Emulation to preserve a Generation

Thomas Crosby

3 May 2020

A project report submitted in partial fulfilment for the degree of

**Bachelor of Science in Software Engineering**

**School of Physical Sciences and Computing**

**University of Central Lancashire**

Abstract

The foundation of my childhood revolved around playing video games from a young age. My earliest memories I remember picking up a controller and pretending to play. The first console I had was the original Gameboy. Since the Gameboy is no longer in production, this means it has become more and more difficult to access. The dwindling number of physical consoles and the lack of official preservation for older systems creates a problem as countless games could be at risk of becoming forgotten. The aim of this project is to create a Gameboy emulator, capable of running digital copies of the Gameboy games so that they can be preserved for future generations. As the Gameboy architecture is a massive and complex undertaking, the goal of the project is to create software that can recreate the experience of the original Gameboy. The scale of the project’s potential scope meant that agile techniques such as MoSCoW and Timeboxing were needed to allocate time for the most important features to achieve this goal. This not only allowed the project to accommodate for unforeseen circumstances and problems with debugging, but also allowed development to prioritise the most valuable components first. The solution required intensive research around the Gameboy’s inner workings and encouraged the development of debugging skills. The product of the project is a functional Gameboy emulator built in C++ using the SDL2 graphical library that can play various original Gameboy games. The emulator is not a true representation of the original Gameboy due to minor graphical issues and a lack of sound, however this was never the intention of the project. Therefore, as the emulator provides a playable experience of the games from my childhood, the project has achieved the goal set out and is a success.

Attestation

I understand the nature of plagiarism, and I am aware of the University’s

policy on this.

I certify that this document reports original work by me during my University project.

**Signature** T. Crosby**Date 23/07/2020**

Acknowledgements

First and foremost, I would like to acknowledge the efforts of the academic staff that have helped me throughout my education and with this project. I would like to thank Nicholas Mitchell for his help and guidance as my supervisor during the journey of this project. I would also like to thank Nicky Danino for the resources that allowed the development of this project possible.

I would like to thank my peers who have helped me with advice and encouragement to allow me to achieve what I thought was impossible. Daryl Baldock for the encouragement and reassurance throughout my time at university and John Green for his insight and advice during the implementation of the project.

I also would like to show appreciation to the Emulation Development Discord server and GB Dev Wiki for providing not only resources that allowed the creation of the project possible, but also advice during the frustration of debugging.

Finally, I would like to thank my parents for putting up with me during my time at university and providing all the encouragement in the world for me to pursue my education.

Table of Contents

[Abstract i](#_Toc47264230)

[Attestation ii](#_Toc47264231)

[Acknowledgements iii](#_Toc47264232)

[Table of Contents iv](#_Toc47264233)

[List of Figures vii](#_Toc47264234)

[List of Tables viii](#_Toc47264235)

[List of Listings ix](#_Toc47264236)

[1 Introduction 1](#_Toc47264237)

[2 Background and Related Work 2](#_Toc47264238)

[2.1 Introduction 2](#_Toc47264239)

[2.2 The history of the Gameboy 2](#_Toc47264240)

[2.3 Preserving Video games for the future 2](#_Toc47264241)

[2.4 Simulation and Emulation 3](#_Toc47264242)

[2.5 Applications of Emulation 3](#_Toc47264243)

[2.6 The approaches to Emulation 3](#_Toc47264244)

[2.6.1 Static and Dynamic compilation 4](#_Toc47264245)

[2.7 The difficulties of emulating software 4](#_Toc47264246)

[2.7.1 The Approaches to Preservation 5](#_Toc47264247)

[2.8 Emulation to preserve the Gameboy 5](#_Toc47264248)

[2.9 Preliminary Research 6](#_Toc47264249)

[2.9.1 The CPU 6](#_Toc47264250)

[2.9.2 Instructions and Registers 7](#_Toc47264251)

[2.9.3 Timings and Interrupts 8](#_Toc47264252)

[2.9.4 The GPU/PPU 8](#_Toc47264253)

[2.9.5 The Cartridge 9](#_Toc47264254)

[2.9.6 The BIOS 10](#_Toc47264255)

[2.9.7 User Input 10](#_Toc47264256)

[2.10 Summary 10](#_Toc47264257)

[3 Project Planning 11](#_Toc47264258)

[3.1 Introduction 11](#_Toc47264259)

[3.2 Methodology 11](#_Toc47264260)

[3.3 Requirements 11](#_Toc47264261)

[3.3.1 Must have requirements 11](#_Toc47264262)

[3.3.2 Should have requirements 12](#_Toc47264263)

[3.3.3 Could have requirements 12](#_Toc47264264)

[3.3.4 Would have requirements 12](#_Toc47264265)

[3.4 Potential Solutions 12](#_Toc47264266)

[3.5 Tools and Techniques 13](#_Toc47264267)

[3.6 Legal, Social and Ethical Issues 14](#_Toc47264268)

[3.6.1 Emulating the Hardware 14](#_Toc47264269)

[3.6.2 Software 14](#_Toc47264270)

[3.7 Summary 14](#_Toc47264271)

[4 Design 15](#_Toc47264272)

[4.1 Introduction 15](#_Toc47264273)

[4.2 Breaking down the heart of the Gameboy 15](#_Toc47264274)

[4.2.1 The Registers 15](#_Toc47264275)

[4.2.2 The Memory Bus 16](#_Toc47264276)

[4.2.3 Processing Instructions 16](#_Toc47264277)

[4.2.4 Interrupts and Timings 17](#_Toc47264278)

[4.2.5 The BIOS 17](#_Toc47264279)

[4.3 The Cartridge 18](#_Toc47264280)

[4.3.1 ROM only games 18](#_Toc47264281)

[4.3.2 The MBC (Memory Bank Controller) 18](#_Toc47264282)

[4.3.3 Saving the game 18](#_Toc47264283)

[4.3.4 Controlling the Gameboy 19](#_Toc47264284)

[4.4 The Display 19](#_Toc47264285)

[4.4.1 Drawing a Blank 19](#_Toc47264286)

[4.5 User Interface Design 20](#_Toc47264287)

[4.6 Summary 20](#_Toc47264288)

[5 Implementation 22](#_Toc47264289)

[5.1 Introduction 22](#_Toc47264290)

[5.2 Initial Development 22](#_Toc47264291)

[5.2.1 Foundations 23](#_Toc47264292)

[5.2.2 Re-evaluating my approach 24](#_Toc47264293)

[5.2.3 The display (Getting visual) 25](#_Toc47264294)

[5.2.4 Processing the pixels 25](#_Toc47264295)

[5.2.5 Re-evaluating the structure of the system 26](#_Toc47264296)

[5.2.6 The Cartridge Class 27](#_Toc47264297)

[5.2.7 Problems with debugging and reliability 27](#_Toc47264298)

[5.3 The final steps to Gameplay 28](#_Toc47264299)

[5.3.1 Understanding Interrupts 28](#_Toc47264300)

[5.3.2 The Halt Bug 28](#_Toc47264301)

[5.3.3 Taking control of the game 29](#_Toc47264302)

[5.3.4 Memory banking 29](#_Toc47264303)

[5.4 Summary 30](#_Toc47264304)

[6 Test Strategy and Evaluation 31](#_Toc47264305)

[6.1 Introduction 31](#_Toc47264306)

[6.2 Functional Testing 31](#_Toc47264307)

[6.2.1 Blargg’s hardware test ROMs 31](#_Toc47264308)

[6.2.2 Using testing as a tool for debugging 31](#_Toc47264309)

[6.2.3 Mooneye GB tests 32](#_Toc47264310)

[6.2.4 The Official Gameboy Test Cartridge 33](#_Toc47264311)

[6.3 Non-Functional Testing 33](#_Toc47264312)

[6.3.1 Is that a Demo in your pocket? 34](#_Toc47264313)

[6.4 User Testing 34](#_Toc47264314)

[6.5 Summary 34](#_Toc47264315)

[7 Evaluation, Conclusions and Future Work 36](#_Toc47264316)

[7.1 Project Objectives 36](#_Toc47264317)

[7.2 Self-Evaluation 36](#_Toc47264318)

[7.3 Applicability of Findings to the Commercial World 37](#_Toc47264319)

[7.4 Conclusions 38](#_Toc47264320)

[7.5 Future Work 38](#_Toc47264321)

[References 39](#_Toc47264322)

[Appendix 1 – Project Proposal 42](#_Toc47264323)

[Project Context 42](#_Toc47264324)

[Specific Objectives 42](#_Toc47264325)

[References 43](#_Toc47264326)

[Potential Ethical or Legal Issues 43](#_Toc47264327)

[Resources 44](#_Toc47264328)

[Potential Commercial Considerations - Estimated costs and benefits 44](#_Toc47264329)

[Proposed Approach 44](#_Toc47264330)

[Appendix 2 – Technical Plan 45](#_Toc47264331)

[Appendix 3 – Additional Code 51](#_Toc47264332)

[Appendix 4 – Development Progression 54](#_Toc47264333)

[Appendix 5 – Graphical Glitches 56](#_Toc47264334)

[Appendix 6 – Game Testing 58](#_Toc47264335)

List of Figures

[Figure 1 The Original Gameboy and Gameboy Color from my Childhood 1](#_Toc47279623)

[Figure 2 BGB Debugging utilities 8](#_Toc47279624)

[Figure 3 Example of the Gameboy scrolling from “The Ultimate Gameboy Talk” 9](#_Toc47279625)

[Figure 4 Diagram of the Fetch-Decode-Execute cycle (Created by Imran Nazar) 17](#_Toc47279626)

[Figure 5 Gameboy Button to Bit mapping (Representation created using the GB Dev Wiki) 20](#_Toc47279627)

[Figure 6 Palette Swapping 28](#_Toc47279628)

[Figure 7 Tetris comparison of BGB and no$gmb 30](#_Toc47279629)

[Figure 8 Infinite testing glitch from lack of MBC and passed tests 35](#_Toc47279630)

[Figure 9 Official Gameboy testing 36](#_Toc47279631)

[Figure 10 Game Testing (Gameboy Gallery, Kirby’s DreamLand, Super Mario Land) 37](#_Toc47279632)

[Figure 11 Gameboy Pocket Demo and graphical demonstration 37](#_Toc47279633)

List of Tables

[Table 1 Memory Bus Mapping (from GB Dev Wiki) 7](#_Toc47279554)

[Table 2 Visual representation of combining bits to get the pixel colour 27](#_Toc47279555)

[Table 3 Gameboy Header Information (Obtained from the GB Dev Wiki) 29](#_Toc47279556)

[Table 4 Non-Functional Game testing 60](#_Toc47279557)

List of Listings

[Listing 1 Defining Datatypes 23](#_Toc47279558)

[Listing 2 Register Union 23](#_Toc47279559)

[Listing 3 Switch Case approach 24](#_Toc47279560)

[Listing 4 Function Pointer Approach 25](#_Toc47279561)

[Listing 5 Updated CPU Implementation 26](#_Toc47279562)

[Listing 6 Register combination error 31](#_Toc47279563)

[Listing 7 Adjusting Frame Timing 32](#_Toc47279564)

[Listing 8 Polymorphic Cartridge 33](#_Toc47279565)

[Listing 9 Writing to the Cartridge to access the MBC 33](#_Toc47279566)

# Introduction

Moore’s law is an observation made in the 1970’s that noted that the development of processor speeds doubled every two years. Although this may not be true today, even household appliances boast more processing power than old processors such as the Z80. The Z80 was the foundation for the first console I ever played growing up, the Gameboy. Sadly, this piece of hardware is no longer being produced and the lack of functional hardware still circulating means that access to play Gameboy games becomes more and more difficult. The aim of this project is to explore the intricacies of the Gameboy to develop a piece of software capable of Gameboy emulation not only to re-experience the childhood nostalgia of the games I played growing up but to preserve the technology for future generations.



Figure 1 The Original Gameboy and Gameboy Color from my Childhood

# Background and Related Work

## Introduction

This chapter will review the literature surrounding the internal workings of the Gameboy hardware and the steps made by others towards emulating the hardware. The Gameboy was the first console I owned as a child playing classic titles such as Tetris and Pokémon. The nostalgia of playing these games again, developed my interest into the emulation to explore the inner workings of my childhood console.

## The history of the Gameboy

The Nintendo Gameboy was released in 1989 selling 118.69 million units worldwide (Nintendo Consolidated sales transition by region); boasting over 1000 games in its library (Gameboy (original) games found in references). 2019 marks the 30th anniversary of the console and although there are games that can be official emulated through the virtual console service offered by Nintendo, there are only 50 Gameboy and 31 Gameboy color games available for download. The Gameboy has become an icon of the retro era making it slowly more difficult to acquire the original hardware. Furthermore, the original Gameboy cartridges required battery power for larger games such as the second-generation Pokémon games. The restrictive battery life of 10-15 years for these cartridges means that unless the batteries are replaced the games are lost. Emulation provides the answer to preserving the nostalgia of these games (Guttenbrunner, M., Becker, C., & Rauber, A. (2010).) by allowing them to be ran on different host system. Emulators are primarily used to play commercial games that are difficult or sometimes impossible to acquire. Unfortunately, this also provides a gateway of opportunity for piracy and the redistribution of original titles. Conley et al (2003). Demonstrates the severity of the impact that emulation can have on the gaming industry and details Sony’s legal action against emulators such as Connectix and Bleem! with Sony losing the dispute. The emulators required the PlayStation BIOS to be used and therefore the software was seen as fair use.

## Preserving Video games for the future

The paper written by Monnens et al (2009). expresses the value of preserving the video games as a media format especially ROM cartridges which may be at risk of “corrosion from moisture and battery acid.” Although they can be restored, the effective lifespan of these cartridges is unknown. Although these consoles can still be found today, an effort to actively preserve this type of media must be made so that it does not fade into obscurity. The games themselves can be viewed as an art medium having a large impact on culture and are an important part of history. This section explores the various approaches that can be used to preserve video games looking at the costs and benefits for each approach.

## Simulation and Emulation

At face value, the difference between simulation and emulation is not clear, as both allow the use of software to be ran in an environment different to its original system. The difference between the two lies within the underlying architecture of the system. An example of a simulation would be a flight simulation. The simulation will model the behaviour of the aircraft. The attributes of the plane are substituted so it can create a realistic appearance of flight. In contrast to this, an emulation focuses on the internal workings of the system to recreate the output of the original object. The National library of the Netherlands Koninklijke Bibliotheek. (2007). explains that “Emulation is best described as imitating a certain computer platform or program on another platform or program.” In the example of a flight emulator, the emulation considers the internal workings of the aircraft that allow it to achieve the state of flight to create a more accurate representation of the system. An emulation of the Gameboy system allows the data from the original game code to be ran on a different system, rather than representing an entirely new game. A Gameboy simulation would be unreasonable as the data fed into the software changes based on the game. Therefore, simulation would only be more useful for the recreation of a specific game that does not need to act like the original. This shows that, the use of Emulation will provide a truer representation of the game and allows the software to run any game that was written for the Gameboy.

## Applications of Emulation

The applications of emulation extend beyond the realm of video game preservation and can be applied to various other areas of programming such as virtual machines. This is the process of emulating an entire operating system through a computer itself and allows incompatibles systems to be able to work together J.E. Smith et al (2005). The coding language Java uses the JVM (Java virtual machine). The key benefit of this is that it allows any operating systems with access to the JVM to run any program written in Java. The Java compiler uses a process of translating the program to an intermediary language between the high-level programming language and the base level assembly language. Bill Venners (1998). This abstraction between the computer and program itself allows both to be independent of each other and therefore creates a high level of flexibility and security as discussed in the JVM Specification. Lindholm et al (2013).

A significant strength and application of virtual machines can be seen when they are paired with a cloud-based server architecture. By using virtual machines on a cloud-based network, this allows businesses to scale and allocate computing resources based on their need. Xiao Zhen et al (2012). This means that costs are significantly lowered as the cloud servers can be accessed from virtually anywhere saving the need for computer infrastructure to be installed and maintained.

## The approaches to Emulation

Emulation can be approached using several different methods, the first method being literally interpretation of the original software. The use of interpretation when writing an emulator means following the code of the game line by line. This is done by simulating every instruction using the local CPU. This system is usually done through an Interpret-decode-execute cycle. The emulator will use a system to both read and write data to the memory of the program. In the Gameboy, “both the code and data are stored in the same memory” (Witawas Srisa-an. (n.d).) Once the data has been retrieved, the information needs to be decoded, to understand which operand must be completed. The Gameboy uses a program counter register to keep track of the which instruction will be completed next. Finally, the code is executed completing the instruction. This must be completed using a specific number of cycles depending on the instruction. This type of emulation allows easier debugging as the program can be stepped through based on the number of cycles that have passed. (Marat Fayzullin. (n.d).) This style of emulation required more time for the CPU to process which was a problem in older C based emulators, such as Marat’s Virtual Gameboy emulator. However, the difference in processing power of CPUs today means that this isn’t as big of a problem anymore with more powerful consoles being emulated. Furthermore, the technique of interpretation allows the program to be cross platform as the host machine only job is to interpret the original code.

### Static and Dynamic compilation

There are two different approaches to compiling an emulator, which can be done in a static or dynamic way. Static compilation is a technique used in modern higher-level languages such as C++ that speeds up execution of a program by compile the code beforehand instead of interpreting each line. This technique of compilation would be extremely difficult to use with emulation as some instructions cannot be pre-determined. This can be seen in Andrew Kelley. (2013). Finally, dynamic recompilation is technique of decoding the instructions of the target software to run on the host CPU of the emulator. This technique can be seen in the Mugen64plus emulator that used a dynamic recompilation approach. This approach is more complex as it requires a deep level of understanding of the emulated software in order to reinterpret the instructions. (zenogais. Introduction to dynamic recompilation.) Evaluating these techniques, the method that seems most appropriate for writing the Gameboy emulator is interpretation, as the Gameboy requires little processing power therefore the cost of interpreting every instruction would not have a great impact on the CPU.

## The difficulties of emulating software

Emulation of a system is a complex task and requires full understanding of the target system to enable development. This can be gathered through source documentation of the original software. The difficulty of emulating hardware differs from system to system. This allows older systems such as the NES (Nintendo Entertainment System) and Gameboy to be emulated. Due to the lack of computational power, these older systems are easier to emulate than modern day software with various emulators currently available. These systems can be reverse engineered through a brute force method of trial and error. However, for systems such as the N64 (Nintendo 64), emulation is much trickier due to the unique implementation of microcodes, including custom microcodes that were undocumented by developers. This means that brute force methods are much more difficult or even impossible, so that although some games can be run, emulation is not perfect and is an ongoing investigative process. Earlier this year, the source code documentation for various Nintendo systems dating back to the original Gameboy system was leaked. This information, although interesting, is not a viable source of documentation as this would break copyright law, meaning the emulator and developer could be subject to legal action. Conley et al (2003). Therefore, the only way to keep the emulators legal would be through completely reverse engineering the system to allow it to be its own entity.

### The Approaches to Preservation

The first approach seen is the museum approach. The benefits of this approach are that the physical hardware can be preserved and displayed to the public. Yet, this technique does not guarantee the safety of the software involved and limits the playability of the media itself. Therefore, this would not be a viable solution as the gameplay of the games themselves is the key focus. Another approach that the paper explores is archiving the software. This can be done in several different ways although all methods would require strong initial effort and documentation to preserve the software as seen in Van der Hoeven (2007). Van der Hoeven expresses scepticism of the value of emulation for archiving due to the complexity and initial costs. In the context, of the Gameboy, this is true as the system itself can be seen as complex and would require significant initial research to start development. However, this would allow the software to be playable with the flexibility to be played in a variety of different formats. An example of archiving software is porting the software to a relevant console at the time. This can be seen in several consoles such as the Nintendo 3DS and Wii U through the e-shop and has the commercial benefit of allowing monetisation of older software. However, this is not a long-term solution, as only a few select games are chosen to be redistributed and eventually the online resources will be shutdown. This can already be seen for some countries as the e-shop will no longer be providing services to some countries. Furthermore, Guttenbrunner el al (2012). Points out the difficulty in validating and replicating the behaviour to be identical to the original system. The difference in modern day compilation and architecture may make replicating the behaviour exponentially more difficult. To combat this, the paper by Kemppainen et al (2019). explores a different route of preservation through remaking the games entirely to suit modern consoles. This is a sustainable solution for the future however, there needs to be significant interest in the game for the remake process to be viable. A remake can be entirely faithful to the original work such as the legend of Zelda, Link’s awakening remake for the switch, or it can be completely transformative such as the Final Fantasy 7 remake completely transforming the experience of the game itself. The final approach that that can be used for video game preservation will be the focus of this paper, emulation.

