Gameboy Emulation In C++

By Brecht Uytterschaut

Digital Arts and Entertainment

Start 24/09/2019 End 04/11/2019

Contents

Abstract

In this paper we will discuss whether to take an object oriented or procedural approach. We will look at the CPU, discussing its specifications, the register layout and emulation accuracy differences. We will also be looking at the way memory busses are implemented and emulated as well as the differences between the busses handling memory transfers. The CPU section will end with an introduction to the way opcodes are laid out and implemented as well as a few interesting findings in this regard. After this we will discuss the PPU talking about its different layers and their implementation ending with a small introduction as to how the Input scheme worked on the original Gameboy and how we'll be implementing it.

Introduction

With the rise of bigger and more resource intensive games, I think it's important for programmers to go back to the roots and learn how games of 32KB fascinated an entire generation, a time before micro transactions and day one DLCs, a time where game developers wrote their games in beautifully raw assembly, no game engines, no sexy UIs to aid in development, just a bunch of hard-core low level gods working to bring their procedural work of art into the world!

This paper will focus on writing a library that allows for parallel accelerated Gameboy emulation with the purpose of training neural networks. A few limitations were set before starting this project. The accompanying case-study only has to support Tetris but be expandable enough to allow for an easy integration of other games' requirements. Sound won't be covered at all.

Unless you're truly interested in this topic, I advise you to read something else. I originally wanted to write this paper in a way that would inspire everyone unfortunate enough to end up with a copy to actually try building their own emulator as it's an amazing journey that I can not recommend enough! Sadly, this paper has been censored to `bit`s in order to conform with the soulless-formal format students are required to deliver.

If you are indeed planning on writing your own emulator, first of all, congratulations, great idea, second, have a look at a much better and more detailed paper written by Justin Micheal Richeson called "Anatomy of a Hardware Emulator" (10.15781/T23R0Q56D), codewise, Codeslinger has a simplistic working (albeit messy) open-source Gameboy emulator worth taking a look at. Good Luck and enjoy!

OOP or Procedural

> *The problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle.*
> -Joe Armstrong

Programming wasn't always object oriented..
A fact that the newer generation of programmers don't seem to really care about.

A lot of "big names" in the programming community seem to be all too eager to drag Object Oriented programming through the mud. While their remarks are not necessarily untrue, I do take issue with their generalization. In my opinion, OOP has its valid use cases as does procedural programming.

Object oriented programming comes with more baggage, which can hurt overall efficiency if used "carelessly". This is why I find it important to know what this baggage entails and how to limit it. A valid argument to this would be that the efficiency drag is in most use cases very limited.

As with everything, especially in programming, you need to know how it works, if you know how it works, are aware of the caveats and are able to justify using the system (paradigm in this case) regardless, then you should use it.

I'll be using an Object Oriented approach because:

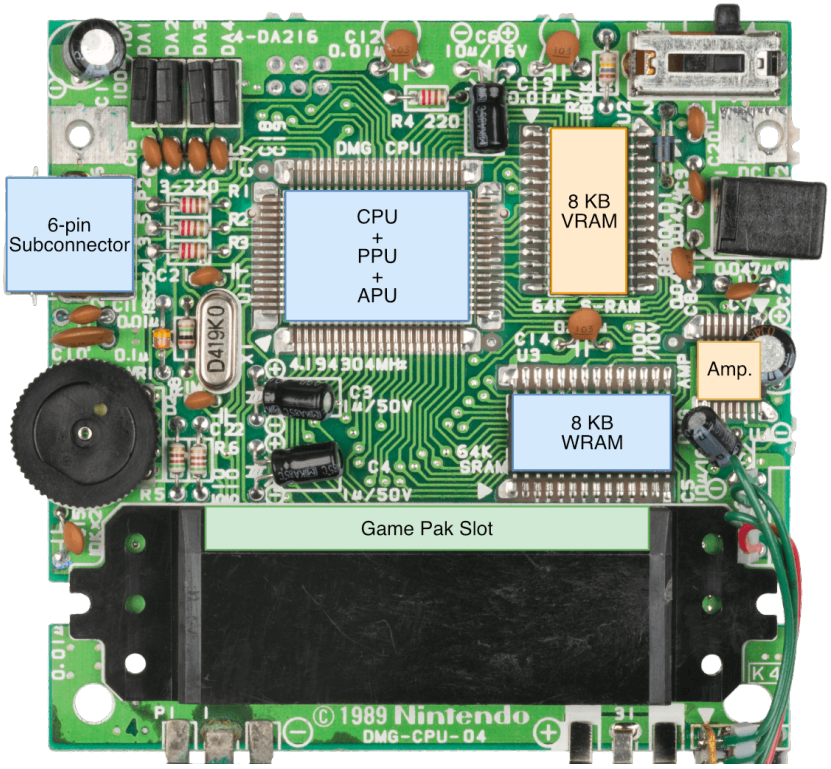
- C++ is object oriented focused, however, procedural programming is still possible.
- It's widely used in the gaming industry
- I'm looking for a job in the gaming industry

This being said, it is definitely possible to create this project in a procedural manner.

