

Emmy: The Gameboy Color Emulator

6CCS3PRJ Final Project

Classic video game console emulation

Author: Oscar Sjöstedt
Student ID: K20040078
Supervisor: Ian Kenny

Draft Two
March 23, 2023
King's College London

I verify that I am the sole author of this report,
except where explicitly stated to the contrary.

Oscar Sjöstedt, March 23, 2023

Abstract

This project aims to create a Gameboy emulator web-application, in other words a program capable of receiving Gameboy game files (commonly referred to as ROMs), and interpreting such ROM to play the game (or execute the program) it contains. The emulator will be usable in browsers, for both desktop computers and mobile devices that may not have access to a physical keyboard. The emulator will also contain debugging capacities, to allow other emulator developers to use it when comparing with their emulator and working on it.

The objective of this project is to create a piece of software that could be used by anyone wanting to emulate retro games, without the need for any technical knowledge on emulators or downloading anything (except the ROMs that need to be obtained separately).

Contents

1	Introduction	4
1.1	Motivation	4
1.2	Scope	4
1.3	Objectives	4
2	Background	6
2.1	Emulation	6
2.2	Video Game Emulation	6
2.3	Gameboy, Gameboy Color	7
2.4	Existing Literature	7
2.4.1	Gameboy Documentation	7
2.4.2	Existing Emulators	8
2.4.3	Gameboy Test ROMs	8
3	Requirements and Specification	10
3.1	Requirements	10
3.1.1	User Requirements	10
3.1.2	System Requirements	10
3.1.3	Non-Functional Requirements	11
3.2	Specification	11

4	Design	13
4.1	CPU	14
4.2	PPU, APU, Joypad	15
4.3	Memory Bus	16
4.4	Other Components	16
4.4.1	Timer	16
4.4.2	Interrupts	16
4.4.3	MBC	16
4.5	Useful Classes	16
4.5.1	Addressable	16
4.5.2	Memory and registers	17
4.6	Emulator Frontend	17
5	Implementation	18
5.1	Emulator Frontend	19
5.2	Emulator Backend	20
5.2.1	Addressable	20
5.2.2	System	20
5.2.3	CPU	25
5.2.4	Timer	27
5.2.5	PPU	27
5.2.6	OAM and OAM DMA	27
5.2.7	APU	27
5.2.8	MBCs and ROMs	27
6	Evaluation	28
	Acronyms	29

Chapter 1

Introduction

1.1 Motivation

Emulators are an area of computer science widely used today. Either implemented in hardware or software, they allow replicating the behaviour of one system on another. One of its applications is video game emulation, where a computer simulates a game console (usually a retro console). This allows users to play games that either may not be obtainable in stores anymore, or made for consoles that don't function properly anymore. A wide range of emulators already exist for most consoles. Emulation in general is also widely used in developing new systems, and is an active area of computer science.

This project will seek to create a new emulator for the Gameboy allowing users to play retro games on their computer or mobile device, through the browser. This report will document how the original console works and how the emulator imitates this behaviour to the best accuracy possible, as well as comparing the resulting emulator with other existing ones.

1.2 Scope

The scope of this project is creating a new Gameboy and Gameboy Color emulator, working for browsers. Emulation will be as accurate as achievable with the time available - there may be minor inaccuracies in the end product. Extra peripherals and features of the console may be omitted, to allow more focus to be put on the core part of the console.

The emulator will be usable across a range of devices. Debugging tools and additional features may be provided to the user to let them customise their experience to their needs.

1.3 Objectives

The resulting software will allow users to open a game file for the gameboy - also called a ROM - and play it. They may use the emulator on a computer, controlling the console via the keyboard, or on a touch device, using on-screen buttons. The emulated features of the emulator include proper rendering of the screen, simulating the audio of the console, the different buttons, and support for a variety of chip controllers for game cartridges.

The frontend of the emulator may also contain additional quality of life features, such as custom

themes, save states, and debugging options allowing to inspect the state of the Gameboy - a feature vital to emulator developers and retro game developers.

Chapter 2

Background

2.1 Emulation

An emulator is “hardware or software that permits programs written for one computer to be run on another computer” [1]. The imitated computer is the *guest*, and the one that imitates is the *host*. Emulators are nowadays mainly found in the form of software, and have many different uses, from preservation to hardware development.

Emulation was technically born with the first computers: the very first computer, the Colossus made in 1941, was built to imitate the Enigma machine [2]. However emulation was properly studied towards the end of the twentieth century, when computing power started to steadily increase. One of the earliest instances of emulation as an actual feature is with the IBM System/360. This computer supported emulation of previous models, such as the IBM 709, 7090, 7094 and 7094 II [3].

Nowadays emulation is used when researching and creating new hardware - using virtual systems to design a system and look for errors is much more cost-effective [4]. The real system doesn’t have to actually be produced, and if something goes wrong it is very simple to look for the error in the simulated system, through logging or stepping through the simulation.

Emulation is also vital for preservation: as transistors and motherboards age, old systems become unusable, and with them the software they ran. Emulating these systems is often the only future-proof and sustainable way to keep this software usable.

Finally, another common use for emulation is virtual machines. These programs allow running another Operating System (OS) on a computer, which can be used for instance when developing for other systems, without needing to use the physical device directly, for instance when developing a Windows-compatible app with a Linux computer.

2.2 Video Game Emulation

Video game emulation is the art of emulation applied to video game hardware systems. This allows the host to run games destined for the original console. This usually requires precise understanding of the console’s hardware and functioning, as games may rely on specific behaviours and edge cases to function. This task is rendered harder by the fact that often game consoles are poorly documented, as they are proprietary hardware and programmer guides written for

them cannot be legally distributed.

Video game emulation started in the 90s when computers were powerful enough to properly simulate console systems. Although precise dates are hard to get, the first console emulators seem to be either from 1990 or 1993 [5], and were able to run some NES games. The first Gameboy emulators were in the late 90s, with the [Virtual GameBoy](#) in 1995 and [NO\\$GMB](#) in 1997 (although its [history page](#) seems to indicate development started in 1993) [6].