## Emulation to preserve the Gameboy

The use of emulation allows software designed for a specific system to be used on another system. This is useful as it allows the preservation of the games that would otherwise be incompatible with modern equipment. The importance of this can be seen in Rothenberg, J. (1998). Furthermore, emulators enable additional features to the original functionality such as save states or monitoring of memory addresses. In the context of the Gameboy, this can be shown in the speed running version of the Gambatte Emulator. This emulator is used as an official software that specializes in hardware accuracy to allow users to attempt world records of finishing popular games the fastest. An example of this would be the Pokémon games that require use of RNG (Random Number Generation). This is useful for understanding how the games worked and has its uses in This functionality allows the third-party development of Gameboy ROMs shown in Campos Colmenarejo, J. (2017). through reverse engineering, which can be used for both game development and testing of the emulator. In the context of Gameboy emulation, the creation of open source ROMs such as blargg’s gb-tests allows testing of the emulator functions without requiring the need for commercial software. This is useful as it allows testing of the emulation without the need for copies of the original software. The effective processing power required to run the system is small in scale compared to today’s computing power. This provides flexibility with emulation as a broad spectrum of hardware could potentially be a host for the Gameboy. An example of this is the Giovanni emulator which is designed and specialized for use with Apple Smart Watch.

However, although emulation provides a long-term solution to video game preservation, it should be noted that emulators themselves can be the victim of obsolescence. The KEEP (Keep Emulation Environments Portable) project discusses the importance of keeping emulation software portable so that it may be used on various types of hardware. (Pinchbeck et al (2009).) The key benefit of this ideology is that through allowing software to migrate to future systems, this provides an ideal long-term solution to video game preservation.

## Preliminary Research

As discussed previously, documentation is a key part of tackling the task of recreating and emulating a system. Therefore, this next section looks at the various areas of the Gameboy that will be needed for development. As there are various sources of information, the validity of the documentation should be taken into consideration as unofficial documentation may have inaccuracies and errors. This is an issue that became more apparent during the Initial development 5.2 of the emulator later in the project. This section looks at the Gameboy’s inner workings and the history behind its development.

### The CPU

The CPU is arguably the most vital part of the Gameboy as it is a customized version of the 8-bit Z80 processor called the “Sharp LR35902” (Nintendo Gameboy color: Console info) developed in 1975 which removed certain functionality of the chip but also enabled use of the PPU to output the display. This chip was used throughout the various models of the Gameboy (such as the Super Gameboy, Gameboy Pocket and Gameboy Color) using a single chip to perform all the actions on the system. The Gameboy interprets the binary data from the ROM file of the cartridge as this was faster access as high-level languages at the time such as C that were not as efficient compiling the assembly code. Furthermore, the chip was chosen for the Gameboy in order to reduce the cost of the overall system. The CPU uses a memory bus mapping the various sections of a 64kB memory space by allocation certain sections of the memory for specific tasks. Furthermore, the CPU uses a 16-bit memory address space to perform operations passed over from the Cartridge. (Michael Steil. (2016).) Below is a table showing the address ranges for the different components on the memory bus.

Table 1 Memory Bus Mapping (from GB Dev Wiki)

|  |  |
| --- | --- |
| 0000-014F | Interrupt handling and Cartridge Header Information |
| 0150-3FFF | ROM Bank 0 - Fixed bank, cannot be switched |
| 3FFF-7FFF | ROM Bank 1 - Switchable bank (If using MBC) |
| 8000-9FFF | VRAM - Background Map data and character RAM |
| A000-BFFF | Cartridge RAM (If available) |
| C000-DFFF | Internal RAM |
| E000-FDFF | Echo RAM - Same as C000-DFFF, Not used |
| FE00-FE9F | OAM – Holds Attributes for the Sprites |
| FEA0-FEFF | Unusable Memory |
| FF00-FF7F | I/O Registers – Additional registers for Gameboy Components |
| FF80-FFFE | High RAM |
| FFFF | Interrupt Enabled Flag |

### Instructions and Registers

The Gameboy uses eight 8-bit registers for various uses and provides two 16-bit registers. The 16-bit registers are implemented by combining two of the 8-bit registers together, for example the A and F register would be combined to create the AF register. The program counter register keeps track of the current instruction and the stack pointer register. In total there are 256 base CPU instructions, along with an additional 256 instructions accessed through the 0xCB instruction. Each instruction carries out one operation which can be found in the Game Boy CPU Manual (Anthrox et al (1989).). An example of these instructions is the LD instruction. This instruction passes a value held in a memory address into the A (Accumulator) register so that it can be manipulated. Many of the Gameboy’s instructions perform similar tasks using different values and registers. The A register is critical as it is mainly used for the mathematical calculations. Furthermore, there are 4 flags that can be set using the first four bits of the F register. These flags are used to loop through areas of memory like a for loop used in modern day compilers. Additionally, the CPU stores additional registers onto the bus directly that are related to the other components of the Gameboy such as the controls or the display. There is a lack of official documentation regarding the Opcodes of the Gameboy, showing the difficulty of implementing the CPU shown in “the internal workings of video game consoles”; “The war with the opcodes was probably the biggest battle during this project. In the beginning I had only one document which was compiled from various text files that used to exist online”. (Ernberg, J. (2011).) This shows the complexity of the system and how important quality resources will be during the implementation. The report “Anatomy of a hardware emulator” explains the difficulties in testing stating that “Tracking invalid state-changes can be very complicated as a single error can cascade into corruption” of the entire memory-space in a matter of seconds. (Richeson, J. M. (2017).) This shows the importance of implementing tools debugging facilities. Figure 2 of this can be seen in the BGB emulator allowing the user to step through each line of the ROM and provides access to unique features such as viewing VRAM to see tiles that have been loaded by the game.

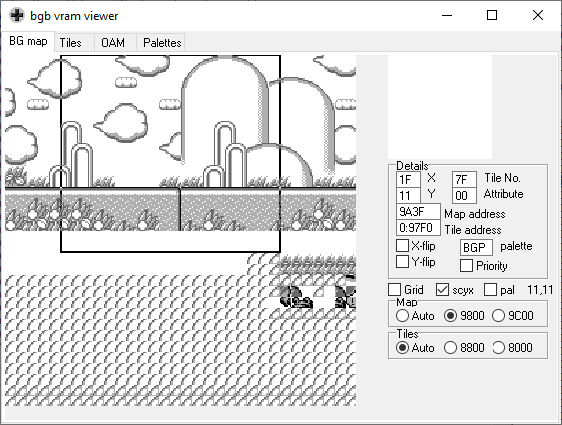


Figure 2 BGB Debugging utilities

### Timings and Interrupts

The start of the Gameboy memory bus (Table 1) is used to process the 5 different interrupts that can occur. These Interrupts are controlled through the Interrupt flag address found at FF0F. The Interrupts enabled flag at the end of the cartridge Dictates whether Interrupts are used throughout the whole system. This was a key part to the implementation of Tetris 5.3.1 further on in the project. All interrupt handling occurs before the CPU processes an instruction, moving the PC register to the to handle the interrupt. There are 4 components that can interrupt the CPU, the Display, Timer, Serial port and the Joypad. This is done through setting bits of the Interrupt flag register to represent if an interrupt has occurred. A quirk of the Gameboy hardware known as the “halt bug” is an unintended consequence discussed later during the implementation 5.3.2.

### The GPU/PPU

The original Gameboy featured a green LCD screen 144x160 pixels which was. The original Gameboy used 4 different colours to shade the pixels of the screen. This was done by mapping the colours to two bits of an 8-bit integer to create a number from 0-3. This number represented the colour of the pixel drawn to the screen. In comparison to this, the Gameboy Color allowed the 32 different colours to be shown on screen using the 0bAARRGGBB format of a 8-bit integer. (Alpha-Red-Green-Blue) (Petar Veličković. (2016).). The screen data was outputted through the PPU (Pixel Processing Unit) The output is based on tiles that are 8x8 pixels in size. The tiles are encoded by each line using 2 8-it integers to store the colour information of each pixel by a process of combining the bits of the integers together. This data is retrieved from two tile maps stored in the VRAM of the memory bus. The screen size is not representative of the complete view, as the background map in the video ram uses a 256x256 pixel map, this allowed the smaller screen to move around the background to give the appearance of scrolling. (Michael Steil. (2016).)

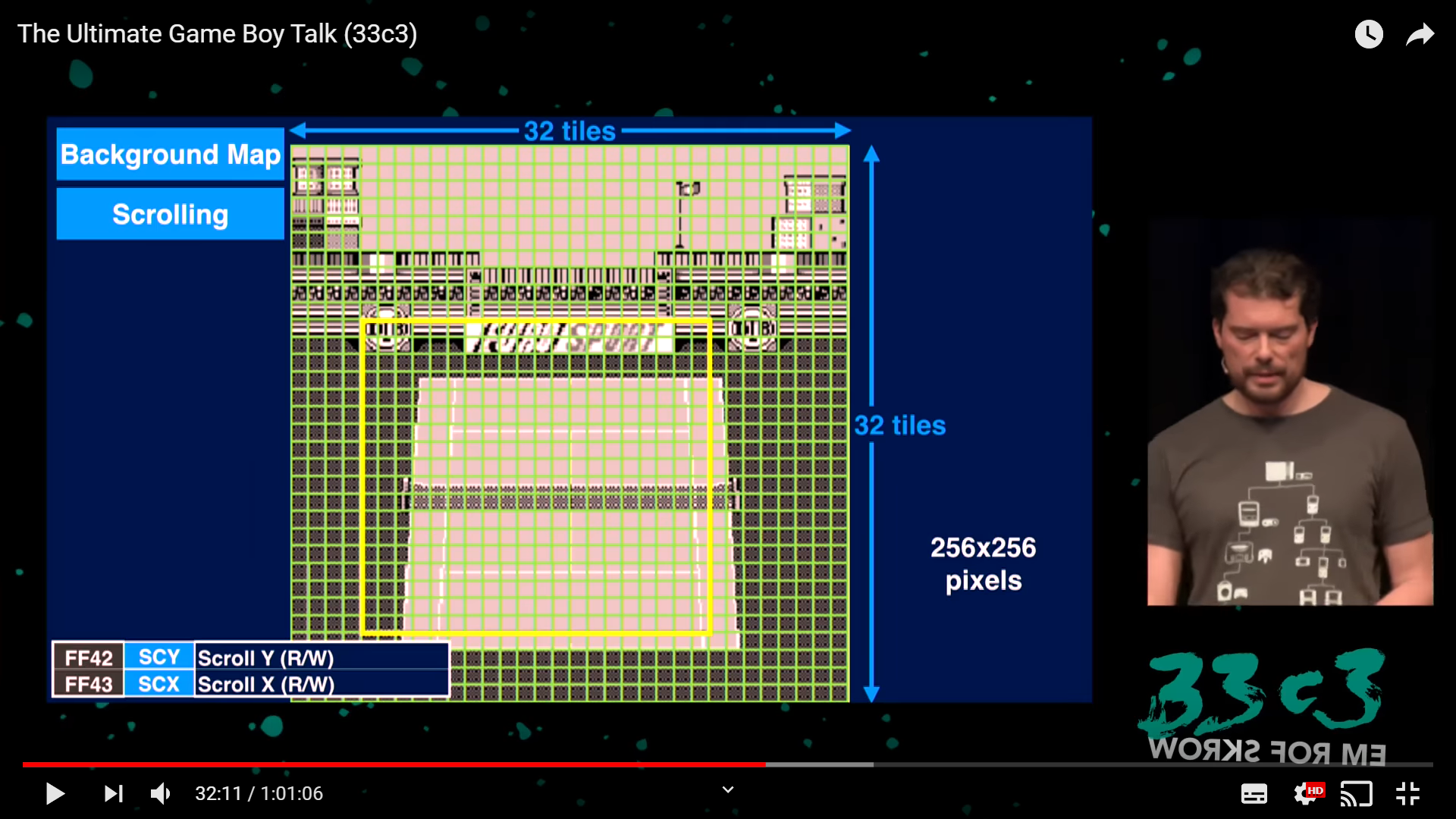


Figure 3 Example of the Gameboy scrolling from “The Ultimate Gameboy Talk”

A single frame of the display is drawn using a scan line technique. Each line of pixels is drawn horizontally to the screen going from top to bottom. This is important as each phase of the requires a specific number of CPU cycles to be completed. The screen uses a window layer that is drawn on top of the background layer allowing data drawn to remain static on the screen such as the score of the player. Finally, the sprite layer allows objects to be drawn onto the screen. The sprite data uses several attributes to modify the sprite such as flipping how it was drawn. This allowed tile data to be reused and reduced the number of tiles needed, saving space on the cartridge.

### The Cartridge

The memory bus of the CPU was a key component storing information related to all the components of the Gameboy, such as whether a button was being pressed, or even if the display was currently turned on. This limited area of memory used two 16kB banks to hold the data of the game from the cartridge. This was fine for games that only needed the ROM itself such as Tetris as the game would fit snuggly onto the bus without any issue. However, this put a limit on the size of game that could be developed. The solution to this was a memory banking system. Although the first memory bank in the bus is fixed and cannot be swapped out, the second bank can be replaced through the MBC (Memory Bank Controller. The MBC is fascinating as it is used by intercepting data that is written to the ROM memory even though the ROM data is read only. This data is then parsed by the MBC and allows various functions to be performed such as changing the ROM or RAM bank. Certain cartridges also featured the use of a battery that allowed the use of an RTC (Real Time Clock) that games could take advantage of. The various cartridge types have been archived showing the requirements to run each game with the cartridge type “MBC1” being the most popular.

### The BIOS

The BIOS is a routine that is ran every time the Gameboy is booted up. The BIOS is used perform several tasks to initialise the CPU. Such as, clearing random values from memory, setting up initial values of the different registers, and scrolling the Nintendo logo down the screen. This routine is more than a visual graphic as the BIOS uses the data from the logo to check that the game is legitimate. This is done by comparing the Nintendo to the Nintendo logo that is stored within every Gameboy game. If there is a discrepancy the routine locks up and the game does not continue. If the BIOS manages to pass this inspection, the BIOS is then overwritten by the fixed ROM bank stored on the cartridge.

### User Input

The Gameboy featured 8 buttons that allowed the user to control the game. The implementation of this is done through the address FF00 (JackTech. (2016). The game boy, a hardware autopsy - part 2: Memory mapping.). The first two values of the byte are not used, while the third and fourth bit are flags that are used to divide the buttons into two groups of 4. (Up, Down, Left, Right and A, B, Start, Select). These bits act as flags for each button to signify if the button is currently being pressed or not. The Joypad is polled every CPU operation to check if there have been any changes to the current state of the system, such as a button being pressed or released. Figure 5 in section 4.3.4 provides a visual representation of the button assignment.

## Summary

This chapter as looked at the techniques that can be used to preserve video games for future generations, looking at emulation as a solution to this problem. Additionally, the chapter provides insight into the history of the Gameboy and its components to gain insight into how the system works. This preliminary research will be the foundation of understanding needed to develop the design of the emulator and its implementation discussed later in the report.

# Project Planning

## Introduction

This chapter focuses on the planning elements that were used to conduct the project. This takes into consideration the approach of the project as well as the potential issues of creating an emulator and how they can be avoided.

## Methodology

There are various factors that need to be considered when planning out the approach of tackling the emulator. The research into the Gameboy showed that the scope of potential features that the Gameboy could have meant that focus would be needed on the components that were required to allow the Gameboy to function. This meant weighing up the costs and benefits of implementing a feature versus the costs of implementation. Therefore, although the project is linear lending itself towards a waterfall approach of moving forward in a specific order, components of the agile methodology can be used to allow development of extra features based on time and scope. This allows features to be sacrificed in order to maintain the quality of the overall product. Using the MoSCoW analysis technique, this means that the features of the Gameboy can be given a weighting based on their contribution to the overall system. Furthermore, this allowed the different areas of the Gameboy to be timeboxed in order to create an overall time estimate for their implementation. This requires consistent revaluation of the allocated time boxes in order allow for problems in development such as debugging and changes in scope to accommodate for the time scale. This contrasts to the waterfall methodology that does not allow refactoring of the requirements in this way. However, the architecture of the Gameboy requires that certain components are implemented before others can even begin to function, as the CPU is the core of the system. Therefore, the development of the base components lends its towards a waterfall approach as components must be implemented to create an overall product.

## Requirements

The sheer size of the potential scope of this project means that prioritisation of the requirements is vital to stop the requirements from being unfeasible to implement. Therefore, below is a MoSCoW analysis of the overall features that can be implemented as well as justification for the features that will not be focused on.

### Must have requirements

* The emulator must be able to emulate the bare bones features of the Gameboy for a user to be able to interact and play with the emulator. The components required for this are the CPU, Display, and Cartridge. This implementation may have glitches and visual errors but no problems that affect the overall playability of the game (crashing or not running at all)
* The Gameboy utilises many different cartridge types, however, the emulator must be able to run ROM only cartridges as a bare minimum as these do not require any memory banking functionality.

### Should have requirements

* The emulator should implement memory banking to allow a larger variety of cartridges to be played. The focus would be on MBC1 type cartridges as the number of games it provides access to is greater than other cart types.
* The emulator should be able to save the state of the game being played for the user to save their game progress. This would improve the playability and overall experience of playing longer RPG games that may require more than one sitting to complete.

### Could have requirements

* The emulator should be able to pass some if not all the blargg’s test ROMs to increase the compatibility of the emulator with various games and as a baseline for the emulator implementation.
* The emulation of Gameboy color games could be implemented to increase the overall number of games the user is able to access.
* The Gameboy could implement support for additional cartridge types (such as MBC3 and MBC5) and include battery timings in order to allow a greater number of games to be played.

### Would have requirements

* The emulator would be able to emulate sound given a greater time frame to allow for research and implementation. Emulating the sound of the Gameboy is a difficult area as the original hardware used electrical signals to create sound. Therefore, although the sound would improve the overall experience of the emulator, the cost of implementation is not worth the benefit it provides.
* The scope of developing the Gameboy to be completely accurate to the original product is an extreme undertaking and would be feasible with years of research to perfect the timings and quirks of the Gameboy. However, the project’s focus is to be able to develop an implementation to focus on functionality and playability and therefore the accuracy of the emulation will not be a main priority.

## Potential Solutions

The emulation of the Gameboy system is based interpretation of the ROM information that Gameboy uses in order to replicate the system. Therefore, the bare requirements for the language to develop the emulator are the facilities to replicate data types that the Gameboy used, such as 8-bit and 16-bit integers. Therefore, the decision was made to develop the emulator in C++ as it allows low level access and data types that lends itself to the architecture of the Gameboy and it is the language that I am most familiar with. In order to create the display of the Gameboy, graphical libraries were considered such as the QT library that is a framework that provides advanced capabilities to create a GUI, however, the SDL2 library will be used as it provides the basics in order to control individual pixels which is needed to replicate the display of the Gameboy and provide input. The use of SDL2 means that the emulator has the potential to be used cross platform for different operating systems such as Linux. Thus, this would require additional research into how this can be done, therefore, should be reasonable to implement. Alternative languages that could have been used are Java, the benefits to using Java would be the ability to develop the display without the requirement of additional libraries through JFrames and, the portability that Java provides. Finally, JavaScript could have been used to allow the emulator to be hosted on a website and accessed without any download necessary. However, these two languages would require additional research to understand the fundamental tools for development and with the time constraints of the project would not be realistic or efficient to use.

