CHALMERS
UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

Game Boy Emulation

Emulating a Complex System

Bachelor's thesis in Computer Science and Engineering

Algot Axelzon
Isak Lindgren
Carl Lindh
David Möller
Andreas Palmqvist
Arvid Rydberg

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

BACHELOR'S THESIS 2021

Game Boy Emulation

Emulating a Complex System

Algot Axelzon
Isak Lindgren
Carl Lindh
David Möller
Andreas Palmqvist
Arvid Rydberg

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

Game Boy Emulation
Emulating a Complex System
Algot Axelzon Isak Lindgren Carl Lindh David Möller Andreas Palmqvist Arvid Rydberg

Supervisor: Roc R. Currius, Department of Computer Science and Engineering
Examiner: Sven Knutsson, Department of Computer Science and Engineering

Bachelor's Thesis 2021
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Image of the original Game Boy from 1989. From [1]. Public Domain.

Gothenburg, Sweden 2021

Abstract

This thesis studies the subject of system emulation through the development of a set of software microcontrollers and the assembling of them into a complex system. The specific system aimed to be emulated is the original Game Boy released in 1989. This requires the developers to reproduce specific hardware behaviour through software and therefore requires certain knowledge of the system which is to be emulated. While the Game Boy is a proprietary product owned by Nintendo, the produced system uses no copyrighted material.

Through the use of documentation provided by the reverse engineering of the original hardware done by members of the community, this thesis shows that an emulator can be created by combining a set of software microcontrollers. Moreover, it is concluded that while the academic interest in the emulation of simple systems might be limited, it could also could be used to generate interest in low-level programming.

Keywords: Emulation, Gameboy, Game Boy, C++, OpenGL, OpenAL, ImGui

Sammandrag

Denna kandidatuppsats studerar systememulation genom att utveckla ett flertal mjukvarumikrokontroller som därefter kombineras för att tillsammans bilda ett komplext system. Det specifika systemet som emuleras är den första Game Boy-konsollen, släppt 1989. För att genomföra detta krävs det att utvecklarna reproducerar den specifika hårdvaran i mjukvara, vilket kräver viss kunskap om det ursprungliga systemet. Då Game Boy är en licensierad produkt, ägd av Nintendo, är det värt att notera att det framtagna systemet ej använder något upphovsrättsskyddat material.

Dokumentationen som använts för att skapa denna emulator har tagits fram genom att demontera och undersöka originalhårdvaran. Detta har gjorts av ett flertal deltagare i en emulatorintresserad internetgemenskap. Avslutningsvis konstateras att det, trots det begränsade akademiska intresset för systememulering av enkla system, finns potential för att använda sig av det för att skapa intresse för lågnivåprogrammering.

Nyckelord: Emulering, Gameboy, Game Boy, C++, OpenGL, OpenAL, ImGui

Acknowledgements

We would like to thank our supervisor Roc R. Currius for providing an initial project and for rigorous feedback regarding the writing of this report. During the project he also provided weekly guidance in our efforts to develop the emulator. Secondly we would like to thank Albin Johansson for providing thorough feedback on the code written for this project, which gave many valuable insights regarding style and general use of C++.

Algot Axelzon, Isak Lindgren, Carl Lindh, David Möller, Andreas Palmqvist, Arvid Rydberg, Gothenburg, June 2021

Terminology

- ROM - Read only memory. A memory which in the context of emulator development contains the game data.
- RAM - Random access memory. As opposed to ROM, RAM can both be read from and written to. The data stored in RAM is however volatile - the memory stores data only as long as it has power.
- `0x` - Using the `0x` prefix indicates that the following value is to be interpreted as a hexadecimal value. For example the value `0xA0` is to be interpreted as 160.
- KiB/MiB - KiB and MiB are used for Kibibyte and Mebibyte, they are of base two and are representing 1024 (2^{10}) byte and 1 048 576 (2^{20}) byte respectively.
- I/O - Input/Output, used for describing units handling either input or output, such as a display or keyboard.
- Scanline - The process of rendering a row of pixels to the display.
- CPU - Central Processing Unit
- PPU - Pixel Processing Unit, responsible for producing pixel data for the Game Boy's LCD.
- APU - Audio Processing Unit, responsible for generating audio data.
- MMU - Memory Management Unit, responsible for handling memory mapping within the emulator.
- Microcontroller - A small computer on a single chip.
- Software microcontroller - Software imitating the function of a microcontroller.
- Sprite - A 2D bitmap representing an image.
- Joypad - A device that connects the user's input with the system. In the context of the Game Boy this refers to four directional buttons and four action buttons.

Contents

List of Figures

List of Tables

1 Introduction

In this chapter, both the concept of an emulator and the Game Boy are introduced. Furthermore, the purpose of this report, which is to develop an emulator of said console is specified.

1.1 Emulators in general

An emulator generally refers to some kind of hardware or a piece of software which emulates the functionality of some other computer system [2] [3]. While the main goal of an emulator is to imitate another computer system, an emulator may also improve upon the original system by providing additional features or by exceeding the performance of the original system [3].

Emulators are used to bridge the gap between different kinds of hardware. For example, it is possible to, within one's operating system, run other operating systems; emulate sound hardware, such as a guitar amplifier or play video games on hardware not specifically made for said games [4]. Within the scope of this report, the term emulator refers to software used specifically to make it possible to play games on systems that are not originally intended to run these games.

1.2 An Introduction to the Game Boy

In 1989, Nintendo released the handheld video game console Game Boy, designed by the inventor Gunpei Yokoi [5] [6]. It was the first entry in a family of video game systems, both 8-bit and 16-bit, the last of which being the Game Boy Advance Micro which was discontinued as late as 2010 [6]. The 8-bit Game Boy is the predecessor to Nintendo's more modern handheld consoles: the Nintendo Switch [7] and The Nintendo DS (Dual Screen) [8] console family.

The 1989 Game Boy, also known as DMG-01 (Dot Matrix Game), is an 8-bit handheld console characterised by its green monochromatic LCD display and bulky grey design (see Figure 1.1) [5]. The other 8-bit Game Boy models are Game Boy Pocket, Game Boy Light and Game Boy Color. The Game Boy Pocket and Game Boy Light are slimmer versions of the DMG-01 with upgraded LCDs while Game Boy Color has a colour LCD and can play games not supported by the older Game Boys [6]. In this report Game Boy refers to either of the DMG-01, Game Boy Pocket, Game Boy Light or all of them grouped together since there are no major differences between them. The Game Boy Color and Game Boy Advance models are not covered by this report.

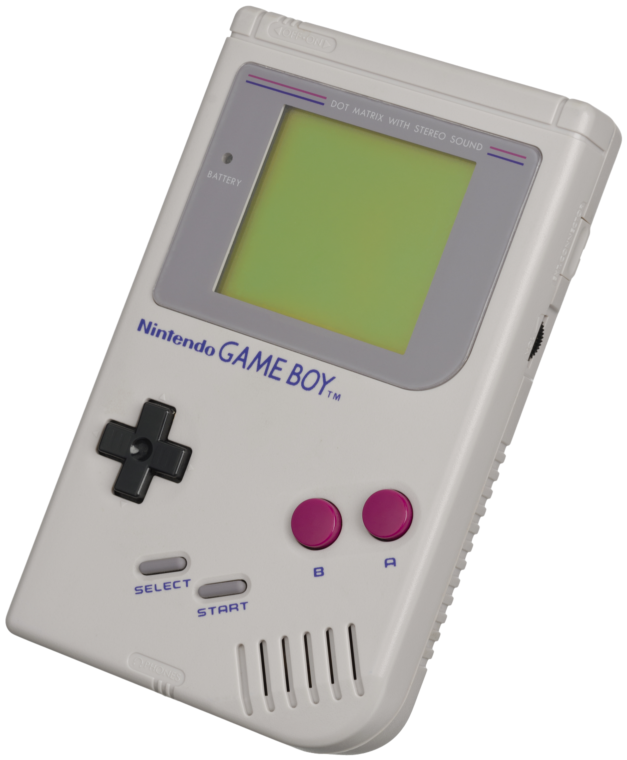


Figure 1.1: The first Game Boy model, DMG-01. From [9]. Public Domain.

The Game Boy and Game Boy Color have together sold approximately 118 million units [10], making them one of Nintendo's most successful product lines ever, while also being the birthplace of popular franchises such as Kirby and Pokémon [11] [12] [13]. According to Katie Smith-Wong at the entertainment news site Den of Geek [5], there are a number of factors that led to the success of the Game Boy, the first being the design of the consoles; Compared to its competitors, the Game Boy and its variations were simplistic in their design. Because of this simplicity, the Game Boy consoles were more affordable than the competing hand held consoles while also having a longer battery life. Furthermore, multiplayer mode was possible thanks to the built in game link port. This in combination with the affordability and the superior battery life made the different Game Boys excellent multiplayer consoles. The many iconic game titles also contributed to the Game Boy's success, most notably Tetris [14] and the Pokémon [12] games, which reached out to the mainstream audience .

Recreating the behaviour of a computer system within another computer system through software is a computationally heavy task since software is much slower than electronics; therefore, simpler computer systems were the first to be emulated on personal computers through software. During the middle of the 1990s the personal computers were powerful enough to make emulation of earlier video game consoles feasible [15]. During this period Nintendo was one of the most popular media brands [16] and thus emulators for Nintendo's consoles started to appear on the internet. The first known emulator of the Game Boy released, that could run commercial games, was an emulator by the name Virtual Game Boy. It was created by Marat Fayzullin in 1995 for some unknown system and was ported to PC somewhere between 1995 and 1996 [17]. Since then many more Game Boy emulators have been released; written in many different languages and ranging in functionality [18].

The central component of the Game Boy is the DMG-CPU [19] which is a chip containing a pixel processing unit (PPU), a central processing unit (CPU) and an audio processing unit (APU). Connected to the DMG-CPU are 8 KiB of RAM and video RAM (VRAM). The PPU outputs pixels, that can be one of four shades, to an LCD that is 160 $\times$ 144 pixels in dimensions and the audio is output either through a headphone jack or a built in speaker. Games are stored on ROM chips that come inside cartridges see Figure 1.2.

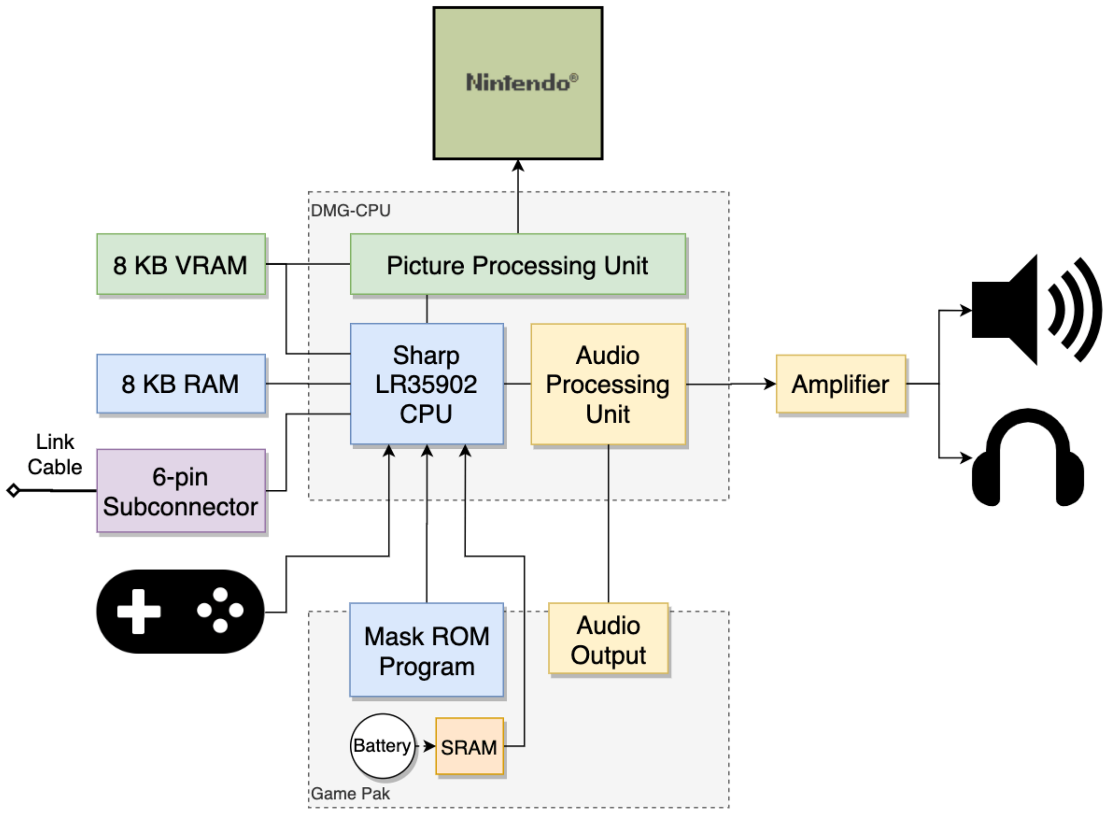


Figure 1.2: Diagram of the architecture of the Game Boy and the general architecture of a game cartridge (Game Pak). From [19]. CC BY 4.0.

1.3 Purpose

The purpose of this thesis is to explore the possibility of developing a set of microcontrollers which can be assembled to a complete complex system. This is done through emulating the original Game Boy from 1989. As per the initial project description, focus will be on having a graphical interface working, not taking sound or other systems into account.

Developing an emulator presents additional purposes as a problem with older gaming consoles is that many of them are no longer produced and are therefore becoming harder and harder to come by. An emulator solves this problem by letting people play old console games on newer hardware. This thesis therefore aims to create a Game Boy emulator for the PC to allow future generations to have the same experience as the people playing the games on the original console whilst also preserving a piece of history. Additionally, the authors would like to share the accumulated knowledge of emulator development with others by providing documentation for the design as well as the code, so that others may learn from it and possibly use it as a base for their own emulators.

1.4 Delimitations

To reduce the complexity and scope of this thesis certain delimitations have been made. The thesis has been limited to only emulate the original Game Boy from 1989 and subsequent with equivalent hardware. This reduces the complexity of the thesis significantly as it reduces the amount of different systems and chips needed to be researched. Furthermore, it also means that the hardware which is to be researched is older and therefore of lower complexity than more modern hardware.

Regarding some features and systems in the Game Boy, further limitations are made. First of all; “Serial Data Transfer” is not supported. This is a feature which is used to connect a Game Boy to another Game Boy to, amongst other things, allow for multiplayer gameplay [20]. Although this could be implemented in software either by connecting two instances of the emulator on the same machine, or separate machines through internet play, this is simply not within the primary scope of emulating the Game Boy hardware.

The games for the Game Boy comes in cartridges called Game Paks [21]. The specifications of these can differ between different games with a total of 28 different configurations supported by the console, including six different memory bank controllers (MBCs) [22]. The aim is not to support all of the configurations, but rather the most significant types.

The Game Boy also has a feature called “Vin”, allowing the use of external audio hardware located in the cartridge. However, none of the games made for Game Boy uses this feature [23] and is therefore excluded from being implemented in the emulator.

1.5 Ethics

Creating an emulator for a previously commercially sold product comes with complications. Because of Nintendo having created the Game Boy, it is natural that they are committed to making sure no one infringes the copyrights related to their product. When discussing emulators, there is also often a discussion of what is right and wrong, what counts as piracy and what is legal to do around the subject of copyright.

Nintendo Australia states the following regarding emulators [24]:

"A Nintendo emulator is a software program that is designed to allow gameplay on a platform that it was not created for. A Nintendo emulator allows for Nintendo console based or arcade games to be played on unauthorised hardware. The video games are obtained by downloading illegally copied software, i.e. Nintendo ROMs, from Internet distributors. Nintendo ROMs then work with the Nintendo emulator to enable game play on unauthorised hardware such as a personal computer, a modified console, or another video game device."

Thus, Nintendo claim that the games which are used together with emulators are obtained illegally. However, they do not explicitly state that they consider the use nor development of emulators illegal, as long as no copyrighted material is used. All of this leads to a lot of confusion around the legality and ethics of emulation, further discussion regarding this topic can be found in Section 5.3.

