framebuffer-less design in our PPU, so pixel colors are computed on-the-fly: there is no intermediate framebuffer where we store the screen's color data. The ultimate goal of our PPU is to determine, for each dot on the LCD screen, which color is present. Our PPU pipeline determines the color at a given pixel by taking the following steps:

1. Determine what background tile is present at this pixel
    a. Apply attributes (flips, colors) to determine what actual color the BG pixel is
2. Determine the candidate sprite that may also be located at this pixel, if it exists
    a. From the list of 40 sprites, determine which sprites this pixel is a part of
    b. Mark any sprites found to maintain only showing 10 per scanline
    c. Resolve conflicts among overlapping sprites to determine the candidate sprite
    d. Determine the color of the candidate sprite, using the same machinery as 1(a)
3. Determine which color, from the sprite or from the background, will show based on priorities

In order to perform step 1, we have built the following architecture:



The arrows show the various lookups that occur, using which pieces of data and in what order, in order to determine the background color. The green box is a component that is common to the sprite portion of the PPU pipeline and the background, so we've extracted the functionality and used it in both computations. It appears as the same green box in this diagram, which shows how sprite color is calculated [step 2(d)]:



After both colors are calculated, we can obtain the priorities using each of the attribute fields, and compare them as specified in the truth table figure in the sprite section of this document, taking into account transparent colors.

This portion of the PPU, which calculates colors given a tile index, performs pointer arithmetic to determine where to get subsequent information from, and it walks the various PPU tables to obtain that information. The original iteration of the PPU performed these memory lookups concurrently, as all addresses could be calculated combinationally. This worked fine as the PPU was being tested on static data (ROM), but as the PPU was integrated with the CPU, the ROM was reinferred as a RAM, which increased the synthesis time of the design to a concerning degree. The design eventually shifted over to reading from memory sequentially in order to mitigate these synthesis concerns. The sequential design used a faster clock in order to perform the memory reads just in time to display them.

The PPU also includes machinery to perform the task of 2(a) through 2(c) above: determining which sprites the current pixel contains. Our PPU included a two-stage pipeline to determine which 10 sprites were to be checked in a given scanline. At each pixel within the row, the X values of the 10 sprites selected for the current scanline would be checked to see if they occurred at the current pixel. The sprite with the lowest index would be selected as the candidate sprite to compare against the background. Meanwhile, the 10 sprites that occur in the *next* scanline out of the 40 total would be determined at the same time, sequentially. The list of 10 sprites would be determined by the end of the first 40 pixels drawn on the current scanline. At a horizontal blank, the next-10-sprites buffer would be inserted in the current-10-sprites buffer and then cleared to continue pipeline operation. This portion of the PPU was particularly timing-sensitive.

For further information, there are reams of documentation (specifically, the PanDocs) which describe the different parts of the PPU, fairly exhaustively. This documentation can be found here: http://nocash.emubase.de/pandocs.htm#videodisplay. In addition, the Game Boy Programming Manual contains a substantial amount of information as to the organization of the PPU, and it is where most of the images in the Background section come from.

## Display Controller

The display controller is designed to take prepared video data and go through the proper steps to show it on a monitor. In the original Game Boy Color design, there is an LCD controller which is similar in nature to our display controller.  Because we're targeting the Virtex-5 LX110T, which has a built-in DVI physical interface, we have decided to make a DVI controller as our display controller.  We have already reserved most of the FGPA's GPIO pins for the cartridge reader, which means we don't have sufficient pins to use for an external VGA adapter as some other teams have.  In addition, using the DVI controller allows us to use a DVI-to-VGA converter to display on VGA monitors.

As a starting point, we took Team DragonForce's console code (which included a Framebuffer, $I^2C$ initialization of the Chrontel DVI controller, and an H/VSync generator) and we began adapting it to suit our needs.  Their display controller is heavily integrated with their own Fast System Access Bus (FSAB) and a Direct Memory Access (DMA) controller, neither of which we required.  Our GBC system eventually required a DMA controller, but we designed a custom controller suited to our purposes.  We eventually only used only the $I^2C$ Chrontel register initialization module, and the H/VSync generator.  We used this sync generator to generate signals for a monitor resolution of 640x480.