## Tools and Techniques

The development of the emulator will require various tools, the first of which is an IDE which will be visual studio because of familiarity and the ability to add breakpoints which are vital to the debugging process. The use of the SDL2 library means that the project has the potential to be cross platform although this is classed as additional functionality and is not a priority of the project. In order to test and debug my emulator, a debugger will be required in order to compare the state of my system’s registers and current state against. The initial choice of debugger was no$gmb to provide a view of the VRAM and the allow breakpoints throughout the program. However, this was changed during the implementation 5.2.2 to the BGB emulator due to inaccuracies discussed later during debugging. Additional tools that will be required are a hex editor in order to view the data of the ROM, to make the process of translating numbers from hex to decimal to binary easier the windows calculator provides a utility for programmers to aid with this.

The project will require various management tools over its lifespan in order to break down and manage the tasks. The first tool considered to keep track of the tasks at hand was Trello. This utility uses the Kanban board system in order to prioritise tasks and on paper goes hand in hand with the timeboxing that will be utilised throughout the project. However, due to circumstances that could not be controlled, the use of the Kanban board did not really work as the effort and maintenance needed did not feel suitable. Although, I believe that for a different emulation project that required multiple people to be involved, a Kanban board would be a strong asset for task allocation and organisation. An alternative technique that could’ve been used to visualise the tasks of the project is a Sprint board, this would be suitable for the smaller components of the Gameboy, however, development of the overall architecture is a continuous process meaning that the usefulness of the board would be limited. The second utility used to manage the project was GitHub, which provided the ability to implement version control to revert changes back. This is useful for reverting code that was implemented incorrectly. The cloud storage allowed for development to be not restricted to a single computer allowing it to be accessed anywhere. GitHub also provides the facilities to prevent files from being uploaded when committing updates. This is useful for the project as Game ROMs can be specified to not be uploaded meaning that they are not distributed but can still be accessed.

## Legal, Social and Ethical Issues

### Emulating the Hardware

There are several ethical issues that may arise from this project. First and foremost, the emulator itself does not break any laws as it simply recreating the Gameboy architecture. However, the distribution of the software could potentially enable other users to break the law through illegally pirating games. Therefore, users will be explicitly informed of breaking the copyright law when using the emulator.

### Software

The project aims to emulate the hardware of the Gameboy and therefore, fundamentally a Gameboy ROM is required. The use of ROMs is a bit of a legal grey area, (Conley et al (2003).) maybe not this one necessarily) as the ROM is copyrighted material it is illegal to download a copy of the game. However, from an ethical standpoint, if the user owns a copy of the game, a digital copy of the game could also be created under fair use. This digital copy would then be used for the emulator.

Finally, the Gameboy hardware uses a BIOS load screen featuring the Nintendo logo. A copy of this sequence also exists on every Gameboy ROM. The BIOS load screen is checked against the ROM to make sure that the game is valid by checking the values are the same. Therefore, care must be taken while creating the emulator so that it does not infringe the Nintendo logo copyright. This can be worked around by creating an instance of the emulator with the initial values already set, eliminating the need for the BIOS.

Overall, this project will be focused on the educational aspect of design and implementing an emulator and therefore any copyrighted material will not be made public thus requiring the user to provide their own ROMs to test or play. The use of copyrighted material such as the ROM for Tetris will be invaluable towards the development of the emulator and demonstration of its capabilities. It can be argued that as the use of the ROMs will be for “research study purposes” However, under the circumstances that a commercial ROM cannot be used, free public use ROMs exist that enable the use of the emulator without breaking copyright law. The problem with using public use ROMs however is that they have the potential to not be created to the same standard as commercial made games. Therefore, this compromise in quality may mean that the ROMs may be inaccurate.

## Summary

This chapter of the report has discussed the approach using aspects of both waterfall and agile methodologies to suit the needs of the project. The requirements of the project have been prioritised based on a MoSCoW analysis focusing on the features that will provide the greatest benefit vs the time and efforts costs. The various tools that will be required for this project have been laid out and the legal issues have been discussed to show that the development of the emulator itself does not break any copyright laws

# Design

## Introduction

The Gameboy uses the CPU chip to perform much of the processing needed by the Gameboy. This means that the control of the memory bus, input and display are communicated through the CPU itself. This technique of putting a “system on a chip” is still prevalent to this day and can be seen in technology such as smartphones and the raspberry pi. The design of the emulator will focus on breaking down the different components of the Gameboy in different classes that are able to communicate with each other supporting an object-oriented approach.

This chapter will focus on breaking down the elements of the Gameboy and will look at the potential methods of implementation using modern tools and techniques and the reasoning for why they would support the development of the emulator.

Appendix (x) shows a class diagram of the overall emulator design. It also highlights the difference between the design and implementation as discussed in section 5.2.2

## Breaking down the Gameboy internals

The CPU is the heart of the Gameboy and performs many of the vital operations required for the system to function. Therefore, the CPU will be used as a starting point of development as a foundation for the other components that will be discussed later in the report. The Gameboy architecture lends itself towards an object-oriented approach as each component will be created using a class for organisation. These components will then be implemented using a “controller” class. It should be noted that during the implementation 5.2.2, this approach was changed to combining components of the Gameboy that were closely related to allow implementation to progress.

### The Registers

The CPU uses 8-bit registers to carry out various mathematical operations and can be combined to create a 16-bit register to move around the memory bus. In addition to this, there are two 16-bit registers that are used as pointers to both the current program instruction and the stack. Using these registers is the bread and butter of most of the functions called by the CPU, therefore, accessing them will be used heavily throughout the emulator. Therefore, the implementation of these registers must be easily readable to reduce confusion when programming and debugging. The registers could be implemented through an array and combined through binary arithmetic when required to be combined as a 16-bit register. However, implementing a method of combing the registers would require using binary arithmetic and could be prone to errors. The second, much friendlier option of implementing these registers would be to use the Union datatype that C++ provides. A union allows smaller parts of data to have the same bytes as the largest data member in the union. The registers are only combined with a specific register; therefore, the union can combine the 16-bit data with the two 8-bit registers allowing for easy access. However, the setup of the union may cause registers to be combined incorrectly which is discussed later in the implementation.

### The Memory Bus

The memory bus is the main area of the CPU used for storage and retrieval through both read and write functions. This is then broken down into different address ranges that relate to various components such as the ROM, display and interrupts. Similarly, to the registers of the CPU, a Union will be used to divide the sections of the CPU (as seen in Table 1) to increase the readability of the memory bus itself.

Reading data from the bus will be simple enough as this is returning the data from the address on the bus specified. Writing to the bus will require more thought as there are several methods that this could be done. Most of the bus will not require specific attention and can be written to inserting the data to the bus. However, certain address ranges will have different functions that will require additional parsing to perform functions such as switching from the boot sequence to the ROM data or accessing video registers that control whether the display is on. Therefore, a check will be needed to see if writing to these address ranges. The obvious choice to implement this is to use “if else” statements however a switch statement could also be used to reduce the number of checks when writing to the memory bus.

### Processing Instructions

The CPU uses a “Fetch-Decode-Execute” method in order to parse information from the ROM. There is a total of 256 base instructions (Opcodes) that can be called with an additional 256 codes accessed through the 0xCB operation. The focus and challenge of implementing this cycle will be making sure that the operations are performed in the correct order.

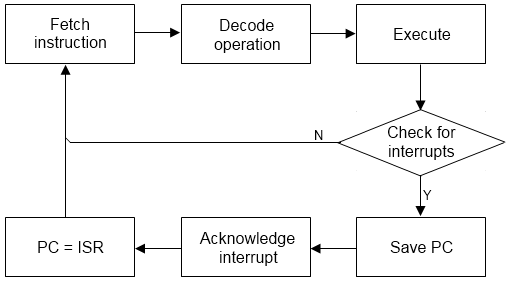


Figure 4 Diagram of the Fetch-Decode-Execute cycle (Created by Imran Nazar)

To begin the cycle, the operation is fetched from the ROM or BIOS as the point where the PC (Program counter) register is pointing. The PC is then incremented once or twice depending on whether the code was a CB code or base code. This will be done by checking if the code retrieved is “0xCB” and then incrementing the PC register to read the next byte according to the CB operations. The next step of the cycle is decoding what the instruction means. Depending on the type of operation, additional data may be needed from the ROM. There are multiple different ways that this could be implemented, the first and least efficient being an if-else chain comparing each operation to the Opcode. This would make the code for handling the CPU difficult to read and furthermore would take a long time to create compared to other implementations. The second implementation that could be used is a switch statement which would reduce the number of comparisons however would still cause a large amount of code to be clumped together.

In order to solve this issue of execution, through looking at the different operations, many of them (especially the CB codes) perform the same function with only the variables changing for each instruction. For example, many of the bit operations found in the CB opcode table can be condensed into a generic BIT function greatly improving the reusability of the code and making it much easier to read. Thus, using a combination of a switch statement along with the use of generic functions will be most elegant solution. The final implementation would use a function pointer as all the instructions would be handled through generic functions and therefore would be able to have the same signature. This would mean that the operations could be stored in a separate file in order which is simple to read and search through. Overall, the switch statement and function pointer solutions would both be strong implementations to parse the opcodes and the use of generic programming will be key to reduce irrelevant code.

### Interrupts and Timings

Within the Fetch-Decode-Execute cycle of the Gameboy, there are flags that can be raised by the different components such as the display or joypad. This is controlled by the 0xFF0F register. For the interrupt to trigger, they must before first be enabled, and occur before the next opcode is read. This caused problems later in the implementation and testing. The interrupts themselves move the PC register to a point on the bus for the specific interrupt. Three are additional quirks to the interrupts that caused problems and will be covered later in implementation section.

The timings of the CPU will be a significant challenge as the time taken for operations corresponds to the timings of the display discussed later. Additionally, the CPU uses an internal timer that is incremented after the operation has been completed. Although the focus of the emulator is not to be perfectly accurate these timings will have to implemented “well enough” to allow games to function properly. This area will require in-depth research to implement and may have to be done as operations are completed.

### The BIOS

Although the BIOS is not strictly needed, it is used to set up the initial values of the different registers and will cause the system to lock up if not initialised properly. Once the BIOS is finished the first 256 bytes of data are switch out with the ROM. As the BIOS is copyrighted, the work around for this will be to set the initial values of the registers manually, removing the BIOS requirement. Further on in the report during development 5.2.1, the BIOS was a key asset in the early stages to getting the emulator up and running.

## The Cartridge

### ROM only games

The cartridge contains all the data required for the emulator to function with various types of cartridge existing based on the requirements of the game. The initial focus will be on ROM only games which only use two banks and can be copied into the memory bus directly. (Table 1) The data on the cartridge is stored in binary therefore a file stream will be used to extract the data to an array using the binary mode so that the data is can be correctly interpreted. The data will then be copied directly onto the memory bus using the memcpy function. At the beginning of the ROM file is the cartridge header, this provides information about the ROM including the whether the ROM implements memory banking.

### The MBC (Memory Bank Controller)

The MBC is a key component required to expand the library of games the emulator will be able to run. As discussed in the preliminary research 2.9.5, the memory bus of the CPU only allows 32kB of data from to be stored onto the bus. The way this is worked around is using the memory controller to switch the second ROM bank when data is written to the ROM memory. This information is intercepted by the MBC which interprets the function it must carry out based on what address range is written to. These functions can include changing ROM banks, ram banks and toggling whether RAM has been enabled altogether. Because of this, the MMU will require information from the cartridge header in order to determine what kind of MBC will be required. Additionally, the MBC must be able to control the read and write functions in order to adequately change between ROM banks. Due to the vary sizes and attributes of each cartridge, the MBC lends itself towards a polymorphic approach using a base “memory chip” class to develop the memory bank controllers for each cartridge type. The benefit of this approach is that future development to support different cartridge types can be added without requiring code to be rewritten to support this. The exception would be in the case of cartridges that require batteries as RTC timings would have to be implemented. When implementing the MBC, there are two different approaches that could be taken, the first would be to completely move the ROM bank directly into the memory bus. The advantage of this is that any additional requests to this bank can be accessed directly from the bus itself meaning reading data from the ROM would not require a specific read function. The second implementation would be to read directly from the cartridge ROM bank, this would relieve the need to copy the data directly to the bus but would require the read function to be worked around to access data from the ROM bank. Overall, as the data transferred to the memory bus is not significant, the copying data to the memory bus approach will be taken.

### Saving the game

The Cartridge of the Gameboy also had the potential to feature RAM memory to store various values relating to the current state of the game. This functionality will be required for games such as Pokémon that can be played over more than one sitting. The RAM banks of the cartridge are accessed through the MBC class and therefore will need to be checked and parsed when the cartridge is initialised. A method of storing this data so that it persists after the Gameboy is closed will be required to allow the user to save their progress. This can be implemented through using a file stream to store the data to the hard drive within the folder that the game is stored. This will allow the emulator to check the folder on start-up and load the saved file to RAM if there is one saved meaning the player can continue where they left off. The option to choose the current save could be a feature of a GUI based implementation discussed later in the design. However, saving and loading the state of the game could be bound to keys using SDL\_Event to allow the user to save and load on the fly.

### Controlling the Gameboy

The Gameboy uses 8 different buttons as input. These buttons are mapped to the lower half a specific register on the memory bus, 0xFF00. The two sets of buttons are switched using the 5th and 6th bits of register. The first 4 bits of the register represent the 4 buttons that are currently enabled. These are either directional buttons, or the input buttons. This is easier to understand visually through the figure shown below.

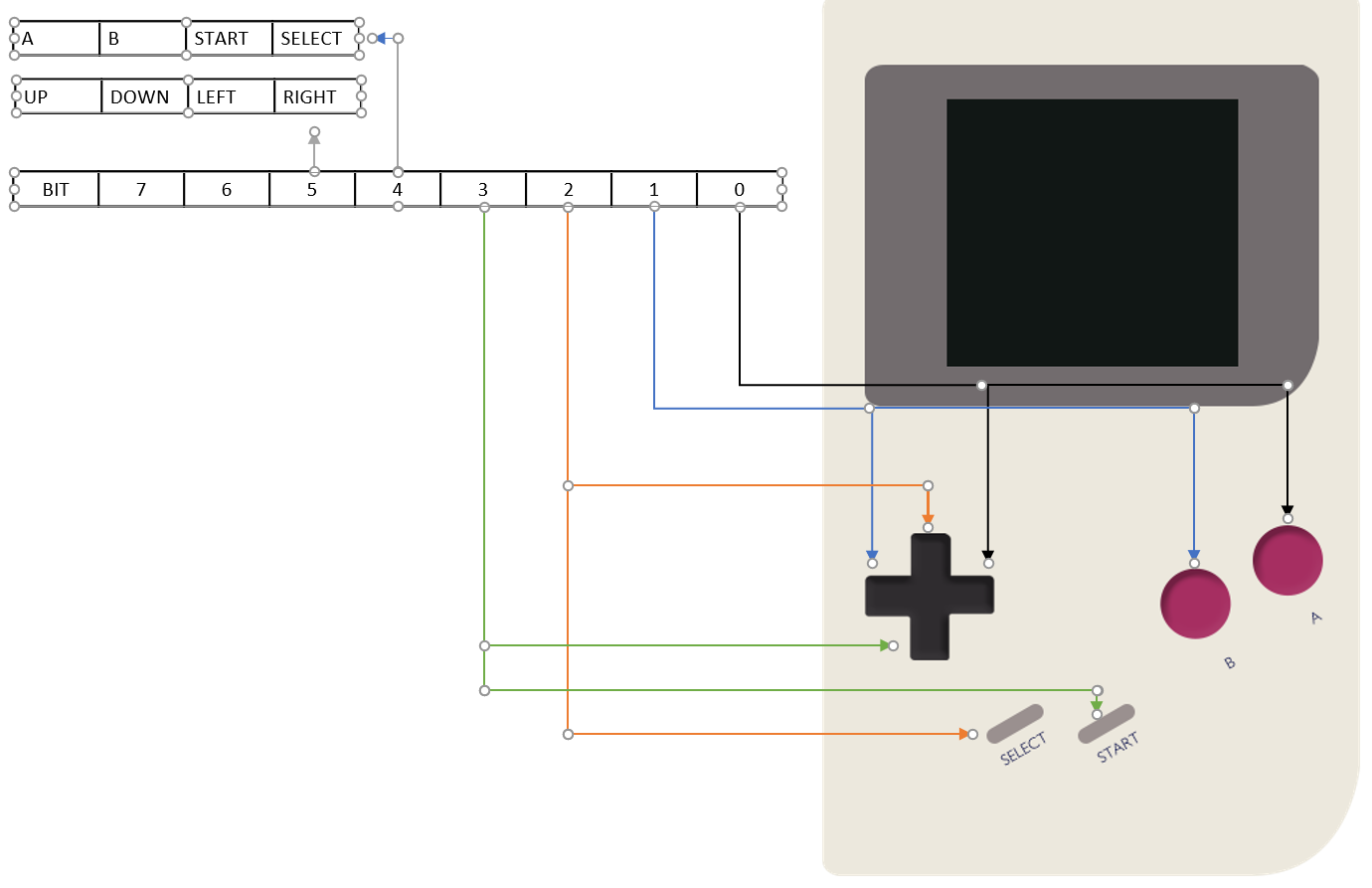


Figure 5 Gameboy Button to Bit mapping (Representation created using the GB Dev Wiki)

The SDL library allows for user input to be polled using SDL\_Event, to check if a key on the keyboard is being pressed. In order to implement this input, the bits of the 0xFF00 register kept within the I/O section of the CPU’s memory bus will be set and unset using binary arithmetic. The bits controlling the which buttons will have to be checked before the buttons can be set. Overall, this seems to be the best method for implementing the controls of the Gameboy.

## The Display

The display of the Gameboy, otherwise known as the PPU (pixel processing unit) works in unison with the CPU by using cycles to dictate which mode the display is currently in. There are multiple registers that control and manipulate how the display behaves, these are stored in the CPU as I/O registers meaning that communication between the CPU and display will be important. Therefore, access to read these registers will be implemented through getter and setter functions so that only specific data related to the component can be accessed. As discussed in the implementation 5.2.2 section of the report, this was ultimately more difficult to implement initially thought.

### Drawing a Blank

The LCD Status register (0xFF41) is used to control display interrupts and dictate which mode the display is currently in using the upper 4 bits of the register. The 4 modes cycle continuously drawing the pixels of the Gameboy from horizontally down the screen from left to right. Once a row of pixels has been drawn, the display enters H-Blank which is a delay used that pauses the display whilst allowing the CPU to continue functioning. The LY register is used to keep track of the currently line that is being drawn. When the final row of pixels is drawn, the display enters a V-Blank period. The key part of implementing these modes will be managing how to the timings are done. In order to draw the pixels, the display accesses the CPU directly through the third mode called OAM and then drawn using the LCD transfer mode. The OAM mode is not required as the information can be parsed directly when the line of pixels is drawn. However, the timings for the OAM mode will still require implementing. Therefore, the main bulk of drawing the pixels can be done using the LCD transfer mode.

The LCD transfer draws the pixels to the screen using 3 different layers, the background, window, and sprite layers. This will be the most difficult part to implement of the display as it requires parsing of the tile data required for each pixel. If this is implemented incorrectly, this could cause problems with debugging as it may be unclear which component is not working correctly. As the 3 layers of the Gameboy all use tiles of 8x8 pixels to be drawn, a generic function can be used to render the tiles of the Gameboy using similar data such as the palette, tile set, and the address in memory where the tile data is stored. This functionality may be difficult to implement with sprites as they have additional data to parse stored in the sprite attributes table that contains information on the direction the sprite is facing and a 8x16 or 8x8 mode that determines whether two different sprites are vertically on top of each other.