2
Theory

This section aims to describe the various components used in the Game Boy. This is needed to then be able to understand how to properly emulate the function of those components.

2.1 The Central Processing Unit

The Central Processing Unit (CPU) used in the Game Boy is a Sharp LR35902 [25] [26] which is designed specifically for the Game Boy. The chip is heavily inspired by the Zilog Z80 [27] and the Intel 8080 [28]. While the CPU itself has a clock frequency of 4 MHz [25], one can consider it having an actual clock speed of 1 MHz. This is due to the fact that the CPU is bound by the speed of the memory [29] (the rate the RAM can provide data to the CPU) which has a clock speed of 1 MHz, see slide 149 of [30]. Furthermore, the CPU has a 16-bit address bus [31] and an 8-bit arithmetic logic unit (ALU).

2.1.1 Registers

The CPU has six general purpose registers, B, C, D, E, H and L [32]. These are all 8 bits each, but can be used pairwise as three 16-bit registers, BC, DE and HL. There are also two additional 8-bit registers, A and F, which both have specific purposes. A is the accumulator register where all the arithmetic is done. The F register, despite being 8 bits, only uses 4 bits to store the value of the four flags of the ALU. These are the zero, negative, half-carry and carry flags [32], abbreviated as Z, N, H and C respectively. Additionally, the CPU has a 16-bit program counter (PC) and stack pointer (SP).

2.1.2 Instruction set

In total the CPU supports 500 assembly instructions, of which 244 are 8-bit and 256 are 16-bit, see Figure 2.1 for the 8-bit instructions. For the CPU to interpret the 16-bit operations, a certain prefix is used (`0xCB`) found in the 8-bit table allowing the CPU to interpret the following 8 bits as an instruction from the 16-bit table. The tables specify for each instruction which flags it affects, how many bytes it is encoded with and how many machine cycles it requires to execute.

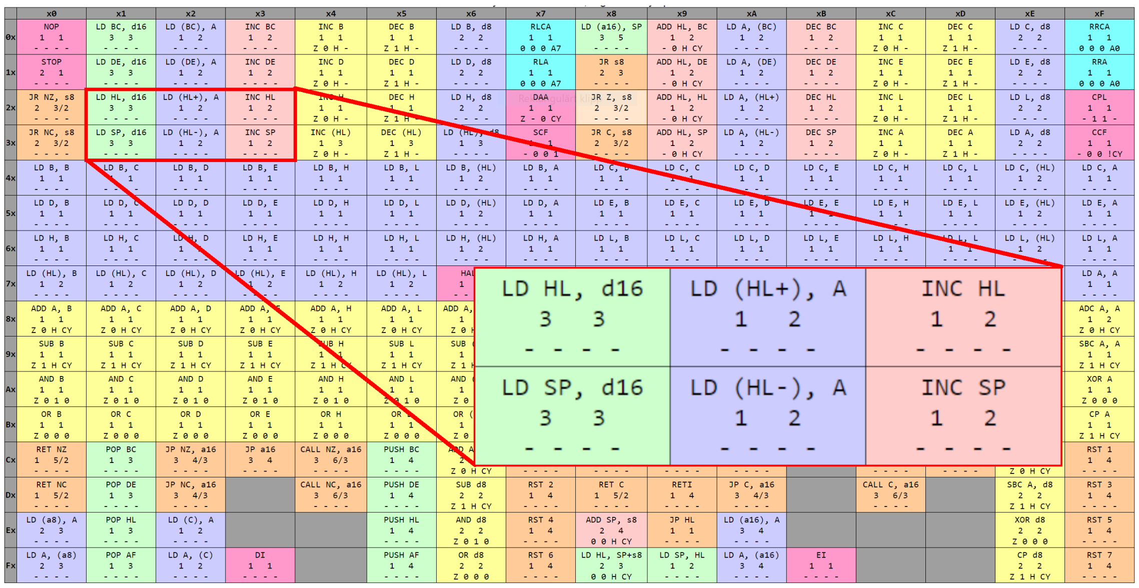

LD HL, d16
3 3
- - - -
LD (HL+), A
1 2
- - - -
INC HL
1 2
- - - -
LD SP, d16
3 3
- - - -
LD (HL-), A
1 2
- - - -
INC SP
1 2
- - - -

Figure 2.1: Displaying the 8-bit operation codes for the LR35902, highlighting six of the operation codes, showing their assembly instruction, number of bytes used, number of cycles to execute and what flags are affected. From [33]. Modified with permission.

2.1.3 Interrupts

Like many other CPUs, the Sharp LR35902 supports interrupts. Interrupts are used in order to break the regular flow of the CPU, forcing it to handle the interrupt before returning to what it was doing previously. Different interrupts have different pre-defined addresses where the respective interrupt routines are stored [34]. When an interrupt occurs, the CPU executes the interrupt routine, and then continues execution from the address that was current when the interrupt occurred. The existing interrupts and their functionalities are the following [34]:

- V-blank - An interrupt sent from the PPU when it has created a whole frame and wants to draw it, for more information see Section 2.3.
- LCD STAT - A customisable interrupt regarding various conditions of the PPU. An interrupt is generated any time the PPU goes from not meeting any condition to meeting at least one condition. Which conditions are used is determined by the STAT register of the PPU, for more information, see Section 2.3.3
- Timer - Sends an interrupt request when a set timer has run out, for more information see Section 2.2.8.
- Serial - The serial transfer interrupt which handles the communication between two Game Boys when they are connected through the serial port. This is not implemented, as is explained in Section 1.4.
- Joypad - An interrupt which handles the input from the joypad, for more information see Section 2.2.7.

The interrupts are implemented by having three different types of flag registers [34]:

- IME - Interrupt Master Enable. This flag enables and disables all other interrupt flags. Can only be manipulated through specific instructions.
- IE - Interrupt Enable. Enables or disables a specific flag. Consists of five bits, one for each type of interrupt. Manipulated through memory, located at `0xFFFF`.
- IF - Interrupt Flag. Shows that an interrupt has been requested. This is the flag which is set when an interrupt is raised. The register consists of five bits, one for each type of interrupt. Manipulated through memory, located at `0xFF0F`.

If IME is disabled, IE and IF can still be altered, but they are not acted upon until IME is enabled once again. For an interrupt to occur and be handled by the CPU three things therefore need to happen: both the IME flag and a specific IE flag must be enabled, and the corresponding IF flag must be triggered [34].

2.2 Memory and I/O devices

The CPU needs to access memory and communicate with peripheral devices. To achieve this the Game Boy uses a 16-bit address bus and an 8-bit data bus [35]. The memory and different peripheral devices are mapped to specific memory addresses visualised in Figure 2.2.

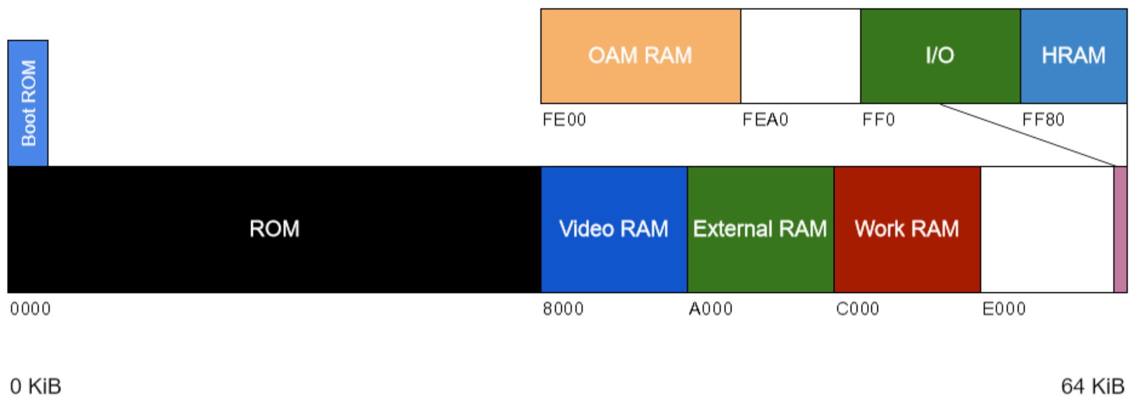


Figure 2.2: Visual representation of how the address space is divided in the Game Boy. From [30]. Adapted with permission.

In the Game Boy's 64 KiB address space, different address ranges are used for different purposes. Below is a description of how this is handled. Note that the I/O devices are controlled through manipulating address space and that the more complex I/O devices, the PPU and APU, are described in Section 2.3 and Section 2.4 respectively.

2.2.1 Boot ROM

Address: `0x00 - 0xFF`
The boot ROM is a 256 byte ROM located on the DMG-CPU and contains the code that is executed when the Game Boy is powered on [36]. The purpose of the boot ROM is to initialise the different hardware units and prepare the Game Boy to execute game code [37]. The official boot ROM used on the Game Boy does this while scrolling down the Nintendo logotype. The use of copyrighted material on the boot ROM has legal implications which is further discussed in Section 5.3. In practice the boot ROM is only accessible a short time when the Game Boy is powered on due to the fact that it disables itself after it has been run. Disabling the boot ROM is done by storing a non-zero value at memory address `0xFF50` and will map the addresses to the ROM instead, see the overlap in Figure 2.2.

2.2.2 ROM

Address: `0x0000 - 0x7FFF`
The ROM located in the Game Boy's memory map is the area where the interactions with data from the inserted game ROMs happen [31]. Although the allocated size for the ROM on the Game Boy is only 32 KiB, the ROMs it supports can be much larger. This is possible through the use of bank switching using a Memory Bank Controller (MBC) [38], located in the Game Pak ROM, see Section 2.2.10 for more information.

2.2.3 Video RAM

Address: `0x8000 - 0x9FFF`
The Video RAM (VRAM) is an 8 KiB RAM [31] that is accessible by both the CPU and PPU. It it used by the CPU to store data which the PPU uses for drawing. While the PPU is drawing, it is reading from the VRAM and the CPU is therefore not allowed access. For more information regarding how the VRAM is used, see Section 2.3.

2.2.4 External RAM

Address: `0xA000 - 0xBFFF`
The external RAM (XRAM) is an optional memory that is located on some of the game cartridges. Its size varies from game to game and can, if desired, also make use of bank switching (see Section 2.2.10). The XRAM is often used for saving progress and high score tables [31]. This is done by powering the XRAM with a small battery inside the game cartridge to make sure the data is conserved between different gaming sessions.

2.2.5 WRAM

Address: `0xC000 - 0xDFFF`
The Work RAM (WRAM) is the main memory of the system which is used by the CPU to store any data the game uses while running. The WRAM is located on the DMG-CPU and has a size of 8 KiB [31].

2.2.6 OAM RAM

Address: `0xFE00 - 0xFE9F`
The Object Attribute Map (OAM) is the area where sprite data is stored. The OAM is divided into 40 blocks of four bytes each with each block corresponding to a sprite. Both the PPU and CPU have direct access to this memory area. While the PPU is rendering scanlines, the CPU has limited access to this area [39]. The CPU can also transfer data into the OAM through the use of Direct Memory Access (DMA) transfers, this process is described in detail in Section 2.3.5.

2.2.7 Joypad

Address: `0xFF00`
The Joypad is the device that connects the user's input with the system. There are eight buttons on the Game Boy, four directional buttons: up, down, left and right, and four action buttons: A, B, Start and Select. The buttons have two states, pressed or not pressed. These states of the buttons are stored at address `0xFF00` as a 2x4 matrix [40].

By writing specific values to bit 4 and 5 either the action buttons or directional buttons will be selected. When selecting either the action or direction buttons bit 0-3 will correspond to the selected buttons [40]. See Table 2.1.

```
Bit 7 - Not used
Bit 6 - Not used
Bit 5 - Select Action buttons    (0=Select)
Bit 4 - Select Direction buttons (0=Select)
Bit 3 - Input: Down  or Start    (0=Pressed) (Read Only)
Bit 2 - Input: Up    or Select   (0=Pressed) (Read Only)
Bit 1 - Input: Left  or B        (0=Pressed) (Read Only)
Bit 0 - Input: Right or A        (0=Pressed) (Read Only)
```

Table 2.1: Layout of the button states located in memory address `0xFF00`. From [40]. Adapted with permission.

When a button is pressed, an interrupt is raised, but only if the pressed button is a button that is selected by bit 4 and 5 (either an action or a directional button) [40]. Regarding how to actually handle the joypad input, it is up to the developer to either poll the 2×4 -matrix or by using the interrupts.

2.2.8 Timer

Address: `0xFF04 - 0xFF07`
The Game Boy has a built-in timer which can be used to perform time sensitive tasks. It is used and controlled by reading and writing to the following registers [41]:

Divider Register - DIV
Address: `0xFF04`
The DIV register is a counter that always increments at a rate of 16384Hz. When writing any value to the address the counter resets to 0 [41].

Timer counter - TIMA
Address: `0xFF05`
This is another counting register, it increments at the rate that is specified in the TAC register. When the counter value is 0xFF and overflows it is reset to the value in TMA as the new counter value. When the counter overflows the "Timer"-interrupt is raised [41]. Writing to this register simply changes the counter to the written value.

Timer Modulo - TMA
Address: `0xFF06`
This register holds the timer modulo value. This value is loaded into the TIMA register when the TIMA counter overflows [41]. It can be used to further modify the chosen clock speed chosen in the TAC register.

Timer Control - TAC
Address: `0xFF07`
This register is used to control the timer. By writing to this register the timer can be enabled or disabled. The rate at which the TIMA register will increment can also be controlled by writing to this register [41].

As shown in Table 2.2, bit 2 enables or disables the timer. Notice that disabling the timer does not stop the DIV counter. Bits 0 and 1 are used to select one out of four possible clock speeds; 4069 Hz, 262144 Hz, 65536 Hz or 16384 Hz [41].

```
Bit  2   - Timer Enable
Bits 1-0 - Input Clock Select
           00: CPU Clock / 1024 (4096 Hz)
           01: CPU Clock / 16   (262144 Hz)
           10: CPU Clock / 64   (65536 Hz)
           11: CPU Clock / 256  (16384 Hz)
```

Table 2.2: Layout of the each bit in the timer control register located at address `0xFF07`. From [41]. Modified with permission.

2.2.9 HRAM

Address: `0xFF80 - 0xFFFE`
The high RAM or HRAM is very much like the WRAM although it is much smaller and faster to access. It is faster because of a special instruction that assumes that the highest byte in the address is `0xFF` and therefore saves time decoding the instruction [33]. During a DMA-transfer (see Section 2.3.5) the CPU is limited to use only this memory. The size of the HRAM is 126 bytes [31].

2.2.10 Memory Bank Controllers

As previously mentioned, the Game Boy uses a 16-bit address bus, which in turn limits the memory addresses available for the game ROM and external RAM. To bypass this limitation many game cartridges make use of a memory bank controller (MBC). The MBC is used to map different sections of a memory, called memory banks, to one specific address range, this method is called bank switching. It works as follows: reading from addresses `0x0000` to `0x7FFF` returns a byte stored on the ROM as expected, but writing to those same addresses will instead change the MBC's control registers, this is further described below and in Figure 2.3. These control registers control what memory bank is to be used for both the game ROM and the external RAM (if any).

There are different versions of MBCs which all support various arrangements of additional hardware. Examples of such hardware are external RAM with or without a battery powering said RAM and in some cases also a timer [42]. For example, MBC3 supports up to 2 MiB of ROM and/or 64 KiB of RAM, and Real Time Clock (RTC) with battery [42].

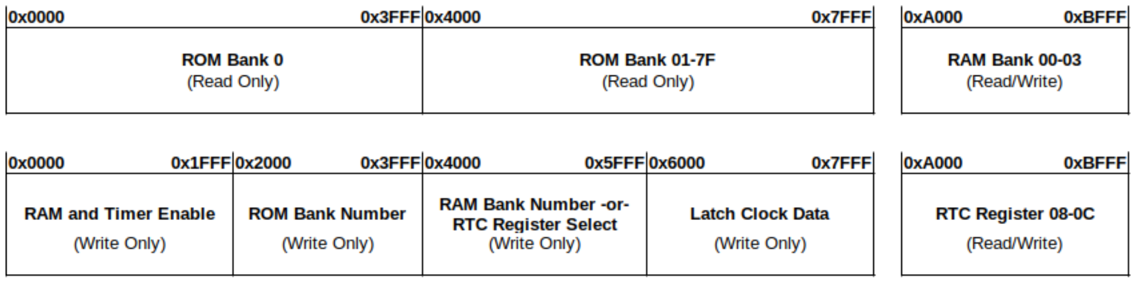


Figure 2.3: Memory map of MBC3. While all MBCs do not necessarily contain the same features, most follow a similar structure [42].