The CPU

The CPU is a **Sharp LR35902** (a mix between the Zilog Z80 and Intel 8080) which runs at 4.19MHz. The LR35902 has **8 8 bit** registers, one of those is used for flags, the remaining 7 can be used directly. Additionally, 16 bits are available for keeping track of the **S**tack**P**ointer and another 16 for the **P**rogram**C**ounter. Even though there are 8 bits allocated for the available flags, the LR35902 only has 4 valid flags: **Z**(Zero, set when the latest instruction resulted in 0, if not it is cleared (set to 0)), **N**(Subtract, set when the instruction is a subtraction, if it's an addition it's cleared), **H**(Half-Carry, set when a carry was produced in bit 3(the 4th)), **C**(Carry, set when a carry was produced in bit 7 (8th)). (Fayzullin, et al., n.d.)

Register A is the accumulator, which is increased/decreased during addition/subtraction. The higher and lower registers can be paired, allowing to store 16 bits, useful for memory addresses.



1 The Gameboy Motherboard with chip markings
Taken from (Copetti, n.d.)

Clock Cycles

The Sharp LR35902 runs at 4.194304MHz meaning that every second, this baby does 4194304 cycles! This is something we need to take into account if we want to do a realistic emulation, however, we're creating a high speed emulator, which can hopefully go 1000 times faster than the original speed.

The Game boy runs at approximately 60 frames per second meaning that every frame accounts for 70000(4194304/60) cycles. In order to accelerate the Game Boy, we can increase the amount of cycles in a second. However, if we increase the frame rate as well, we can end up with *time jumps* (The game starts; Draws a frame; runs for way too many cycles; Draws a frame; Game over). To solve this, we'd simple have to increase the frame rate as well.

Note that this will speed up the game due to the games being fps/cycle dependent, today in games, when a ball has to move forward, we don't add x meters every frame, instead, we use a delta time value to determine the duration since we last updated.

Increasing the frame rate will decrease performance. Because our framework will be used to train neural networks, we can remove drawing to a window al together. The entire framework will be written as a static library exposing the framebuffer, so it'll be up to the developer to decide what to do with the data (Whether it'll be rendered or not)

Accuracy

There are two common levels of accuracy, T-Cycle and M-Cycle accuracy.

Every M-cycles is equal to 4 clock cycles, ever opcode takes a multiple of 4 clock cycles, so when we're emulating on M-cycle accuracy, we simply add the duration of each opcode to a counter.

T-cycle are synonymous with clock cycles, so technically 1 M-cycle is 4 T-cycles. When using T-cycle accuracy you're expected to emulate the cycles within each opcode, some opcodes work on the dereferenced values of registers, actually interacting with ram memory directly, these kinds operations take a specific amount of clock cycles.

Emulating on T-cycle accuracy is very challenging but required for certain games like Pinball Fantasies and Deluxe which require a one cycle accuracy between checking whether an interrupt has occurred and which interrupt has occurred. This is most likely unintended behavior that happened to work on the Gameboy rather than a conscious design decision. A more in-depth analysis of this *issue* can be found on https://mgba.io/2018/03/09/holy-grail-bugs-revisited.

If you're looking for an engineering beauty check out Battletoads for the NES, which requires extremely tight timing and was designed that way.

Battletoads
Infamous among emulator developers for requiring fairly precise CPU and PPU timing (including the cycle penalty for crossing pages) and a fairly robust sprite 0 implementation. Because it continuously streams animation frames into CHR RAM, it leaves rendering disabled for a number of scanlines into the visible frame ***to gain extra VRAM upload time*** *and then enables it. If the timing is off so that the background image appears too high or too low at this point, a sprite zero hit will fail to trigger, hanging the game. This usually occurs immediately upon entering the first stage if the timing is off by enough, and might cause random hangs at other points otherwise.*
--Taken from (nesdev.com, n.d.)*'s Tricky-to-emulate games*

I will be emulating on M-Cycle accuracy for this project since the main focus of my framework is efficiency over accuracy.

(Emulation Development)

Busses

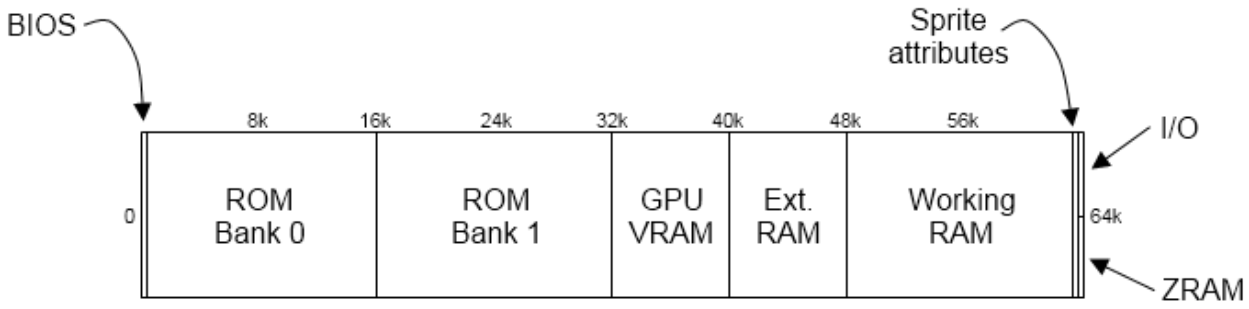
There are 2 busses that we'll need to emulate, the **address bus** (**16 bits**, 64KB addressable) and the **data bus** (**8 bit**). The address bus specifies which memory address needs to be either read or written to, the data bus contains the actual value. (Fayzullin, et al., n.d.)