Once this has been completed the data will be stored in an array the size of the screen viewport. The benefits of this is that the data can then be interpreted by a graphical library, in this instance, the SDL2 library to draw the individual pixels of the Gameboy.

## User Interface Design

There are two ways to go when thinking about the UI for the emulator. The first path would be to create a full UI for the Gameboy that allows ROMs to be selected with functionality to save and load games. This could be implemented through IMGUI to create a GUI suitable for the user. The second option would be to implement a barebones UI using the console to pass the ROM specified to play. Overall, both implementations will still allow additional features such as controlling the speed of the game, changing key bindings, and saving the state of the game through SDL\_Event. The decision to implement either design will be based on the amount of time left once the base Gameboy has been implemented as although the GUI will make the emulator more user friendly it is not a full requirement and can still function without it. With the implementation of a barebones UI, the user will have to be informed of the various buttons beforehand.

## Summary

This chapter has discussed the various components of the Gameboy and the potential methods of implementing them using modern day languages. The areas of implementation that had different methods were explored weighing up their costs and benefits reasoning why they should be implemented that way. Additionally, the structure of the emulator was discussed exploring how the different components will communicate with each other.

# Implementation

## Introduction

This chapter will look at the implementation of the various components focusing on the aspects that worked well as well as the difficulties encountered during development.

## Initial Development

The C++ data type int uses 32-bit integers by default which would be difficult to use for the binary implementation. Therefore, in order to start working on the implementation of the CPU, base data types were needed for both unsigned and signed types for the 8-bit and 16-bit integers of the Gameboy. This could have been implemented using existing data types such as characters. However, the choice was made to use a typedef to define the datatypes used for convenience to avoid typing out the full datatype name and make the code more readable.

typedef unsigned \_\_int8 ui8; //8-bit Integer

typedef signed \_\_int8 i8;

typedef unsigned \_\_int16 ui16; //16-bit Integer

typedef signed \_\_int16 i16;

Listing 1 Defining Datatypes

The base functionality was implemented using unions as mentioned in the design section to create two arrays of 8-bit and 16-bit registers together.

union

{

// 16-bit registers array

ui16 register16bit[6]; //AF/BC/DE/HL/PC/SP

// 8-bit registers array

ui8 register8bit[8]; //A/F/B/C/D/E/H/L

};

Listing 2 Register Union

The different registers (and flags) were assigned using constant values to make it clearer when they were being accessed. A map of the memory bus was created using arrays and unions to partition the various sections of memory although, in hindsight, this was only necessary for access to the Joypad but did provide a clearer view of the memory bus itself.

The logical starting point for development was to look at the BIOS sequence of the Gameboy itself. The BIOS is 256 bytes long. The disassembled Boot ROM found on the GB Dev Wiki provides a logical view of the instructions that the sequence goes through. This was useful as I was able logically follow along with what the emulator was doing versus the actual output. Initial I focused on trying to get through the CPU implementation as fast as possible so that at least something could be displayed, however, this approach sacrificed many features that were critical to debugging and gaining understanding of the system.

### Foundations

The starting point of the project was the CPU. This also meant that development could start without the implementation of the cartridge class as the BIOS could be put straight into the bus. Basic functionality of the Fetch-Decode-Execute cycles was implemented through the Tick function to parse the data of the bios. This was done by using a pointer to represent the program counter register to access data on the memory bus. Finally, a switch statement was implemented as not all codes were required to be initialised unlike a function pointer and Opcodes could be added when needed. This approach was taken as implementing all the Opcodes at once would be extremely difficult to debug as there could potentially be many points of failure affecting the program not working properly. The implementation of the various Opcodes required base functionality such as reading and writing to the memory bus and accessing registers.

void CPU::ProcessCode(ui8 OPCode)

{

switch (OPCode)

{

case 0x00:

break;

case 0x11: // LD DE, ui16

SetWordRegister(DE\_REGISTER, ReadWord());

break;

default:

std::cout << "Current OPCode is: " << (int)OPCode << std::endl;

system("pause");

}

}

Listing 3 Switch Case approach

Instructions were implemented using the Izik Opcode table that provided a more compact and readable view of the Opcodes and their timings in comparison to the Gameboy CPU Manual. and the instruction timings were added through the table of instruction timings found on the GB Dev Wiki. As a CPU cycle of the Gameboy 4hz the values from the table were multiplied by 4 to get the cycle timing. It should be noted that the Gameboy Color is twice as fast as the original Gameboy, therefore meaning that if GBC mode is enabled, cycle timings need to be divided by two to accommodate for this. This approach allowed the development throughout the initial parts of the Gameboy BIOS to progress. However, the BIOS utilises jumping to different areas of memory and loops to process data. For example, the first loop in the BIOS uses the A register to initialise all the values in the VRAM to 0. Therefore, a lack of proper debugging tools meant that progress was slow. The functionality of the CPU was being misinterpreted as incorrect due to lack of understanding and instructions were implemented wrong meaning various flags and data was not being set correctly. The benefit of this initial flailing to get something working was that it helped improve my understanding of binary arithmetic and initial misunderstandings of the order of bits. However, in order to speed up development and reduce the frustration that was building from the lack of results, a change of approach was required. A resource I used to gain better understanding of the various instruction implementations was Z80 Heaven which provides insight into the functions of the original Z80 chip. Although this resource provided understanding of the binary instructions, the Gameboy CPU is modelled slightly differently to the original Z80 and therefore some actions are altered or even removed. Therefore, I looked at the implementations of the operations in other emulators such as the Gearboy to confirm that my implementation was correct.

### Re-evaluating my approach

I was introduced to a discord server that focused on the discussion of emulator of various consoles, including the Gameboy. This provided a world of resources that aided in the overall development and understanding of how the Gameboy worked. The switch statement that was used was scraped for a function pointer implementation. This was simple enough to switch to as approximately only 30 codes had been implemented at this time and the functionality of working codes could be easily copied over.

typedef void(GB::\*OPCodePtr)(void);

OPCodePtr BASECodes[256];

OPCodePtr CBCodes[256];

void GB::OPEB() {assert("CPU Instruction not implemented" && 0);};

Listing 4 Function Pointer Approach

In addition, this allowed the use of assert statements that exited the program once a new Opcode was reached allowing easier debugging and provided insight as to whether the program was getting stuck.

Furthermore, a debugging function was used to display the current state of the system such as the registers, I/O registers, and flags using the bit set datatype from the STD library). Although the functionality of the debugging tool was limited as the console was much slower at outputting the system status, this allowed for numerous bugs to found in the implementing functions to continue progress through the BIOS. The final and most important resource that greatly improved the development getting through the BIOS was the BOOTROM disassembly found on the Realboy development blog, that explained the functionality of the different areas of the boot up sequence. This was an amazing resource to find as not only did it give a brief synopsis of what was happening but also why. By implementing these techniques to my emulator, this allowed the development of the emulator to progress to a point where the Display was turned on.

bool GB::TickCPU()

{

cycle = 0;

CheckInterrupts();

OPCode = ReadNextCode();

cycles += cycle;

if (OPCode == 0xCB) // Secondary Codes

{

OPCode = ReadNextCode();

(this->\*CBCodes[OPCode])();

}

else

{

(this->\*BASECodes[OPCode])();

}

TickClock();

bool vSync = updatePixels();

JoyPadTick();

}

Listing 5 Updated CPU Implementation

### The display (Getting visual)

Now that the display was turned on, this allowed the display class to finally be implemented. Although I had researched about SDL2 I had never used the library before and required some basic tutorials to get up to speed. This allowed the initial set up of the library to get the display class rolling. This would not be used until later in the development but a preliminary understand of the setup was beneficial and would also be used for the Joypad discussed later. The first point of action was to set up a switch statement to determine which state the Gameboy was in, like the initial implementation for reading Opcodes in the early days of development. The 4 modes would progress to the next in a cycle whilst using the instruction cycles from the CPU for the mode timings. This required a reference to the CPU to be stored within the Display class itself (causing problems further down the line). Once these 4 modes were implemented, a one-off buffer timer was added to mimic the delay of powering on the display.

### Processing the pixels

The next part of the implementation was arguably the most difficult to understand, the parsing of the tile data. The information obtained from GBDev Wiki was a valuable resource in understanding how the various aspects of the display come together. The resource was a little daunting to understand at first, but eventually I was able to understand the intricacies of the display component. The framebuffer array was set up using a struct to store the RGB values of each pixel. The function “drawScanline” is used to implement this, first checking if the bits of the LCDC are set for the 3 different layers of the display. Functions for the Background and window were created to parse all the information needed in order to carry out drawing the pixels. This is then passed to function to render the tiles. Appendix 3 shows how the tile rendering was implemented. The tile rendering was done by determining the current position of the tile through a combination of the LY register and the current X coordinate on the screen. The address for the tile data is then retrieved from the tile map. There are various factors that can affect this such as which tile map is currently being used or if the tile unsig flag is checked meaning the address should be offset. The display combines the bits of two 8-bit integers to determine the colour of the pixel. The easiest way to understand this is through the representation below.

Table 2 Visual representation of combining bits to get the pixel colour

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0x8000 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 |
| 0x8001 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
| Colour Drawn |  |  |  |  |  |  |  |  |

This process was repeated for the window layer as it is drawn similarly to the background. Appendix 4 provides a more in-depth view of the tile rendering process. Following on from this, the sprite layer required more thought as each sprite had individual attributes to consider. Therefore, the sprite layer was done through a separate function. Although this worked on paper, this implementation required data to be written to the CPU, as the display already had a reference to the CPU another class would be required to control reading and writing data to the memory bus for the display and CPU to be cohesive. As the Gameboy does not specify the values of the colours passed to the screen, this allowed the palettes to be created changing the colour scheme of the games as shown in figure 5 below



Figure 6 Palette Swapping

### Re-evaluating the structure of the system

The problems with writing data to the CPU from the display meant that a decision had to be made. The options were to implement the memory bank controller early, to act as a middleman for reading and writing data however, this meant implementing a large chunk of code even though the emulator wasn’t even displaying anything yet. Therefore, the decision was made to completely change the design of the class-based interaction to moving the display and CPU together to allow easier communication. The justification I had for this change in design is that many of the components require access to each other in order to function, therefore although the emulator would be class based, they would all have to reference the components they related to creating more hassle than needed. The cost of this decision was that the main cpp file of the emulator would be significantly larger making finding specific parts more difficult however, this meant that the CPU and display would be able to communicate directly. Although the communication issue was fixed, there were still several bugs that were preventing the pixels to be drawn properly, such as colour values being reversed, tile data being parsed incorrectly and incorrect implementations of CPU functions such as cp not setting the zero flag register properly causing jumps in memory to not function properly. Appendix 4 shows the gradual development of the display at this stage.

### The Cartridge Class

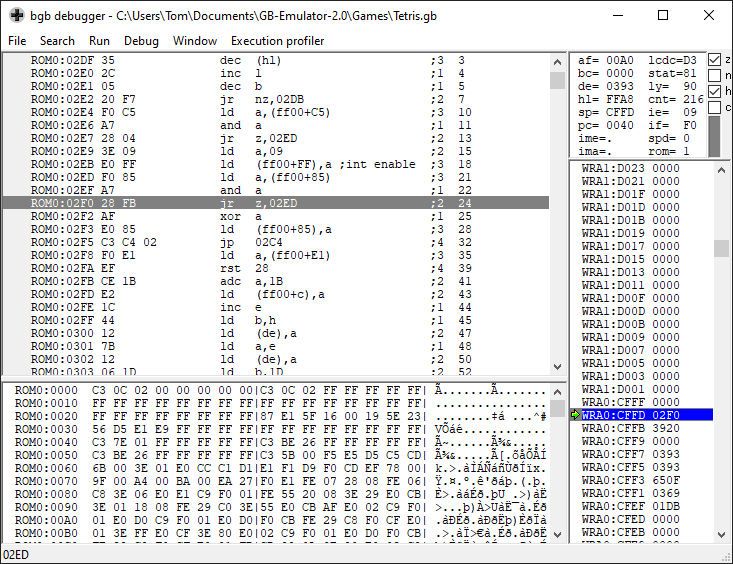
Now that the display and the CPU were working to a suitable point, the next point of action was implementing the cartridge class. The BIOS sequence of the Gameboy uses a routine to compare the Nintendo logo against a copy of the Nintendo logo that is stored on every game. Interestingly, this is the reason why the logo is a black bar when the cartridge is not loaded properly. (Although this time we can’t blow the dust out of the cartridge!) Therefore, the base functionality of the cartridge class was set up to test that this comparison was working correctly. The test ROM file used to parse the information from the cartridge header was Tetris as the cartridge type was ROM only and required no memory banking to work. A file stream using ios::ate allowed the stream to the end of the file in order to parse the file size. From this, various data such as the title of the game, and the cartridge type was stored within the class. Although not strictly necessary, this information would be useful when implementing the MBC later. A key issue that occurred during the implementation of the cartridge was understanding how the data was represented on the cartridge. Using hex editors showed the values of the cartridge as hex so I mistakenly assumed the data was stored as hex values. The data of the ROM is in binary therefore values being loaded into the bus were 0s and 1s. The fix for this was to change the file stream to binary mode using ios::binary.

Table 3 Gameboy Header Information (Obtained from the GB Dev Wiki)

|  |  |
| --- | --- |
| 0100-0103 | Entry Point, Jumps to the main ROM program |
| 0104-0133 | Nintendo Logo for comparison to the BIOS |
| 0134-0143 | The Title of the Cartridge |
| 0143 | CGB flag, determines if the game enables Gameboy Color functionality |
| 0147 | Cartridge Type, Specifies the MBC and any additional hardware |
| 0148 | ROM Size, Number of ROM banks available |
| 0149 | RAM Size, the amount of RAM available if any |
| 014A | Destination Code, Specifies if the game is region locked to Japan |

### Problems with debugging and reliability

With a couple of tweaks and bug fixes to the display class, the BIOS was working correctly displaying the Nintendo logo scrolling down the screen. This felt like a big accomplishment as the disheartened hours of developing and debugging had finally led to an output being displayed. The next step was to get Tetris up and running, the game uses approximately a third of the CPU instructions and therefore was a good starting point. This was a process of using the assert statements to determine which instruction was required until, disaster struck. The emulator was reading the next instruction as one of the various Opcodes that are not used and therefore invalid. In order to understand the problem that was happening, I used the no$gmb emulator to compare the output against my own emulator. This caused more confusion as at the point where things were going wrong, a JR, Z instruction was being read as HALT instead by no$gmb. Puzzled by this, I did what every good programmer does and went to google for answers. I found a document related to the rom2pdb software that explained that no$gmb replaces the jump with a HALT as a fix as shown in the figure below. After talking to peers on the emulation discord, I looked through the binary file of the game through a hex editor and realised that no$gmb was changing the value without giving an explanation. From this, I deemed that the emulator was an unreliable source for debugging, and thus switching to the BGB emulator instead. This allowed me to discover the root of the problem, another zero flag not being set by the INC register function. With this implemented, the start-up screen for Tetris was finally loading to the Copyright screen.



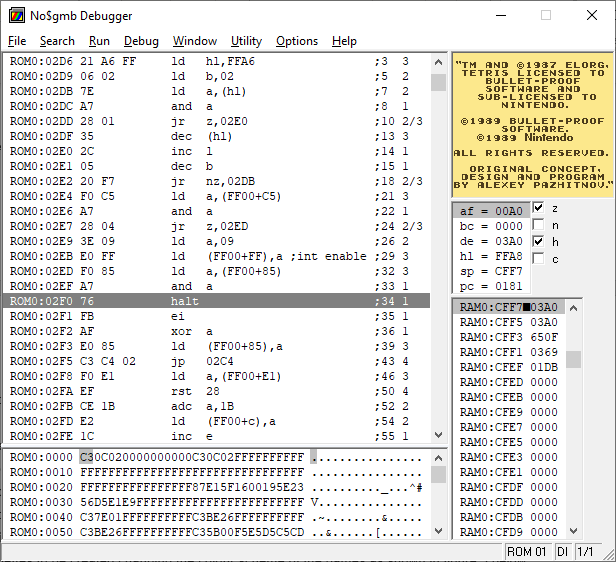


Figure 7 Tetris comparison of BGB and no$gmb

## The final steps to Gameplay

### Understanding Interrupts

The test game Dr Mario was able to get past the menu; however, Tetris was still getting stuck at the start up screen, thus the priority was to get past this bottleneck. At this point, some of the blargg’s test ROMs were functional enough to debug through BG, therefore, focus shifted towards getting the easier tests such as test 7 that tests the load functions of the emulator. Through a laborious task of going through comparing the instructions and registers of the BGB emulator against my own, I was finally able solve the problem with the test ROM. Surprisingly, the problem stemmed from the incorrect implementation of the AF register. The values for F and A were the wrong way around, meaning the 16-bit register AF was being combined backwards. This did not affect the registers individually, and therefore went un-noticed during the BIOS implementation.

const ui8 A\_REGISTER = 0; // Values for A and F were swapped causing issues

const ui8 F\_REGISTER = 1; // combining registers.

const ui8 B\_REGISTER = 3;

const ui8 C\_REGISTER = 2;

Listing 6 Register combination error

This solved the issue and allowed the load test to succeed. However, this caused different problems to appear in testing which are discussed in the testing later. Implementing Tetris took a step backwards, as the emulator was in a loop of calling the reset function causing lines to appear on the screen instead. An issue was discovered when looking at the documentation of Imran Nazar’s emulator explaining the fetch-decode-execute cycle. From this I was able to discover, that I was reading the next Opcode before checking if there were any interrupts. Although, this was not the root cause of the problem. The start screen of Tetris was calling an interrupt however, when comparing the BGB emulator output against my own, it became clear that the jump to the Interrupt was not functioning properly. When going through the Tetris ROM, I noticed that the game behaved abnormally during start up, the ROM attempts to write to memory that is unused by the Gameboy. This is interesting to note, as Tetris was one of the first games to be released with the Gameboy, therefore the software was bound to have numerous bugs and unintentional behaviour that was left in as it didn’t affect the game.

### The Halt Bug

The Halt state of the Gameboy is used for a variety of different games to stop all processing by the CPU. An example of this is pushing start to pause a game. In order for the Gameboy to proceed, an interrupt must be called. This works fine on the Z80; however, the problem arises when the Gameboy interrupts are disabled (DI). The bug causes the HALT state to skip a PC instruction and is a problem found in the physical hardware itself. Therefore, additional data handling was required when calling interrupts in case this scenario happened to prevent the halt bug from happening.

### Taking control of the game

The issue with the incorrect jumps in memory was fixed using debugging through blargg’s test ROMs as discussed in the testing section and therefore was no longer creating a bottleneck in the system, this allowed Tetris to progress to the main menu, however, as appendix 5 shows, the game was not functioning properly as all the blocks were same, making the game incredibly easy. This was fixed by implementing the Timer which increments based on two factors, the CPU cycles ticked, and the current clock frequency which is controlled by the Timer Control register in the I/O registers Additional registers that should be noted are the Divider Register (DIV) and Timer Counter Register (TIMA). The documentation by Hacktix provides a great representation to understand how the overall Timer works. This allowed Tetris to drop multiple different blocks, however, later in development, a key problem was that the registers used to by the timer were not referenced correctly therefore values were not being updated. The focus now shifted to implementing the controls to test that the games were playable. This was a relatively straight forward process; A function was used to determine if the current button being pressed was the top or bottom set of buttons. From this, an SDL\_Events handler was implemented that set and unset the bits of the register that represented that button on the Gameboy. This took a little understanding as to why two separate functions were required for pressing and un-pressing the keys however in the end, the joypad worked, allowing the games to finally be played. Though the games, were playable, there was no limiting on how fast frames were being drawn, thus, the games progressed extremely fast, especially in release mode where compiler optimisations took place. This was a problem as the games were unplayable. This was fixed by using the SDL\_Delay to implement a delay between the actual time taken to render the frame, and the 60-fps frame rate that the Gameboy uses.