The following is a description of the structure and functionality of MBC3, shown in Figure 2.3 [42]:

ROM Bank 0
Contains the first 16 KiB of the ROM.

ROM Bank 0x01-0x7F
This part of the memory is used for bank switching. There are at most 127 available memory banks each with a size of 16 KiB, resulting in a total ROM size of 128×16 KiB = 2 MiB, including ROM bank 0.

RAM and Timer Enable
Writing specific values to this address will enable reading and writing to XRAM or RTC registers depending on which is currently active.

ROM Bank Number
Writing to this address controls which ROM bank is in use.

RAM Bank Number/RTC Register Select
Writing to this area maps the corresponding XRAM bank or RTC register into the memory of `0xA000-0xBFFF`.

RAM Bank 0x00-0x03 or RTC Register 0x08-0x0C
Depending on the current **Bank Number/RTC Register selection**, this memory space is used to either access an 8 KiB external RAM Bank, or a single RTC Register, see Table 2.3.

Latch Clock Data
If writing first 0x00, and then 0x01 to this register, the current time becomes latched into the RTC register. This means that the current time can be read from the register while the RTC continues to tick. Additionally the RTC needs a quartz oscillator as well as an external battery to work while the Game Boy is turned off.

0x8	RTC S	Seconds	0-59 (0x00-0x3B)
0x9	RTC M	Minutes	0-59 (0x00-0x3B)
0xA	RTC H	Hours	0-23 (0x00-0x17)
0xB	RTC DL	Lower 8 bits of Day Counter (0x00-0xFF)	
0xC	RTC DH	Upper 1 bit of Day Counter, Carry Bit, Halt Flag	
	Bit 0	Most significant bit of Day Counter (Bit 8)	
	Bit 6	Halt (0=Active, 1=Stop Timer)	
	Bit 7	Day Counter Carry Bit (1=Counter Overflow)	

Table 2.3: List of the different RTC registers and its contents. From [43]. Adapted with permission.

2.3 The Pixel Processing Unit

The purpose of the Pixel Processing Unit (PPU) is to interpret the data residing in VRAM and compose it into a picture which can then be printed on the LCD. The CPU can communicate with the PPU either by loading data into the VRAM or by reading from or writing to a number of hardware registers, most notably the LCD control and status registers. The PPU, on the other hand, uses interrupts to communicate with the CPU [44].

2.3.1 Composing the frame

The frame is composed of three layers: Background, Window and Sprites, rendered in that order. These layers are in turn composed of tiles, 8x8 bitmaps of colour indices, which are stored in VRAM. When rendering the picture, these indices map to one of the four colours using palette tables, which in the original Game Boy was four shades of grey over a green background [19]. The following is a description of the components used to form the frame.

Tiles - As previously mentioned, the tiles are 8x8 bitmaps stored in RAM, 16 bytes per tile. This is because each row of a tile is formed by combining the bits of two bytes. The bytes are organised in two memory regions called *Tile sets*; one ranging from `0x8000` to `0x9000` and one ranging from `0x8800` to `0x9800`, see Figure 2.4. These two tile sets overlap and which one to use is determined by a bit in the LCD Control register (LCDC). The first of the two uses unsigned addressing for tile IDs, while the second one uses signed addressing. [19].

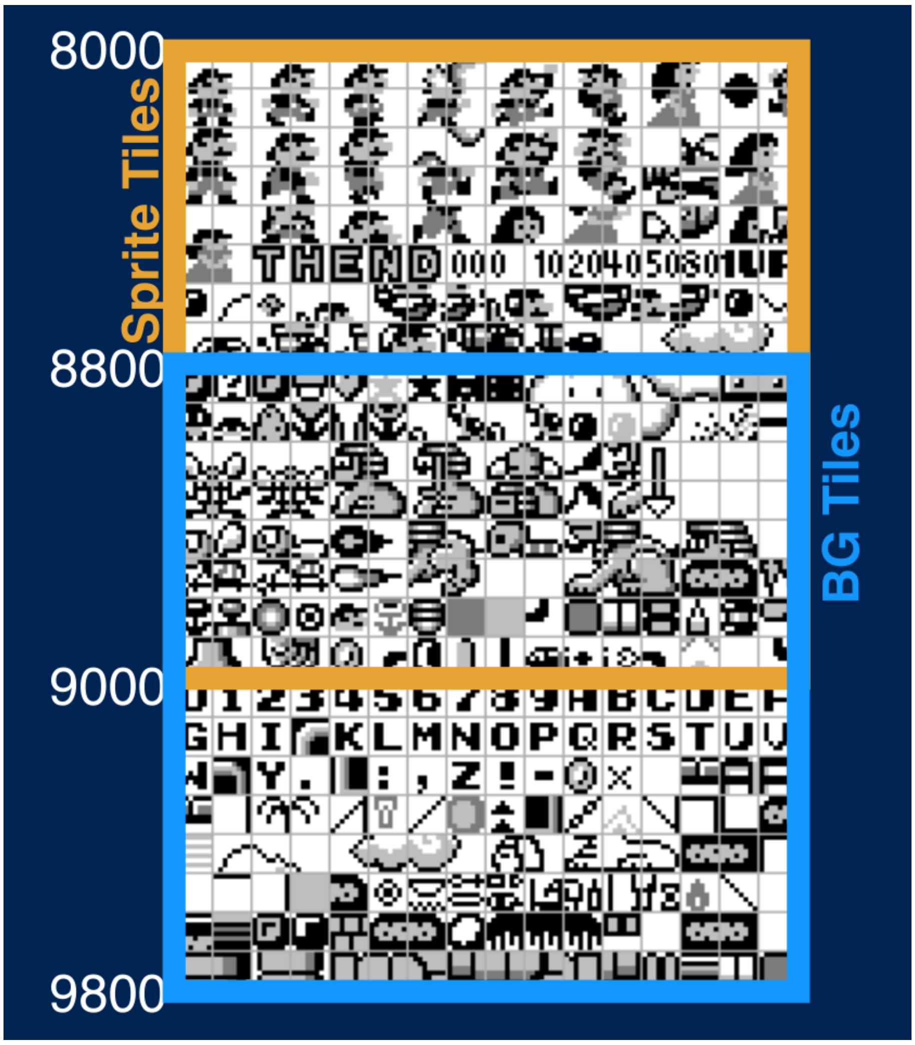


Figure 2.4: The two tile sets. Sprites may only use tiles from the tile set ranging from `0x8000` to `0x9000` while the background and window can use any of the two sets. From [30]. Used with permission.

Background - The background is defined by a map of 32x32 tile IDs. Since a tile is 8x8 pixels, this results in a map of 256x256 pixels; however, the Game Boy's display is only 160x144 pixels. The result of this is that only part of the background is being displayed at any given time, as shown in Figure 2.5. Which part of the map to display is determined by the two hardware registers, SCX and SCY, which specify which pixel should appear in the top left corner of the display. If the edge of the map is reached, the map loops around and continues at the opposite side. There are also two maps available at any given time, which one is being used is determined by a bit in the LCDC [44].

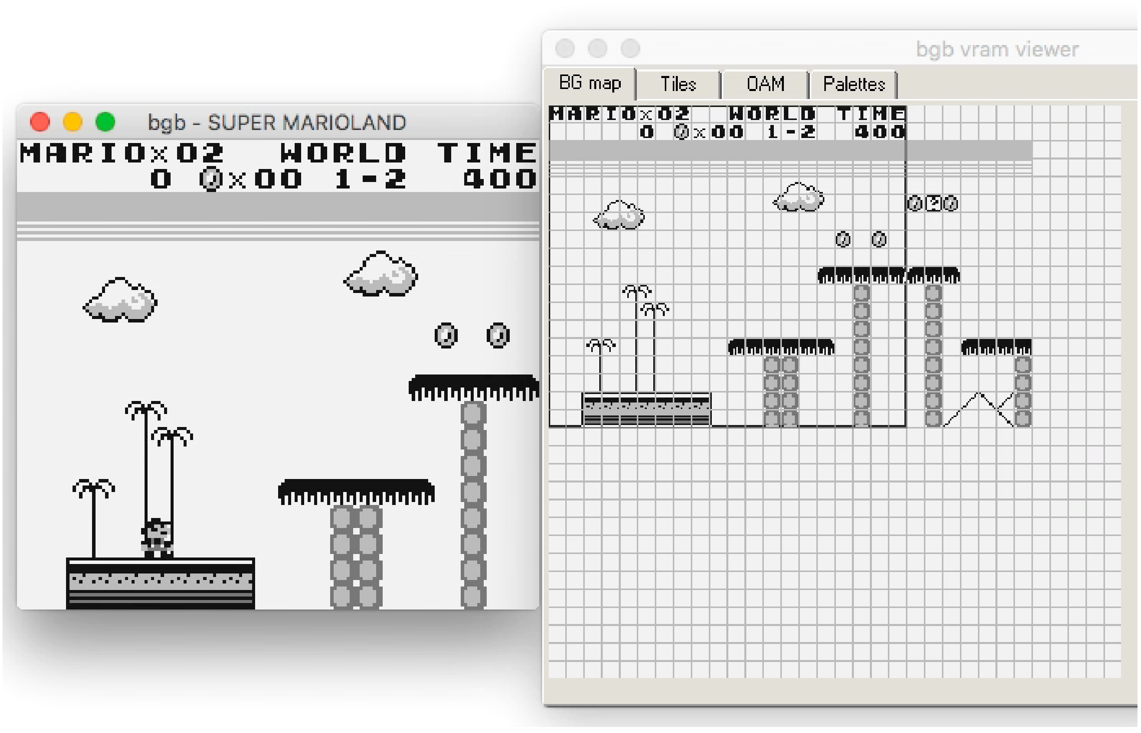


Figure 2.5: What is shown on the LCD in relation to the background map. From [30]. Used with permission.

Window - The window is drawn on top of the background. Just like the background, it consists of a map of tiles; however, this map is only 20x18 tiles, just covering the entire screen and, unlike the background, does not loop around. This layer can be used to display information that should not scroll with the rest of the background, such as a GUI [44].

Sprites - Sprites are objects that can move around on the screen freely. They consist of one or two tiles, depending on a bit in the LCDC register. As opposed to the tiles in the background and window layers, sprites can have transparent pixels, meaning that the background and window are still visible through said pixels. Sprites only consist of tiles from the tile set ranging from `0x8000` to `0x9000`, as shown in Figure 2.4 [44].

Sprite data is stored in a memory region called the Object Access Map(OAM). Each sprite takes up four bytes of memory. These bytes store the sprites' x and y coordinates, the sprites' tile IDs and a set of flags. The flags describe which of two object palettes the sprite uses, whether the sprite is flipped horizontally or vertically, and lastly whether the sprite should be drawn in front of or "behind" the background. Drawing a sprite "behind" the background results in background and window pixels with colour indices 1-3 being drawn in front of the sprite, though the sprite is still drawn in front of pixels with colour index 0 [44].

2.3.2 The modes

At any given time, the PPU is in one of four modes: horizontal blanking, vertical blanking, OAM search or drawing [44].

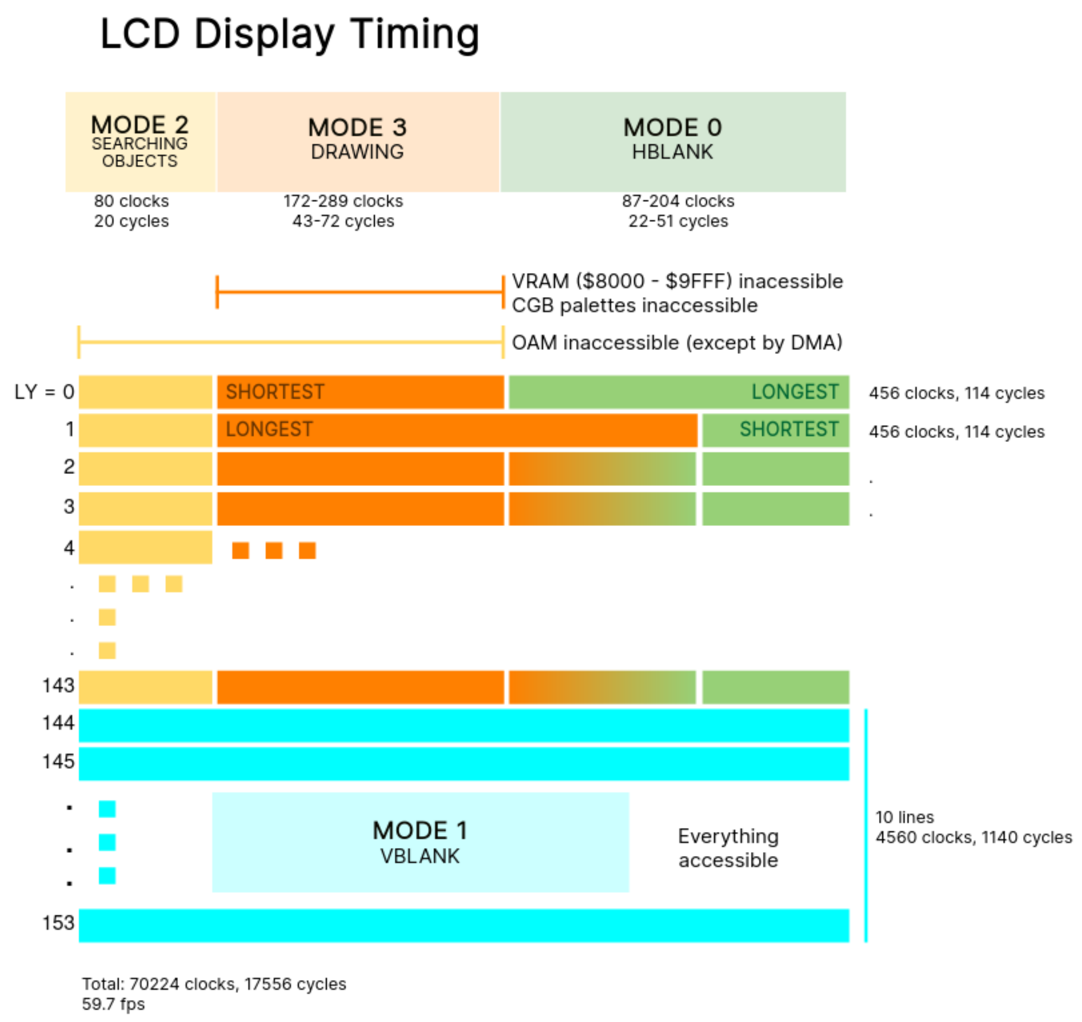


Figure 2.6: The various PPU modes and their durations. From [45]. Public Domain.

Depending on which mode the PPU is in, there are limits on whether the CPU has direct access to VRAM and OAM. When the CPU does not have access to one of these regions, any write to that region will be ignored and any read will return 0xFF. The PPU changes modes according to Figure 2.6 [44]. The four modes will now be described in further detail.

Horizontal blanking

The PPU enters a horizontal blank (H-blank) whenever it has finished drawing a line. During this period, the CPU can access VRAM and OAM freely. Depending on how long the PPU was in mode 3 before entering this mode, the length of the H-blank is adjusted so that the total time spent on OAM search, drawing and H-blanking for each line is exactly 114 cycles [44].

Vertical blanking
The PPU enters a vertical blank (V-blank) whenever all 144 scanlines have been drawn to the screen. The V-blank is treated as ten blank lines and therefore lasts for 1140 cycles. During this time the CPU has free access to VRAM and OAM [44].

OAM search
The PPU enters this mode before drawing each line. When in this mode, the PPU finds the first ten sprites in OAM that intersect the current scanline to prepare for the draw phase. During this time the CPU can access VRAM freely [44].