The original game files and assembled code for video games is copyrighted material, and is referred to as the Read-Only Memory (ROM) of the game. Although distributing these ROMs is usually illegal, there also exist copyright free ROMs: games created by developers that chose to license them under Creative Commons licenses, for instance. Websites such as [Retro Veteran](#) host wide collections of legal ROMs.

2.3 Gameboy, Gameboy Color

The Gameboy (GB) is an 8-bit handheld video game console, released in 1989. It has a small 160×144 pixel screen, and has a Sharp LR35902 as its Central Processing Unit (CPU), clocked at 4.19MHz [7, Specifications]. In 1998, the Gameboy Color (GBC) was then released. Seen as the successor of the GB, it contains a screen of the same resolution, but supporting colour, from a palette of 32768 options (15 bits per color). It contains the same CPU as its predecessor, a Sharp LR35902, with now two modes: a 4.19MHz mode and a 8.38MHz mode (double-speed mode). This allows the GBC to be backwards compatible with most GB games - there are a few exceptions to this, games that used hardware bugs of the original GB that were fixed in the GBC.

From an emulation perspective, the Gameboy Color can thus be seen as an extension of the Game Boy - it has an identical CPU (although with a toggle-able double speed mode), and most of the memory layout is identical. To keep the remaining of this document simple, if not stated, "GB" will refer to both the original Gameboy and the Gameboy Color, as they are very similar. Dot Matrix Game (DMG) refers to the original Gameboy model.

2.4 Existing Literature

2.4.1 Gameboy Documentation

The Gameboy is one of the best documented consoles for emulation, and an array of resources exist to get started writing a new one. Some useful resources explaining it's behaviour are:

- [Pandocs](#) is a technical reference of how the GB works. It is extremely complete and covers a wide range of topics, so it is useful to get a global view of a problem. It is one of the most referenced pieces of literature on the console.
- [GB CPU Instructions](#) is a table containing all instructions the GB CPU has, as well as information on the amount of cycles taken by the instruction, the bytes of memory used, the flags affected by the operation, and a description of the instruction.
- [Gameboy Complete Technical Reference](#) (GBCTR) is an unfinished document that contains very detailed information on the CPU and other components of the GB. Although incomplete, it provides a much lower-level view of the details of the GB (compared to Pandocs), making it useful to emulate very specific behaviour like the cycle-by-cycle timing of the CPU.

- [GB dev wiki](#) is a wiki containing additional information on the GB, including guides to making games and explanations on some hardware quirks, and in particular a very precise description of the Audio Processing Unit (APU).

2.4.2 Existing Emulators

A wide range of emulators for the GB and GBC already exist, many of them being open-source. These are useful when developing a new emulator, to see how they work internally. For performance reasons they're usually written in compiled languages, such as C++ and Rust, but some interpreted language alternatives exist. These emulators include:

- [Game Boy Crust](#) is a simple GB emulator written in Rust. It is quite incomplete but has a comprehensive structure, so it's a good project to first figure out how emulators work.
- [AccurateBoy](#) is a highly accurate emulator, in particular for its Picture Processing Unit (PPU) that has pixel-perfect accuracy.
- [oxideboy](#) is another GB emulator written in Rust, that is much more complete and helpful for some edge cases.
- [SameBoy](#) is one of the most accurate open source GB and GBC emulators, written in C. It is much more technically complex but still useful to understand edge cases, especially since it is the emulator I use and compare mine with.
- [Mooneye GB](#) is a GB research emulator written in Rust. It passes most of the Mooneye test ROMs, making it helpful when encountering issues with these tests.
- [GameBoy-Online](#) is a high-accuracy JavaScript emulator, that I used when unsure on how to interface the emulator with the browser (notably for the APU).
- [Gameboy.js](#) is another JavaScript emulator. It is fairly simply and inaccurate, but is easily hackable. As such it's the emulator I used when starting the emulator, to compare mine with.
- [rboy](#) is an emulator written in Rust, that I used when developing the APU to compare mine with, as it passes some test ROMs I struggled with.

2.4.3 Gameboy Test ROMs

A core set of resources to develop an emulator is the test ROMs for that console. These are valid source code for the console, that instead of playing a game will run a set of tests on the console. These tests are first written to pass on the physical console itself, and are then used to ensure they also pass on the emulator. This means issues in specific components can be easily diagnosed (so long as the rest of the emulator responsible for running the test ROM works itself). These test ROMs also have the advantage of being open source, meaning their source code can be referred to, to understand what they expect of the console.

An other advantage to using test ROM is that they tend to re-use the same framework across a given test suite to report results. This means the result of the test is usually logged somewhere in the console, and testing can easily be automated by inspecting specific registers/memory addresses, rather than having to store an "expect result" image for each test.

The test ROMs used for this project are:

- [Blaarg test ROMs](#) are some of the most well-known and used GB test ROMs. They include tests for the CPU, the timings of instructions, and some other functionality.
- [Mooneye test ROMs](#) is a very complete test suite, that verifies most components of the GB: CPU instructions, memory timings of specific instructions, behaviour of Memory

Bank Controllers (MBCs), timing of the Object Attribute Memory (OAM) Direct Memory Access (DMA), PPU timings, timer timings, etc.

- Acid Test ([DMG](#), [GBC](#)) is a test that verifies the PPU of the GB displays data properly (to line-rendering accuracy), for both Gameboy and Gameboy Color displays.
- [SameSuite](#) is a test suite that is valuable for its APU tests: it uses the PCM12 and PCM34 registers exclusive to the GBC to inspect the exact output of the APU (whereas other test ROMs tend to inspect the on/off status of the channels, which is much less accurate).