The way that we'll be doing this is actually quite simple, the only thing these busses do is handle data retrieval and editing, which we can easily emulate by reading or writing to an array in a controlled manner (using a function for example). That being said, cartridges are not completely standardized in the sense that there are many ways for developers to "hack in" custom functionality. This won't be part of my emulator as this depends on the cartridge being used.

Memory Addressing

The address bus is 16 bit wide allowing for up to 64KB of addressable memory. This memory is spread across the entire console and cartridge meaning we'll have to emulate a control function which will decide what part of the emulator needs to be addressed.

This is how the memory layout looks like in your off-the-shelve Gameboy



*Stolen from Imran Nazar's **GameBoy Emulation in Javascript***

[0000-3FFF](16KB) Cartridge ROM (bank 0)
    [0000-00FF] BIOS: A 256 byte long block which initiates the Game
    [0100-014F] Cartridge Header: Contains information about the game
[4000-7FFF](16KB) Cartridge ROM (bank !0)
[8000-9FFF](8KB) Video RAM
[A000-BFFF](8KB) Cartridge RAM (if available)
[C000-DFFF](8KB) (working)RAM
[E000-FDFF](7,680 bytes) **Echo RAM**: Redirects to the actual RAM
[FE00-FFFF] CPU Internal RAM
  [FE00-FE9F](160 bytes) Sprite information
  [FF00-FF7F](128 bytes) I/O control values
  [FF80-FFFF](128 bytes) **Zero-Page RAM**: A high speed area

Two interesting sections that you might've noticed is Echo RAM and Zero-Page.

Echo RAM is an artifact of the way RAM is wired to the address bus. Everything written to this address range is directly written *adrs-C000*, in other words, there is no unique memory behind this range

Zero-Page RAM, in this case, is interestingly enough implemented at the end of the Memory range. The memory residing in this area is part of the CPU itself, warranting it's high speed access. This area is often referred to as **H**igh**RAM**.

If you do the math, you'll notice that 96 bytes are unaccounted for. Those missing bytes reside at [FEA0-FEFF]. What do they do? Well, from what I could find, no one seems to really knows.. I have a hypothesis though, We've got 8 8-bit registers, a 16 bit stack pointer and a 16 bit program counter, which happens to total 96 bytes. This could of course be a coincidence.

Most websites and people I've talked to seem to think that the range is just completely unconnected/implemented, that the data returned is just the value of the pins in the state they happen to be in. Very mysterious indeed!

Info gathered here (Nazar, n.d.)

Opcodes

The LR35902 uses 8-bit opcodes, there are 245 **valid** instructions in the main instruction set, however, this processor also has an extended instruction set at its disposal with 256 valid opcodes, bringing the total of opcodes that need to be implemented to 501.

Every opcode has an address, for example 0x87 is ADD A,A. The extended set, however, can't be accessed in this manner, due to every opcode being 8 bits wide meaning we can only have a total of 256 opcodes

The way this is solved is by prefixing every extended opcode instruction with 0xCB followed by the desired extended opcode value. 0xCB is not mapped in the main instruction set.

That's a lot opcodes, granted most of them have similar behavior (Especially in the extended set) allowing for the use of some *shortcuts*, but even then, it'll be a lot of functions.

We won't be implementing them all, the purpose of this project is to get one single game running at 1000 times the original speed. The game I've chosen for this is good old classic Tetris!

Tetris is a fairly simple game. It's only 32KB big which allows for it to be written onto a cartridge without the need of additional memory banks (Bank 0 and 1 are the minimum in every cartridge).

In order to get an idea as to how many opcodes I'll be working with, I've written a simple python script. This script read 23893 bytes of which 259 were invalid opcodes. The only thing this script did was read the first valid opcode (At line 0x100, more on this later) look it up in the instruction set and advance our position in the file *x* bytes. Where *x* is the amount of bytes the opcode requires.

I didn't write any logic to verify if the opcode that's being analyzed is in fact an opcode and not just data that happens to have an opcode representation. There are a few architecture tricks (some of which are architecture dependent) that one can apply to alleviate this problem, however, due to the limited number of false positives, this isn't worth the trouble.

My script recognized that **284** of the 501 opcodes are being used and that of those 284, **37** are unique.

Just for funsies, I decided to run Tetris trough a disassembler (mattcurrie, n.d.) *which (After applying some regex) gave the same result.*

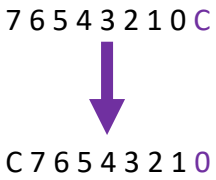
Interesting findings

The first opcode that proved interesting was **SRA**. SRA stands for ***S**hift **R**ight **A**rithmetic* which means that every bit is shift 1 place to the right. There's a left variant of this opcode as well (SLA). Arithmetic means that the **M**ost**S**ignificant**B**it doesn't change (if it was a 1, the "new" MSB will be a one as well), same with the **L**east**S**ignificant**B**it for the left variant. In contrast, SRL, ***S**hift **R**ight **L**ogic*, does not manipulate the MSB, it will always be a 0.

What's interesting is that we've got a SRA, SRL and a SLA opcode but no SLL variant of the later. Not only that, the SLA opcode functions as if it were the logic variant, in that it doesn't manipulate the LSB. The main purpose of these shift opcodes is to perform an efficient multiplication or division by 2^n. The arithmetic variant then assures that the sign bit is kept.