Below is the timing diagram of the important signals that need to be sent to the Chrontel chip:

**CHRONTEL** **CH7301C**

## 5.1 Timing Information

### 5.1.1 Clock - Slave, Sync - Slave Mode

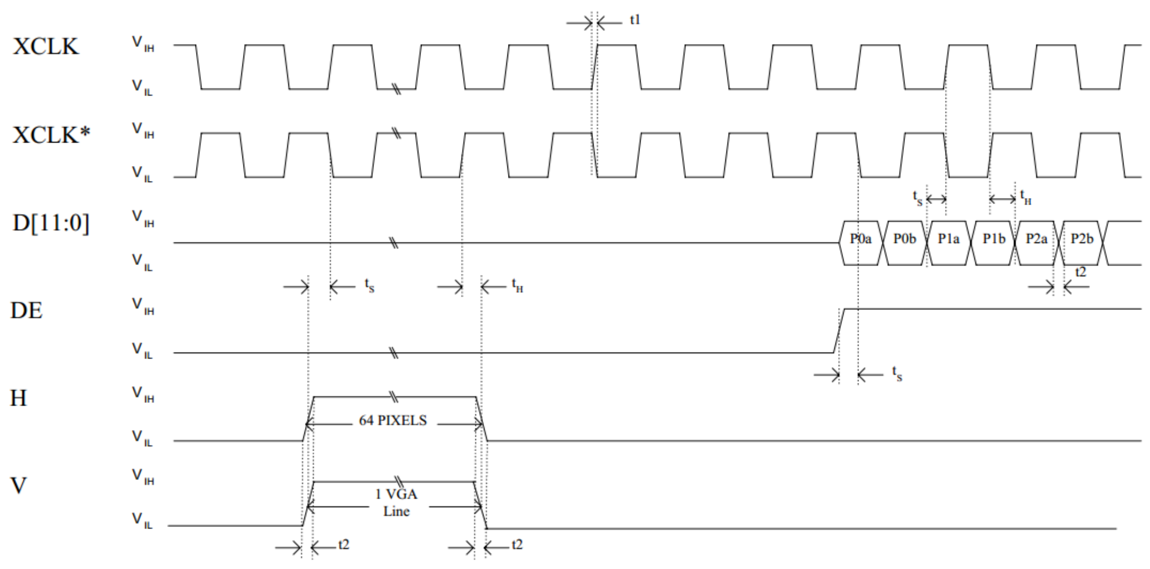


**Figure 6:  Timing for Clock - Slave, Sync - Slave Mode**

The Chrontel chip is capable of sending 24 bit color (8 bits per R, G, B channel) to a monitor. These 24 bits are sent in two batches, once on the positive edge of the clock and then again on the negative edge (dual-data rate transfer). The color which is sent for the current pixel comes directly from the PPU's output.

## DMA Controller

In order to facilitate efficient memory transfers between main memory and auxiliary components (cartridge, video RAM, etc.), a DMA controller is employed in the Game Boy Color system. The DMA controller specifically offloads memory intensive transfers from the CPU; instead of executing long sequences of load/store instructions, the CPU can program the DMA controller through a set of control registers (including source start address, destination start address, and transfer length). During this period, the CPU can either be idle (waiting for the DMA to complete), or it can perform other useful work while the memory is being transferred. For the Game Boy Color, the DMA controller can write directly to video memory, specifically in the OAM (sprite attribute table) section, or even a general purpose DMA may be performed.

### LCD OAM DMA Transfers

**FF46 - DMA - DMA Transfer and Start Address (W)**

Writing to this register launches a DMA transfer from ROM or RAM to OAM memory (sprite attribute table). The written value specifies the transfer source address divided by 100h, ie. source & destination are:

```
  Source:      XX00-XX9F   ;XX in range from 00-F1h
  Destination: FE00-FE9F
```

It takes 160 microseconds until the transfer has completed (80 microseconds in CGB Double Speed Mode), during this time the CPU can access only HRAM (memory at FF80-FFFE). For this reason, the programmer must copy a short procedure into HRAM, and use this procedure to start the transfer from inside HRAM, and wait until the transfer has finished:

```
   ld  (0FF46h),a ;start DMA transfer, a=start address/100h
   ld  a,28h      ;delay...
  wait:           ;total 5x40 cycles, approx 200ms
   dec a          ;1 cycle
  jr  nz,wait    ;4 cycles
```

Most programs are executing this procedure from inside of their VBlank procedure, but it is possible to execute it during display redraw also, allowing to display more than 40 sprites on the screen (ie. for example 40 sprites in upper half, and other 40 sprites in lower half of the screen).