Chapter 3

Requirements and Specification

3.1 Requirements

3.1.1 User Requirements

- U1. Run Gameboy games to a satisfiable fidelity, with proper rendering and controls emulation.
- U2. Run Gameboy Color games to a satisfiable fidelity, with proper rendering and controls emulation.
- U3. Allow the user to run GB and GBC games for both a Gameboy and a Gameboy Color (ie. allow using a GBC for a `.gbc` file, and a GB for a `.gb` file).
- U4. Allow the user to save the state of the game, to continue their playthrough later. The state can simply be saved as a downloaded file, and re-uploaded later to continue the game.
- U5. Allow the user to change the speed at which the game is played: double speed mode, half speed mode, etc.
- U6. Have some debug functionality, to inspect the state of the console at any given time.
- U7. Allow users to pause the console, and add breakpoints to stop execution at specific moments.
- U8. Allow the user to switch between rendering modes (nearest-neighbour, LCD display, scale2, etc.)
- U9. Allow the user to switch the colour palette of the DMG emulation.

3.1.2 System Requirements

- F1. The system can receive a ROM file, construct an instance of the emulated console, and run the code inside said ROM.
- F2. The system emulates different components of the GB and GBC, with as much precision as possible (M-Cycle precision).
- F3. The system renders the output of the emulator to a Web `<canvas />`.
- F4. The system creates the required DOM elements for the web-app, and updates them as needed.

F5. The system listens to key presses and releases to emulate controls through the keyboard.

F6. For touch devices, the system may render buttons to simulate the console's controls.

3.1.3 Non-Functional Requirements

N1. The emulator should be accessible on computers through a web browser equipped with a recent version of JavaScript.

N2. The emulator should be accessible on mobile devices through a web browser equipped with a recent version of JavaScript.

N3. The emulator should be accessible on computers through a standalone app.

N4. Maximise the tests passed by the emulator (see [Gameboy Test ROMs](#))

N5. Have the code be well documented, allowing new-comers to the project and to GB emulation to easily understand what is going on - if possible with links to relevant Gameboy emulation resources.

3.2 Specification

Code	Specification	Importance
U1	User can upload a GB ROM file (.gb), and the emulator will run the game. The keyboard can be used to control the game, and the output is displayed.	High
U2	User can upload a GBC ROM file (.gbc), and the emulator will run the game. The keyboard can be used to control the game, and the output is displayed.	Medium
U3	User can upload a GB ROM file (.gb). The user can switch between a DMG and a GBC emulator.	Medium
U4	User can press a button to download a save of their game (or, alternatively, the save can be stored inside the browser with a technology like IndexedDB)	Low
U5	User can select the speed of emulation, to dynamically accelerate/decelerate the game.	Medium
U6	User can see debug information of the emulator. This information includes the current tileset, background map, time to draw a frame, and register information.	Low
U7	User can pause the console emulation through a button. They can also input conditions for which the console should break execution.	Low
U8	User can dynamically switch the rendering filter via a dropdown button.	Low
U9	User can dynamically switch the colour palette of the GameBoy via a dropdown button.	Low
F1	A ROM file can be uploaded, is transformed into an <code>UInt8Array</code> (because the GB is an 8-bit system), and the appropriate object is created to run the code.	High
F2	Different components exists as different classes, respecting typical OOP principles such as encapsulation and inheritance when relevant.	High

Code	Specification	Importance
F3	A <code><canvas /></code> element is created, and is updated with the output of the emulator after every frame is drawn (ie. at the start of each VBlank mode).	High
F4	The Preact framework is used to handle the UI of the web-app.	High
F5	Listeners are added to the environment's <code>window</code> to listen to all key presses and releases. The emulator can then request for a control update, by reading the state of keys.	High
F6	If a touch device is detected, buttons are added to the UI and are used by the emulator as inputs.	Medium
N1	A deployed version of the web-app is accessible on a desktop browser and provides full functionality, via keyboard and mouse inputs.	High
N2	A deployed version of the web-app is accessible on a mobile device browser and provides full functionality, via touch controls.	Medium
N3	A downloadable version of the web-app can be used on a computer and provides full functionality, via keyboard and mouse inputs.	Low
N4	As many possible tests as possible should be passed, while ensuring previously passing tests don't start failing.	Medium
N5	Main methods and variables must be properly documented, and have links to appropriate online resources to documentation about said element.	High

Chapter 4

Design

In this chapter we will outline the main components of the Gameboy and of this emulator. For the different GB components, we will briefly go over their role, and how they interact with other components and to what end.

In emulators, the different components are usually split in three parts:

- The CPU, responsible for reading instructions and changing the state of the console. This is what drives the emulation, as other components usually idle unless acted upon.
- Input and output components, such as the APU, the PPU and the joypad. These components interact with the outside user, by either outputting the game state, or reading inputs from the user.
- A memory system, that handles addressing within the console. This part is essential to ensure components are communicating between each other properly, since different addresses may map to different components.

Because the emulator should not rely on the environment it is running it to work, the functionality of the project can be split into two parts: the emulator core, that is responsible for simulating the Gameboy, and the emulator's frontend, that allows interfacing with the user, and may be changed to work for different platforms (see [4.1](#)).