It therefore makes sense that the opcode with the SLA functionality is missing, since there's no sign bit to be maintained during a left shift. That still doesn't explain why they have an SLA opcode with the functionality of SLL.

Another interesting case are the rotation opcodes: RLC, RL, RRC and RR.
These opcodes rotate the register right or left, using or ignoring the carry bit



In the above **R**otate**R**ight example, after the operation has completed, bit 0 will be placed into the carry, if we were using the RRC variant, bit 0 would become bit 7 as well as being copied to the carry flag.

Now for the interesting part, these opcodes have an address for each of the 7 (+ HL) registers, RR A (rotate the A register right, note that A is an argument) is at 0x1F in the extended opcode set, however, there's another opcode RRA in the basic opcode set which does the exact same thing. This is true for RLC, RL and RRC as well.

It is reasonable to assume that this is done for efficiency reasons, as the basic instruction set is accessed faster than the extended one. In this case, RRA takes 4 cycles while its identical RR opcode takes 8. However, It is odd how they're keeping both versions (RRA and RR A, …) as they are exactly the same, except for the one where A is an argument taking 4 cycles longer. Even though it's a waste of address space, they might've concluded that keeping them both was easier than taking them out.

Implementation

I've implemented the opcodes in basic inline functions using arguments to differentiate between the behavior of the different variants, using function overloading for certain 16 bit alternatives. The opcodes functions themselves are called from a switch statement. A switch statement, when compiled optimized for speed is often implemented as a branch table with a O(1) lookup complexity, meaning that *jumping* to different opcodes is likely to take up a constant time regardless of how many cases the switch statement contains (Wikipedia: Switch Statement, n.d.)

The PPU

The **P**icture**P**rocessing**U**nit is on certain consoles a separate chip, on the Gameboy it's been integrated into the CPU. The PPU is in charge of writing the textures to the screen.

When the pixels are calculated, they're stores in a 160*144*2 bitset. 160*144 is the amount of pixels the Gameboy uses and since every pixel has a 2 bit color assigned to it, we multiply the 160*144 by 2. Meaning that instead of using 23 040 bytes we only use 5 760 bytes.

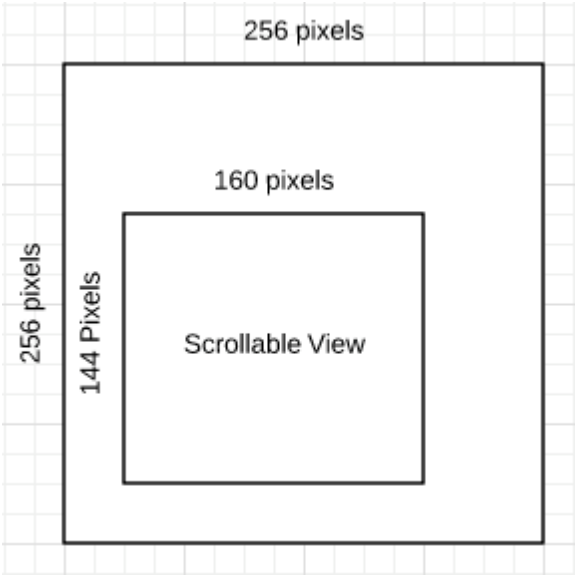
Like I mentioned, a 2 bit color scheme is used, 00 for white, 01 for light gray, 02 for dark gray and 03 for black. For older Gameboy models this used to be a green shade instead of a grayscale.

In order to quickly change the *colors* of a certain tile (a tile is 8*8 pixels), the colors are not directly mapped to each tile, instead they refer to the index of a color palette, this palette can be modified to immediately change the appearance of ever pixel. Different color palettes are used in different layers. (Fayzullin, et al., n.d.)

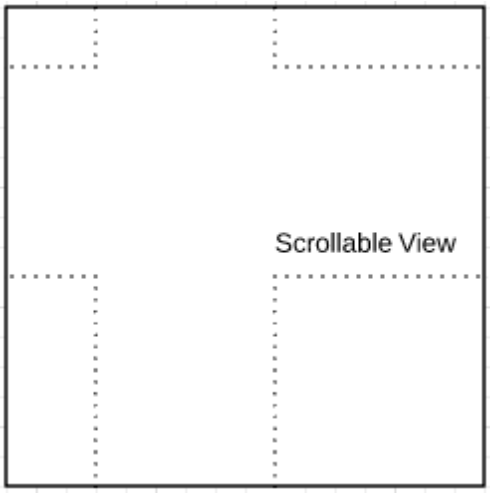
The PPU works in three layers: The background, the window (ui) and the sprites layer. The background layer contains the scene, pixels that should be drawn behind every other pixel. The window layer is usually used to display UI elements that don't frequently move during gameplay. The sprites layer is used to render the individual elements of a game that move very frequently and accurately during gameplay (Fayzullin, et al., n.d.)