Drawing
The PPU enters this mode after having finished OAM search. During this mode the PPU combines the background, window and sprites of the current row and draws the pixels to the LCD. The process is described in more detail in Section 2.3.4. During this time the CPU can access neither VRAM nor OAM [44].

2.3.3 The registers

The PPU contains a variety of hardware registers. These are described in this section, also noting whether they are used for reading or writing to.

LCD Control Register (R/W)
The LCDC register is the main way for the CPU to control the operation of the PPU. It can be modified at any time and contains the following bits [46]:

- Bit 7 - LCD enable. If this bit is 0, the display is off.
- Bit 6 - Window tile map select. This bit determines which map should be used for the window layer.
- Bit 5 - Window enable. This bit determines whether the window layer should be drawn or not.
- Bit 4 - Tile set select. This bit determines which tile set should be used by the tile maps.
- Bit 3 - Background tile map select. This bit determines which map should be used for the background layer.
- Bit 2 - Object size. This bit determines whether sprites consist of one or two tiles.
- Bit 1 - Object enable. This bit determines whether sprites should be drawn or not.
- Bit 0 - Background and window enable. This bit determines whether the background and window layers should be drawn or not.

LCD Status and Configuration Register (R/W)
The STAT register represents the status of the PPU. It can also be used to create custom interrupts, depending on the status of the PPU. The STAT register contains the following bits [47]:

- Bit 6 - LYC interrupt enable. Enables interrupts when the current scanline equals the value of the LYC register.
- Bit 5 - OAM search interrupt enable. Enables interrupts when the PPU enters OAM search.
- Bit 4 - V-blank interrupt enable. Enables interrupts when the PPU enters a V-blank.
- Bit 3 - H-blank interrupt enable. Enables interrupts when the PPU enters an H-blank.
- Bit 2 - LYC=LY flag. Is set to 1 if the current scanline equals the value of the LYC register.
- Bit 1-0 - Mode flag. Represents the current mode of the PPU.

The PPU also has a variety of utility registers, described here [44]:

- SCY and SCX (Scroll X & Y) - Control background scrolling. Represents the coordinate on the background map that should be displayed in the top left corner of the display. (R/W)
- LY (Line Y) - Keeps track of the current scanline. (R)
- LYC (Line Y Compare) - Used in the STAT register to generate interrupts. (R/W)
- WY and WX (Window X & Y) - Represents the coordinate on the display where the top left pixel of the window should be displayed. (R/W)
- BGP (Background Palette) - The palette for the background. Maps the indices 0-3 to white (0), light grey (1), dark grey (2) or black (3). (R/W)
- OBP0 (Object Palette 0) - The first of two sprite palettes. Functions similarly to BGP except colour index 0 represents transparent pixels. (R/W)
- OBP1 (Object Palette 0) - The second of two sprite palettes. Functions exactly like OBP0. (R/W)
- DMA (Direct Memory Access) - Used for triggering a DMA transfer. More information can be found in Section 2.3.5. (R/W)

2.3.4 Drawing the image

During mode 3 the PPU continuously transfers pixels to the LCD. This is done by loading tile data into two FIFO queues, one row of pixels at a time. One queue stores background and window pixels and the other queue stores sprite pixels. The pixels are then popped from the queues and mixed depending on whether the sprite or the background should have priority [48]. This is also where the colour indices are mapped to colours using the palette tables. Depending on how many sprites are on the current scanline and whether the background is scrolled or not, the duration of mode 3 may be lengthened or shortened [48]. This results in mode 3 lasting between 43-72 cycles, as shown in Figure 2.6.

2.3.5 Direct Memory Access transfer

As mentioned previously, the CPU can access OAM freely during H-blanks and V-blanks. There is however another way to update the OAM: through Direct Memory Access transfers (DMA transfers). A DMA transfer is triggered by writing a value between 0x00 and 0xDF to the DMA register [49]. This value is bit-shifted to the left eight times and is then used as the start address for the transfer, where the PPU copies the $40 \times 4 = 160$ following bytes into the OAM. During the transfer, which takes 160 cycles, the CPU can only access HRAM and no sprites are displayed. This process can be used no matter which mode the PPU is in, though there may be visual bugs if a transfer is started when the PPU is in mode 3 [49].

2.4 The Audio Processing Unit

The role of the Audio Processing Unit (APU) is to play audio. The Game Boy can play sound through four separate channels, each of which is an independently controllable source of sound [50]. Each individual channel has a designated waveform assigned to it with each waveform defining what type of sound the respective channel plays. A waveform can be visually represented through a graph that shows change of amplitude over time. These waveforms sound drastically different to each other, which makes for a highly customisable sound experience even with only four channels available. The first two channels are only able to play pulse waves (see Figure 2.9), the third channel has a programmable waveform, which enables some customisability for the game developer, and the fourth channel plays a pseudo-random waveform, producing noise [51]. For further information about how waveforms and sound in general works, see [52] and [53].

2.4.1 The registers

The APU has a large number of registers for controlling the audio output for each channel. Settings such as volume and frequency can be changed by writing to the registers of each channel. The layout of these registers can be seen in Figure 2.7. The labels for each bit in Figure 2.7 corresponds to a specific functionality described in Table 2.4.

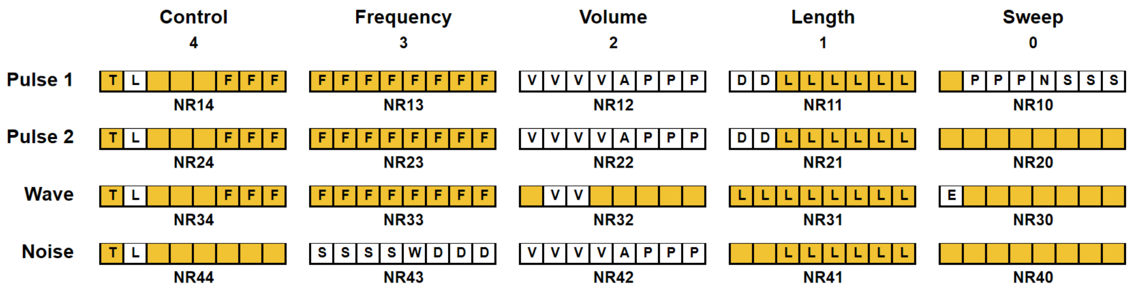


Figure 2.7: The registers NR10-NR44, addresses 0xFF10-0xFF23, for controlling audio channels. Bits coloured in yellow are write only and will always return 1 if read from [51]. From [30]. Adapted with permission.

There is also a set of APU master control registers which affects the four audio channels. These can turn the whole APU on or off, indicate what channel is currently playing, and control what side the sound should come from which results in stereo sound. The layout of these registers can be seen in Figure 2.8. The labels for each bit in Figure 2.8 corresponds to a specific functionality described in Table 2.5.

Label	Name	Functionality
T	Toggle	Turns the channel on or off
L (NRx4)	Length enable	If the channel should be turned off after a short period of time
F	Frequency	How long to play each audio sample
S, W, D (NR43)	Clock Shift LFSR mode Divisor code	
V, A, P (Not NR32)	Volume Add mode Period	Initial channel volume and how it should sweep up or down
V (NR32)	Volume code	Channel volume
D (NR11,NR21)	Duty	Sets pulse length, see Section 2.4.3
L (NRx1)	Length	Time until channel is turned off
P, N, S (NR10)	Sweep period Negate Shift	Sweeps channel frequency up or down to create sound effects
E	DAC power	Turns channel DAC on or off

Table 2.4: Description of the labels in Figure 2.7. From [51]. Adapted with permission.

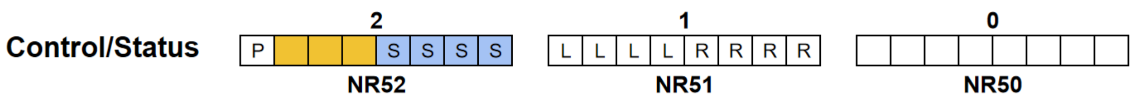


Figure 2.8: The registers NR52-NR50, addresses `0xFF26-0xFF24`, are used for controlling the power and returns the state of all channels. Bits coloured in yellow are write only and will always return 1 if read from. Bits coloured in blue are read only, any data written to this location will be ignored [51].

Label	Name	Functionality
P	Power	Enables or disables the entire APU
S	State	State of each channel, if 1, the channel is playing
L	Left enable	Enables or disables left and right side for each channel
R	Right enable	

Table 2.5: Description of the labels in Figure 2.8. From [51]. Adapted with permission.

2.4.2 Events

The registers described above trigger events at different frequencies. At what frequency these events should be triggered and exactly what happens at these events is controlled by the registers. Note that all values which are changed with these events are reset when the bit labelled with "T" is set [51]. The table below summarises the events which occur and at what frequency.

Event	Frequency [Hz]	Action
Play sample	`0x20000`/(`0x800` $- F$)	Play next audio sample
Play sample (Noise)	`0x80000`/$(D * 2^{(S+1)})$	Play next audio sample
Volume envelope	`0x40`/P	$V = \begin{cases} V-1 & \text{if A} = 0 \\ V+1 & \text{if A} = 1 \end{cases}$
Length sweep	`0x100`	Decrease L if length is enabled
Frequency sweep (Pulse 1)	`0x80`/P	$F = \begin{cases} F + F/2^S & \text{if N} = 0 \\ F - F/2^S & \text{if N} = 1 \end{cases}$

Table 2.6: F, D, S, V, P, L - The value of the bits labelled with "F", "D", "S", "V", "P", "L" respectively, for each channel.

2.4.3 The audio data

The pulse wave channels can play a pulse wave with four different waveforms. What waveform to play is specified by the "duty"-bits, i.e. the bits labelled with "D" in Figure 2.7. The resulting waveforms can be seen below in Figure 2.9.

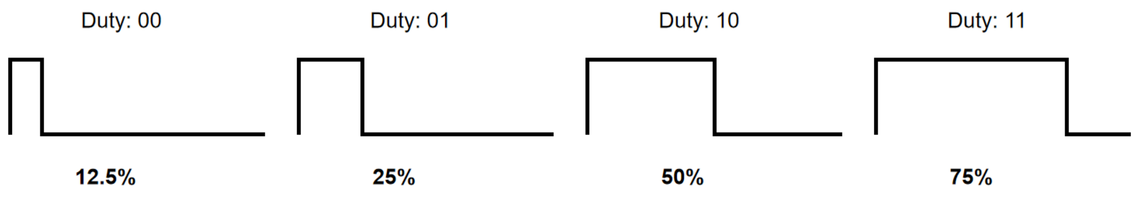
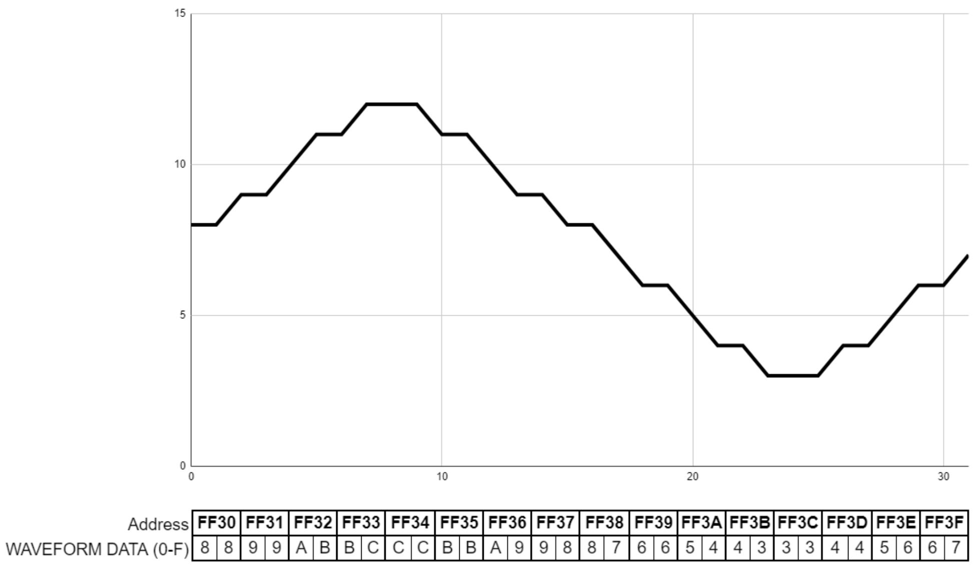


Figure 2.9: The four pulse waveforms with their corresponding duties [51]. The percentages indicate what amount of the wave is high. Duty "10" is most commonly used.

The waveform of the wave channel can be set to any shape you want by writing to the addresses `0xFF30` to `0xFF3F`. Each set of four bits represent one sample which means one waveform consists of 32 samples [51]. This approach enables the game developers to create something completely custom made to make their own unique sounds.



Address	FF30		FF31		FF32		FF33		FF34		FF35		FF36		FF37		FF38		FF39		FF3A		FF3B		FF3C		FF3D		FF3E		FF3F	
WAVEFORM DATA (0-F)	8	8	9	9	A	B	B	C	C	C	B	B	A	9	9	8	8	7	6	6	5	4	4	3	3	3	4	4	5	6	6	7

Figure 2.10: A rough representation of how a custom waveform is created through addresses `0xFF30` to `0xFF3F`.

The noise channel generates a pseudo-random waveform using a Linear-Feedback Shift Register (LFSR) [54]. The LFSR is 15 bits long and all bits are set to 1 in the beginning of the process. Bit 0 in the LFSR determines the output of each sample, if bit 0 is set to one, the output is low, otherwise it is high. To generate a new value of LFSR, the following is calculated where $LFSR(x)$ is the new value, $LFSR(x-1)$ is the old value, $LFSR(x-1)_n$ is the value of bit n in the old value, and W is the value of the bit labelled with "W" in Figure 2.7 [51].

$$LFSR(x) = \begin{cases} (LFSR(x-1)_0 \oplus LFSR(x-1)_1) * \texttt{0x4000} + LFSR(x-1)/2 & \text{if W} = 0 \\ (LFSR(x-1)_0 \oplus LFSR(x-1)_1) * \texttt{0x40} + LFSR(x-1)/2 & \text{if W} = 1 \end{cases}$$

3
Tools and methodology

This section describes the process of planning and implementing an emulator for the Game Boy, the tools used and why they were chosen.

3.1 Programming language

The language used in the project is C++ [55]. The language was chosen because it is fast, and supports the bit-level manipulation that is required when emulating hardware.

3.2 Graphics and GUI

For displaying graphics SDL2 [56] and OpenGL [57] were chosen, mainly due to them being appropriate when developing in C++, as well as the supervisor for this thesis being able to provide a base project already using these libraries which the emulator could be based on. In addition to this the team had some previous experience working with OpenGL.

When it came to providing a GUI enabling the user to interact with systems outside of the emulation itself, such as changing/displaying keybinds or loading a game, the library ImGui [58] was used.

3.3 Audio

For generating audio and playing sounds, OpenAL [59] was used. The choice to implement sound was made partway through the project and the choice then fell on OpenAL as it has a similar design as OpenGL, which the team had already acquired some experience with.

3.4 Scrum

When developing software, it is beneficial to use a flexible work process. When encountering problems or if the scope of the project changes, it should be possible to change direction without requiring the team to rewrite large parts of the project. In order to achieve this flexibility, the working process was formed around the agile development framework Scrum [60].

Whereas traditional Scrum perhaps has not been implemented in this project, the work process has been heavily influenced by Scrum. This is mainly due to the flexibility it allows while developing as well as tracking the work which has been done, and is to be done. This has specifically been done by implementing sprints and using a scrumboard for tracking tasks and progress.

3.5 Plan

The development of this emulator can roughly be split into three phases. The research, implementation and refinement phases, each being different from each other.

The main focus of the first phase was researching the Game Boy and finding relevant documentation as this is something which is not provided by Nintendo. By doing this, a basic plan and architecture could be produced. This also included a number of documents summarising information about the Game Boy.

Although a rough plan was sketched during this phase, providing both a plan for what to implement and in what order, this was always subject to change as the team had been implementing an agile workflow allowing for flexibility. Therefore the planning was allowed to be less strict than in other kinds of projects. The planning and documentation produced in this phase still yielded results which could be used as a basis for the second phase.