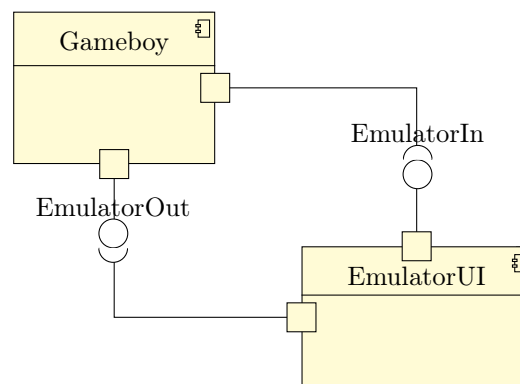


Figure 4.1: Split of emulator core and frontend

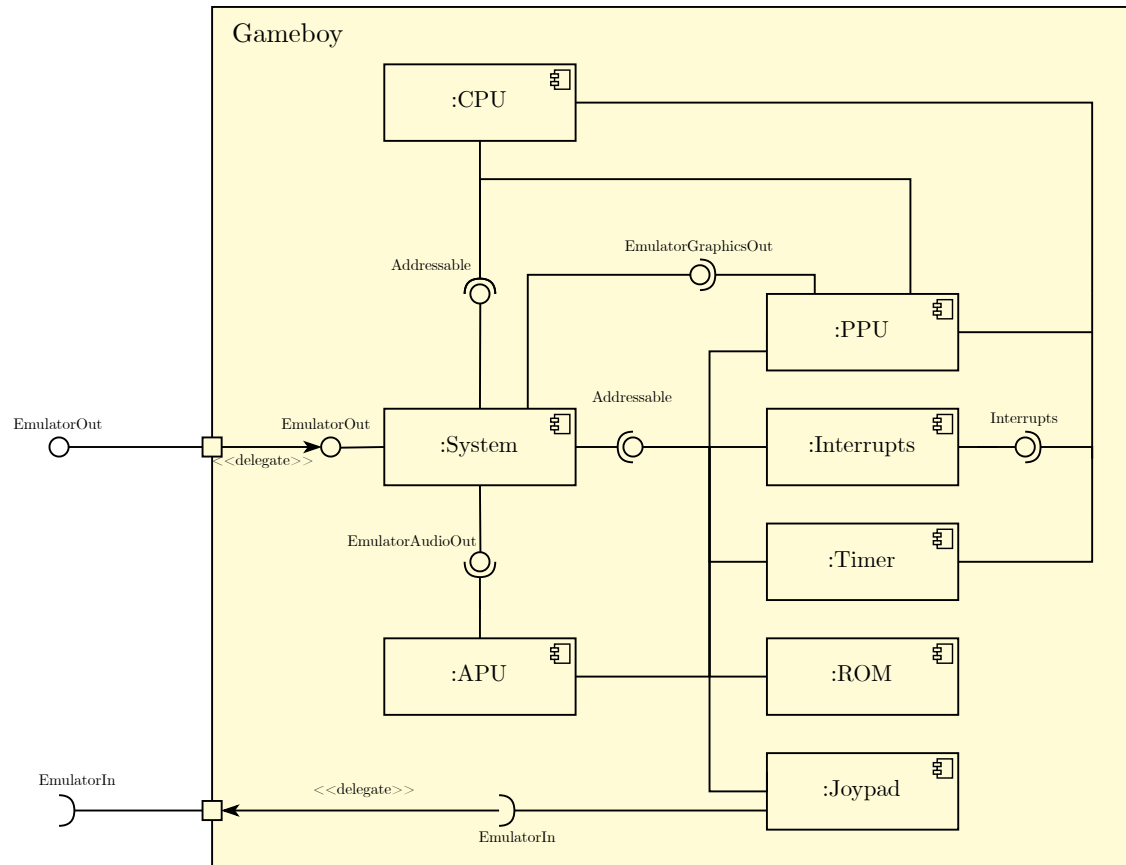


Figure 4.2: Simplified design of the emulator’s core

4.1 CPU

The Central Processing Unit (CPU) is perhaps the most complex part of the GB. It is however also one of best documented parts of it, making it quite easy to emulate properly. The CPU model is a LR35902, which is a hybrid of the Intel 8080 chip and the Zilog Z80 [gbcpumannual](#).

Most emulator are typically “CPU-driven”. This means that the CPU is what drives the emulation. If an instruction takes multiple hardware cycles to execute, the CPU is responsible for asking the rest of the system to tick at the appropriate time. In the design chosen for this project however, the emulator will be “System-driven”, with the CPU only being part of the overall running of the emulator, and will not be responsible for ticking the rest of the system. This difference will be made obvious by the interfaces exposed to the CPU; it allows for nicer encapsulation and better separation of concerns.

The CPU is thus responsible for stepping through the program it’s given. It has 6 different 16-bit registers, with different roles. Some registers’ bytes can be accessed independently, operating as two 8-bit registers or one 16-bit register as needed.

- **AF**: the accumulator-flag register. Most arithmetic operations can be done on **A**. **F** holds the CPU’s flags, and can only be altered through arithmetic operations.
- **PC**: the program counter register. It cannot be accessed directly, and is used to store the current position of the CPU in the program. It can be altered via **CALL**, **RET**, **JP** operations, for instance.
- **SP**: the stack pointer register. It can be modified or incremented, and stores the pointer

to the top of the “stack”. A typical use of the stack is for handling functions, by pushing the current address to the stack whenever a procedure is called, and popping it back to the stack pointer when it returns.

- BC, DE: two simple registers that can be used for arithmetic operations.
- HL: similar to BC and DE, it can also be used as an address by some operations - allowing the use of indirect addressing.

Aside from its registers, the CPU needs to interact with system interrupts, and needs to access memory for reading/writing operations.

4.2 PPU, APU, Joypad

The Picture Processing Unit (PPU) is a complex part of the Gameboy, that handles outputting the game state to a 160×144 screen, that may either be monochrome on the DMG or with color support on the GBC. It may raise interrupts, and needs to be able to access memory, due to it being responsible for the Object Attribute Memory (OAM) - a feature allowing transfer of data from an arbitrary location in memory.

The Audio Processing Unit (APU) is responsible for playing the audio output of the console. It has a few registers to control it, as well as a short memory area called the wave RAM. It is partially controlled by the timer, but is otherwise independent. Its sound output is derived from four separate channels: two square wave channels, a custom wave channel, and a noise channel [7, Audio]. These channels are here modelled as separate entities, but are purely internal.

The former two components need to output information to the user, either graphic or audio data. To do this in a portable way, the emulator will expose an interface, that can be implemented by the front-end. This allows the emulator core to be platform-agnostic, and be easily portable.

The joypad, on the other hand, is the one input component the GB has. It contains 4 directional arrow buttons, and 4 input buttons: A, B, start and select (see 4.3). To use these inputs, the CPU needs to use two different registers.

Similarly as for the output, the emulator core will expose an interface to read the user’s input, ie. the state of all 8 buttons.