Background drawing

The Background layer is 256*256 pixels (32*32 tiles) in size, note that our LCD screen is only 160*144 pixels big. The way this is implemented is by allowing the background layer to scroll over the 160*144 "*view window*", showing only the area the developers intended to be visible. The 256*256 *"image"* also wraps around the horizontal and vertical edges, allowing for simple and efficient moving level background. (Richeson, 2017)



2 The viewable area wraps around the background
Taken from Richeson's Paper



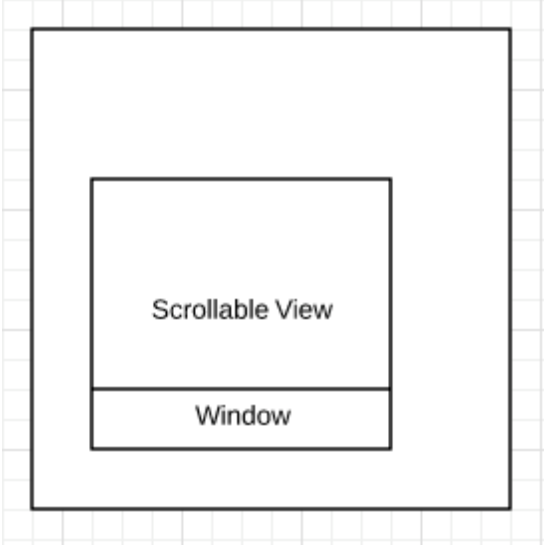
3 The scrollable Viewable area on top of the background
Taken from Richeson's Paper

These 32*32 tile numbers are all stored in VRAM at 0x9800-0x9BFF and 0x9C00-0x9FFF these two ranges are used for the background or window tiles. Every range is 1024 bytes in size which is 32 rows of 32 bytes with each byte holding a number referring to a tile. (Fayzullin, et al., n.d.)

The actual tile pixel data is stored at 0x8000-0x9700 with ever tile occupying 16 bytes (8*8 pixels *2 bit colors each /8 bits). The position of the background tiles (the scrolling aspect of the scrollable view) is determined by a byte in the IO memory space at 0xFF42 and 0xFF43 for the Y and X position respectively.

Window Drawing

Drawing the window layer happens in a very similar fashion as the background layer except that the window layer is always drawn on top of the background layer and that the window layer is always drawn relative to the scrollable view area instead of the 256*256 background. Usually, the window layer is offset in such a way that it aligns to the bottom of the scrollable view, this however, isn't a requirement. (Richeson, 2017)



4 The window on top of the scrolabble view on top of the background layer
Taken from Richeson's paper

The bytes controlling the position of the window layer can be found in IO memory at address 0xFF4A and 0xFF4B for its Y and X(-7) position respectively. Due to the window layer being drawn relative to the scrollable view, a Y position of 0 and an X position of 7 would put it at the top left corner making it cover the entire screen, note that sprites are still drawn on top of the window layer. (Fayzullin, et al., n.d.)

Sprite Drawing

The sprite layer is used to draw the elements that do not align to the 256*256 background layer or window layer. Sprites frequently move about the scrollable view and are drawn unaligned with pixel level accuracy (Richeson, 2017)

In order to keep track of the multiple sprites active at the same time, basic information is kept in the **O**bject**A**ttribute**M**emory space, this memory space can hold information of up to 40 sprites at a time. Certain games like Tetris can have more than 40 active sprites at a time, the developers usually remedy this by changing the background layer. In Tetris for example, more than 40 blocks can be on the playfield at the same time, the developers *fixed* this by embedding a block into the background layer as soon as it's placed (Fayzullin, et al., n.d.)

The OAM address space ranges from 0xFE00-0xFE9F allowing up to 4 bytes per sprite.

> Byte **0**: Holds the Y position of the sprite -16, if the value is off-screen, the sprite is hidden

*There are two kinds of supported sprites 8*8 or 8*16 pixels, the -16 is to account for the maximum height*

Byte **1**: Holds the X position of the sprite -8
Byte **2**: Specifies the Tile number of the sprite selecting a tile from memory 0x8000-0x8FFF
Byte **3**: Contains the sprite's attribute flags, this includes the palette being used, sprites have 2 possible palettes from which they can choose

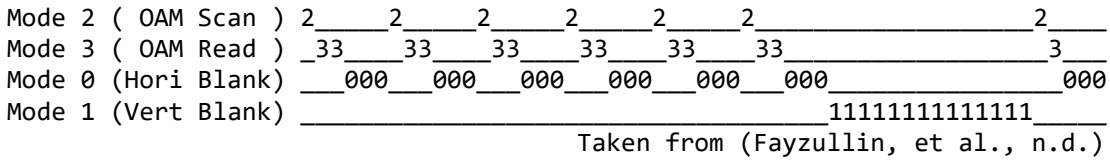
(Fayzullin, et al., n.d.)

An interesting hardware limitation is that only 10 sprites are processed for each scanline. This is done by checking each sprite's Y position to the current scanline position, the first 10 sprites that appear on that line are drawn, the rest is ignored. Note that this follows the order they appear in OAM (Fayzullin, et al., n.d.)

LCD Status

Layers are drawn line by line. An entire frame is 154 lines which means 1 frame takes 16.7 ms. These 154 frames are divided into 4 modes: 0 Horizontal Blanking, 1 Vertical Blanking, 2 Scanning OAM for overlapping sprites and 3 Reading OAM to create an image. (Fayzullin, et al., n.d.)