Uint32 startTime = SDL\_GetTicks();

Uint32 endTime = 0;

Uint32 delta = 0;

endTime = SDL\_GetTicks();

delta = endTime - startTime;

if (delta < timePerFrame)

{

SDL\_Delay(timePerFrame - delta);

}

Listing 7 Adjusting Frame Timing

This allowed the games to run much smoother and improved drastically the overall playability of the emulator.

### Memory banking

So far, the library of games that the emulator could access was limited by the cartridge class. The lack of memory controller meant that ROM only games were the only type that could be used. Therefore, the cartridge class had to be slightly refactored to accommodate this. This was done using a polymorphic approach. The base MBC class uses a virtual write method that must be implemented for each of the different cartridge types.

CartType cartType;

MBC\* memoryBankController;

//Polymorphism allowed the cartridge to be created based on its type

switch (cartType)

{

case CartType::ROM\_ONLY:

{

memoryBankController = new RomOnly(this, m\_bus);

}

break;

case CartType::ROM\_AND\_MBC1:

{

memoryBankController = new MBC1(this, m\_bus);

}

break;

}

Listing 8 Polymorphic Cartridge

The choice was made to replace the data in the bus with the ROM bank used instead of reading the data from the cartridge as this meant that data could be read from the bus, thus reducing the amount of code that would have to be changed. Additional classes for the ROM only and MBC1 cartridge types were implemented using this base class. The write function uses the 3 most significant bits of the data passed over to parse what action is needed, such as switching ROM banks or toggling RAM to be active. (Once the memory bank classes were implemented, the next step was adding them to the cartridge class. This is done through a switch statement that creates the cartridge based on the cartridge type data kept within the header. (The header information was found on the GB Dev Wiki) The final step was to replace the existing write calls, with the newly implemented MBC class.

if (InMemoryRange(0x0000, 0x7FFF, address))

{

m\_cartridge->GetMBCRule()->Write(address, data);

return;

}

Listing 9 Writing to the Cartridge to access the MBC

The ability to run MBC1 games allowed for additional problems to spotted in the display, these can be found in Appendix 6. Additionally, when testing the MBC with Super Mario Land, I noticed that the game was attempt to write to RAM even though it was not enabled.

## Summary

This chapter has gone through the chronological development of the emulator highlighting the initial problems and key issues that caused changes in the implementation of the emulator due to revisions from the design. The focus of the chapter was on the key features that were implemented looking at the problems that were encountered and how they were resolved. The chapter does not go into detail regarding the implementation of each feature as the volume of information would not fit into the report however the resources used provide more information on the specific instructions

# Test Strategy and Evaluation

## Introduction

This chapter looks at the testing strategies used for debugging in order to evaluate the quality of the code implemented. The effort required to test each individual instruction separately would be laborious, prone to human error, and additionally, may be influenced by other parts of the system that are incorrect. Therefore, public test ROMs were used as test cases to reduce the labour of testing each instruction. As the test ROMs and the games, themselves required the same functionality to be implemented to work, testing was a key part of the debugging process. Additionally, the emulator was tested based on playability as different games implement different instructions therefore what may work for one game may fail for another.

## Functional Testing

### Blargg’s hardware test ROMs

The blargg’s test ROMs were the starting foundation of the functional testing, as they tested the core functionality of the Gameboy, the instructions. This was key as poor implementation of the instructions could influence and cause errors in other parts of the system making testing unfeasible. The tests were ran on real hardware thus their providing a test suite that is has strong validity. Additional ROMs provide access to testing for different areas such as sound and hardware bugs that appear in the Gameboy. The emulator focuses on the playability of the games and due to a lack of time, these features were not implemented and therefore testing focused mainly on the instructions. Initially, the test ROMs were unusable due to poor implementation of various instructions causing the tests to fail to load, thus meaning their usefulness was limited in the beginning.

### Using testing as a tool for debugging

During Implementation, an issue with jumps was causing both games and tests to fail cycling into an infinite loop. This had a knock-on effect of not allowing testing to produce an output of what was failing. Therefore, to tackle this issue, I employed the same strategy as the load test discussed earlier in the implementation. This was carried out by going through the Blargg’s interrupt test (test 2) through the individual sections. At this point, I discovered that the test ROMs had a source file that explained the different tests, making moving through the tests much easier than going instruction by instruction as done previously. From this, I was able to deduce that the values that were loaded in for the different tests were being loaded incorrectly, however I was unable to understand why. In hindsight, the implementation of better debugging facilities would have been extremely valuable at this stage, however, the time constraints from external factors meant that this was unfeasible. Therefore, after hours of debugging going nowhere, I decided to change my approach. Instead of going through each line of code, I used a process of elimination to determine which instruction was the main offender. This was done by looking at the potential jump instructions that were used by the test and then discounting the ones that were functioning on other tests. This allowed me to discover that the JR function was using an 8-bit integer when it required a 16-bit integer therefore causing the jump to be inaccurate. This change in approach meant that it took a total of 15 minutes to solve the issue compared to the countless hours trying to understand the test line by line. The debugging of the tests was interesting as it provided insight as to how the tests functioned, but this change in point of view was a great accomplishment for me. This fix allowed all other tests to function properly and sped up the debugging process exponentially. An interesting problem that was encountered when getting each test to run was once all the individual tests were working correctly, the group test ROM that implements all the tests together was caught in an infinite loop and never finished as seen in the figure below. After talking to the discord community, I found out that the test ROMs implement ROM banking through MBC1 which was causing the error as shown in the figure below.

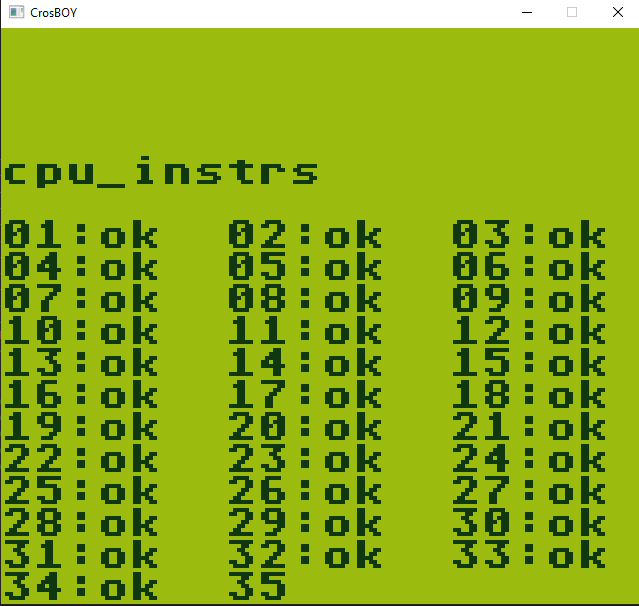
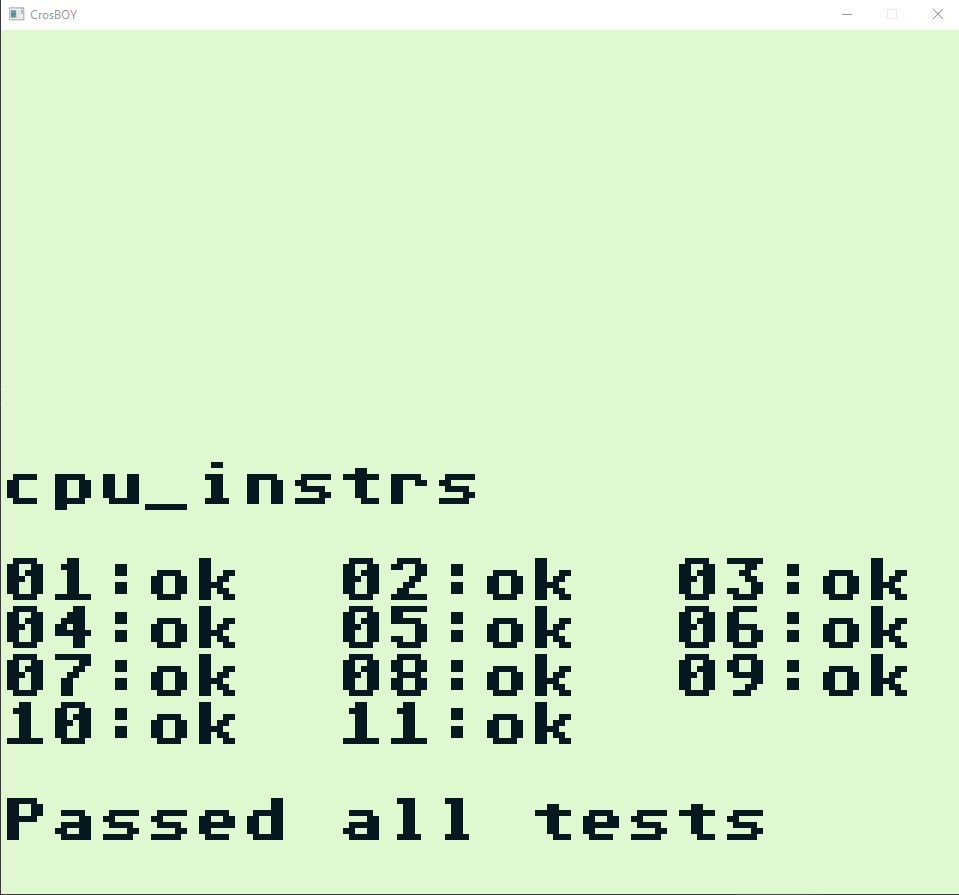
 

Figure 8 Infinite testing glitch from lack of MBC and passed tests

This drastically reduced the time needed to chase a problem caused by another part of the system’s influence. Overall, the blargg’s instruction tests were all able to be passed. While the scope of values that the ROMs test is limited, they provided a great foundation for the emulator and were key in providing a playable implementation that was the aim of the project.

### Mooneye GB tests

The mooneye test ROMs provide a test suite similar to the blargg’s test suite focusing much more closely on the individual aspect of the Gameboy such as timings, interrupts, and memory banking implementation. The Mooneye tests provide a much more thorough testing suite with ROMs for testing individual edge cases such as memory banking with different quantities of RAM. The tests were useful for timings as I was able to identify a problem with variables in the timer being referenced incorrectly. The problem with the tests in comparison to the blargg’s test suite is that it requires much more of the system to be implemented for the tests to work. This means that various tests currently do not function properly with the emulator with some being unable to run at all. The overall focus of the project was to create an emulator that provided the core functionality of the Gameboy therefore the blargg’s test suite was better suited towards testing the functionality of the Gameboy. The scope and timescale of the project did not allow for this type of extensive testing to be done. A goal for future development could be to work towards passing these tests to improve the emulator’s accuracy.

### The Official Gameboy Test Cartridge

The final ROM used to test the functionality of the Emulator was the Official Gameboy test cartridge used to test that hardware was functioning properly at Nintendo repair centres. The ROM features several tests including an automated RAM test, along with screen, button, scrolling, and sound testing facilities. The cartridge itself is interesting as the shape and size is twice as big as a conventional cartridge. The emulator manages to pass all the tests on the test cartridge apart from the sound test as this was not implemented and the scroll test. The tests can be seen in the figure below. The benefit of this type of testing is that it can test the basic functionality of the Gameboy such as input and the display. Additionally, all the tests are featured on one test ROM meaning multiple ROMs do not have to be used. Therefore, this ROM is a good starting point for testing the emulator. However, in comparison to the other ROM tests, the tests are not as in-depth as the mooneye tests that focuses on thorough testing of the several edge cases to ensure that the emulation accurate.

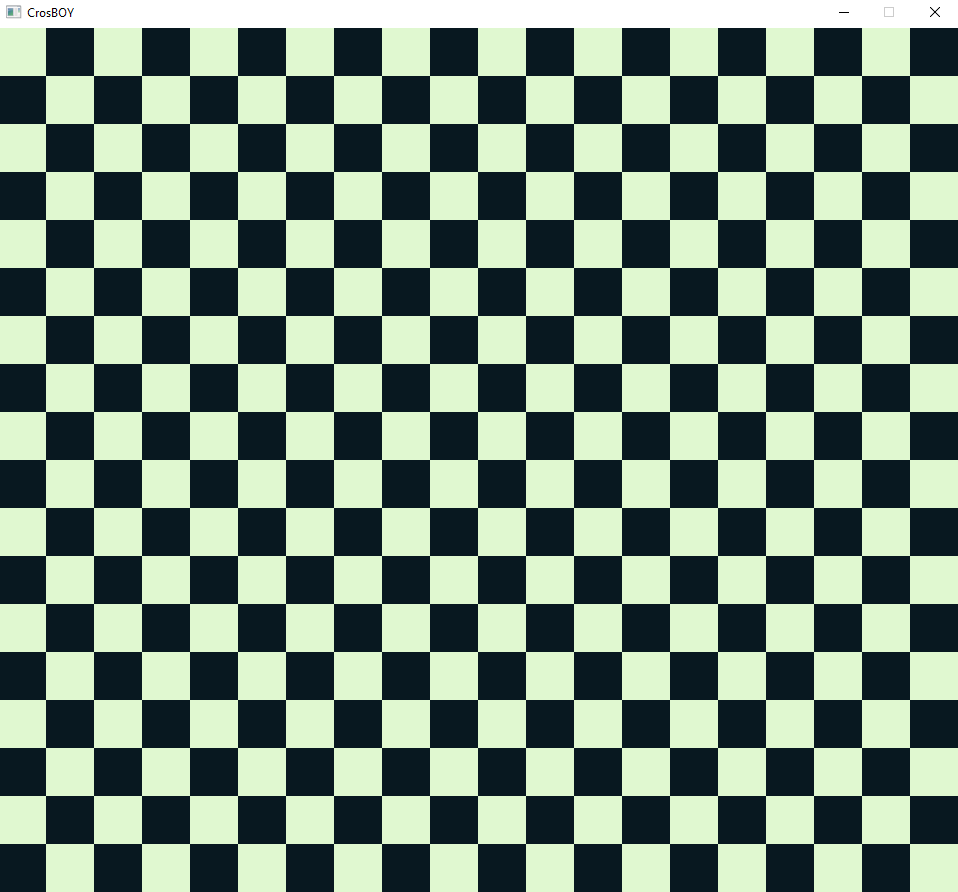
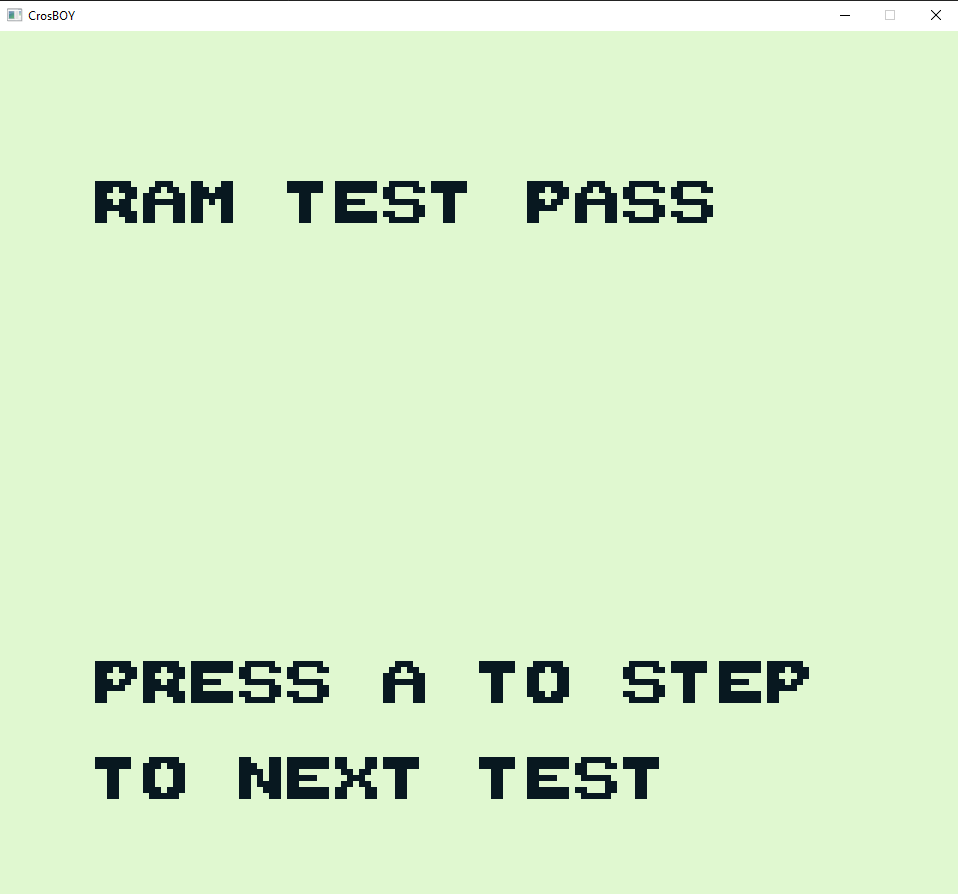




Figure 9 Official Gameboy testing

## Non-Functional Testing

The overall goal of the emulator was to be able to play and run original Gameboy ROMs. The emulator implemented ROM only and MBC1 cartridge types therefore, several games were able to be played to test how well the emulator performed. The games tested used different levels of implementation to show off what the emulator was capable of handling. The table in appendix 6 gives a brief overview of the testing for the various games. In order to fully test each game, the games would have to be tested extensively exploring each level and path of the games. Given the scope and timescale of the project, this would require multiple testers to be completed in a timely manner. Therefore, the testing of the games was cut down to focus on the main playability of the game disregarding minor graphical errors and considering that all features were not going to be implemented. Most of the games that were tested were functional however, this highlighted a problem with sprite drawing as the palettes used were incorrect. This did not impact gameplay for some games but caused issues in The Legend of Zelda, Link’s Awakening as shown in the figure, although the game was playable it was an eye sore and therefore was deemed a major graphical glitch. This problem will require looking at the code for drawing sprites and could be corrected with further development. This therefore shows that the goal of creating a playable emulator has been met, although there is room for further development.