Figure 4.3: Picture of a Gameboy, with the screen and joypad visible

4.3 Memory Bus

Components within the Gameboy need to communicate. To do this, components typically have a set of registers that control how they behave - for instance, the timer has a **TAC** register at `0xFF07` to control its frequency. To allow this interaction without having all components directly depend on each other, a “memory-bus” component is created, responsible for handling all components and managing memory-addressing among them.

This component, called **System**, is thus responsible for instantiating the other components, and providing them access to each other, via a read and write method.

4.4 Other Components

Aside from the aforementioned components, the GB has a few components that allow it to run but aren’t part of the CPU or the input/output components. These will be briefly outlined here, as they have a lesser impact on the emulator’s design.

4.4.1 Timer

The timer, responsible for incrementing a counter (the **DIV**) at a set frequency. It can raise interrupts, and owns a few registers.

4.4.2 Interrupts

The interrupt system, that allows components such as the timer or the PPU to interrupt the CPU to run another process. It can also be enabled and disabled by the CPU, with the **EI** and **DI** instructions.

This sub-system is designed as a separate component to reduce coupling between other components, and instead have a small object that can be passed to components as needed.

4.4.3 MBC

The Memory Bank Controller (MBC) is an extra chip contained within some game cartridges, to allow access to more ROM data (as well as external RAM in some cases) via banking [7, MBCs]. There are multiple different MBCs, and so it is convenient to define a common interface for all of them, which can then be implemented according to specification for each MBC type.

The type of the MBC is in the header of the ROM [7, The Cartridge Header]. To separate this logic from the base system, a **ROM** class is used. It is responsible for reading the cartridge’s header, and creating the appropriate MBC instance.

4.5 Useful Classes

To have similar interfaces over all components, we will also declare specific classes and interfaces to be implemented by them.

4.5.1 Addressable

The **Addressable** interface provides a read and write method. This allows all components to communicate between each other without needing to be aware of what the component they’re

communicating with it. Aside from the CPU, all components of the emulator implement this method.

4.5.2 Memory and registers

Simple utility classes can be declared to manage memory in a simple way.

The **RAM** class implements **Addressable**. It can be instantiated with a set size, and can be read and written to. It is used in components that have large blocks of writable data.

The **Register** class implements **Addressable**, but ignores the address parameter of the read and write operations, as it contains only one byte. A **DoubleRegister** class may also be implemented, backed by two **Registers**, to provide support for 16-bit registers, used for instance in the CPU.

4.6 Emulator Frontend

To allow the user to access the emulator, a frontend needs to be built with it. This frontend is responsible for outputting the graphics and audio data of the emulator, and also for receiving input from the user to pass down to the emulator. Aside from that, extra UI components have been added, to make the usage of the emulator more convenient and add more features: a play/pause button, custom themes, debugging tools, upscaling filters and keybinding options. All of these are kept independent from the emulator, and control transformations of the input/output, or changes in the running of the emulator.

Because this implementation is merely to allow access to the emulator and doesn't have any outstanding functionality, its description will be brief - most of the focus will be put on the emulator's core.

Chapter 5

Implementation

The project uses [Preact](#), a light-weight alternative to the more popular [React](#) framework. I chose to use Preact because the front-end of the web-app is extremely light weight, so I wanted to avoid bloating the app with a heavy framework such as React. The React framework once GZipped and minified is around 31.8Kb, when Preact is only 4Kb (87% less).

The language used for the project is [TypeScript](#), a typed version of JavaScript. This will prove very useful as the project gets larger, to ensure the correctness of code.

The project is divided in two parts:

- The root directory contains the UI for the web-app. Most of the code is in `app.tsx`, since the front-end is quite light-weight (a few buttons, a file input, and the logic to create the emulator as well as handling inputs and outputs).
- The `emulator/` directory contains the actual GB emulator. Although most classes and interfaces used are exported, only three elements are needed to properly interact with the emulator:
 - `GameBoyColor.ts` handles the core loop of the system: it ticks both the CPU and the rest of the system, bundled under the class `System`.
 - `GameBoyOutput.ts` is a simple interface, with optional methods to receive any output produced by the emulator (see figure 5.1)
 - `GameBoyInput.ts` is a simple interface with a required `read()` method that should return an object with the current inputs for the console (see figure 5.2).

```
interface GameBoyOutput {
  receive?(data: Uint32Array): void;
  debugBackground?(data: Uint32Array): void;
  debugTileset?(data: Uint32Array): void;
  serialOut?(data: number): void;
  errorOut?(error: unknown): void;
  stepCount?(steps: number): void;
  cyclesPerSec?(cycles: number): void;
  frameDrawDuration?(ms: number): void;
}
```

Figure 5.1: `GameBoyOutput` interface methods

```

type GameBoyInputRead = {
  up: boolean;
  down: boolean;
  left: boolean;
  right: boolean;

  a: boolean;
  b: boolean;
  start: boolean;
  select: boolean;
};

interface GameBoyInput {
  read(): GameBoyInputRead;
}

```

Figure 5.2: GameBoyInput interface method

5.1 Emulator Frontend

The frontend of the emulator is written in Preact, allowing us to create a simple, fast and lightweight UI to control it. It contains the emulator title, the control buttons, some debug information, and the emulator video output (see figure 5.3).

The main controls for the emulator are the 6 buttons above the screen. These are, from left to right:

- Pause/play: pauses or starts the emulator.
- Step one M-Cycle: this is particularly useful when debugging to figure out exactly when something goes wrong. It is only enabled when the emulator is paused.
- Debug view: displays the tileset and background map alongside the executing game (see figure 5.4)
- Compare mode: runs a second open-source emulator ([Gameboy.js](#)) at the same time as my emulator. This is useful to compare graphics and check it works somewhat properly.
- Test mode: stops execution of the current game, and instead runs a series of tests automatically, recording which ones pass and fail. The result of the tests is logged to the console as a table, allowing to quickly check to what is and isn't working (see figure 5.5). The state of a test can be: a green check (passed), a red cross (failed), a headstone (failed due to error) or an hourglass (failed due to timeout).
- Speed-boost toggle: a toggle button that speeds up the emulator to triple its base speed.