The second phase, implementation, heavily relying on the planning made in the previous phase, followed an agile workflow with sprints being one or sometimes two weeks long. This resulted in a flexible work flow allowing the team to continuously re-assess what to focus on each week while also allowing to document the progress on a weekly basis.

The final phase, refinement, allowed for extensive testing, re-factoring and also further development and bug fixing.

3.6 Architecture

During the first weeks, before the development of the actual emulator began, the initial plan was that the emulator would be divided into two larger modules. One module would contain all code related to the actual Game Boy emulation (Game Boy module) while the other would contain code related to the application part of the project such as keyboard input handling, graphics rendering, etc (Application module). For the Game Boy module, a basic design was developed where each of the well defined hardware units such as the CPU and PPU were decided to be further separated into modules. Additionally memory handling was decided to be represented by an aggregate unit resulting in a Memory Management Unit (MMU). For the Application module it was decided that a sort of Model-View-Controller architecture would be used. The final part of this design was an interface where the separate units of the Game Boy could work together and could bridge the gap between the Game Boy module and the Application module.

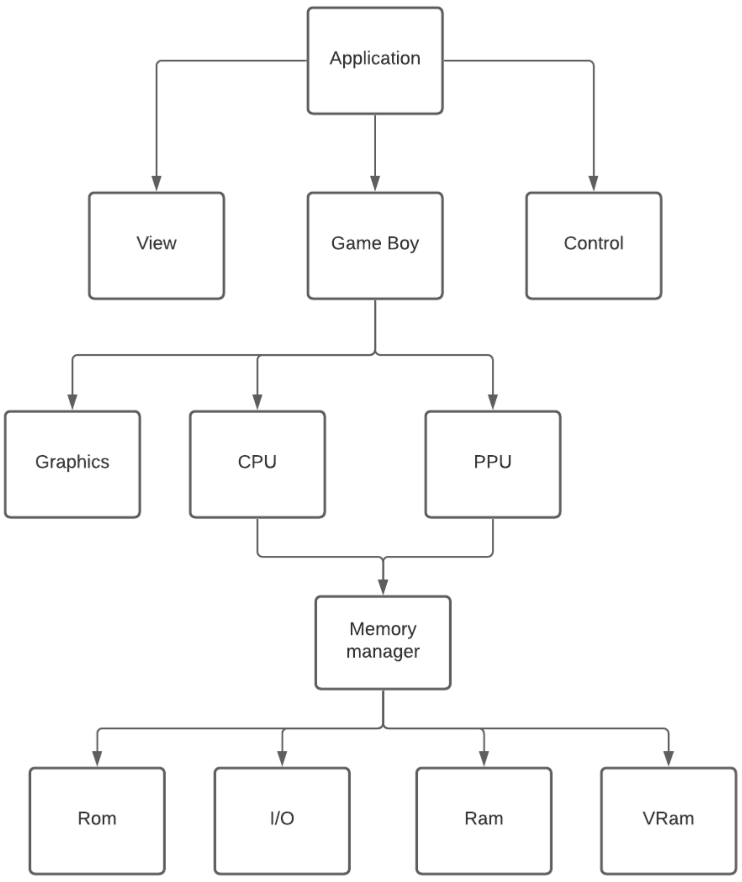


Figure 3.1: Initial sketch of how different features were to be separated into different modules.

3.7 Testing

Unit tests were one of the methods chosen to ensure that the emulator behaves as intended. For this, Google's testing framework Google Test was used [61]. Google Test was chosen as it supports unit tests for C++ and works well with the Continuous Integration (CI) tool Travis [62].

Nintendo have not released any official documentation of the hardware and the exact behaviour of the Game Boy is therefore partly unknown. Due to this, one can not be sure of the accuracy of an emulator, and if certain behaviour is correct or not without comparing the emulator to actual hardware. This of course makes development for the Game Boy more difficult. Fortunately, the Game Boy Wiki provides a table for a set of test ROMs [63] made by the emulator community member Blargg [64], containing the results of running the ROMs on a number of emulators as well as actual hardware. By comparing these results and the results produced when run on a specific emulator, one can in some ways confirm whether or not the emulator is behaving correctly. As the community provides a table with the results from Blargg's test ROMs on actual hardware, these were chosen as the first and main test ROMs to be used. In addition to this, some of the tests were deemed more central than others. Specifically the ROMs testing the instruction timing and instruction behaviour.

To streamline the testing process, most of the time spent in unit testing was done testing the basic operations executed by the CPU, MMU, PPU and in time, the APU. This was done to make sure that the most central parts of the emulator work correctly individually. Once this was done, the test ROMs came into use as these require a working CPU, MMU and PPU as they write the results of the tests directly onto the screen. These produce more extensive results and one might argue that they act as integration tests, testing multiple parts of the code and their interactions. In addition to this, testing was also done by playing a multitude of games, visually looking for bugs and otherwise strange behaviour.

4 Results

In this section the technical design and implementation of the theory is described.

4.1 Architecture and Design

The final design of the architecture follows the basics of what was laid out in the initial design with some changes. These changes mostly concern the Application module which was initially planned to follow the Model-View-Controller architecture. The main reason for this is that the chosen GUI library acts as both view and controller. The Application module of the emulator will not be further explored in this section, instead the Game Boy module will be looked at in more detail.

The Game Boy module is interacted with through the “GameBoy”-class which also has the task of synchronising every part of the Game Boy emulation (see below). As mentioned in Chapter 2, the different units communicate by reading from and writing to shared address space. This is reflected in the implementation as well as the diagram in Figure 4.1, which is a dependency diagram of the Game Boy module.

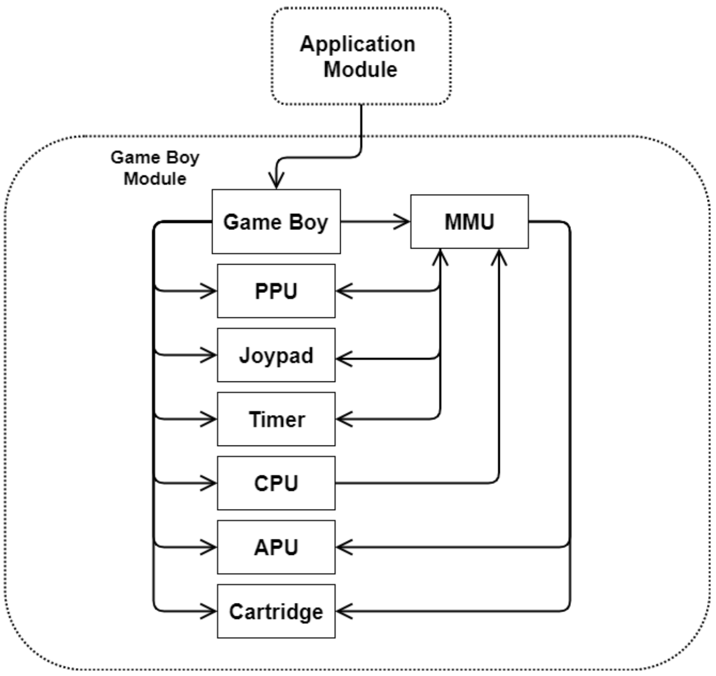


Figure 4.1: Dependency diagram over the Game Boy module.

Something which the original design knowingly did not take into consideration was the fact that the different hardware units run in parallel with different clock speeds. This choice was delayed as the research made showed that there were multiple possible solutions to synchronise the different units. Deciding which solution to implement therefore had to be done during development. The chosen solution was to make the CPU return the number of machine cycles after executing an instruction and to then allow the other units to catch up before executing another instruction, see Listing 4.1. The shown code snippet displays part of what could be considered the main loop of the emulation.

```
void GameBoy::step(IVolumeController *vc){
    ...
    int cycles = cpu->update();
    ppu->update(cycles);
    apu->update(cycles, vc);
    timer->update(cycles);
    cartridge->update(cycles);
    ....
    }
```

Listing 4.1: Code displaying how the CPU executes an instruction and returns the number of machine cycles, whereafter the other units catch up by executing the same number of cycles.

4.2 The Central Processing Unit

The implementation of the CPU mainly consists of translating the CPU instructions from assembly to a modern programming language, in this case, C++. The total number of instructions the CPU supports is 500 as mentioned previously, of which many are very similar to each other and all of which are implemented. Due to this, most of the instructions could be generalised into base functions, such as `add_8bit` seen in code in Listing 4.2. These base operations could in turn handle the variation of the instructions and instead of implementing one function for each operation code, the correct input simply needed to be provided to the correct base instruction given an operation code. There are some exceptions, amongst them the HALT-instruction [65] which has a very particular behaviour and therefore needed to have a separate implementation.

```cpp
void CPU::addA(uint8_t value, bool withCarry) {
    add_8bit(A, value, withCarry);
}

void CPU::add_8bit(uint8_t &a, uint8_t b, bool withCarry) {
    auto CFlag = withCarry ? F.c : 0;
    setCFlag(a, b + CFlag, false);
    setHFlag(a, b, false, CFlag);
    a += b + CFlag;
    //Note that all 'false' parameters specify that subtraction is
   not used, which in turn affects how and which flags are set.
    setZNFlags(a, false);
}
```

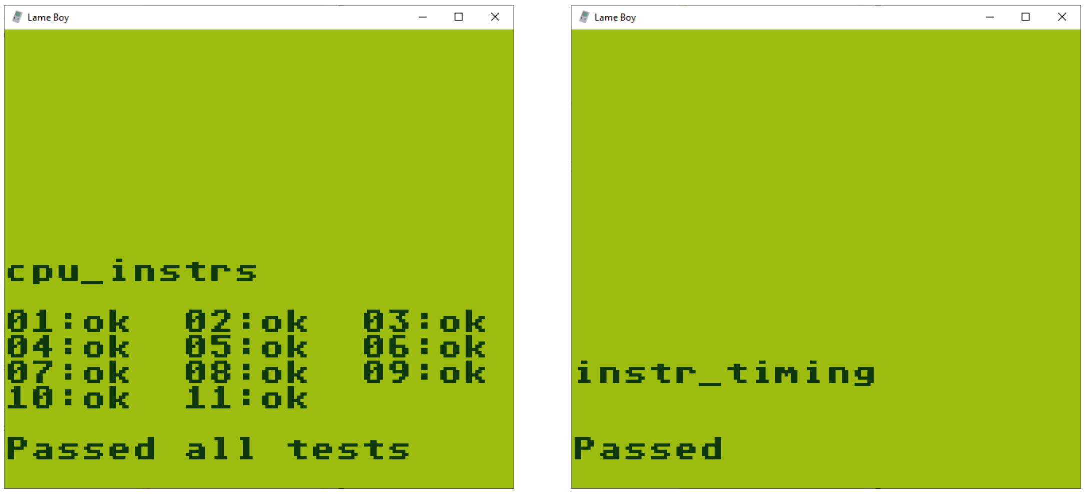
Listing 4.2: Code displaying a generalised method used, in this case an addition function which allows for addition between register A and all other registers, both with and without the use of the carry bit.

4.2.1 Interrupts

As previously stated, the CPU has five types of interrupts. Of these five only the “Serial interrupt” is not implemented, which there never was any intention of implementing as mentioned in Section 1.4. All the implemented interrupts are in some ways central in having a working CPU as these allow for other units to request the CPU to perform specific tasks when needed.

4.2.2 Testing of the Central Processing Unit

Mainly, three tests were used for checking the accuracy of the CPU. These check the correctness of the instructions [64], their timings and the interrupt timing respectively. The tests regarding the CPU-instructions, `cpu_instrs` and `instr_timing`, pass on both the real hardware [63] and the emulator see Figure 4.2. The `interrupt_time` test, however, does not pass despite being implemented and performing as expected. It most likely fails due to the implementation of the synchronisation of the different units. This is further discussed in Section 5.1.1.



```
cpu_instrs

01:ok  02:ok  03:ok
04:ok  05:ok  06:ok
07:ok  08:ok  09:ok
10:ok  11:ok

Passed all tests
```

```
instr_timing

Passed
```

Figure 4.2: Displaying the emulator results from Blargg's test ROMs checking the correctness of the CPU instructions and CPU timing respectively.

4.3 The Memory Managing Unit

The Game Boy uses memory mapped I/O which means that almost all communication can be done through reading and writing to the different addresses. Therefore the main purpose of the MMU is to provide functionality which supports reading memory and writing to memory.

To separate code and improve the possibility for parallel development all the different devices are separated into their own classes: PPU, APU, Cartridge, Joypad and Timer, each having their own read and write functions. The memory addresses used by the device (mostly control registers) are also maintained in the devices' class. The other memory areas which are not part of a device are located directly in the MMU class. Examples of such memory are: boot ROM, VRAM, WRAM, OAM, and HRAM.

4.3.1 Timer

The timer is implemented as a device with read and write functions. Reading and writing to the timer's four registers is implemented to behave as described in the Section 2.2. To make the timer update at the same pace as the CPU (and the system as a whole) it has an update function. The update function allows the timer to catch up to the system based on the number of cycles required for the last CPU instruction executed, see Section 4.1. Because of the timer not being updated continuously it is not always accurate. On original hardware the counter would increase during the execution of a CPU instruction and could therefore change between the start of the execution and the actual read from the counter register.

4.3.2 Cartridges and Memory Bank Controllers

Due to the fact that there are a number of different game cartridges, all supporting different hardware, such as different MBCs, RTC, ROM and XRAM sizes, a modular approach was taken. This was possible as although the cartridges may have different hardware they operate in a similar manner, therefore an MBC interface class was made. This enables different behaviour when reading or writing to the cartridge's addresses depending on the used MBC. The MBC interface also uses an update function to allow the possibility of an RTC to update at the same pace as the rest of the system. This function updates the RTC according to the specified number of clock cycles, see Section 4.1. For MBCs without an RTC this function does not do anything. This way multiple MBC classes can be implemented to easily expand the support for different MBCs. The most common MBCs are MBC1 and MBC3, which are also the ones which have been implemented.

The information about hardware used in each game is stored in the game ROM file. When loading the ROM file into memory the MBC type, ROM size and XRAM size can be obtained and initialised accordingly. Another feature for games with XRAM supported with a battery is the ability to save data between different sessions. A common way to support this in emulators, this one included, is to create a separate file to save the current contents of the XRAM to. Meaning that when loading a ROM file, the emulator also looks for an XRAM-file, which if found is also loaded and initialised, providing a way to use a games' built in save functionality.

4.3.3 Testing of the Memory Management Unit

Testing of the implemented MBCs has been done using test ROMs that test different features of the MBC such as ROM and XRAM bank switching as well as the RTC. The MBC tests used are Gekkio's mooneye-gb tests [66] and aaaaaa123456789's rtc3test [67], which the emulator pass.

Testing the timer was done with some test ROMs from Gekkio's mooneye-gb tests [66]. Most of them did not pass, most likely due to the inaccuracies caused by the update function. Although, its basic counting capability has been tested through games that for example use the counter to produce randomness. Without the counter the supposed randomness did not produce different outcomes, but with the timer implemented the outcome seemed more random.

4.4 The Pixel Processing Unit

The PPU class aims to replicate the functionality of the PPU in the Game Boy. In this section, the term “PPU” will refer to the PPU class and the term “original PPU” will refer to the Game Boy’s PPU.

The PPU contains the same registers as the original PPU and implements read and write methods that the CPU uses to communicate with the PPU.

4.4.1 Timing

As mentioned in Section 4.1, the PPU is controlled by a different clock than the CPU and need to be kept in sync. To solve this problem, the PPU is updated every time a CPU instruction is executed. By providing the number of cycles the CPU instruction used to the update-method of the PPU, it is possible to determine how much time has passed since it last switched mode and thereby decide whether the PPU should switch mode or not. For simplicity, the PPU also does all the real “work” of a mode whenever it switches from that mode.

4.4.2 Basic operation

The basic operation of the PPU is described by the update-method. Firstly, the PPU adds the number of cycles elapsed since the last CPU instruction was executed to the number of accumulated cycles since the PPU last switched modes, as seen in Section 4.1. It then checks whether or not it should switch mode. This calculation is based on the durations in Table 4.1; That is, the draw mode has been set to take the fewest amount of cycles that it can possibly take in the real Game Boy and the H-blank has been set to take as much time as is possible. When this is done, the PPU checks whether a STAT-interrupt should be requested and if so requests one.