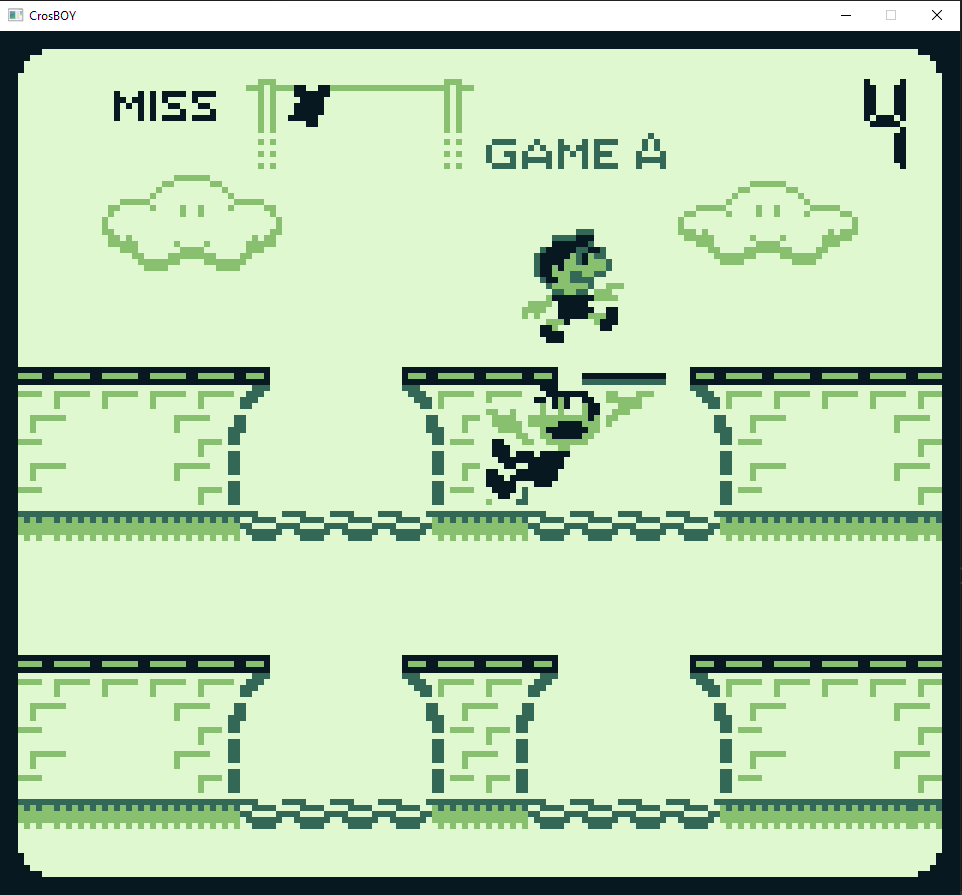


Figure 10 Game Testing (Gameboy Gallery, Kirby’s DreamLand, Super Mario Land)

### Is that a Demo in your pocket?

“Is that a Demo in your pocket” was an interesting custom public ROM I found that was made to show off the capabilities of the Gameboy hardware. I used this ROM to gauge how far from completion the implementation of the display was. The emulator was able to handle some of the graphical techniques used in the ROM as shown below, however completely fell apart during the 3-D rendering that can be seen in the Figure 7 and eventually renders the emulator unresponsive. Overall, the fact that the emulator was able to run some of the graphical artefacts in the ROM was surprising and could be explored as a future project along with the other tests to get the ROM up to full functionality.

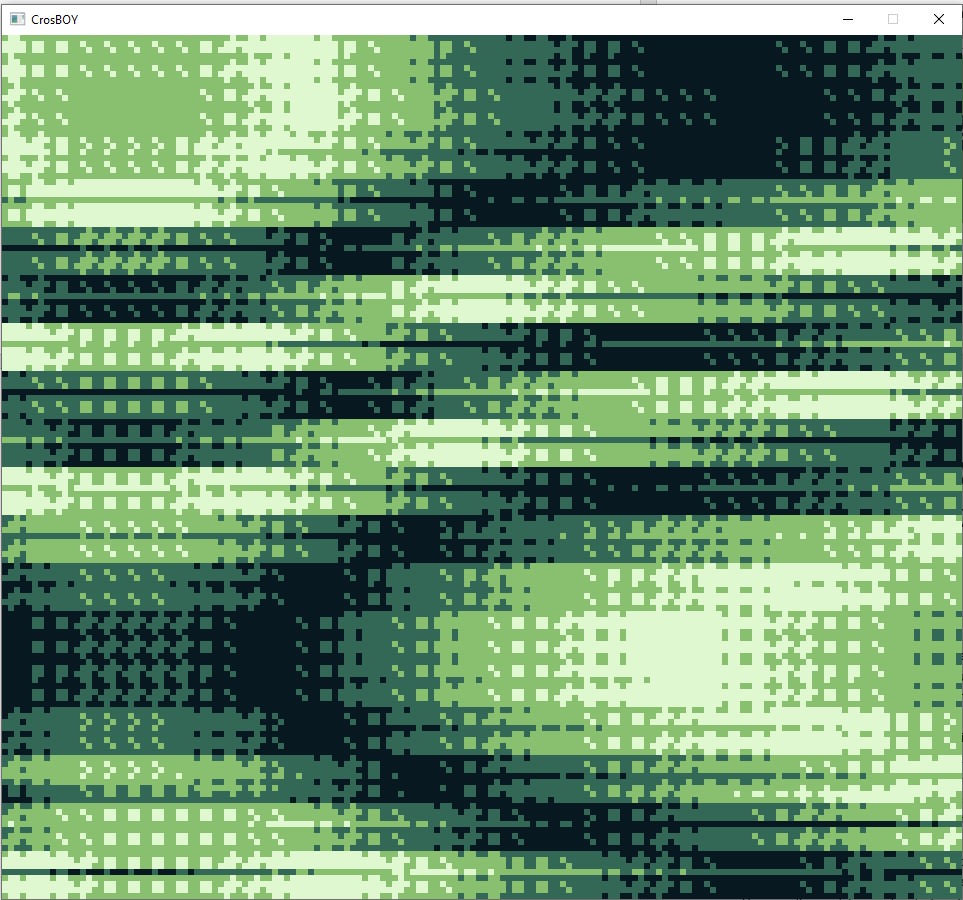
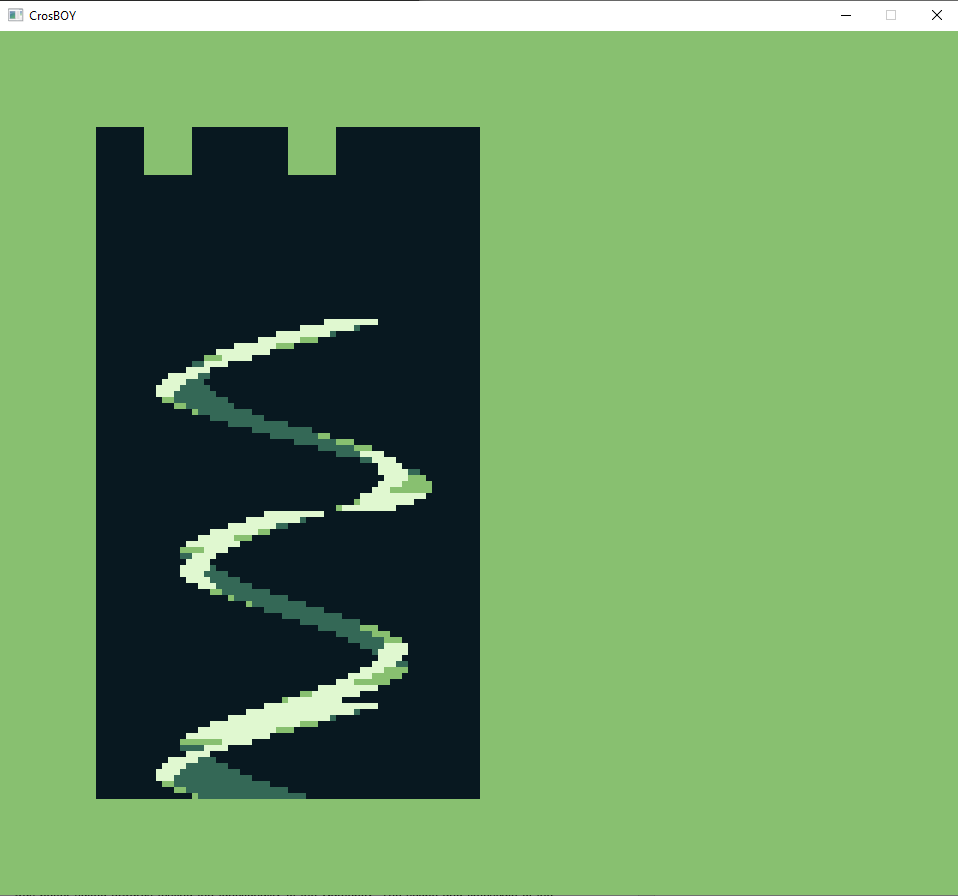
  

Figure 11 Gameboy Pocket Demo and graphical demonstration

## User Testing

Due to circumstances out of my control, the scope of users that were able to test the emulator were limited. The focus of the testing was the usability and playability of the Emulator. The usability of the emulator was limited for my family as setting up the emulator to play was complicated for them. This shows that the development of a UI will greatly improve the overall experience. However, the overall experience of the emulator was positive as my sister was able to play Gameboy games from our childhood. Thus, I believe that the emulator can be deemed a success as the main focus was to re-create the experience of the Gameboy.

## Summary

This chapter has looked at the various types of testing used to test the overall functionality of the emulator as a whole and provides insight into how well the emulator was implemented. The tests conducted used not only original Gameboy games, but also public test ROMs. The effectiveness and validity of the ROMs was evaluated including their usefulness during development. Overall, the testing shows that the emulator met the goal of performing well enough, accomplishing playability in most games tested. However, the testing also shows that there is still much more development that could be done to further improve the emulator.

# Evaluation, Conclusions and Future Work

## Project Objectives

Summarise what you have achieved.

The development of the project has produced a functional Gameboy emulator capable of running multiple different game cartridge types allowing the user access to games they otherwise may not be able to use. As the software was developed using SDL, this allows the program to be cross platform n different operating systems such as Linux and windows. Additionally, as the implementation of the Gameboy does not rely on the SDL library, the framebuffer used to push pixels to the screen can be adapted to different software if needed.

Focusing on the scope of the project, this was laid out in section 3.3 using a MoSCoW analysis to identify the key requirements needed for the project. The Must and Should sections of implementation have been met, additionally, parts of the could requirements have been met and exceeded, as the emulator passes all the blargg’s test ROMs. In addition, the other aspects of the could section have the potential to be implemented as implementing MBC5 would only require a class built upon the existing MBC architecture. The implementation of the Gameboy color functionality proved to be much more in-depth than initial research suggested, however, the functionality to adjust the clock speeds for the Gameboy color would be easy to implement and still has potential to be implemented in the future.

However, the emulator is not perfect, and there are still issues that limit the playability of some games, such as the incorrect sprite data changing the appearance of the game altogether in some cases. This would require additional debugging time that was not possible to allocate a time box to due to the strict timing requirements. This means that the emulator does not allow for more complicated games to be played, however, the games that can be played are fully functional except for minor graphical glitches that do not affect gameplay. Given the tight schedule allocated to the project and complexity of the implementation needed, I believe I have been able to tackle the project in a logical and professional manner overall allowing development the emulator to be a success.

## Self-Evaluation

I believe that I was too naïve going into the development of this project and did not fully comprehend the level of understanding of the system needed to commit to development. This meant that a lot of potential time for development was wasted and ultimately the project suffered because of this. My approach to try and implement everything as fast as possible without care or debugging was reckless and has allowed myself to reflect on the way I approach tasks different. I believe that from this, a strong point of the development was the estimation of the time needed to implement the various features needed. Due to the time that the project was started, time boxing was a crucial element that allowed many of the features to be develop and helped keep myself focused. Overestimating the time to implement some of the features gave a buffer in-case of problems with debugging which inevitably happened. This also provided breathing room to allocate more times to other areas of the emulator if a section was implemented quickly, such as the Joypad. The experience of the project has allowed me to develop the ability to draw understanding from technical documentation instead of taking it at face value. It has also allowed me to question documentation based on its source to determine how valid the information really is, comparing it against various other sources. Overall, understanding the technical documentation was a whole new world for me and applying myself to the new techniques and ideas used was difficult in the beginning. This caused problems with implementation that ultimately extended the time needed for debugging. However, the main cause of the problems I had to debug was my own carelessness during implementation making silly mistakes such as not referencing something correctly or using wrong datatypes. This was due to the reckless implementation at the beginning and has allowed me to understand that taking time in the beginning to create a strong foundation for debugging will save time in the end. Although this is still a problem for me, the project has allowed me to become much more aware of the impact of this and therefore keep myself in check to make sure that I am not rushing myself.

I am extremely proud of the overall implementation of the emulator and have been able to exceed even my own expectations as this has been the biggest project I have worked on throughout my 3 years of education, especially now understanding the number of things that could’ve gone wrong during the implementation. The difficulties I have experienced this year made completing this project seem impossible due to time constraints and lack of confidence in my abilities programming. The challenge of developing the emulator, although extremely stressful and even daunting in the beginning, was overall a rewarding process in the end. The process of debugging the emulator was a key problem and massive undertaking that has allowed me to develop my skills in this area. Additionally, developing the ability to understand technical documentation and at times take a step back to look at the bigger picture of how a feature would affect the entire system was a key skill that was needed throughout development. In comparison to myself at the start of the project, I believe that the development of the skills needed to debug and implement the emulator through analysis of documentation and tracking down bugs in a logical manner are all positive aspects that can be applied to other projects that I work on in the future.

## Applicability of Findings to the Commercial World

The overall applicability of the emulator in a real-life scenario is limited due to the legal issues discussed during the background research in section 2, however, aspects of the development of the emulator can be taken and applied to future projects. At the core, the emulator still uses assembly language that is still relevant to programming in modern day compilers. The understanding of the assembly language gained allows for insight into software behaviour and potentially alternate implementations. In addition, the understanding gained from developing a system from scratch can be applied to alternate forms of emulation such as machine virtualisation or even java used in android phones. This can be used to plan out and model the architecture of future projects.

## Conclusions

The development of the emulator has allowed me to achieve the goal of developing an emulator with the functionality to play Gameboy games from my childhood. This is a feat due to the overall complexity and workload that the system required. Various factors were required to achieve this outcome, the scope of the project was reasonably thought out, and most of the project objectives were completed in a timely manner by using timeboxing and project planning to attack the development logically. In addition, the project required development of various skills for implementation and debugging that can be applied to future pieces of work and has allowed myself to reflect on my approach to programming.

## Future Work

The timescale of the project meant that many features had to be sacrificed for the final product to have an achievable level of functionality. Therefore, this would be the first point of future development. The main problem with the current implementation is the lack of sound that is a key feature of other emulators and drastically takes away from the nostalgia of playing the Gameboy games. This would require the use of external libraries to emulate the way that the Gameboy produced sound. Additionally, because of the polymorphic approach to developing the memory banking system, the implementation of additional cartridge MBC classes such as MBC3 and the implementation of a battery would greatly improve the number of games that the emulator could access including rarer games that are potentially more difficult to find. This would also require additional testing and potentially developing the emulator to pass the various mooneye tests. Additionally, the current graphical glitches of the emulator could be weeded out to allow the proper demonstration of the “Is that a Gameboy in your pocket” ROM. The emulator could potentially be developed to access physical hardware features of the Gameboy such as communication however, this would not have a great impact on playability and would only be needed for niche games. Implementing the features needed for Gameboy Color games to be played such as modifying the display buffer and changing the CPU timings would allow the library of games that the emulator can play to increase further.

The second point of future development that would coincide with developing the emulator features would be to improve the debugging facilities of the emulator. This could be done using additional libraries such as IMGUI to allow the visualisation of different components such as the tile data loaded allowing for easier debugging and can be used as an educational tool by others to debug their own emulators. Additionally, the emulator could be further developed as a tool for creating future public use ROMs. An example of this is ZGB which is an engine used to develop Gameboy ROMs.

Finally, the implementation of the emulator could be modified to be used with front-end GUIs such as emulation station. This project is open source and can be used to create a versatile GUI that would be a great improvement over the current barebones GUI implementation and thus would improve the overall user experience of the emulator.

References

A Look at The Game Boy Bootstrap: Let the Fun Begin! (2013). serginho89 Retrieved from <https://realboyemulator.wordpress.com/2013/01/03/a-look-at-the-game-boy-bootstrap-let-the-fun-begin/>

Alec Lofquist, Nils Bonenberger (2020). Emulation Station Retrieved from <https://emulationstation.org/>

Anthrox, GABY, Marat Fayzullin, Pascal Felber, Paul Robson, Martin Korth, et al. (1989). *Game boy TM CPU manual*

Agile Business Consortium. (2019). MoSCoW prioritisation.*The DSDM Agile Project Framework (2014 Onwards),*

Al-Samarraie, H., & Ahmad, Y. (2016). Use of design patterns according to hand dominance in a mobile user interface.*Journal of Educational Computing Research, 54*(6), 769-792.

Andrew Kelley. (2013). Statically recompiling NES games into native executables with LLVM and go. Retrieved from <https://andrewkelley.me/post/jamulator.html>

Beware (2020). BGB Emulator Retrieved from <https://bgb.bircd.org/>

Blargg. (2014). Retrio gb-test-roms. Retrieved from <https://github.com/retrio/gb-test-roms>

Campos Colmenarejo, J. (2017). Developing a game for nintendo game boy.*Developing a Game for Nintendo Game Boy,*

Conley, J., Andros, E., Chinai, P., & Lipkowitz, E. (2003). Use of a game over: Emulation and the video game industry, a white paper.*Nw.J.Tech.& Intell.Prop., 2*, 261.

Console Database. (n.d). Nintendo gameboy color: Console info. Retrieved from <https://www.consoledatabase.com/consoleinfo/nintendogameboycolor/>

Dr Helius (2020). Gearboy Emulator Retrieved from <https://github.com/drhelius/Gearboy>

Edwards, S. A., & Patel, H. (2014). *MEMOCODE 2014 software design contest: Space invaders emulator* IEEE.

Ernberg, J. (2011). *The internal workings of video game consoles: The GameBoy* Mälardalens högskola, Akademin för innovation, design och teknik. Retrieved from <https://www.openaire.eu/search/publication?articleId=od_______258::c0cf77b8f413f5eb373647ee809b6901>

Fowler, M., & Highsmith, J. (2001a). The agile manifesto.*Software Development, 9*(8), 28-35.

gabrieloc. (2019). Giovanni. Retrieved from <https://github.com/gabrieloc/GIOVANNI>

GB Dev Wiki. (2008). Gameboy Documentation. Retrieved from <https://gbdev.gg8.se/wiki/articles/Pan_Docs>

Guttenbrunner, M., Becker, C., & Rauber, A. (2010). Keeping the game alive: Evaluating strategies for the preservation of console video games.*International Journal of Digital Curation, 5*(1), 64-90.

Guttenbrunner, M., & Rauber, A. (2012). A measurement framework for evaluating emulators for digital preservation.*ACM Transactions on Information Systems, 30*(2), 1-28.

Hacktarux. (2011). Mugen64plus. Retrieved from <https://pandorawiki.org/Mupen64plus_dynamic_recompiler>

Imran Nazar (2010). GameBoy Emulation in JavaScript Retrieved from <http://imrannazar.com/GameBoy-Emulation-in-JavaScript:-The-CPU>

J. E. Smith, & Ravi Nair. (2005). *The architecture of virtual machines*

JackTech. (2016). The game boy, a hardware autopsy - part 2: Memory mapping. Retrieved from <https://www.youtube.com/watch?v=ecTQVa42sJc>

Jeffrey van, D. H., Lohman, B., & Verdegem, R. (2007). Emulation for digital preservation in practice: The results.*International Journal of Digital Curation, 2*(2), 123-132.

Joonas Javanainen (2016). Game Boy test ROM do's and don'ts Retrieved from <https://gekkio.fi/blog/2016/game-boy-test-rom-dos-and-donts/>

Kemppainen, S. (2019). Remaking a video game: Case: Hertan ruskaretki.

Koninklijke Bibliotheek. (2007). What is emulation? Retrieved from <https://www.kb.nl/en/organisation/research-expertise/research-on-digitisation-and-digital-preservation/emulation/what-is-emulation>

Lindholm, T., Yellin, F., Bracha, G., & Buckley, A. (2013). *The java virtual machine specification, java SE 7 edition: Java virt mach spec Java\_3* Addison-Wesley.

Marat Fayzullin. (2019). Virtual gameboy. Retrieved from <http://fms.komkon.org/VGB/>

Marat Fayzullin. (n.d). How to write a computer emulator. Retrieved from <https://fms.komkon.org/EMUL8/HOWTO.html#LABA>

Michael Steil. (2016). The ultimate game boy talk.

Monnens, D., Armstrong, A., Ruggill, J., McAllister, K. S., Vowell, Z., & Donahue, R. (2009). *Before it's too late: A digital game preservation white paper* Lulu.com.

Mooneye (2019). Mooneye Test ROMs Retrieved from <https://github.com/Gekkio/mooneye-gb>

Mosqueda, A. (2018). *The grey area behind emulators*. Carlsbad: Uloop, Inc.