Further more the emulator currently displays some debug data, to check the performance of it. From top to bottom, these are:

- The total number of instructions executed so far.
- The number of T-Cycles executed per second - this is to compare with the ideal of 4.19MHz.
- The number of milliseconds needed to render a frame. The GB renders at 60 frames per second (60Hz), so this measures the time taken to perform $(4.19 * 10^6)/60 \approx 69000$ T-Cycles.

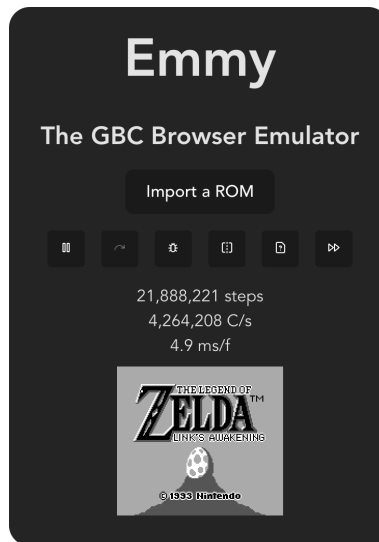


Figure 5.3: Home page of the emulator

5.2 Emulator Backend

To keep the logic of the hardware, divided in different components, the structure of the emulator has a similar format, with different classes implementing the behaviour of the different parts of the Gameboy.

5.2.1 Addressable

The **Addressable** interface is implemented by most classes of the emulator: **System**, **Register**, **MBC**, etc. It is simple, and provides a read and a write method (see figure 5.6). The advantage of using it is that when implementing addressing logic with large switch-statements I can simply return any type of components that adheres to the interface, and the receiving method is agnostic of if it's dealing with a register, memory, or a component.

5.2.2 System

The **System** class implements the motherboard, and the general connection between elements. It handles the ticking of the PPU, timer and OAM DMA. Furthermore, it is the component that links all of the data together: whenever a component is ticked (any of the above or the CPU), the **System** instance is passed, so that the components can read and write to the rest of the system. It does implement **Addressable**, and internally has a **getAddress** method that returns both an **Addressable** and a number (the index to read or write to). This way the addressing logic to determine what component is accessed depending on the address isn't duplicated.

getAddress optimisation

Because **System** is a highly used component and is accessed for almost every read and write, the **getAddress** method is under a lot of pressure. As such, it was the source of a major re-write during development, which improved performances significantly.

Initially it was implemented as a list of if-statements for different ranges of the Memory Map, as well as an object where the keys were different register-addresses. The code would first check if the key exists in the register object (it thus served as a map), and if not it would then go

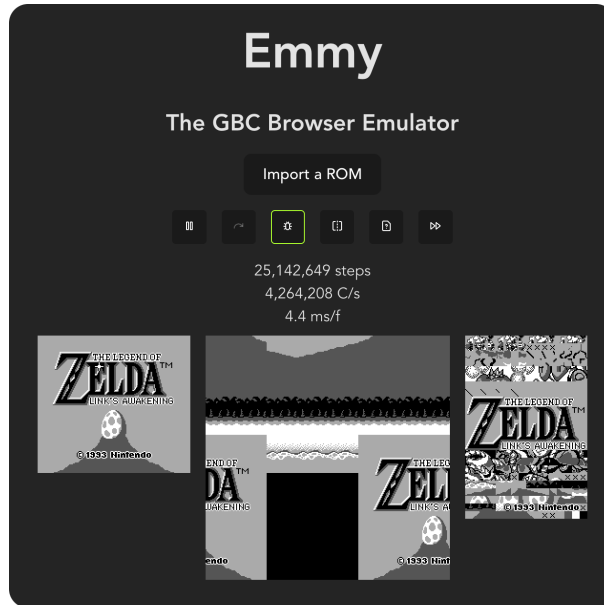


Figure 5.4: Debug view of the emulator

(index)	type	group	state
dmg_sound	'blaarg'	'other'	⌚
halt_bug	'blaarg'	'other'	⌚
oam_bug	'blaarg'	'other'	⌚
mem_oam	'mooneye'	'oam'	✓
oam_dma_restart	'mooneye'	'oam'	✓
oam_dma_start	'mooneye'	'oam'	✗
oam_dma_timing	'mooneye'	'oam'	✓
oam_dma_basic	'mooneye'	'oam'	✓
oam_dma_reg_read	'mooneye'	'oam'	✓
oam_dma_sources-GS	'mooneye'	'oam'	🔔

Figure 5.5: Snippet of test mode output

through a series of if-conditions (see figure 5.10).

This proved quite costly, for three main reasons:

- This code creates a new object every time it is called, when checking for the registers' addresses.
- The if-conditions for the ranges of the biggest areas of memory (everything between 0x0000 and 0xffff) happened after the register checks, which delayed response for these reads and writes.
- Chaining if-conditions is unnefficient, as the JS engine must step through all conditions and check the values each time. Furthermore, although having both the lower and upper bound of the memory section indicated in the condition (e.g. `0x0000 <= pos && pos <= 0x7fff` for `[0x0000; 0x7fff]`) makes the translation from Memory Map to code easier, it is slower, since only the upper bound of the range is needed for the condition *if* all the ranges below have already returned.

My intial optimisation - and the one that had the most impact on this method - was fixing the last two points: removing these if-conditions, and moving these areas higher in the code.

The way the addressing circuitry works is that addresses are not compared in their entirety: the

```

interface Addressable {
  read(pos: number): number;
  write(pos: number, data: number): void;
}

```

Figure 5.6: Addressable interface method

circuit checks for small sections of the address, and then maps those to more precise locations. As such the main areas of memory can usually be identified by their Most Significant Bits (MSB) - usually the most significant nybble (see figure 5.7).