## LCD VRAM DMA Transfers (CGB only)

**FF51 - HDMA1 - CGB Mode Only - New DMA Source, High**

**FF52 - HDMA2 - CGB Mode Only - New DMA Source, Low**

**FF53 - HDMA3 - CGB Mode Only - New DMA Destination, High**

**FF54 - HDMA4 - CGB Mode Only - New DMA Destination, Low**

**FF55 - HDMA5 - CGB Mode Only - New DMA Length/Mode/Start**

These registers are used to initiate a DMA transfer from ROM or RAM to VRAM. The Source Start Address may be located at 0000-7FF0 or A000-DFF0, the lower four bits of the address are ignored (treated as zero). The Destination Start Address may be located at 8000-9FF0, the lower four bits of the address are ignored (treated as zero), the upper 3 bits are ignored either (destination is always in VRAM).

Writing to FF55 starts the transfer, the lower 7 bits of FF55 specify the Transfer Length (divided by 10h, minus 1). Ie. lengths of 10h-800h bytes can be defined by the values 00h-7Fh. And the upper bit of FF55 indicates the Transfer Mode:

**Bit7=0 - General Purpose DMA**

When using this transfer method, all data is transferred at once. The execution of the program is halted until the transfer has completed. Note that the General Purpose DMA blindly attempts to copy the data, even if the LCD controller is currently accessing VRAM. So General Purpose DMA should be used only if the Display is disabled, or during V-Blank, or (for rather short blocks) during H-Blank.

The execution of the program continues when the transfer has been completed, and FF55 then contains a value if FFh.  For sake of simplicity and the fact that this design decision will not affect game correctness, we always run the HDMA in general purpose mode.

**Bit7=1 - H-Blank DMA**

The H-Blank DMA transfers 10h bytes of data during each H-Blank, ie. at LY=0-143, no data is transferred during V-Blank (LY=144-153), but the transfer will then continue at LY=00. The execution of the program is halted during the separate transfers, but the program execution continues during the 'spaces' between each data block.

Note that the program may not change the Destination VRAM bank (FF4F), or the Source ROM/RAM bank (in case data is transferred from bankable memory) until the transfer has completed!

Reading from Register FF55 returns the remaining length (divided by 10h, minus 1), a value of 0FFh indicates that the transfer has completed. It is also possible to terminate an active H-Blank transfer by writing zero to Bit 7 of FF55. In that case reading from FF55 may return any value for the lower 7 bits, but Bit 7 will be read as "1". For sake of simplicity and the fact that this design decision will not affect game correctness, we always run the HDMA in general purpose mode.

**Confirming if the DMA Transfer is Active**

Reading Bit 7 of FF55 can be used to confirm if the DMA transfer is active (1=Not Active, 0=Active). This works under any circumstances - after completion of General Purpose, or H-Blank Transfer, and after manually terminating a H-Blank Transfer.

**Transfer Timings**

In both Normal Speed and Double Speed Mode it takes about 8us to transfer a block of 10h bytes. That are 8 cycles in Normal Speed Mode, and 16 'fast' cycles in Double Speed Mode.

Older MBC controllers (like MBC1-4) and slower ROMs are not guaranteed to support General Purpose or H-Blank DMA, that's because there are always 2 bytes transferred per microsecond (even if the itself program runs it Normal Speed Mode).