Mode	**Cycles**
H-blank	51
V-blank	144
OAM-search	20
Drawing	43

Table 4.1: The durations of the PPU’s modes

What happens when the PPU changes mode depends on which mode it was just in. This is very similar to the basic operation described in Section 2.3.2 but there are a few differences, the main ones being that the image is rendered one line at a time and that the CPU has free access to VRAM and OAM at all times. The operation during the various modes is described below.

H-blank
When leaving an H-blank, the PPU moves on to the next scanline. The LY register is therefore increased by one. If all lines have been drawn, the PPU moves to V-blank mode and a V-blank-interrupt is requested. If not, the PPU goes into OAM-search mode.

V-blank
The V-blank in the emulator is treated as ten blank scanlines, since the V-blank in the Game Boy lasts for ten lines. Therefore, the LY register is increased by one every time the PPU is done with a blank scanline. If all ten blank scanlines have passed, LY is reset to 0 and the PPU goes into OAM-search.

OAM-search
When leaving OAM-search the PPU prepares the sprites to be drawn on the current line by adding the ones that intersect with the line to a priority queue. This ensures that when retrieving the sprites from the queue, the ones with highest priority will be drawn above those with lower priority. The PPU then goes into drawing mode.

Drawing
When leaving the drawing mode, the PPU processes the next line by calling the `processNextLine` method and then goes into H-blank mode. The drawing process is described in detail in Section 4.4.3.

4.4.3 Drawing

Unlike the original PPU, which draws pixels directly to the LCD, the PPU instead inserts pixels into a frame buffer. Scanlines are drawn one at a time, but the layers background, window and sprites are drawn in that order on top of each other, depending on which of these layers are enabled in the LCDC register.

Background
For a given scanline, for each x-coordinate, the PPU determines the tile corresponding to the coordinate, then which pixel of the tile should be drawn and lastly which colour it should have. The colour is written to the frame buffer and the colour index is saved in an array containing the background and window colour indexes for the current line. This array is later used for determining sprite priority over the background.

Window
For a given scanline, the PPU first determines whether the window covers this scanline. If it does, the window pixels are written to the frame buffer in a similar manner to how the background is written. Any background pixels covered by the window are overwritten.

Sprites
For each sprite found during OAM search, in order of ascending priority, the PPU draws the correct tile. If a pixel is transparent, the PPU simply moves on to the next pixel. If a sprite is to be drawn below the background, the array containing background and window indexes is used to determine the correct pixel.

4.4.4 DMA transfer

The DMA transfer in the emulator functions exactly the same as the one in the original Game Boy described in Section 2.3.5, with the exception that the process does not consume any cycles. That is to say, the entire DMA transfer takes place before the CPU is allowed to execute its next instruction.

4.4.5 Testing of the Pixel Processing Unit

The correctness of this module was mainly tested through visual observations. Unit tests were written to ensure correct behaviour of the PPU with various settings turned on or off in the LCDC register. Examples are correct scrolling, using different tile maps and tile sets and ensuring that the window covers the background.

Testing was also done by running game ROMs and comparing them to gameplay on the original Game Boy as well as other emulators.

4.5 The Audio Processing Unit

Since the APU is composed of multiple audio channels, features and controls, it was decided that the first square wave channel, without the frequency sweep functionality, should be implemented first. After getting this first audio channel working, the other audio channels were implemented iteratively along with more complicated features such as volume and frequency sweep.

4.5.1 The Audio Controller

An audio controller class was created whose responsibility is to play sound using the OpenAL library. Due to how the OpenAL interface works, this class cannot play one sample at a time, but instead relies on being fed an array of samples which should be played at a specific frequency for a specific amount of time. In order to play a sound for any length of time, only the samples of one waveform is provided which is then looped until stopped. Since the pulse wave and programmable wave channels already have a predefined waveform, these waveforms were simply fed to the OpenAL library. The waveform of the noise channel on the other hand is pseudo-random, and therefore has to provide more than one waveform to OpenAL. Thankfully, it was discovered from testing that the $LFSR(x)$ function, which is used to generate pseudo-random noise specified in Section 2.4.3, loops at $x =$ `0x7F` when $W = 1$ and at $x =$ `0x7FFF` when $W = 0$. This means only two arrays need to be generated, one of size `0x7F` and one of size `0x7FFF` containing the pseudo-random noise, which can be provided to OpenAL while also instructing OpenAL to loop the sound, resulting in an infinite noise sound.

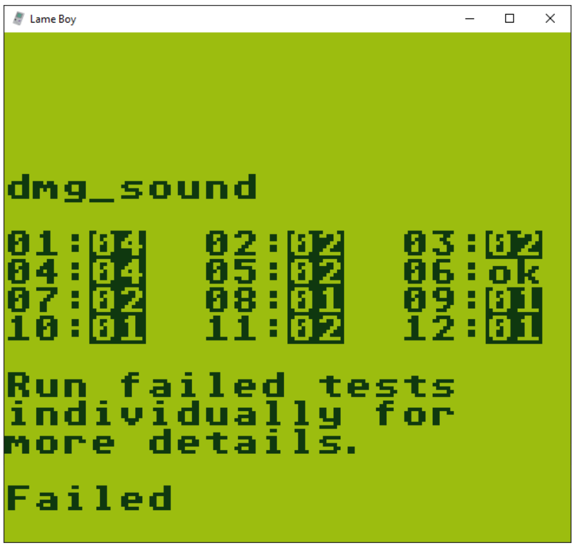
Due to the interface of OpenAL, the frequency of a sound cannot be changed while playing. Changing the frequency therefore requires the sound to be stopped before being played again at the new frequency. The effects of this are noticeable when a game uses the frequency sweep register since there are fast consecutive switches between frequencies. Instead of sounding like dragging your finger across a piano, it sounds like dragging your thumb across the teeth of a comb.

4.5.2 Testing of the Audio Processing Unit

When testing the APU, games were played on the emulator and the sound generated was compared with sound from videos on YouTube where the same game was played. Aspects such as what tone was played, at what moment, for how long and at what volume was observed when testing each register of each audio channel. This resulted in the APU to be an approximation of the real APU where small technical details was ignored.

The more advanced features such as the frequency and volume sweep were difficult to replicate accurately. To test these features, the “Sound Test (PD) [a1]” [68] ROM was run which allows the user to set specific parameters in the APU registers and play the specific sound channels. This ROM was also run on existing emulators, like the Visual Boy Advanced [69], in order to compare the results.

Similarly to when testing the CPU, Blargg’s test ROMs [64] were also used to test the APU. In particular, the `dmg_sound` test ROM was run in order to test the sound of the emulator. However, since many of the small technical details of the APU were ignored, not many of the tests passed, as seen in Figure 4.3.



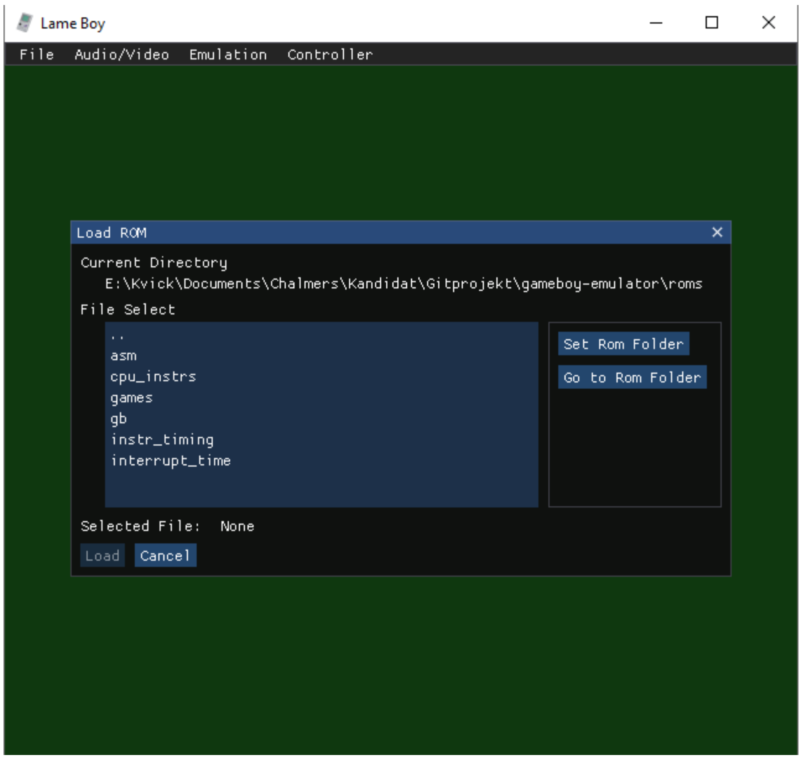
```
dmg_sound

01:04  02:02  03:02
04:04  05:02  06:ok
07:02  08:01  09:01
10:01  11:02  12:01

Run failed tests
individually for
more details.

Failed
```

Figure 4.3: Results of running the `dmg_sound` test ROM from Blargg’s test ROMs [64].

4.6 The Game Boy

After having developed multiple microcontrollers, they were combined into a complete system as described in Section 4.1. This system is the public interface of the hardware units and could be considered the actual emulation of the Game Boy. In combining these microcontrollers an emulator is created which in its current state can run multiple games without issue and in many ways provide an authentic experience. The emulator does however display some irregularities and unwanted behaviour in certain games, both mechanical and visual.

In code, this is represented by a separate "GameBoy"-class which combines all of the developed microcontrollers. It is through this the emulation is run and through this, all information is funnelled to external libraries such as OpenGL, ImGui and OpenAL.



```
Current Directory
   E:\Kvick\Documents\Chalmers\Kandidat\Gitprojekt\gameboy-emulator\roms
File Select
   ..
   asm
   cpu_instrs
   games
   gb
   instr_timing
   interrupt_time
Selected File:  None
```

Figure 4.4: A screenshot of the emulator when loading ROMs.

5 Discussion

This chapter discusses the results of the project. If there were any specific problems or choices in implementation which was made, and why.

5.1 Hardware

There is very little, if any, official documentation available regarding the specifics of the hardware used in the Game Boy. While this complicates emulator development, there are members of the community who have dedicated a lot of resources into providing accurate documentation on the hardware. Among these are two of the co-authors of PanDocs [45], Gekkio [70] and Antonio Niño Díaz [71]. Both of which are also authors of their own very detailed documentations of the Game Boy, the "Game Boy: Complete Technical Reference" [72] and "The Cycle-Accurate Game Boy Docs" [35] respectively. These two documents are based on tests written and ran on multiple units of the real hardware to ensure that they are testing correct behaviour, and thereby providing a way to reverse engineer the hardware. These sources, or sources similar to these, have therefore been central in the development of this emulator.

5.1.1 The Central Processing Unit

While implementing the CPU, a problem regarding the number of machine cycles an operation required was encountered. The problem was that the source used for the operation codes [33], which displays flags affected, number of bytes used and number of machine cycles, was not the same as used in Blargg's test ROMs [64]. As the test ROMs have been run on the real hardware and passed the tests, the team considered the test ROMs to be using the correct amount of cycles [63]. A process similar to what Gekkio and Díaz applies in their testing.

Looking at Blargg's test suite regarding the CPU tests, one might note that the emulator does in fact not pass all tests. Among the tests which it does not pass are the `interrupt_time`, `mem_timing` and `mem_timing-2` [64]. As mentioned in Section 4.1, the different units need to be synchronised and act in accordance with each other. There are indications that the reason for the tests failing is that it is possible for the CPU to execute too many machine cycles before allowing the other units to catch up or interrupts to intercept. Meaning that the current implementation of the CPU does not allow for timer updates mid-instruction. In hindsight, it is possible that the problems encountered as a result of this choice could have been avoided. One possible solution could be to allow the other units to catch up to the CPU mid-instruction every time it reads from or writes to memory.

5.1.2 The Memory Management Unit

Early in the development all memory and registers were located directly in the MMU class. When first implementing the PPU it always needed to "fetch" the registers it needed from the MMU to continue to do its job. This made it clear that it would be better to make the different device classes keep their own registers. With this change it was necessary but also natural to make use of individual read and write functions for each device, which in turn would be called from the main read/write function located in the MMU class.

In the memory map, seen in Figure 2.2, there are some areas which are empty. Although Nintendo says use of these areas are prohibited they still have a specific behaviour confirmed on all official hardware [73]. The seemingly empty range `0xE000-0xFDFF` is by the community called "Echo RAM" and is on the original hardware mapped to the actual RAM addresses. For example, writing a value to address `0xE001` would have the same effect as writing to the address `0xA001`. The same goes for reading values. We have chosen to not implement this behaviour mainly because it is prohibited by Nintendo and is expected to be unused in licensed games. Additionally, this behaviour does not add any functionality that cannot be achieved by other means. Furthermore, if it is used in unlicensed games it is easily implemented down the line.

The choice to only implement two different MBCs was made because the most popular games make use of these MBCs. Implementing more MBCs would yield less return for the time spent and time was therefore put into other areas that felt more important. Also, as previously mentioned, a modular approach was taken when implementing MBCs, making it easy to implement in the future if wanted.

Limiting access to different memory areas under certain circumstances has not been implemented, as this is expected to be respected by game developers. For example, during a DMA-transfer the CPU, on the original hardware, is limited to only access HRAM during that period.

5.1.3 The Pixel Processing Unit

As mentioned both in Sections 5.1.1 and 2.3, the different units are synchronised with the CPU by supplying the other units with the number of cycles the CPU has consumed after having completed each instruction. This made simulating the PPU accurately impossible and the approach of drawing one scanline at a time was instead chosen. The biggest limitation resulting from this choice is that the emulator does not support writing to PPU hardware registers mid-scanline. This results in graphical glitches in a few games, but is not a problem for a vast majority of games. However, the emulator supports writing to hardware registers between scanlines, enabling visual effects such as the one shown in Figure 5.1:

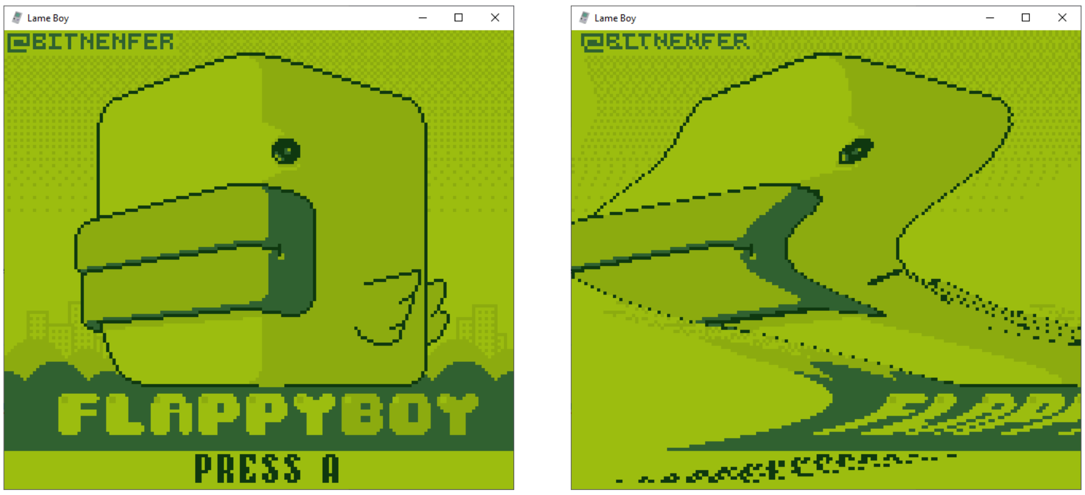


Figure 5.1: The starting screen of the game Flappyboy. From left to right, its original form and its distorted form achieved by writing to the SCX and SCY registers between scanlines. From [74]. Screenshot by authors, used with permission.

The decision to render the image scanline by scanline was made as it supported the aforementioned wobble effect and other visual effects, while still being achievable within the given time frame. To support writing to hardware registers mid-scanline would require a much more accurate simulation of the original PPU and was not feasible in the allotted time.