No two modes can be active at the same time, so when the display is enabled (the display can be disabled, skipping the whole rendering pipeline) the interval between the modes will look like this:



Mode **0**: Horizontal blanking, its entire duration is 85-208 cycles depending on mode 3's duration. A horizontal blank is fired when we're at the end of our scanline (our **horizontal** scanline)

Mode **1**: Vertical Blanking, takes up 4560 cycles which is the duration of 10 scanlines, all memory is accessible. A vertical blank is fired when all our scanlines have been drawn, every pixel row is a scanline and every row is laid out in a **vertical** manner.

Mode **2**: Scanning OAM to see if any sprites have active pixels on our current scanline. This mode takes 80 cycles and prevents the CPU from accessing OAM memory

Mode **3**: During this mode we actually generate our image through the data gathered from OAM, VRAM and the color palettes. This mode takes up between 168 and 291 cycles depending on the sprite count

(Fayzullin, et al., n.d.)

Very important to know is that mode 0, 2 and 3 will usually have a combined total of 456 cycles! Another important note, if you're looking at PanDocs (The Gameboy emulation bible) you'll notice that they tend to use the terms cycles and dots interchangeably.

Mode 3 and 0 have a variable duration, however, we know that every scanline (mode 0,2 and 3) takes 456 cycles. Therefore I've emulated this in a static manner: mode 2 always takes 80 cycles, so that's fine; For mode 3, I've allocated 172 cycles; The remainder goes to Mode 0.

Input

There are 8 buttons on the Gameboy: Up, Down, Left, Right, Start, Select, A and B. Conveniently, there are also 8 bits in a byte, however, to save on GPIOs, they've implemented input control using only 6 instead of 8 GPIOs. As a result we will be utilizing 6 instead of all 8 bits of a byte. (Steil, n.d.)

The 8 buttons are split in 2 columns with 2 buttons each. The first column contains all direction keys, the second one contains the normal buttons.

The status of the joypad is kept in the IO memory space at 0xFF00

```
Bit 7 - Not used
Bit 6 - Not used
Bit 5 - P15 Select Button Keys      (0=Select)
Bit 4 - P14 Select Direction Keys   (0=Select)
Bit 3 - P13 Input Down  or Start    (0=Pressed) (Read Only)
Bit 2 - P12 Input Up    or Select   (0=Pressed) (Read Only)
Bit 1 - P11 Input Left  or Button B (0=Pressed) (Read Only)
Bit 0 - P10 Input Right or Button A (0=Pressed) (Read Only)
Taken from (Fayzullin, et al., n.d.)
```

Bit 5 and 4 can be written to by the game and is used to determine which key statuses are being requested. Bit 5 and 4 cannot both be set at the same time. Note that this setup leaves the last two bits unused, from a data standpoint this is suboptimal but remember that this design pattern stems from the technical layout of the Gameboy which used this system to reduce the amount of required GPIOs.

Whenever a key is pressed that's currently part of the selected key set an interrupt is fired. This is the only way to get out of the STOP opcode which basically puts the Gameboy in a low power state until a joypad interrupt is fired. Game developers rarely use the joypad interrupt to check for key presses.

Implementation

The game can write tot the Joypad byte whenever it likes, however it can only set bit 5 or 4. Whenever the game reads the joypad byte, bits 0 to 3 will be set reflecting the status of the requested keys (directional or normal buttons) by reading bit 5 and 4 first.

The way I've implemented this is by letting the game write to the byte without intercepting it. When the game tries to read from it, the request will be caught and forwarded to a function updating the joypad byte reflecting the current keyboard status

```
uint8_t GameBoy::GetJoypadState() const {
	uint8_t res{ Memory[0xff00] };

	if (!(res & 0x20)) { //Button keys
		res |= !!(JoyPadState & (1 << aButton));
		res |= !!(JoyPadState & (1 << bButton)) << 1;
		res |= !!(JoyPadState & (1 << select)) << 2;
		res |= !!(JoyPadState & (1 << start)) << 3;
	} else if (!(res & 0x10)) {
		res |= !!(JoyPadState & (1 << right));
		res |= !!(JoyPadState & (1 << left)) << 1;
		res |= !!(JoyPadState & (1 << up)) << 2;
		res |= !!(JoyPadState & (1 << down)) << 3;
	}
	return res;
}
```

Note that 0 means the key is pressed or the button set is selected and 1 means the button is released or set unselected. 0010 0111 would mean that the down button is pressed. The *JoyPadState* variable is a uint8 data member that I'm using to keep track of which keys are pressed at the moment. I could check the key state at the requested time but that would slow things down, in this project it's speed over memory. Also keys need to be parsed when pressed anyway, due to the possible interrupt being fired.

Memory

The cartridges were designed to be very expandable and versatile. If the developers deemed it necessary, they could add additional rom or ram banks effectively expanding the available memory capacity..

Because the amount of rom/ram banks is variable, the Gameboy itself had to find a flexible way of addressing this additional memory. The way they've done this is through a **M**emory**B**ank**C**ontroller chip. This chip allowed the address range 0x4000-0x7FFF to be physically rewired to a different memory section. For example, the game might set the MBC to Bank 1 and then request the byte at address 0x4001 which might be 0x10, then it might set the MBC to bank 2 and request the same address, however, this time the result might be 0x56.

The Gameboy itself just reads the data and is none the wiser.

The way the MBC is changed is somewhat interesting..
We change the status of the MBC by modifying its control registers, this is done by writing to the ROM address range. Remember, we can't write to ROM memory because, that's what ROM memory is.. Read-Only Memory.