## Audio Controller

The Game Boy Color has an audio controller which uses information in memory mapped I/O registers to produce audio. There are two output channels which can be used and manipulated. The console can produce four types of sound. Each of these sounds is produced on its own channel, and which sound channels are sent to the output channels is specified by a memory mapped register. The sound channels are as follows:

Sound Channel 1: Tone and Sweep (A sound that transitions from one tone to another)

Control Registers (mmapped to 0xFF10-14)

NR10 – Sweep register – defines sweep duration and direction

NR11 – Sound length and wave pattern duty register

NR12 – Volume envelope register

NR13 – Low 8 bits of initial frequency

NR14 – High 3 bits of initial frequency

Sound Channel 2: Tone (Same as channel 1, but without a sweep)

Control Registers (mmapped to 0xFF16-19)

NR21-4 – Same as NR11-4

Sound Channel 3: Wave Output (Plays digital sound defined by 32 digit wave pattern)

Control Registers (mmapped to 0xFF1A-1E)

NR30 – Sound on/off register

NR31 – Sound length

NR32 – Sound level

NR33-34 – Same as NR13-4

Additionally, 0xFF30-3F store the wave pattern data consisting of 32 4bit samples

Sound Channel 4: Noise (Outputs white noise with a partially controllable tone and harshness)

Control Registers (mmapped to 0xFF20-23)

NR41 – Sound length

NR42 – Volume envelope register

NR43 – Polynomial counter register

NR44 – Counter/consecutive register

Registers NR50-52 control the output channels as follows:

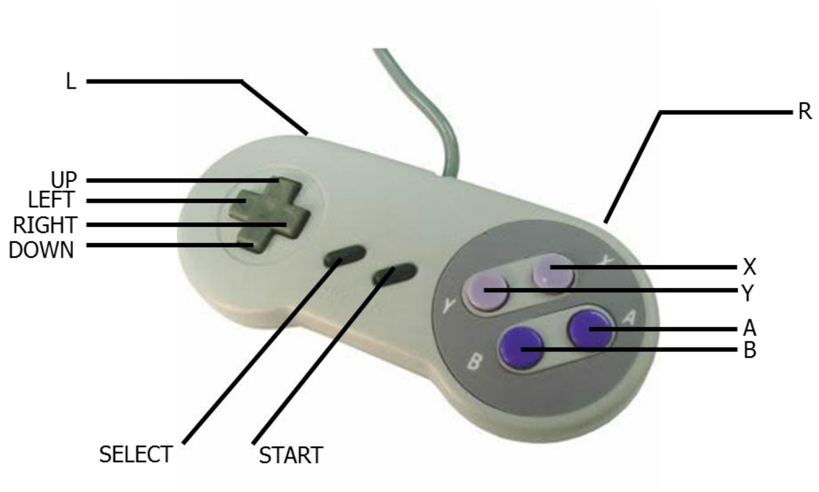
NR50 – Channel control – adjusts volume of each output channel

NR51 – Sound output select – selects which sound channel goes to which output channel

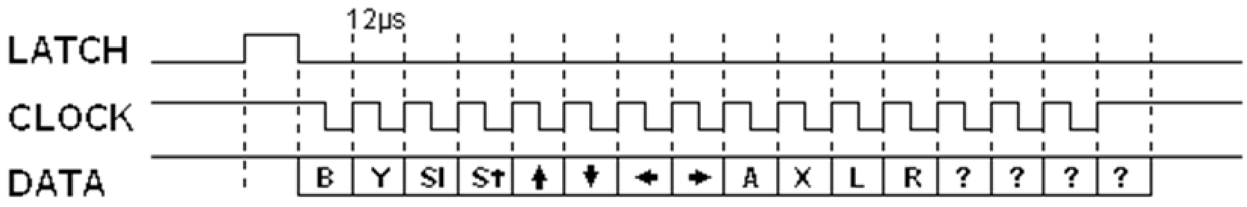
NR52 – Sound on/off – can be used to turn off sound, which saves power

The audio controller which we will implement needs to be a bridge between the registers defined here and the AC'97 audio driver. These values are put into memory by the CPU (based on instructions from the cartridge). We will need to continually pull the values out, interpret them, and convert them to waveforms supported by AC'97. This should be fairly straightforward, since any math necessary to create the waveforms is outlined in the documentation we have.