Start	End	Description
0000	3FFF	16 KiB ROM bank 00
4000	7FFF	16 KiB ROM Bank 01~NN
8000	9FFF	8 KiB Video RAM (VRAM)
A000	BFFF	8 KiB External RAM
C000	CFFF	4 KiB Work RAM (WRAM)
D000	DFFF	4 KiB Work RAM (WRAM)

Source: [Pandocs](#)

Figure 5.7: Memory map for the largest chunks of memory

This means that for this area of memory we can simply isolate the last nybble, and then create a switch-statement over it to map directly to specific areas. This removes the need for the if-conditions, and is also faster as it is evaluated much earlier on (see figure 5.11).

I then ran a simple test to verify the performance improvement. I ran the first 25 million instructions of the [cpu_instrs](#) test ROM - I chose this sample because it is considerably large, because the test itself requires around 25 million instructions to complete, and because this test checks for all instructions of the CPU, meaning that it exercises most configurations of the CPU and system. For the measurement, I used `window.performance.now()` before and after each drawn frame, and summed the values.

The result was the following: 33955.9ms before the change, and 20039.1 after the change. The relative difference is thus $\frac{20039.1 - 33955.9}{33955.9} = -0.4098$, thus reducing time taken by 40.98%.

Note that, although this may vary based on engine, switch statements aren't faster than if-conditions in JavaScript. [This simple test-suite](#) has a switch statement with 16 cases, along with an object with 16 keys and an if-statement with 16 clauses. On my browser (Google Chrome v108), the switch-statement proved the fastest, with the switch and if statements lagging behind by being respectively 69.81% and 70.23% slower. As such I suspect this improvement to be mainly due to the switch statement being moved at the beginning of the function, thus skipping the more expensive object creation.

This also means that improvements to this method can still be done. Using the "Performance" tab of Chrome Developer Tools, I measured the performance of the whole emulator when running the first 10 million instructions of this same test. The results I got indicated that `getAddress` is still the third method with the highest **self-time** (see figures 5.8 and 5.9). Here self-time refers to the time taken inside the method itself, and total time is the time taken by the method and the methods it calls. Note how `getAddress` still has the third highest self-time, meaning that the content of the method itself takes time (although this value dropped from 40.7% to 7.7%). The

increase is self-time percentage (not time!) for all other methods is simply due to the fact that by making `getAddress` faster the self-time of these other methods (that all call `getAddress`) gets proportionally higher compared to their total time.






Self Time	Total Time	Activity
6937.0 ms 40.7 %	6945.4 ms 40.7 %	▶  getAddress
2956.4 ms 17.3 %	2960.9 ms 17.4 %	▶  address
831.6 ms 4.9 %	5110.9 ms 30.0 %	▶  tick
512.3 ms 3.0 %	684.6 ms 4.0 %	▶  executeNext
437.4 ms 2.6 %	7014.6 ms 41.1 %	▶  read

Figure 5.8: Performance measurement of the emulator before `getAddress` optimisation






Self Time	Total Time	Activity
2966.2 ms 29.0 %	3029.0 ms 29.7 %	▶  address
801.5 ms 7.8 %	5101.8 ms 50.0 %	▶  tick
784.8 ms 7.7 %	800.2 ms 7.8 %	▶  getAddress
466.9 ms 4.6 %	649.8 ms 6.4 %	▶  executeNext
440.8 ms 4.3 %	5508.8 ms 54.0 %	▶  tick

Figure 5.9: Performance measurement of the emulator after `getAddress` optimisation

```
protected getAddress(pos: number): AddressData {
  if (pos < 0x0000 || pos > 0xffff)
    throw new Error(`Invalid address to read from ${pos.toString(16)}`);

  const register = {
    0xff00: this.joyypad,
    ...
    0xffff: this.intEnable,
  }[pos];
  if (register !== undefined) return [register, pos];

  if (0x0000 <= pos && pos <= 0x7fff) return [this.rom, pos]; // ROM Bank
  ...
  if (0xfe00 <= pos && pos <= 0xfe9f) return [this.oam, pos]; // OAM

  // Unmapped area, return 'fake' register
  return [{ read: () => 0xff, write: () => {} }, 0];
}
```

Figure 5.10: Initial implementation of `getAddress`

```

protected getAddress(pos: number): AddressData {
  if (pos < 0x0000 || pos > 0xffff)
    throw new Error(`Invalid address to read from ${pos.toString(16)}`);

  switch ((pos >> 12) as Int4) {
    case 0x0:
      return [this.rom, pos]; // ROM
    ...
    case 0xe:
      return [this.wram, pos & (WRAM_SIZE - 1)]; // ECHO RAM
    case 0xf:
      break; // fall through - ECHO RAM + registers
  }

  const register = {
    0xff00: this.joyypad,
    ...
    0xffff: this.intEnable,
  }[pos];
  ... rest is unchanged
}

```

Figure 5.11: Optimised implementation of `getAddress`

5.2.3 CPU

The CPU is the most important part of the emulator, and allows running the code from the ROM by reading the operation code (or opcode) and executing the matching action.

Initial Instruction Set

Because the Gameboy is an 8-bit system, opcodes are 8 bits (or one byte) long, giving in theory a maximum of $2^8 = 256$ operations. However in the GB the operation `0xCB` gives access to an extended instruction set, meaning that when reading `0xCB` the CPU will read the next byte and use a different logic to execute the operation. This means there are now $2^8 - 1 + 2^8 = 511$ operations. The GB also has 11 forbidden opcodes (the console will lock itself if they are executed), meaning there are in total $2^8 - 12 + 2^8 = 500$ operations to implement.