Another difference between the emulator and the original PPU is that the emulator does not limit the CPU's access to VRAM and OAM. This could lead to bugs in games where the developers have not ensured that the CPU has access to VRAM or OAM before reading or writing to these memory areas.

All in all, graphical glitches are few and far in between and the PPU must therefore be considered correctly implemented within the scope of the project.

5.1.4 The Audio Processing Unit

Implementing sound was initially considered out of scope, as it was deemed too complex and would take too much time to implement. Therefore any progress made in this regard is seen as exceeding the initial expectations. As the development of the other parts progressed faster than expected, there was time to spare which allowed for the implementation of the APU.

Only implementing one channel at a time was a good strategy as once the first channel had been implemented, it was easy to implement more. The biggest difference between the sound channels was how the audio was generated. Thankfully they could be implemented similarly without changing much but the audio data provided to the OpenAL API.

However, the devil is in the details, and for the APU there are a lot of them. There have been several issues where sound continues to play when it should have been muted, sound beginning to play from nowhere when it is not supposed to etc. Most of these issues have been resolved, but there are still existing issues with some games where the audio does not play at all.

The room for improvement is also apparent in the results of running Blargg's `dmg_sound` [64] test ROM which, as seen in Figure 4.3. Almost all tests fail due to these small details not being implemented. On the other hand, having sound implemented in any form and in some ways working as expected, is exceeding the initial expectations.

5.2 Accuracy

One thing to always consider when discussing the development of emulators is the accuracy of the emulation, and specifically how accurate the emulator aims to be. Some developers choose to try to emulate the exact behaviour of the hardware, including what today could be considered bugs, flaws, or mistakes. Others seize the opportunity and correct some behaviour of the emulator which they deem incorrect or faulty.

5.2.1 Compromises

Given the scope and the time spent on this project, this emulator had to be made with some compromises between accuracy and progress. Assuming the aim would have been to be as accurate as possible, a lot more research would have had to been done, and implementation could and would have had to be planned more carefully. On the other hand, many of the systems are implemented with the intention of making the emulator extensible and easy to further develop.

One of the compromises made was considering the choice of further developing the CPU, making it more accurate and passing more test ROMs, like Blargg's interrupt timing test [64], or attempting to implement sound. Adding sound to the emulator would increase the overall impression and feeling of the emulator, while there is no guarantee that making the CPU more accurate would increase the user experience in any way. One might even argue that implementing additional features such as saves, a functional GUI and other quality of life features yields a better overall impression than improving the accuracy of the emulation. Therefore the choice fell on trying to implement sound and other, smaller, features.

5.2.2 Vital and auxiliary components

Some components of the Game Boy are more important than others, which means these have to be more accurately emulated than auxiliary components. The most vital component to be emulated accurately is the CPU since it is the central component of both the Game Boy and the emulator. Any bugs caused by the CPU may result in obscure behaviours which can be close to impossible to debug. Getting Blargg's `cpu_instrs` and `instr_timing` [64] test ROMs working was therefore highly prioritised in order to validate the accuracy of the CPU and avoid any of these potential bugs.

The APU, which on the other hand could be considered an auxiliary component, was not given the same attention in regards to accuracy as the consequences of having a defective APU is not as severe. The worst thing that could happen with the APU is the absence, or constant presence, of sound, but all other parts of the emulator would still work as intended. This is because, in contrast to the CPU, no other component strictly relies on the APU to perform its function accurately. The results of this approach to accuracy is visible in Figure 4.3 which shows only test `06` passing when running Blargg's `dmg_sound` [64] APU test ROM.

5.3 Ethics

As mentioned in Section 1.5, the emulation of a proprietary product comes with both ethical and legal complications, some of which are discussed in this section.

5.3.1 Legality of emulators vs. ROMs

Most digital copyrighted material have some sort of controversy around them. Gaming consoles and their respective emulators are no different. Like with most subjects touching on potential piracy and copyright infringement, there is a huge grey zone whether or not something is legal or illegal, and subsequently if something is right or wrong.

5.3.2 The emulator

The creation of an emulator is generally not illegal as long as no proprietary code is used [75]. This is mainly because the emulator itself does not necessarily have to contain any proprietary code to replicate the original console. Some consoles do have proprietary code in their BIOS, however this can usually be avoided either by flashing a BIOS legally, or creating a custom BIOS.

5.3.3 The Boot ROM

As previously mentioned, the boot ROM initialises the Game Boy while scrolling down the Nintendo logotype, something which might seem like a banality at first. However, this means that the official boot ROM cannot be used in emulators as it uses both copyrighted code, and displays a copyrighted logo. Additionally, on the original hardware the Game Boy enforces that any game run on the Game Boy has to contain the Nintendo logotype [76]. This is done by comparing the logotype data stored on the boot ROM with data stored on the inserted game. If this data does not match, the Game Boy locks itself up. This allows Nintendo to control what is released to the platform, as any game released for the Game Boy has to contain copyrighted material to run on the official hardware, and therefore needs the approval of Nintendo [76].

5.3.4 The ROMs

Copyright laws differ from country to country, which complicates the matter further. In the United States (US), there are multiple cases in court regarding the creation of emulators and distribution of ROMs [77]. In the European Union (EU), however, it is more difficult to find actual court cases around emulation, which creates confusion for people without a legal education. The European Parliamentary Research Service has documentation which describes the copyright laws in the EU [78]. This documentation is on the other hand not as easily interpreted as pre-existing cases, of which there seem to be very few using EU law.

Due to the difficulty of interpreting the EU law as a layperson when discussing the legality of ROMs, the US law is usually referred to. Furthermore, according to US law the ROMs themselves are in a bit of a grey zone [79]. Making back ups of game cartridges one already owns is completely legal under the right circumstances and for the right purposes according to US copyright laws [80]. Selling or distributing said copies is illegal and counts as copyright infringement, and so does downloading other people's copies. The main legal way to get a playable version of an existing game's ROM seems to be through making your own copy of that specific game, which you must already own. Piracy and copyright laws do vary from country to country, and there is continuous debate online whether or not these laws are for the greater good or not [81]. Furthermore, it is not always clear which country's laws are to be followed, since the company who owns the copyright might not be based in the same country as the person using their content.

5.3.5 Test-ROMs

If game ROMs are not acquirable for some reason, there are a multitude of test-ROMs online that are available to make sure the emulator and its parts work as intended [82]. These are free to use and has benefited the troubleshooting of this project immensely without any risk for potential copyright infringement. There are also games developed by the community, i.e. non-proprietary ROMs, specifically for emulators which have been used for testing.

5.3.6 Sharing knowledge

By creating an emulator for an old console one gets to thoroughly understand how the very basic components of a gaming system work together to create a complete console. By making the emulator's source code available publicly, it might help increase the knowledge of these types of systems among students, enthusiasts and developers alike. As long as no proprietary code is shared, there is no obvious way this could harm anyone. If the Game Boy was not discontinued one could argue that an emulator would deter people from buying an expensive console. However, since the Game Boy is no longer being sold, that risk is effectively eliminated. On the contrary, finding an emulator might even contribute to further interest in Nintendo products which would most likely benefit them.

5.3.7 Gaming history preservation

One of the main pro-emulation arguments is that of historic preservation. The Game Boy was made with hardware that degrades over time, which limits each units lifetime. Ever since the product was discontinued in 2003 [83] people have been urging to save what many believes to be one of the greatest gaming products ever released. Much like museums would save objects from historical events, Game Boy emulators and ROM archiving could be seen as a kind of digital museum where the soon to be lost hardware is forever stored. Several games for the Game Boy are no longer being sold, and acquiring a used copy becomes increasingly more difficult with time. This makes retro game emulation for historic preservation quite an attractive option for a lot of people. However the legal situation around emulators is very much a grey zone, which makes justifying an emulator for this purpose much more difficult. Creating an emulator would definitely contribute to increased historic preservation, however through incorrect use it could also be a tool for playing illegal copies of games.

5.3.8 Console Classix

An example of a company which has tried to make old ROMs playable to the public is Console Classix [79]. The idea is to rent their games through a client-server solution to make the games playable on a home PC. Although they have received a letter from Nintendo regarding their business [84], no legal action has been taken since. Console Classix's defence is that they, unlike illegal ROM sharing websites, do not publish the ROMs publicly but rather provide limited access to them with a subscription method. By sending the ROM images directly from the server to the client's RAM, the game effectively disappears from the client when they stop playing, which would prevent any permanent distributing from happening. Additionally, they do not rent more copies out than they own, therefore it is difficult to build a case around copyright infringement as well. This is a great example of using an emulator seemingly legally to still fill the purpose of historic preservation, while simultaneously providing a way to play retro games for those subscribed to their service.

5.4 Future Work

Going forward, there are numerous ways to improve the emulator. Initially, one could work on improving its accuracy, ensuring that more games are playable without any unintended behaviour, for example by continuing to work towards passing more test ROMs. Furthermore, there are many additional features regarding quality of life which could be implemented, such as quick saves, shortcuts for commands etc. One could even develop a debugger/disassembler for the emulator or modularise the code further in an effort to emulate multiple systems within one emulator.

Furthermore most emulators are created for one of three purposes: they either aim to provide ways to play old games, a historical documentation of a product which is at risk to be lost to time or as a form of learning experience. Most of which results in a code base which is more or less cluttered, either due to inexperience in the field, that the code bases are very big, or due to the fact that they aim to reproduce exact behaviour of hardware. Additionally, due to the fact that there is no official documentation from Nintendo regarding the Game Boy, most emulators will be dependent on people such as Gekkio [70] or Díaz [35] as most people do not have the technical competence or resources available to reverse engineer the hardware. This results in the available emulators being quite homogeneous, where most emulators are built using the same source material. Most will probably even use the same tests to check the emulator's accuracy compared to the real hardware. As a developer of an emulator it is therefore highly unlikely that you bring something new to the table.

One might therefore ask oneself if there is any academic interest in such a product. One could then consider the fact that video games are extremely popular among the general population. This is something which could be leveraged into piquing an interest for low-level programming and hardware through the use of emulator development. This could, for instance, be done by providing an emulator which has a clear structure, is effectively modularised and commented. The purpose would not be to make the most accurate or the fastest emulator, but rather to provide a clear point of reference for the interaction between hardware and software. This emulator could then work as an inspiration for other potential developers, creating a base for further learning.

6
Conclusions

Emulating an older system, such as the Game Boy, can be done to some accuracy with limited resources, both in time and available knowledge, as shown by this thesis. Despite not perfectly replicating the system to be emulated, a Game Boy emulator can be created which runs many games and provides the functionality expected from an actual Game Boy, thereby both providing a more or less authentic experience as well as a way to preserve games. It does however mean that the emulator can exhibit strange behaviour, freeze, or even crash when unexpected problems regarding the emulation are encountered and can therefore not be considered stable. If better, more detailed, and comprehensible documentation could be provided, better emulators could also be produced. One could even consider that there is something of a finish line for emulator development, where if you reproduce the exact behaviour of the hardware, the topic is exhausted. If this would be achieved, one could look at other purposes of emulator development such as using it as a tool for education. This could for example be done by creating a simple, well-documented emulator with a clear structure showing how the translation from hardware to software is done, leveraging the interest for gaming in the general population to work as an inspiration to learn more about low-level programming.

Bibliography

Note that between the first and second edition of this thesis, the website of Pan Docs hosted by the Game Boy Development community at *gbdev.io* was updated and all links used as references therefore stopped working. These are now referred to using *The Wayback Machine* [85] and therefore contain a prefix of *web.archive* specifying a date preceding the changes which broke the initial references.

[1] C. Woodcock, *Original nintendo game boy*, Public Domain Pictures. [Online]. Available: `https://www.publicdomainpictures.net/en/view-image.php?image=191203&picture=original-nintendo-game-boy`, Accessed on: 2021-05-28.

[2] Techopedia, *Emulator*, Techopedia. [Online]. Available: `https://www.techopedia.com/definition/4788/emulator`, Accessed on: 2021-02-12.

[3] R. E. W. III, *What is an emulator?* Jan. 2021. [Online]. Available: `https://www.lifewire.com/what-is-an-emulator-4687005`, Accessed on: 2021-02-12.

[4] D. Johnson and W. Antonelli, "Emulators can turn your pc into a mac, let you play games from any era, and more — here's what you should know about the potential benefits and risks of using one," *Business Insider*, Oct. 2020. [Online]. Available: `https://www.businessinsider.com/what-is-an-emulator?r=US&IR=T`, Accessed on: 2021-03-30.

[5] K. Smith-Wong, *Why the nintendo game boy was so successful*, Jul. 2015. [Online]. Available: `https://www.denofgeek.com/games/why-the-nintendo-game-boy-was-so-successful/`, Accessed on: 2021-02-12.

[6] G. Sheppard, "The chronology of game boy models," *GameGrin*, Apr. 2019. [Online]. Available: `https://www.gamegrin.com/articles/the-chronology-of-game-boy-models/`, Accessed on: 2021-03-30.

[7] *Nintendo switch*, Nintendo. [Online]. Available: `https://www.nintendo.com/switch/system/`, Accessed on: 2021-05-28.

[8] *Nintendo 3ds-familjen*, Nintendo. [Online]. Available: `https://www.nintendo.se/nintendo-3ds-familjen`, Accessed on: 2021-05-28.

[9] *Game boy family*, Wikipedia, Mar. 2021. [Online]. Available: `https://en.wikipedia.org/wiki/Game_Boy_family`, Accessed on: 2021-04-01.

[10] Nintendo, *Consolidated sales transition by region*, 2016. [Online]. Available: `https://www.nintendo.co.jp/ir/library/historical_data/pdf/consolidated_sales_e1603.pdf`, Accessed on: 2021-03-25.

[11] *Official home of kirby*, Nintendo. [Online]. Available: https://kirby.nintendo.com/, Accessed on: 2021-05-28.

[12] *Pokemon.com*, Nintendo. [Online]. Available: https://www.pokemon.com/us/, Accessed on: 2021-05-28.

[13] M. Minotti, *25 years of the game boy: A timeline of the systems, accessories, and games*, Apr. 2014. [Online]. Available: https://venturebeat.com/2014/04/21/25-years-of-the-game-boy-a-timeline-of-the-systems-accessories-and-games/, Accessed on: 2021-03-31.

[14] *Tetris*, Nintendo Wiki. [Online]. Available: https://nintendo.fandom.com/wiki/Tetris, Accessed on: 2021-05-31.

[15] *History of console emulators*, Emulation Nation. [Online]. Available: http://www.emulationnation.com/console-emulation/history-of-console-emulators/, Accessed on: 2021-04-25.

[16] *What are the top youth media brands in 1980s, 1990s, 2000s and current?* Wonder, May 2017. [Online]. Available: https://askwonder.com/research/top-youth-media-brands-1980s-1990s-2000s-current-keja4oioq, Accessed on: 2021-04-25.

[17] *History of emulation*, Game Tech Wiki, Jul. 2020. [Online]. Available: https://emulation.gametechwiki.com/index.php/History_of_emulation, Accessed on: 2021-02-12.

[18] Z. Domain, *Nintendo - gameboy emulators*. [Online]. Available: https://www.zophar.net/gb.html, Accessed on: 2021-02-12.

[19] R. Copetti, *Game boy architecture - a practical analysis*, 2021. [Online]. Available: https://www.copetti.org/writings/consoles/game-boy/, Accessed on: 2021-03-30.

[20] *Serial communication (link cable) tutorial*, Game Boy Development Wiki, Nov. 2014. [Online]. Available: https://gbdev.gg8.se/wiki/articles/Serial_Communication_(Link_Cable)_Tutorial, Accessed on: 2021-04-19.

[21] *Game pak*, Nintendo Fandom Wiki, Mar. 2021. [Online]. Available: https://nintendo.fandom.com/wiki/Game_Pak, Accessed on: 2021-04-08.