If we're writing !0x?A between 0x0000-0x1FFF (which is the ROM section), RAM memory banking will be disabled. If we're writing 0x?A, RAM memory will be enabled. Note how we don't care about the upper 4 bits, if the last 4 bits are 0xA (1010), we're enabling, if they're anything else, we're disabling.

Writing to 0x2000-0x3FFF changes the active ROM bank. Here we're looking at the first 5 digits, 00011 will enabled bank 3 (and disable all the others, remember we can only read/write to one bank at the time), 11111 will enable bank 31.

Writing to 0x4000-0x5FFF changes the active RAM bank OR allows to set the upper 2 bits of the active ROM bank. Whether we set the ram bank or upper two bits of the active rom bank depends on the what we write to

0x6000-0x7FFF. This range will change the mode of the previously discussed setting. 0x00 will make it use rom banking, 0x01 will make it use RAM banking. Note that when we're in ROM mode, ROM can be extended up to 2MB leaving RAM at 8KB, if we're in ram mode, ram can grow to 32KB leaving ROM at 512KB. This mode can be switched during runtime.

These settings depend on the MBC chip that's being used (of which there are 5). I love these "hacks"!

Future Work

So what's next?
Well, I only focused on getting Tetris to work, that being sad, I'm fairly certain I implemented every opcode since I wrote a python script to automatically generate most of them. However, it is possible that I missed one, I'd have to check.

In order to ensure that the Gameboy is emulated correctly, there are several Test roms freely available that can be used to debug the emulator. When the test rom shows that there's an issue (or the rom itself doesn't work), you'll have to dig into the ROM's assembly and compare the execution and expected execution with the emulator you're testing it on. **I highly recommend you do this!** It's an amazing journey, though, it can be a bit time consuming especially if it's your fist time doing this. Don't let that scare you off though!

Again, for this project I only focused on Tetris, made sure that that works as expected. This means that **there are bugs in my implementation**, I just haven't found them yet. When you do find any, let me know! :)

Sound

For this emulator sound was irrelevant due to its insignificance to training neural networks. That being said, I hear that getting sound to work is an interesting challenge on its own! Here's an interesting reddit thread to get you started. Also be sure to check out the Ultimate Game Boy Talk's sound section. Though the best way to find out how it's done is by looking at other people's code, I do suggest you try it yourself first, unless time is of the essence of course..

Networking

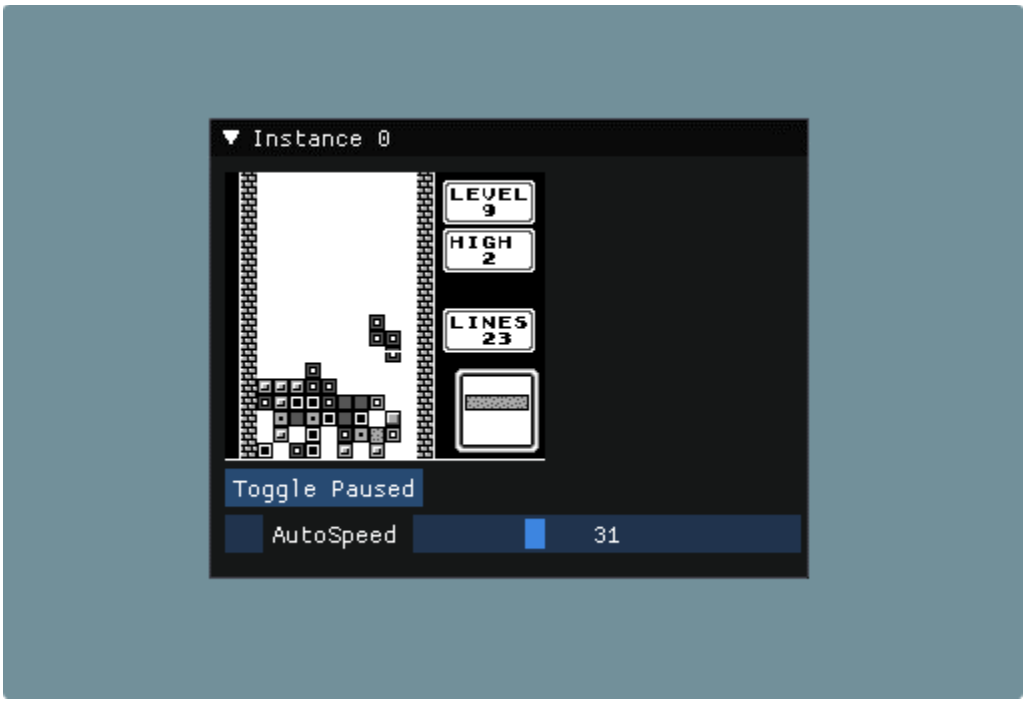
Implementing networking through the link cable was a stretch goal for my project. I never got around to it but I hear it's in essence fairly simple to implement. The PanDocs, as always, covers the technical details, so I highly recommend you have a look. The Ultimate Game Boy talk covers the subject as well albeit very briefly.

Modular Cartridge Interface

Game Boy cartridges are extremely modular, I've heard stories of developers equipping their cartridge with its own separate processor. I think it would be cool to develop a system that allows for simple custom cartridge behavior, though, this will probably mean having to write an electrical signal interface. Maybe adding scripting language support for easy plug-n-play? Food for thought!