## SNES Controller



As input to our system, we will be using the controller for the 1991 Super Nintendo Entertainment System (SNES). The SNES controller was chosen due to its similarity to the Game Boy Color's control layout, simplicity as compared to more modern wireless controllers, and being easily obtainable. The SNES controller uses a serial communication interface and a shift register to output which buttons are being pushed at the time of data transfer. To power the SNES controller, it must be supplied +5V.



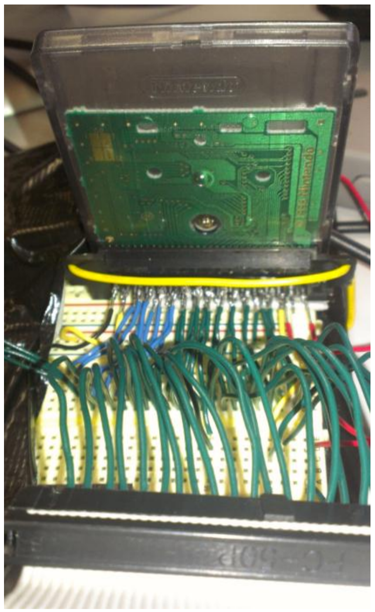
The serial communication and shift register works as follows: (1) a 12 microsecond positive pulse on the latch pin, (2) 6 microseconds of delay between the first negative edge of the clock and the latch is sent,

(3) a total of 16 negative clock edges are sent on the clock pin reading a value out of the shift register during each negative edge, (4) 16.67 milliseconds of no reading must occur before you may perform another cycle. The shift register is updated on the positive edge of the clock (or the positive edge of the latch in the first bit's case). The buttons are negatively asserted, that is, the voltage of a pushed button will be 0V. The last four bits of the data word are always high and hold no button specific information. This 16 bit data word is interpreted by the FPGA as I/O and will also be stored and read from the I/O portion in memory.

To interact with memory, the controller loads four values at a time to the JOYPAD register (FF00). Bit 7 and 6 of FF00 are not used, bit 5 is set if you want to read push buttons, bit 4 is set if you want to read direction buttons, and bits 3 through zero are set to down, up, left, right/start, select, B, A depending on bits 5 and 4.

## Cartridge Reader

For full implementation of a Game Boy Color, any Game Boy Color or Game Boy cartridge needs to be compatible with our system. To accomplish this, we chose to use a Game Boy Color cartridge reader. To obtain said cartridge reader, one was de-soldered out of a working Game Boy Color, re-soldered, and wired into the Virtex 5 board.



The cartridge reader has 16 address pins, 8 data pins, a $V_{DD}$ pin for power (+5V), a reset pin (also tied to the +5V line), a GND pin, a chip select (CS) pin, a negatively asserted read (RD) pin, a negatively asserted write (WR) pin, a $V_{in}$ pin for audio (not used), and a clock pin (not used).

The Game Boy Color cartridges will be mapped to specific locations in memory as discussed in the memory portion of this paper.

| | |
|---|---|
| 0x0000 – 0x3FFF: | Cartridge ROM Bank 00 (Read Only) |
| 0x4000 – 0x7FFF: | Cartridge ROM Bank 01 – NN (Read Only) |
| 0x2000 – 0x3FFF: | ROM Bank Number (Write Only) |

To read from the cartridge, WR is held high and RD is held low. Data corresponding to the address lines is sent over the data lines to the FPGA where it can then be interpreted by the CPU. To write to the cartridge, WR is held low and RD is held high. The data on the data lines is written into the memory specified by the address lines.

In order to select a bank for the second area of cartridge ROM, a special write must be performed. WR and RD must be held low to indicate a write. The address lines must hold an address in the ROM Bank Number portion of memory (as shown above). When this is done, the Memory Bank Controller (MBC) interprets the write as a bank switch to the value of data being written. For example, an attempt to write 0x03 to address 0x2100 would not actually write any data, but it would instead switch bank 3 into the second ROM Bank space.

Three different versions of MBCs are supported by the Game Boy Color (MBC1, MBC2, and MBC3). MBC1 can contain 125 ROM banks, MBC2 can contain 16 ROM banks, and MBC3 can support 128 ROM banks. Also, MBC3 also supports an internal day counter due to an internal 32.768 kHz Quartz Oscillator and an external battery (should the clock continue to tick while the Game Boy Color is turned off).