Multiple techniques exist to handle this large number of operations:

- Have a large switch-statement for all operations
- Have a map, that maps an opcode to a function to execute
- Decode the operation by reading specific parts of the byte, and generate the instructions dynamically

I initially had a large object, with all the opcodes as keys, that would then contains a simple function that returns the number of cycles taken by the instruction (see figure 5.12). This however proved quite repetitive and prone to errors. Furthermore, this wasn't M-Cycle accurate: the CPU instruction was executed as one monolithic block, when in reality all reads and writes are executed at a separate M-Cycle - this becomes very important when the timer, OAM and PPU are involved, as they run in parallel with the CPU, so memory accesses to these components may return different values depending on the M-Cycle.

```
protected instructionSet: Partial<Record<number, InstructionObject>> = {  
    // NOP  
    0x00: () => 1,  
    // LD BC/DE/HL/SP, d16  
    0x01: (s) => { this.regBC.set(this.nextWord(s)); return 3; },  
    0x11: (s) => { this.regDE.set(this.nextWord(s)); return 3; },  
    0x21: (s) => { this.regHL.set(this.nextWord(s)); return 3; },  
    0x31: (s) => { this.regSP.set(this.nextWord(s)); return 3; },  
    // INC BC/DE/HL/SP  
    0x03: () => { this.regBC.inc(); return 2; },  
    0x13: () => { this.regDE.inc(); return 2; },  
    0x23: () => { this.regHL.inc(); return 2; },  
    0x33: () => { this.regSP.inc(); return 2; },  
    ...  
}
```

Figure 5.12: Initial instruction set implementation

Becoming M-Cycle accurate

In most (if not all) emulators I looked, the way M-Cycle accuracy is reached is by making the emulator **CPU-driven**. What this means is that inside each instruction, between each M-Cycle, the CPU is responsible for ticking the rest of the system - the main loop is then only responsible

for continuously running the CPU, and nothing else. This approach is probably the simplest and most straightforward one, as it is quite simple to implement - all one needs to do is call the system tick method when relevant (see figure 5.13) .

```
protected ld_bc_hl(system: System) { // LD BC, (HL)
    const lower = system.read(this.regPC.inc());
    system.tick();
    const upper = system.read(this.regPC.inc());
    system.tick();
    this.regBC.set(upper << 8 | lower);
}
```

Figure 5.13: CPU-driven LD BC, (HL)

Because this was already done in most emulators, I wanted to try another idea I had, that I hadn't seen anywhere else: keep the emulator "system-driven", and instead make the CPU remember what state and M-Cycle it's on and what micro-instruction it must execute next.

This was done by splitting all the instructions into smaller chunks, that return each other, as arrow-functions. The CPU now must simply store whatever the instruction returns: if it's `null` then it needs to fetch an instruction at the next cycle, otherwise it's a function and must be executed (and it's result stored for the next step). An advantage of this method is that the CPU is still only responsible for executing instructions - it only needs the system to read/write to memory. See figures 5.14 and 5.15 for an example of this.

```
protected ld_bc_hl(system: System) { // LD BC, (HL)
    const lower = system.read(this.regPC.inc());
    return () => {
        const upper = system.read(this.regPC.inc());
        return () => {
            this.regBC.set(upper << 8 | lower);
            return null;
        }
    }
}
```

Figure 5.14: System-driven LD BC, (HL)

```
step(system: System) {
    if (this.nextStep === null) {
        // Execute next instruction
        const opcode = this.nextByte(system);
        const instruction = this.instructionSet[opcode];
        this.nextStep = instruction;
    }
    this.nextStep = this.nextStep(system);
}
```

Figure 5.15: System-driven step of the CPU

- 5.2.4 Timer
- 5.2.5 PPU
- 5.2.6 OAM and OAM DMA
- 5.2.7 APU
- 5.2.8 MBCs and ROMs

Chapter 6

Evaluation

Acronyms

APU Audio Processing Unit.

CPU Central Processing Unit.

DMA Direct Memory Access.

DMG Dot Matrix Game.

GB Gameboy.

GBC Gameboy Color.

MBC Memory Bank Controller.

MSB Most Significant Bits.

OAM Object Attribute Memory.

OS Operating System.

PPU Picture Processing Unit.

ROM Read-Only Memory.

Glossary

Dot Matrix Game The model name of the Gameboy. It is often used to refer to the "base" Gameboy (in contrast with GBC for the Gameboy Color).

M-Cycle An M-Cycle (or machine cycle) is the smallest step the CPU of the Gameboy can do. Because all instructions of the CPU are multiples of 4, instruction lengths and timings are usually referred to in M-cycles (e.g. LD A, B takes 4 T-cycles, thus 1 M-cycle).

Memory Map The memory map is what determines where each address leads to - it can be seen as a list of non-overlapping ranges.

Picture Processing Unit The part of the Gameboy that is responsible for rendering the game.

T-Cycle A T-cycle is the smallest step the internal clock of the Gameboy can do. This means the rate of T-cycle is that of the CPU, ie. 4.19Mhz.

Bibliography

- [1] Merriam-Webster. “Emulator definition and meaning.” (Mar. 2022), [Online]. Available: <https://www.merriam-webster.com/dictionary/emulator> (visited on 03/02/2022).
- [2] A. Kaluszka. “Computer emulation, history.” (Dec. 2001), [Online]. Available: <https://kaluszka.com/vt/emulation/history.html> (visited on 03/03/2022).
- [3] IBM. “709/7090/7094/7094 II compatibility feature for IBM System/370 models 165, 165 II, and 168.” (1973), [Online]. Available: http://bitsavers.org/pdf/ibm/370/compatibility_feature/GA22-6955-1_709x_Compatibility_Feature_for_IBM-370_165_168.pdf (visited on 03/03/2022).
- [4] P. Magnusson. “Full system simulation: Software development’s missing link.” (Oct. 2004), [Online]. Available: <https://www.computerworld.com/article/2567652/full-system-simulation--software-development-s-missing-link.html> (visited on 03/07/2022).
- [5] MyaMyaMya. “First Famicom/NES emulator? - Zophar’s Domain.” (Jan. 2009), [Online]. Available: <https://www.zophar.net/forums/index.php?threads/first-famicom-nes-emulator.10169/> (visited on 03/07/2022).
- [6] Emulation General Wiki. “History of emulation.” (Jan. 2023), [Online]. Available: https://emulation.gametechwiki.com/index.php/History_of_emulation#Game_Boy.2FColor (visited on 03/07/2022).
- [7] “Pandocs.” (2023), [Online]. Available: <https://gbdev.io/pandocs/> (visited on 03/07/2022).