Conclusion

Unfortunately I was never able to reach an acceleration of 1000 times the original speed, I only managed to get a measly 30. In retrospect, 1000 might've been a teensy bit optimistic.



5 Single instance running at maximum acceleration

Note that all information regarding its runtime happened on an intel core i7 6700HQ @2.60 GHz which has 8 logical cores. The UI elements in the above picture are curtesy of [dear imgui](), a GUI library for C++.

The final library is 8.1MB, 10 files in total with the largest file being less than 1200 lines (which is exceptionally large for this project), the average being around 300 lines. Runtime memory usage scales depending on the amount of instances, the demo (Which includes the ui) ran at 9.7MB for one instance, the UI and SDL overhead taking up most of the memory.

CPU wise the bottleneck is definitely the function in charge of drawing the background with the function handling memory reading as a close second.

Function / Call Stack	CPU Time		
	Effective Time by Utilization ▼ Idle Poor Ok Ideal Over	Spin Time	Overhead Time
LR35902::DrawBackground	13.171s	0s	0s
GameBoy::ReadMemory	11.056s	0s	0s
LR35902::ExecuteOpcode	4.177s	0s	0s
GameBoy::Update	4.037s	0s	0s
LR35902::HandleGraphics	3.686s	0s	0s
LR35902::DrawSprites	1.642s	0s	0s
Update	0.892s	0s	0s
GameBoy::AddCycles	0.495s	0s	0s
GameBoy::WriteMemory	0.138s	0s	0s
LR35902::DrawWindow	0.109s	0s	0s

6 Bottom-Up graph from VTune

The library can be used through a single header file exposing the following functions:

```
void LoadGame( const std::string &gbFile );
void Start() const;

std::bitset<160*144*2> GetFrameBuffer( const uint8_t instanceID ) const;
void SetKeyState( const uint8_t key, const bool state, const uint8_t instanceID ) const;
void RunForCycles( const unsigned short cycles, const uint8_t instanceID ) const;
void RunForFrames( const unsigned short frames, const bool onlyDrawLastFrame, const
uint8_t instanceID ) const;

void SetPauseState(const bool state, const uint8_t instanceID) const;
bool GetPauseState( const uint8_t instanceID ) const;

void SetSpeed( const uint16_t cycleMultiplier, const uint8_t instanceID ) const;
uint16_t GetSpeed( const uint8_t instanceID ) const;
void SetAutoSpeed(const bool onOff, const uint8_t instanceID) const;
```

Every instance is addressed through an unsigned byte ranging from 0-[Instances-1]. `LoadGame`, `Start`, `Set/GetPauseState`, `Set/GetSpeed` and `GetFrameBuffer` are fairly self-explanatory.

`SetKeyState` sets the status of the Gameboy keys with true being pressed.

`RunForCycles` lets the given Gameboy instance run for # cycles after which the instance will pause.

`RunForFrames` lets the given Gameboy instance run for # frames after which the instance will pause, the `onlyDrawLastFrame` argument, when set, will ensure that only the final frame is drawn, allowing the instance to go much faster (since the main bottle necks are the draw calls)

`SetAutoSpeed` will ensure that the speed is set to the maximum achievable amount. Note that this doesn't really change anything compared to having the speed set to a ridiculously high number, if anything, it slows things down.

References

(n.d.). Retrieved from nesdev.com: https://wiki.nesdev.com

Anthrox, P. o., GABY, Fayzullin, M., Felber, P., Robson, P., Korth, M., . . . DP. (n.d.). *Game Boy CPU Manual.* DP.

Bove, B. (n.d.). *Building a Gameboy Clone from Scratch (Part 1: Core Emulation)*. Retrieved from brianbove.com: https://brianbove.com/blog/2017/05/29/building-a-gameboy-clone-from-scratch-part-1-emulation/

Codeslinger. (n.d.). *gameboy*. Retrieved from codeslinger.co.uk: http://www.codeslinger.co.uk/pages/projects/gameboy.html

Copetti, R. (n.d.). *game-boy*. Retrieved from copetti.org: https://copetti.org/projects/consoles/game-boy/

Fayzullin, M., Felber, P., Robson, P., Korth, M., nocash, & kOOPa. (n.d.). *PanDocs*. Retrieved from gbdev: https://gbdev.gg8.se/wiki/articles/Pan_Docs

mattcurrie. (n.d.). *mgbdis*. Retrieved from https://github.com/mattcurrie/mgbdis

Nazar, I. (n.d.). *GameBoy Emulation in JavaScript*. Retrieved from imrannazar: http://imrannazar.com/GameBoy-Emulation-in-JavaScript:-Memory

Richeson, J. M. (2017, 05). Anatomy of a hardware emulator. 96. The University of Texas. doi:10.15781/T23R0Q56D

Steil, M. (n.d.). *The Ultimate Game Boy Talk*. Retrieved from media.ccc.de: https://media.ccc.de/v/33c3-8029-the_ultimate_game_boy_talk

Wikipedia: Switch Statement. (n.d.). Retrieved from https://en.wikipedia.org/wiki/Switch_statement

XVar#0831(167020033002700802), PSI#1750(272461651394035712), Pixel#4370(189512784537583616), & LukeUsher#0879(293295788371607552). (n.d.). Emulation Development. (velocity#4995(111576667218075648), Ed.) Retrieved from https://discord.gg/dkmJAes