## Final Demonstration

*What we wanted to accomplish:*

For our planned demonstration, we wanted to play at least one game out of a cartridge (our ideal choice was Pokemon Blue), control it with our SNES controller, and display it using our PPU and DVI controller onto the monitor.

*What we successfully accomplished:*

We could play a custom ROM (a ROM that we designed from scratch) to play a custom game, controlled it with the SNES controller, and displayed it onto the monitor using our PPU and DVI controller.

*Why we could not accomplish what was originally planned:*

The open core TV80 had many bugs that kept us from executing code completely. In order to demonstrate the working Game Boy Color, we needed to avoid certain instructions that our CPU could not execute (this is why we created a custom ROM/game). Cartridge bank switching also did not work all the time. Some banks would never work, and some would always work (and the working/nonworking banks would be consistent for a given game). This kept us from running large games out of a cartridge. Finally, our PPU displayed images onto the monitor combinationally, which, when synthesized with the larger design, would increase synthesis time by an unrealistic amount of time. So our game had a static VRAM.

*Possible fixes:*

<u>CPU</u>

- With more time to debug, we may have been able to flush out the bugs, but seeing as how many bugs we have found, it may have been better to write most of the CPU ourselves

<u>Cartridge Reader</u>

- Since all of the bank switching is done in the hardware, the problem must exist in the hardware
- The pins to the cartridge reader may not be driven correctly (although we did some testing with drive strength)
- The cartridge reader itself may have been broken (we purchased a used Game Boy that did not always work)

<u>PPU</u>

- We created a sequential PPU, but we could not run it fast enough to have all of the data prepared for the Chrontel chip (we need to make 6 memory reads by a 25MHz clock, so a 100MHz clock is not fast enough). Backgrounds work (4 reads) but sprites do not (2 reads).
- With a faster clock or a different DVI/VGA controller, the timing constraints could be achieved

## Individual Report: Mike Gardner

What I contributed:

I built the hardware for both the cartridge reader and the SNES controller setup, coded the modules that controlled all controller and cartridge communication, coded all of the timing structure of the DMAs, designed the PPU (teammate did the actual coding), and handled a lot of the CPU debugging with teammate.

What I enjoyed/learned:

When we started, I knew I was the member with the most hardware experience so I knew that I would be doing the controller and cartridge reader. But it was in the other aspects of the project that I learned the most. For example, I had no experience in graphics processing at all, but when I took over the designing of the PPU, I learned an incredible amount. I also got a lot more experience with debugging. When we ran into some difficult hurtles, pushing through them became one of my strong suits for the team. But if I can take anything away from this course, it would have to be what I learned about managing people (not necessarily the work). Bringing my teammates together, handling design disputes, always making sure that everyone is making some sort of forward progress, and determining when a teammate isn't communicating when they are stuck are some of the skills I have learned and developed over this semester.

## Individual Report: Arjuna Hayes

When we first set out to make the Game Boy Color, it seemed like a reasonable objective to accomplish. We had a pretty reasonable place to start from and a reasonable amount of documentation. While we didn't finish, I'm still happy with what we accomplished and am glad to have had the opportunity.

Integrating the CPU and building the memory components was my main task for this project. This included building the infrastructure for the CPU to interface with RAM and the cartridge as well as designing the memory itself. Later, my role expanded to building the DMAs and interrupt handler as well as building the final demo.

Initially, we started with a Z80 implementation we found on opencores (the TV80). Initially, I thought I would have to modify it to fit the Game Boy Color specifications. However, it actually had a GBC mode that used the modified instructions. Since the TV80 is a fairly robust Z80 implementation, I initially assumed that the Game Boy Color mode would also work well. Unfortunately, this assumption was incorrect. As the semester progressed, I found that more and more of my problems originated with faulty CPU instructions. Some of the instructions were flat out incorrect or partially implemented, interrupts were handled strangely, and the status indicator flags were only half-modified (the GBC used a different set of flags). Because of this, I had to spend much more time debugging the CPU than I had planned. As more our design was implemented, we were able to test more of the CPU. We would run a game and compare the execution of our FPGA to that of the emulator. After running through a few thousand instructions from a bunch of different games, we were able to work out most of the bugs in the CPU. I'm still not confident we got to them all, since our debugging of the CPU was interrupted by a bank switching problem and, shortly thereafter, time constraints.