Newman, J., Jewitt, R., & Yar, M. (2013). Illegal deposit: Game preservation and/as software piracy.*Convergence, 19*(1), 45-61.

Nilsson, E. G. (2009). *Design patterns for user interface for mobile applications*

Nintendo. (2016). Consolidated sales transition by region.

Nintendo files U.S. patent for game boy 'software emulator'.(2014), *Dow Jones Institutional News,* Retrieved from <https://search.proquest.com/docview/2074246988?accountid=17233>

Orland, K. (2016). How the demonization of emulation devalues gaming's heritage.*ArsTechnica.Com*

Park, Y. S., & Han, S. H. (2010). One-handed thumb interaction of mobile devices from the input accuracy perspective.*International Journal of Industrial Ergonomics, 40*(6), 746-756.

Petar Veličković. (2016). *How to make a (GameBoy) emulator?*

Pinchbeck, D., Anderson, D., Delve, J., Alemu, G., Ciuffreda, A., & Lange, A. (2009). Emulation as a strategy for the preservation of games: The KEEP project. Paper presented at the *DiGRA 2009: Breaking New Ground: Innovation in Games, Play, Practice and Theory,*

Rafael Kitover. (2019). Visual boy advance. Retrieved from <https://github.com/visualboyadvance-m/visualboyadvance-m>

Richeson, J. M. (2017). Anatomy of a hardware emulator.

Ross, J., & Gao, J. (2016). *Overcoming the language barrier in mobile user interface design: A case study on a mobile health app*. Ithaca, United States Ithaca, Ithaca: Cornell University Library, arXiv.org. Retrieved from <https://search.proquest.com/docview/2079951085?accountid=17233>

Rothenberg, J. (1998). "Criteria for an ideal solution." avoiding technological quicksand: Finding a viable technical foundation for digital preservation".

Rom2pdb (2000). Retrieved from <https://fladnag.net/downloads/telephone/palm/APPS/Liberty1.25/rom2pdb.c>

Saltzman, M. (1999, The emulation craze; depending on which side of the fence you're on, playing old arcade games on a modern PC is either a nightmare or nirvana for gaming.: [1 edition].*Toronto Star,*pp. 1. Retrieved from <https://search.proquest.com/docview/437916390?accountid=17233>

Sinamas. (2019). The Gambatte Emulator. Retrieved from <https://github.com/sinamas/gambatte>

Snorpung (2015). Is that a Demo in your pocket? Retrieved from <http://www.pouet.net/prod.php?which=65997>

Díaz A. N. (2015). The Cycle Accurate Game Boy Docs Retrieved from <https://github.com/AntonioND/giibiiadvance/blob/master/docs/TCAGBD.pdf>

The Gameboy Group. (n.d). *The gameboy project*

The Z80 CPU User Manual (1975).

Venners, B. (1998). *The java virtual machine* McGraw-Hill, New York.

Witawas Srisa-an. (n.d). Emulation: Interpretation. Retrieved from <http://cse.unl.edu/~witty/class/embedded/material/note/emulation.pdf>

Xiao, Z., Song, W., & Chen, Q. (2012). Dynamic resource allocation using virtual machines for cloud computing environment.*IEEE Transactions on Parallel and Distributed Systems, 24*(6), 1107-1117.

Zenogais. (n.d). Introduction to dynamic recompilation. Retrieved from <http://www.multigesture.net/wp-content/uploads/mirror/zenogais/Dynamic%20Recompiler.html>

Zal0 (2020). ZGB engine Retrieved from <https://github.com/Zal0/ZGB>

Appendix 1 – Project Proposal

Computing Degree Project Proposal

|  |  |
| --- | --- |
| **Student Name**  Thomas Crosby |  |
| **Course** Software Engineering |  |
| **Project Title** |  |
| Emulation of a handheld generation |  |

## Project Context

This project intends to develop software that emulates the Gameboy architecture from 1989 to be used on a more up to date piece of hardware. The project explores the low-level code used in the hardware. The Gameboy uses a processor similar to the Intel 8080 with features from the Z80. In addition, the Gameboy also uses a 160x144 pixel display output and accesses data though banks of memory stored on the game cartridge. The benefits of exploring these techniques is that it will allow a greater understanding of the core concepts and methods used at the time to run software with the limited resources that were available. Thus, gaining insight into the differences of low-level language compared to the object-oriented languages of today. In addition, this project allows preservation of older game titles allowing them to be accessed without requiring the original hardware which can be both rare and expensive.

Due to the lack of official documentation available on Gameboy Architecture, the topic of Gameboy emulation has been an ongoing process throughout the years, with documentation of the architecture being created to develop this kind of software.

The issues that may arise from this is that development will rely on unofficial sources, which although thorough, could potentially be inaccurate or invalid. Therefore, thoroughly testing of software will be required. however, A perfect recreation of the Gameboy would be impossible unless using the original hardware. This means that the focus of the emulator should be to recreate a playable experience of the original software

Finally, the legality of using the original Gameboy ROMs for emulation may potentially be a conflict with the project. This could potentially cause issues for the software development as it will rely on ROM files for testing and to function.

## Specific Objectives

The Emulator will be able to:

* Load Gameboy ROMs into the software successfully.
* Correctly emulate the CPU of the Gameboy
* Output the display of the Gameboy to a Screen
* Output sounds from the Gameboy ROM
* Allow input from the user to play/test the Game ROM

## References

Campos Colmenarejo, J. (2017). *Developing a game for Nintendo Game Boy* (Bachelor's thesis, Universitat Politècnica de Catalunya).

Ernberg, J. (2011). The internal workings of video game consoles: The GameBoy.

Guttenbrunner, M., Becker, C., & Rauber, A. (2010). Keeping the game alive: Evaluating strategies for the preservation of console video games. *International Journal of Digital Curation*, *5*(1), 64-90.

Newman, J. (2013). Illegal deposit: Game preservation and/as software

piracy. *Convergence*, *19*(1), 45-61.

Richeson, J. M. (2017). *Anatomy of a hardware emulator* (Doctoral dissertation).

Game Boy CPU Manual (found at http://marc.rawer.de/Gameboy/Docs/GBCPUman.pdf)

## Potential Ethical or Legal Issues

There are several ethical issues that may arise from this project. First and foremost, distribution of the software could potentially enable other users to break the law through illegally pirating games. Therefore, users will be explicitly informed of breaking the copyright law when using the emulator.

The project aims to emulate the hardware of the Gameboy and therefore, fundamentally a Gameboy ROM is required. The use of ROMs is a bit of a legal grey area, as the ROM is copyrighted material it is illegal to download a copy of the game. However, from an ethical standpoint, if the user owns a copy of the game, a digital copy of the game could also be created under fair use. This digital copy would then be used for the emulator.

Furthermore, the Gameboy hardware uses a BIOS load screen featuring the Nintendo logo. A copy of this sequence also exists on every Gameboy ROM. The BIOS load screen is checked against the ROM to make sure that the game is valid by checking the values are the same. Therefore, care must be taken while creating the emulator so that it does not infringe the Nintendo logo copyright.

## Resources

The resources that will be required for this project is an IDE (Integrated development environment) such as Visual Studio or Eclipse to develop the software. To develop this project, the main languages I would use are C++ or Java as I have experience using them. To implement the output of the emulator onto a screen, I would use the SDL library for C++ which is a low-level library that can be ported. This would allow the project to be cross platform.

Additionally, digital ROMs will be required for testing the software as it is being developed.

## Potential Commercial Considerations - Estimated costs and benefits

Overall, in a commercial environment, it would not be feasible to create this software as a product in order to produce a profit as this would promote piracy. However, the development of this project could potentially work through donations to the project.

Overall, this would require, focus from a very small team (2 to 3 people) of developers in order to maintain and add additional functionality to the emulator. I estimate that £1000 a month would be needed for the team to develop the software and add additional features.

## Proposed Approach

The project will require a level of understanding of the inner workings of the Gameboy Architecture. Therefore, the start of the project will require 1-2 weeks, of research into the components of the hardware and how they work.

From this research, the next stage of the project would be to successfully load digital ROMs and begin setting up the CPU such as registers and commands. This will take 2-3 weeks as it will require thorough testing and trial and error to implement. Continuing from this, I will spend a 1-2 weeks focus on the output of the emulator using either the SDL library or JFrames depending on the language used. Additionally, implementing input and sound will be required for the base emulator which will require 2-3 weeks

Appendix 2 – Technical Plan

**Computing Project Technical Plan**

**Name: Thomas Crosby**  **Mode**: Full Time

**Course: Software Engineering** **Supervisor:** N/A

**Title**

Emulation of a Nintendo Generation

**Summary**

The objectives of this project will be to create software that will emulate the experience of the original Gameboy hardware. The challenges that will be present during the course of this project will be maintaining focus of the core elements of the emulator whilst also implementing additional features. Furthermore, the challenge of setting up the emulator will mean that thorough research will be need to get to a working state. In addition, the scale of the code implemented in this project will mean that thorough testing of the system will be required.

**Deliverables**

The artefact submitted will be a executable capable of emulating system using original Gameboy rom files. This software will be usable on the windows platform.

**Constraints**

The reliance of the CPU for the emulation means that development of additional and core features (such as graphical output) will be constrained by the early development. Furthermore, the development and testing of the emulator will be constrained by the use of Gameboy rom files, as the emulator will be useless without them.

**Key Problems**

The key problems that will arise during the course of this project will be the scale of development and the lack of documentation. The CPU, a single component of the emulator, will require 500 operations to handle the data from the Gameboy ROM files alone. Therefore, the scale of the project will be dictated by the time left once the core components have been completed. Furthermore, testing will a key component of the development stage as there is a large volume of operations required by the Gameboy to function properly. This means that the margin of error during development will be high and require debugging through trial and error.

**System and Work Outline**

The components of the Gameboy can be broken down into areas such as GPU, CPU, Audio, etc. However, certain components such as the GPU will be reliant on core components like the CPU. The initial work required for this project will be research into the features of these core elements of the Gameboy hardware to develop a base understanding for development. The Gameboy’s functionality will be limited without these core elements being put into place. Therefore, developing an understanding of the CPU and the techniques used in the low-level language will be critical to development and will need to be prioritised.

The outline of the system can be seen below through a MoSCoW analysis of the components of the Gameboy and their relevance to the whole system.

|  |
| --- |
| **The emulator must be able to:**   * access the data from the Gameboy ROM * load the data into the CPU architecture of the software * Output the display of the game through a window * allow the user to interact with the game |
| **The emulator should be able to:**   * run games that only require rom files (such as Tetris) * allow the progress of the game to be saved (through access to additional memory banks or saving the state of the game) * allow switching between memory banks correctly to allow a greater variety of games to be played |
| **The emulator could be able to:**   * Emulate the original sounds of the game boy * allow emulation of game boy color games (implement additional colours to be outputted) * develop a gameboy ROM to allow further software to be ran on the Gameboy beyond the limits of just games. (Campos Colmenarejo, J. (2017)) |
| **The emulator would be able to:**   * Accurately recreate the experience of each game including areas where behaviour that isn’t intentional is replicated * Be loaded onto a raspberry Pi device allowing portability |

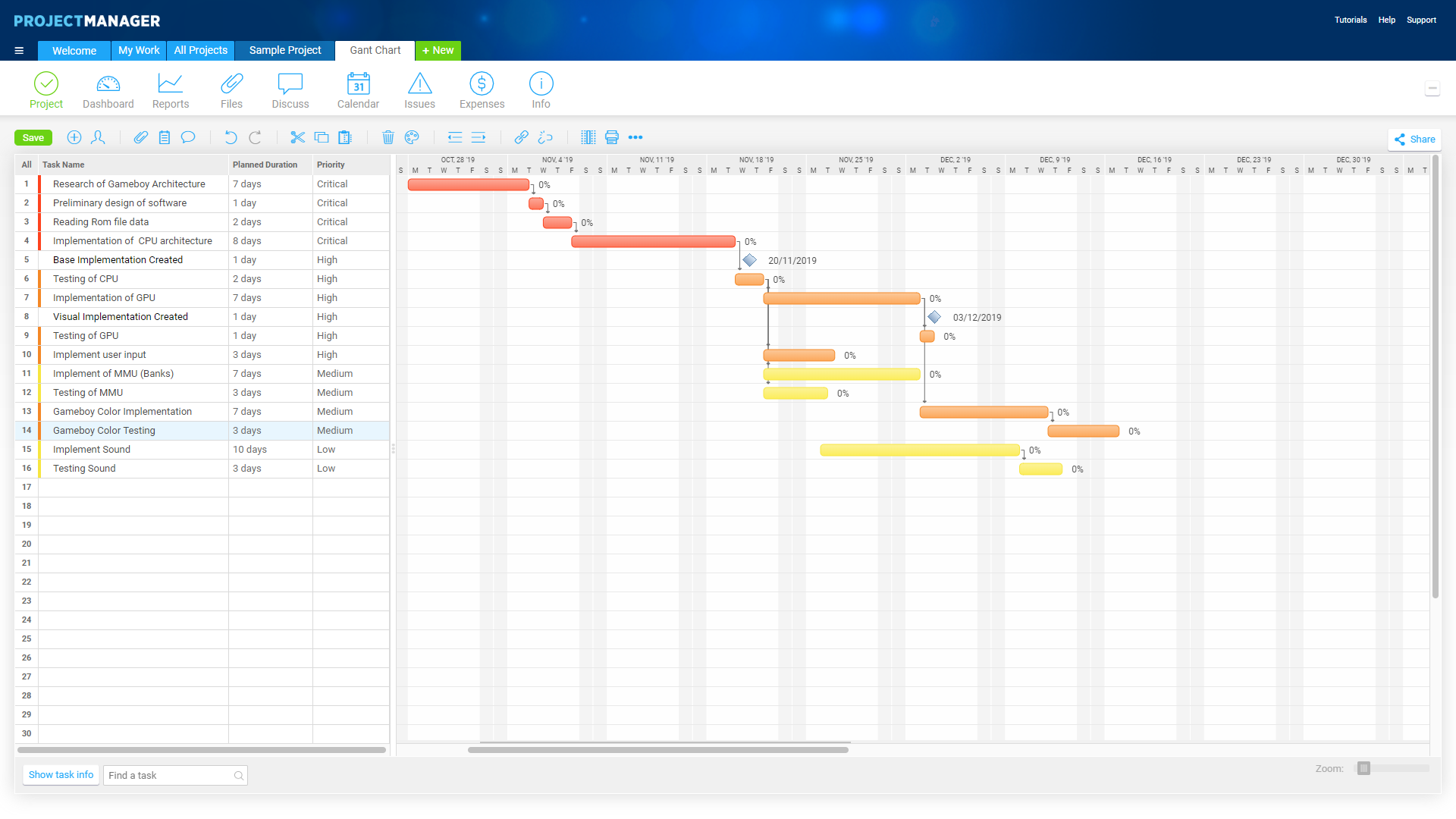
The emulator will be created in C++ and will require graphical output. In order to provide this, the SDL 2.0 library will be used due to it’s popularity for emulation and ability to provide low level access which is ideal for this project. This library will require some research to understand how the game emulation can be outputted to a screen and may also be useful for other features such a controller support.

The project will have known issues that will present themselves during the implementation stages. The key problems will be that the hardware will not be a perfectly accurate recreation of the original hardware. It would be unreasonable to implement features such as the Gameboy printer as this would require the original hardware to operate. Furthermore, as there is no official documentation and testing of the emulation for each game in the Gameboy’s library is unfeasible. Therefore, only major bugs such as games crashing will be focused on over smaller issues such as graphical oddities that do not impact the overall gameplay of the emulator.

The testing aspect for this project will mainly focus on the operations performed by the emulator in the form of Opcodes. This will be used to monitor and maintain that essential functionality ( memory addresses, flags, registers, etc) are being used correctly. In addition to this, testing will be required for the usability of the software and accuracy of the emulation.

**Project Activities**

The activities that will be carried out throughout the project have been represented as a Gant chart below. The chart shows the core elements of the project that are required to be completed and shows a timescale of the project. It should be noted below that the project is bottlenecked by the initial research and development of the core elements of the emulator. However once the foundation of the project has been laid down, it will be possible to work on numerous components at once.



**Risk Analysis**

|  |  |  |  |
| --- | --- | --- | --- |
| **Risk (1-low 10-high)** | **Severity (1-10)** | **Likelihood (1-10)** | **Action** |
| Loss of Data | 10 | 2 | In order to safeguard the project from this, github will be used to store and control the versions so that project changes can be reverted if needed. |
| Unable to use ROMs due to copyright | 4 | 5 | There are public use ROMs available that allow testing to be performed and will still allow the emulator to be used. Although emulating original software would show its full capability. |
| Difficulty of the implementation | 5 | 5 | As the project contains unexplored areas such as the CPU and the SDL library. Although research can be done through documentation of the library, roadblocks may occur which will require assistance from other people with prior experience in emulation. |
| Time constraints of the project/Number of features | 3 | 9 | The project will not be able to implement all of the features of the Gameboy. This means that some features may have to be unimplemented to focus on others. |
| Illness | 5 | 7 | Failling ill during the project may hinder the amount of time that can be dedicate to it, therefore, the time needed to complete task will have a small buffer to account for any time lost. |

**Options**

The project will focus on using a waterfall methodology as the requirements of the project are near enough understood and each phase will have clear deliverables to show. This will also give a clear view of the development process showing how much progress has been made. However, the project will also incorporate features of the agile lifecycle in order to test the software and to prioritise the features that will be implemented in the final product through assessment and re-assessment.

The development tools that will be used for this project will be an Integrated development environment to develop the project. The project will be written in C++ using visual studio due to prior experience with the language. The SDL 2.0 library will be used due to popularity for emulation and the features that it provides through low level access. An alternative to this would be to use Java which could potentially have used JFrames to output the graphics to the screen.

The use of the SDL library allows the project cross platform; therefore, the software could be used on a raspberry Pi device to mimic the original handheld nature of the console. This would be interesting as the Pi could be used in conjunction with the original case of the Gameboy. This could be a further iteration of the project. Despite this, the main priority target platform for the emulator will be windows.

**Potential Ethical or Legal Issues**

There are several ethical issues that may arise from this project. First and foremost, distribution of the software could potentially enable other users to break the law through illegally pirating games. Therefore, users will be explicitly informed of breaking the copyright law when using the emulator.

The project aims to emulate the hardware of the Gameboy and therefore, fundamentally a Gameboy ROM is required. The use of ROMs is a bit of a legal grey area, as the ROM is copyrighted material it is illegal to download a copy of the game. However, from an ethical standpoint, if the user owns a copy of the game, a digital copy of the game could also be created under fair use. This digital copy would then be used for the emulator.

Furthermore, the Gameboy hardware uses a BIOS load screen featuring the Nintendo logo. A copy of this sequence also exists on every Gameboy ROM. The BIOS load screen is checked against the ROM to make sure that the game is valid by checking the values are the same. Therefore, care must be taken while creating the emulator so that it does not infringe the Nintendo logo copyright.

**Commercial Analysis**

Overall, in a commercial environment, it would not be feasible to create this software as a product in order to produce a profit. This would promote the piracy of the games and could potentially cause a Nintendo to take legal action. However, the development of this project could potentially work through donations to the project and would be focused on preserving the software so that it does not become lost with time.