[22] A. Niño Díaz, A. Vivace, Beannaich, C. Sandlin, E. Habert, Elizafox, Furrtek, Gekkio, J. Frohwein, J. Harrison, L. Halphon, Mantidactyle, M. Fayzullin, M. Korth, P. of ATX, P. Felber, P. Robson, T4g1, TechFalcon, endrift, exezin, jrra, kOOPa, mattcurrie, nitro2k01, pinobatch, and P. Fagan, *Pan docs, cartridge type*, GBDev, Apr. 2021. [Online]. Available: http://web.archive.org/web/20210426154117if_/https://gbdev.io/pandocs/#_0147-cartridge-type, Accessed on: 2021-04-26.

[23] ——, *Pan docs, game boy technical reference*, GBDev, Apr. 2021. [Online]. Available: http://web.archive.org/web/20210426154117if_/https://gbdev.io/pandocs/#sound-control-registers, Accessed on: 2021-04-26.

[24] N. Australia, *Legal*, 2021. [Online]. Available: https://www.nintendo.com.au/legal, Accessed on: 2021-03-31.

[25] A. Niño Díaz, A. Vivace, Beannaich, C. Sandlin, E. Habert, Elizafox, Furrtek, Gekkio, J. Frohwein, J. Harrison, L. Halphon, Mantidactyle, M. Fayzullin, M. Korth, P. of ATX, P. Felber, P. Robson, T4g1, TechFalcon, endrift, exezin,

jrra, kOOPa, mattcurrie, nitro2k01, pinobatch, and P. Fagan, *Pan docs, game boy technical reference*, GBDev, Apr. 2021. [Online]. Available: `http://web.archive.org/web/20210426154117if_/https://gbdev.io/pandocs/#specifications`, Accessed on: 2021-04-26.

[26] T. C. for Computing History, *Nintendo game boy*, The Centre for Computing History, 2021. [Online]. Available: `http://www.computinghistory.org.uk/det/4033/Nintendo-Game-Boy/`, Accessed on: 2021-02-12.

[27] Zilog, *Z80 cpu user manual*, Aug. 2016. [Online]. Available: `http://www.zilog.com/docs/z80/um0080.pdf`, Accessed on: 2021-03-31.

[28] Intel, *Intel 8080 microcomputer systems user's manual*, NJ7P Amateur Radio Web Server, Sep. 1975. [Online]. Available: `http://www.nj7p.info/Manuals/PDFs/Intel/9800153B.pdf`, Accessed on: 2021-03-31.

[29] M. Steil, *The ultimate game boy talk*, [Video] Youtube, Oct. 2016. [Online]. Available: `https://youtu.be/HyzD8pNlpwI?t=948`, Accessed on: 2021-04-22.

[30] M. Steil, *The ultimate game boy talk [slides]*, Apr. 2019. [Online]. Available: `https://www.pagetable.com/?p=1099`, Accessed on: 2021-04-22.

[31] A. Niño Díaz, A. Vivace, Beannaich, C. Sandlin, E. Habert, Elizafox, Furrtek, Gekkio, J. Frohwein, J. Harrison, L. Halphon, Mantidactyle, M. Fayzullin, M. Korth, P. of ATX, P. Felber, P. Robson, T4g1, TechFalcon, endrift, exezin, jrra, kOOPa, mattcurrie, nitro2k01, pinobatch, and P. Fagan, *Pan docs, game boy technical reference*, GBDev, Apr. 2021. [Online]. Available: `http://web.archive.org/web/20210426154117if_/https://gbdev.io/pandocs/#memory-map`, Accessed on: 2021-04-26.

[32] ——, *Pan docs, game boy technical reference*, GBDev, Apr. 2021. [Online]. Available: `http://web.archive.org/web/20210426154117if_/https://gbdev.io/pandocs/#registers-and-flags`, Accessed on: 2021-04-26.

[33] M. Sullivan, *Game boy cpu instructions*, Megane Sullivan personal website, Nov. 2020. [Online]. Available: `https://meganesulli.com/generate-gb-opcodes/`, Accessed on: 2021-03-30.

[34] A. Niño Díaz, A. Vivace, Beannaich, C. Sandlin, E. Habert, Elizafox, Furrtek, Gekkio, J. Frohwein, J. Harrison, L. Halphon, Mantidactyle, M. Fayzullin, M. Korth, P. of ATX, P. Felber, P. Robson, T4g1, TechFalcon, endrift, exezin, jrra, kOOPa, mattcurrie, nitro2k01, pinobatch, and P. Fagan, *Pan docs, game boy technical reference*, GBDev, Apr. 2021. [Online]. Available: `http://web.archive.org/web/20210426154117if_/https://gbdev.io/pandocs/#interrupts`, Accessed on: 2021-04-26.

[35] *The cycle-accurate game boy docs*, GitHub, May 2020. [Online]. Available: `https://github.com/AntonioND/giibiiadvance/tree/master/docs`, Accessed on: 2021-04-12.

[36] *Gameboy bootstrap rom*, Gameboy Development Wiki. [Online]. Available: `https://gbdev.gg8.se/wiki/articles/Gameboy_Bootstrap_ROM`, Accessed on: 2021-04-06.

[37] Techopedia, *Bootstrap*, Techopedia. [Online]. Available: `https://www.techopedia.com/definition/3328/bootstrap`, Accessed on: 2021-04-08.

[38] A. Niño Díaz, A. Vivace, Beannaich, C. Sandlin, E. Habert, Elizafox, Furrtek, Gekkio, J. Frohwein, J. Harrison, L. Halphon, Mantidactyle, M. Fayzullin, M.

Korth, P. of ATX, P. Felber, P. Robson, T4g1, TechFalcon, endrift, exezin, jrra, kOOPa, mattcurrie, nitro2k01, pinobatch, and P. Fagan, *Pan docs, game boy technical reference*, GBDev, Apr. 2021. [Online]. Available: `http://web.archive.org/web/20210426154117if_/https://gbdev.io/pandocs/#external-memory-and-hardware`, Accessed on: 2021-04-26.

[39] ——, *Pan docs, game boy technical reference*, GBDev, Apr. 2021. [Online]. Available: `http://web.archive.org/web/20210426154117if_/https://gbdev.io/pandocs/#vram-sprite-attribute-table-oam`, Accessed on: 2021-04-26.

[40] ——, *Pan docs, game boy technical reference*, GBDev, Apr. 2021. [Online]. Available: `http://web.archive.org/web/20210426154117if_/https://gbdev.io/pandocs/#joypad-input`, Accessed on: 2021-04-26.

[41] ——, *Pan docs, game boy technical reference*, GBDev, Apr. 2021. [Online]. Available: `http://web.archive.org/web/20210426154117if_/https://gbdev.io/pandocs/#timer-and-divider-registers`, Accessed on: 2021-04-26.

[42] *Memory bank controllers*, Game Boy Development Wiki, Apr. 2021. [Online]. Available: `https://gbdev.gg8.se/wiki/articles/Memory_Bank_Controllers#MBC3_.28max_2MByte_ROM_and.2For_64KByte_RAM_and_Timer.29`, Accessed on: 2021-04-16.

[43] ——, *Pan docs, game boy technical reference*, GBDev, Apr. 2021. [Online]. Available: `http://web.archive.org/web/20210426154117if_/https://gbdev.io/pandocs/#memory-bank-controllers`, Accessed on: 2021-04-26.

[44] ——, *Pan docs, game boy technical reference*, GBDev, Apr. 2021. [Online]. Available: `http://web.archive.org/web/20210426154117if_/https://gbdev.io/pandocs/#video-display`, Accessed on: 2021-04-26.

[45] ——, *Pan docs, game boy technical reference*, GBDev, Apr. 2021. [Online]. Available: `http://web.archive.org/web/20210426154117if_/https://gbdev.io/pandocs/`, Accessed on: 2021-04-26.

[46] ——, *Pan docs, game boy technical reference*, GBDev, Apr. 2021. [Online]. Available: `http://web.archive.org/web/20210426154117if_/https://gbdev.io/pandocs/#lcd-control`, Accessed on: 2021-04-26.

[47] ——, *Pan docs, game boy technical reference*, GBDev, Apr. 2021. [Online]. Available: `http://web.archive.org/web/20210426154117if_/https://gbdev.io/pandocs/#lcd-status-register`, Accessed on: 2021-04-26.

[48] ——, *Pan docs, game boy technical reference*, GBDev, Apr. 2021. [Online]. Available: `http://web.archive.org/web/20210426154117if_/https://gbdev.io/pandocs/#mode-3-operation`, Accessed on: 2021-04-26.

[49] ——, *Pan docs, game boy technical reference*, GBDev, Apr. 2021. [Online]. Available: `http://web.archive.org/web/20210426154117if_/https://gbdev.io/pandocs/#lcd-oam-dma-transfers`, Accessed on: 2021-04-26.

[50] ——, *Pan docs, game boy technical reference*, GBDev, Apr. 2021. [Online]. Available: `http://web.archive.org/web/20210426154117if_/https://gbdev.io/pandocs/#sound-controller`, Accessed on: 2021-04-26.

[51] *Gameboy sound hardware*, Gameboy Development Wiki, Nov. 2020. [Online]. Available: `https://gbdev.gg8.se/wiki/articles/Gameboy_sound_hardware`, Accessed on: 2021-03-30.

[52] *What are waveforms and how do they work?* SoundBridge, Apr. 2019. [Online]. Available: `https://soundbridge.io/what-are-waveforms-how-they-work/`, Accessed on: 2021-04-22.

[53] S. Deuty, *Llc power conversion explained, part 2: Sine wave from a square wave*, Planet Analog, Aug. 2017. [Online]. Available: `https://www.planetanalog.com/llc-power-conversion-explained-part-2-sine-wave-from-a-square-wave/`, Accessed on: 2021-04-22.

[54] *Gameboy sound hardware: Noise channel*, Gameboy Development Wiki. [Online]. Available: `https://gbdev.gg8.se/wiki/articles/Gameboy_sound_hardware#Noise_Channel`, Accessed on: 2021-05-24.

[55] *C++*, 2021. [Online]. Available: `https://www.cplusplus.com/`, Accessed on: 2021-04-13.

[56] SDL2, *Simple directmedia layer.* [Online]. Available: `https://www.libsdl.org/index.php`, Accessed on: 2021-03-25.

[57] K. Group, *Opengl*, 2021. [Online]. Available: `https://www.opengl.org/`, Accessed on: 2021-02-11.

[58] O. Cornut, *Dear imgui: Bloat-free graphical user interface for c++ with minimal dependencies*, Github, Apr. 2021. [Online]. Available: `https://github.com/ocornut/imgui`, Accessed on: 2021-04-07.

[59] OpenAL, *Openal*, Nov. 2018. [Online]. Available: `https://openal.org/`, Accessed on: 2021-04-06.

[60] K. Schwaber, *What is scrum?* Scrum.org, Nov. 2020. [Online]. Available: `https://www.scrum.org/resources/what-is-scrum/`, Accessed on: 2021-02-10.

[61] Google, *Google test*, GitHub, Apr. 2021. [Online]. Available: `https://github.com/google/googletest`, Accessed on: 2021-04-07.

[62] Travis, *Travis*, Apr. 2021. [Online]. Available: `https://www.travis-ci.com/`, Accessed on: 2021-04-07.

[63] *Test roms*, Game Boy Development Wiki, Mar. 2020. [Online]. Available: `https://gbdev.gg8.se/wiki/articles/Test_ROMs`, Accessed on: 2021-03-25.

[64] Blargg, *Blargg's gameboy hardware test roms*, Game Boy Development Wiki, Jun. 2019. [Online]. Available: `https://gbdev.gg8.se/files/roms/blargg-gb-tests/`, Accessed on: 2021-03-25.

[65] A. N. Díaz, *The cycle-accurate game boy docs*, Github, May 2020. [Online]. Available: `https://github.com/AntonioND/giibiiadvance/blob/master/docs/TCAGBD.pdf`, Accessed on: 29 Mar 2021.

[66] Gekkio, *Mooneye-gb*, Github, Mar. 2021. [Online]. Available: `https://github.com/Gekkio/mooneye-gb`, Accessed on: 2021-03-30.

[67] aaaaaa123456789, *Rtc3test*, Github, Apr. 2021. [Online]. Available: `https://github.com/aaaaaa123456789/rtc3test`, Accessed on: 2021-04-21.

[68] *Sound test (pd) [a1]*, ROMNation.NET. [Online]. Available: `https://www.romnation.net/srv/roms/14091/gb/Sound-Test-PD-a1.html`, Accessed on: 2021-04-21.

[69] R. K. et al, *Visual boy advance - m.* [Online]. Available: `https://github.com/visualboyadvance-m/visualboyadvance-m`, Accessed on: 2021-04-23.

[70] Gekkio, *Gekkio.fi*, 2021. [Online]. Available: `https://gekkio.fi/`, Accessed on: 2021-03-31.

[71] *Antonio niño díaz*, GitHub, Apr. 2021. [Online]. Available: `https://github.com/AntonioND/`, Accessed on: 2021-04-12.

[72] ——, *Game boy: Complete technical reference*, Nov. 2020. [Online]. Available: `https://gekkio.fi/files/gb-docs/gbctr.pdf`, Accessed on: 2021-02-9.

[73] A. Niño Díaz, A. Vivace, Beannaich, C. Sandlin, E. Habert, Elizafox, Furrtek, Gekkio, J. Frohwein, J. Harrison, L. Halphon, Mantidactyle, M. Fayzullin, M. Korth, P. of ATX, P. Felber, P. Robson, T4g1, TechFalcon, endrift, exezin, jrra, kOOPa, mattcurrie, nitro2k01, pinobatch, and P. Fagan, *Pan docs, game boy technical reference*, GBDev, Apr. 2021. [Online]. Available: `http://web.archive.org/web/20210426154117if_/https://gbdev.io/pandocs/#echo-ram`, Accessed on: 2021-04-26.

[74] bitnenfer and M. Fisher, *Flappy-boy-asm*, GitHub, May 2018. [Online]. Available: `https://github.com/bitnenfer/flappy-boy-asm`, Accessed on: 2021-04-22.

[75] J. Pot, *Is downloading retro video game roms ever legal?* How-To Geek. [Online]. Available: `https://www.howtogeek.com/262758/is-downloading-retro-video-game-roms-ever-legal/`, Accessed on: 2021-02-12.

[76] M. Steil, *The ultimate game boy talk*, [Video] Youtube, Oct. 2016. [Online]. Available: `https://youtu.be/HyzD8pNlpwI?t=1157`, Accessed on: 2021-04-19.

[77] B. Farrand, *Emulation is the most sincere form of flattery: - retro videogames, rom distribution and copyright*, IDP. Revista d'Internet, Dret i Política. [Online]. Available: `https://www.raco.cat/index.php/IDP/article/view/260390/347561`, Accessed on: 2021-05-10.

[78] C. L. L. Unit, *Copyright law in the eu.* [Online]. Available: `https://www.europarl.europa.eu/RegData/etudes/STUD/2018/625126/EPRS_STU(2018)625126_EN.pdf`, Accessed on: 2021-05-10.

[79] S. Harding, *Yes, downloading nintendo roms is illegal (even if you own the game)*, Tom's Hardware. [Online]. Available: `https://www.tomshardware.com/news/why-most-roms-are-illegal,37512.html`, Accessed on: 2021-02-12.

[80] L. I. Institute, *17 u.s. code § 117 - limitations on exclusive rights: Computer programs*, Cornell Law School. [Online]. Available: `https://www.law.cornell.edu/uscode/text/17/117`, Accessed on: 2021-02-12.

[81] G. Dave, *Are roms and emulation bad? | digitally distracted ep 18*, Youtube. [Online]. Available: `https://www.youtube.com/watch?v=i-3tppY6DwQ`, Accessed on: 2021-02-12.

[82] J. Javanainen, *Game boy test rom do's and don'ts.* [Online]. Available: `https://gekkio.fi/blog/2016/game-boy-test-rom-dos-and-donts/`, Accessed on: 2021-02-12.

[83] N. Wiki, *2003.* [Online]. Available: `https://nintendo.fandom.com/wiki/2003`, Accessed on: 2021-02-12.

[84] C. Classix, *Letter from noa.* [Online]. Available: http://www.consoleclassix.com/letter_from_noa.php, Accessed on: 2021-02-12.

[85] *Wayback machine*, Internet Archive. [Online]. Available: http://web.archive.org/, Accessed on: 2021-05-31.