I designed the DMA controllers and interrupt handler because they were closely tied in to my preexisting tasks and, as such, I would have the easiest time integrating them. Both had their fair share of kinks to work out, but both were interesting and (I believe) functional. The hardest part of all of this was trying to test functionality. I had to write my own programs for testing, and when something went wrong, it was challenging to determine which unit broke first (did an instruction get executed incorrectly? Is there a timing problem with the interrupt handler? Did I write incorrect code?). Much like my previous hardware designing experiences, when debugging, finding the bug is 95% of the work and fixing it is the other 5%.

My last task was writing a demo game for the final display. That was probably my most entertaining job. While it was annoying to write a game in modified-Z80 assembly, it was very satisfying to see code I wrote running on a system the team designed.

I really enjoyed this class. Though we didn't finish, I had a lot of fun in the process and learned a lot. While I've had previous experience doing hardware design, I'd never done something of this scale. My team worked really well together as well; I think our skillsets complemented each other pretty well. In retrospect, I wish we knew how big of a time investment fixing the CPU would be. Also, I know it's cliché, but we would have had a better time at the end of the semester if we were more organized earlier on.

## Individual Report: Costas Akrivoulis

Coming into the course, I was very excited to build an entire game console on an FPGA. Before the course had begun, our team bounced around a few different project ideas, and after hearing about the results of past projects in the first week, we finally settled on the Game Boy Color, a handheld console that held immense nostalgic value to all of us. Our entire team had just taken Computer Architecture (18-447) the prior semester, so we were ready to step up and tackle the challenges of building a full console system. By the end of the semester, although we did not have our ideal working product, I'm very pleased with the understanding I gained of the Game Boy Color system, a device that comprised many of my fondest childhood memories.

My role in the FPGBC team was focused entirely on the graphics subsystem, which included the Pixel Processing Unit, the display controller, and a bit of associated research regarding memory creation on the FPGA. In addition, I supported the team by improving our infrastructure, replacing OpenSuSE with Ubuntu on our lab machines and configuring AFS and the Xilinx ISE. This allowed us to install custom packages that improved our workflow, including debugging and scripting tools. With regards to video, I initially started with a previous team's display controller and removed the tightly integrated components which we did not need. This left us with a module that allowed us to send video data to the monitor. Having just spent a semester learning about CPU design, building the PPU was an engaging task that I thoroughly enjoyed, since it was a new challenge. In addition, the allure of visual output and the satisfaction of a working design was rewarding, though I had equally depressing experiences on the journey towards that working design. I began by writing the component that could correctly display the background map of a static image (the title screen to Pokemon), and then later I added a component to render sprites. Unfortunately, when it came time to integrate the PPU with the CPU and memory, the PPU needed to be slightly modified. Coupled with other technical issues the team was facing and the looming demo deadline, the older graphics subsystem was used to demonstrate our project.

Looking back on the course and our team's decisions, I think we made a great effort toward building the Game Boy Color: we took advantage of the literature available online about the GBC and spent a lot of time researching subsystems, we designing subsystems before implementing them, and we used various debugging and scripting tools to improve our productivity. However, there are a few things I would improve if we were to reset the semester. Personally, I didn't take advantage of hardware simulation, which could have cut down substantially on development time. My debugging cycle consisted of synthesizing a design and using ChipScope to verify it, which led to slower iterations than simulation would have. Additionally, we did not integrate our subsystems as soon as possible, in order to work quicker in isolation (shorter development cycles, including synthesis times). I would probably have reconsidered this decision if we were to do it over, because of the integration pain and stress late in the semester. We looked at the previous GBC team's report at the beginning of the semester to judge if we could complete it, but I wish I had looked at their report later in the semester as well, when we were deep in design or implementation and could understand their technical discussion better. Overall, the course was a fantastic learning experience, and I hope to revisit and complete our efforts in the near future.