This project would require focus from a very small team (2 to 3 people) of developers in order to maintain and add additional functionality to the emulator. I estimate that initially the project would require £1000 a month would be needed for the team to dedicate the time to develop the software features and add additional features such as implementing the original behaviour of the games.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Factor name** | **Description** | **Is this a cost or a benefit** | **Estimated Amount** | **Estimate of when paid** |
| **Pay for hours worked** | **Initially, I would estimate the project would require 10-20 hours of work a week at a rate of £12 per hour.** | **Cost** | **£720** | **Based on donations (potentially monthly through services like patreon).** |
| **Website to host/advertise** | **The emulator would need to be advertised to create a userbase with interest.** | **Cost** | **£30** | **On a monthly basis.** |
| **Donation from users** | **This would be from users who are willing to invest in the project** | **Benefit** | **£200-300** | **With additional features added to the emulator** |

**Employability Contribution**

The project contains many aspects and techniques that will be useful for myself as a software engineer. To implement the system, a level of knowledge must be gained through research and experimentation showing the ability to understand documentation and the architecture of a low level language. Furthermore, the project will show the ability to develop software under time constraints and the ability make decisions based on the features needed for the system and client. Finally, the project will show the importance of using correct testing and debugging techniques as the accuracy of the operations used will be vital to the overall performance of the emulator.

**References**

Campos Colmenarejo, J. (2017). *Developing a game for Nintendo Game Boy* (Bachelor's thesis, Universitat Politècnica de Catalunya).

Ernberg, J. (2011). The internal workings of video game consoles: The GameBoy.

Guttenbrunner, M., Becker, C., & Rauber, A. (2010). Keeping the game alive: Evaluating strategies for the preservation of console video games. *International Journal of Digital Curation*, *5*(1), 64-90.

Newman, J. (2013). Illegal deposit: Game preservation and/as software

piracy. *Convergence*, *19*(1), 45-61.

Richeson, J. M. (2017). *Anatomy of a hardware emulator* (Doctoral dissertation).

Game Boy CPU Manual (found at http://marc.rawer.de/Gameboy/Docs/GBCPUman.pdf)

# Appendix 3 – Additional Code

union

{

struct BUS

{

ui8 cartridge[0x8000];

ui8 video[0x2000];

ui8 ramBank[0x2000];

ui8 ram[0x2000];

ui8 echoRam[0x1E00];

ui8 oam[0xA0];

ui8 unusable[0x60];

union

{

struct

{

ui8 joypad; // 0xFF00

… Additional I/O Registers

ui8 boot\_rom\_switch; // 0xFF50

}io;

ui8 m\_io[0x80];

};

ui8 internal\_ram[0x7F]; // 0x7F

ui8 interrupt;

}bus;

ui8 m\_bus[0x10000]; //64kb

};

The Memory bus of the Gameboy divided using Unions.

void GB::RenderTile(bool unsig, ui16 tileMap, ui16 tileData, ui8 xPos, ui8 yPos, ui8 pixel, ui8 pallette)

{

ui8& line = ReadData(LYRegister);

ui16 tileRow = (((ui8)(yPos / 8)) \* 32);

ui8 tileColumn = (xPos / 8);

//Get the address for the tileData

ui16 tileAddress = tileMap + tileRow + tileColumn;

ui16 tileLocation = tileData;

i16 currentTile;

if (unsig)

{

currentTile = (ui8)ReadData(tileAddress);

tileLocation += (currentTile \* 16);

}

else

{

currentTile = (i8)ReadData(tileAddress);

tileLocation += ((currentTile + 128) \* 16);

}

// Determine which line is being drawn

ui8 vline = (yPos % 8) \* 2;

//Get the upper and lower bytes of tile data (1 bit from each combine to create the colour)

ui8 upper = ReadData(tileLocation + vline);

ui8 lower = ReadData(tileLocation + vline + 1);

int colourBit = xPos % 8;

colourBit -= 7;

colourBit = -colourBit;

//Combining the tile data together to create the value for the pixel colour

int colourNum = (lower >> colourBit) & 1; // Get the set bit

colourNum <<= 1;

colourNum |= (upper >> colourBit) & 1;

int pixelIndex = pixel + (DISPLAY\_WIDTH \* line);

//Retrieve the colour values of the pixel from the pallette

pixelRGB colour = currentPallete[getColourFromPallette(pallette, colours(colourNum))];

//Store them in the framebuffer

frameBuffer[pixelIndex \* 4] = colour.blue;

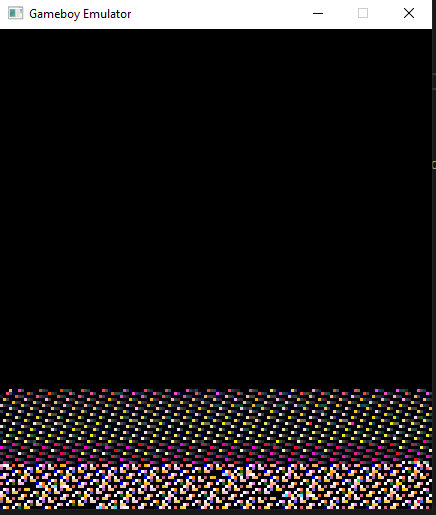
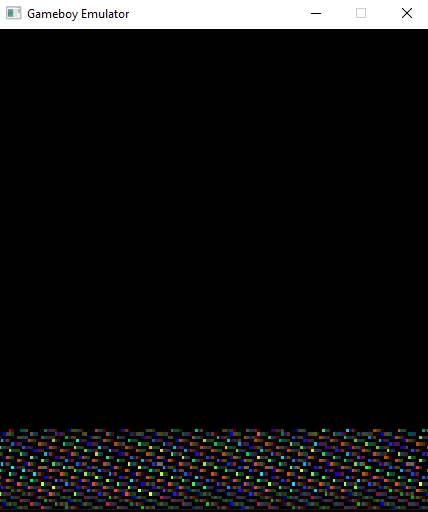
frameBuffer[pixelIndex \* 4 + 1] = colour.green;

frameBuffer[pixelIndex \* 4 + 2] = colour.red;

}

Code used to render the 8x8 tiles. This is similar to the implementation of drawing sprites.

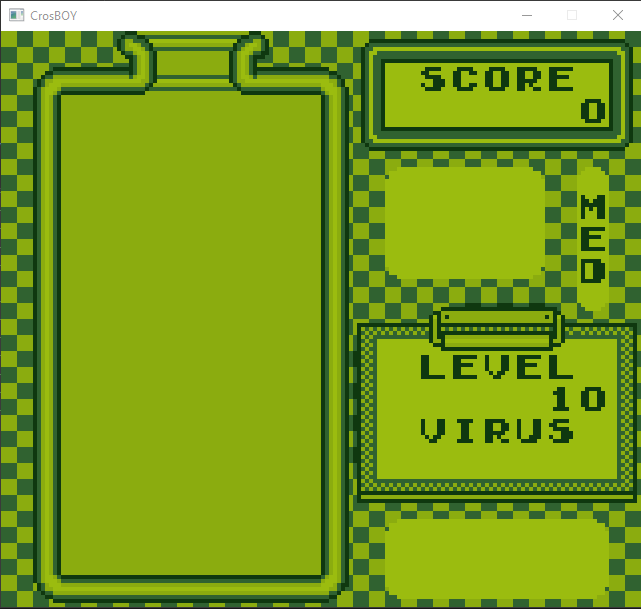
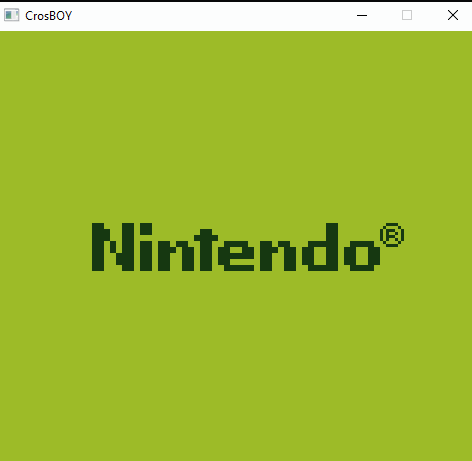
# Appendix 4 – Development Progression



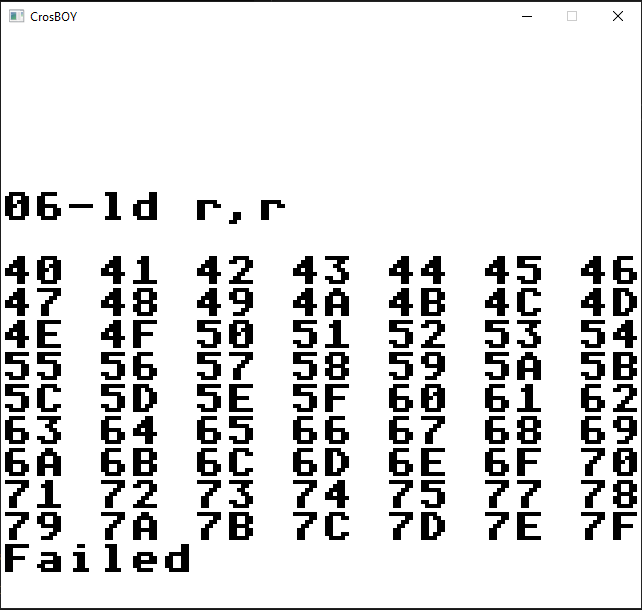
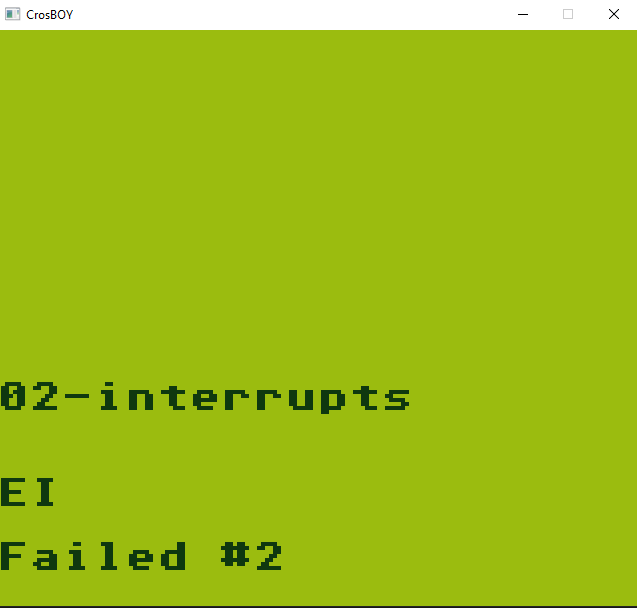
Initial problems with the framebuffer array caused pixels to be formatted incorrectly.



Once the pixels were formatted correctly, the battle continued with tile data being parsed incorrectly.

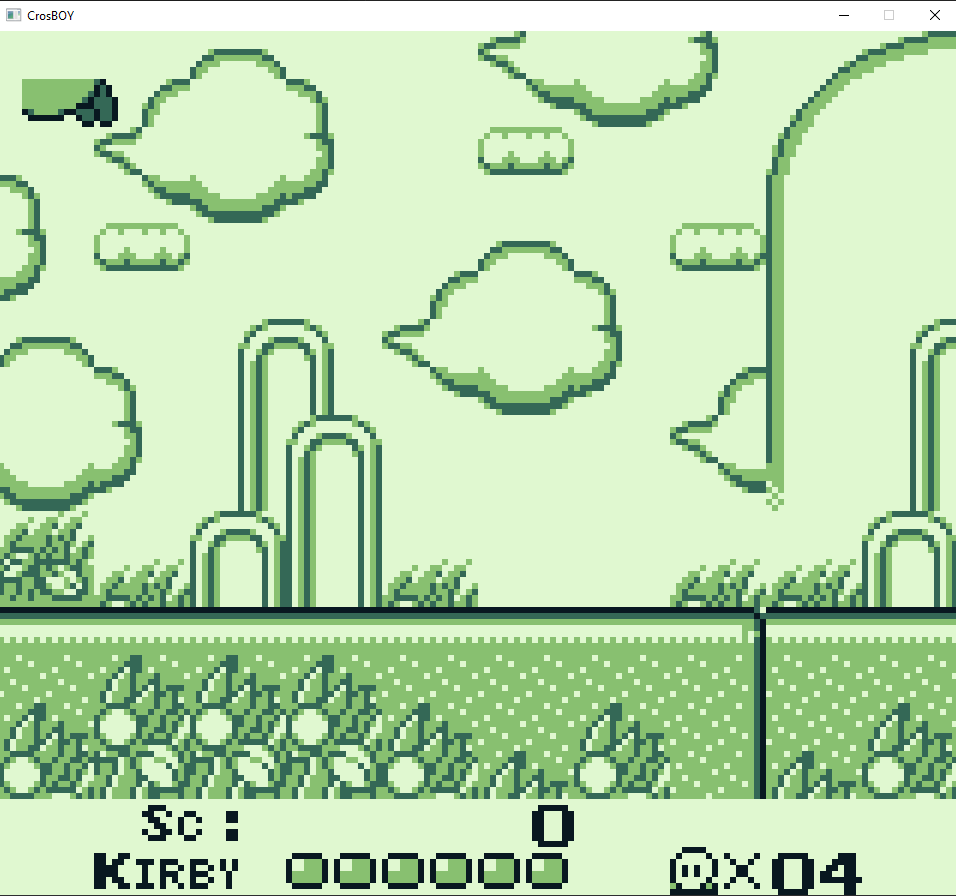
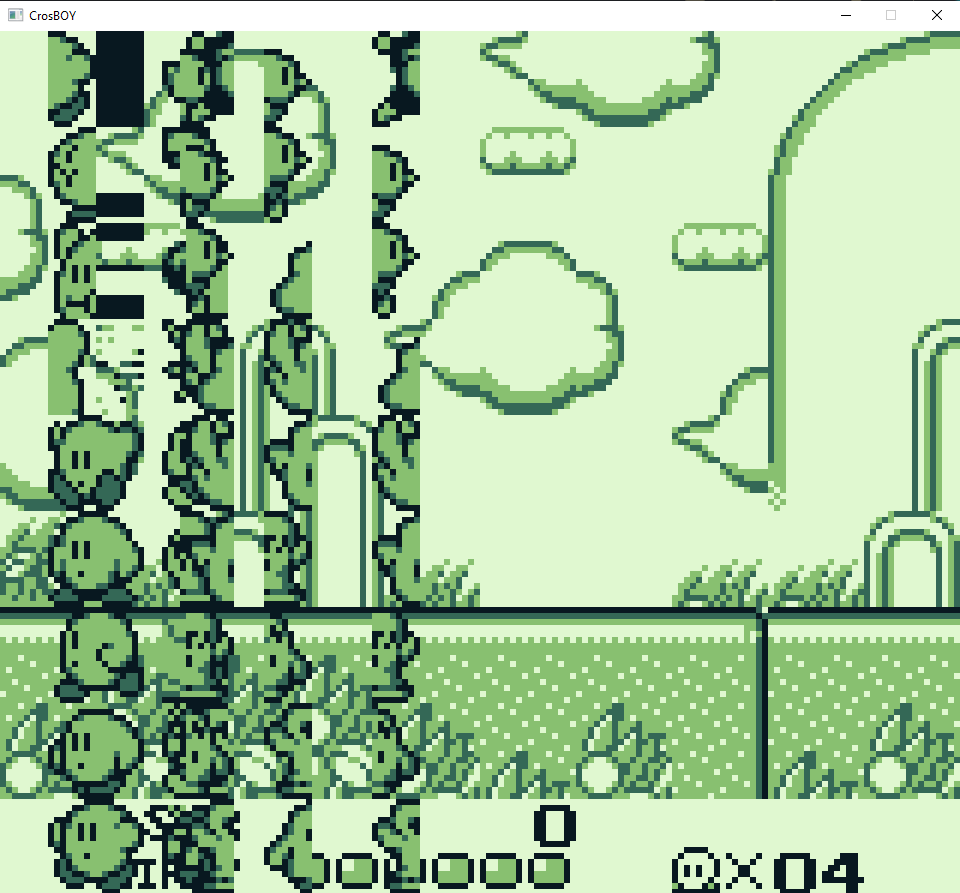


The BIOS loading successfully and Dr Mario drawing the background successfully (but no sprites!)

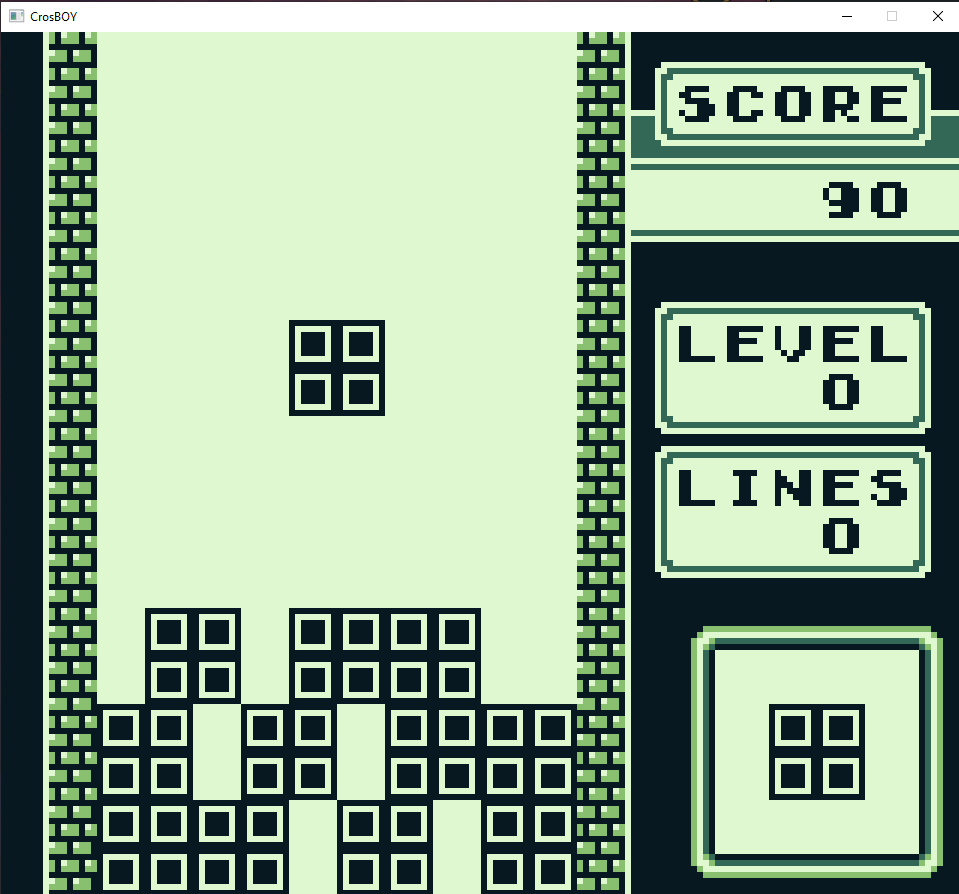
 

An example of the blargg’s test ROMs providing insight into the tests that failed. The numbers in the first image represent the Instruction that failed the test, the second image provides an error code that can be used along with the blargg’s source information to understand which test is failing and why.

# Appendix 5 – Graphical Glitches

The first image shows the sprite of Kirby being drawn incorrectly and cut off when near the edge of the screen. This was due to incorrectly using unsigned integers for the x and y position thus causing data to be lost. The second image was caused by problems with checking the Y position of the sprite causing the tile data to be corrupted.



The lack of Timer implementation led to this humorous side effect. (It made Tetris super easy!)



The Legend of Zelda, although running, has problems with parsing the sprite data, causing this creepy effect to occur.

# Appendix 6 – Game Testing

Table 4 Non-Functional Game testing

|  |  |
| --- | --- |
| Tetris – ROM Only | |
|  | Descriptions of the games will go here. |
| Dr Mario – ROM Only | |
|  |  |
| Kirby’s Dreamland – MBC1 + RAM | |
|  |  |
| The Legend of Zelda, Link’s Awakening – MBC1 + RAM + Battery | |
|  |  |

|  |  |  |
| --- | --- | --- |
| Super Mario Land 1– MBC1 | | |
|  | |  |
| Super Mario Land 2 6 Golden coins – MBC1 + RAM + Battery | | |
|  | |  |
| Pokémon Red – MBC3 + RAM + Battery | | |
|  | |  |
| Gameboy Gallery 5 | | |
|  |  | |

# Appendix 7 – Class Diagram

