

Emmy: The Gameboy Color Emulator

6CCS3PRJ Final Project

Classic video game console emulation

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

Draft Two
February 27, 2023
King's College London

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

Acronyms	3
Glossary	4
1 Background	5
1.1 Emulation	5
1.2 Video Game Emulation	5
1.3 Gameboy, Gameboy Color	5
1.4 Existing Literature	6
1.4.1 Gameboy Documentation	6
1.4.2 Existing Emulators	6
1.4.3 Gameboy Test ROMs	7
2 Requirements and Specification	8
2.1 Requirements	8
2.1.1 User Requirements	8
2.1.2 System Requirements	8
2.1.3 Non-Functional Requirements	9
2.2 Specification	9
3 Design	11
3.1 Emulator Frontend	12

3.2	Emulator Backend	13
3.2.1	Addressable	13
3.2.2	System	13
3.2.3	CPU	18
3.2.4	PPU	20
3.2.5	Timer	20
3.2.6	OAM and OAM DMA	20
3.2.7	MBCs and ROMs	20
3.2.8	Audio	20
	List of Figures	21

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.

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.

Chapter 1

Background

1.1 Emulation

TO-DO: Outline what emulation is/was used for, how it started. Examples of its usage, examples of what it's useful for.

Current state of emulation, in what industry is it applied? Talk also about virtual machines.

1.2 Video Game Emulation

Video game emulation is the art of simulating a video game hardware system, to allow it 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.

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 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.

1.3 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. 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 colors, from a palette of 32768 colors. 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, which are 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.

1.4 Existing Literature

1.4.1 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 of the resources I used to write my emulator 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.
- [GBCTR](#) (Gameboy Complete Technical Reference) 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.
- [GB dev wiki](#) is a wiki containing additional information on the GB, including guides to making games and explanations on some hardware quirks.

1.4.2 Existing Emulators

Furthermore, to aid with developing the emulator, I sometimes had to look through the open source code of existing emulators, to understand how they implemented certain things. These projects 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 Audio Processing Unit (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.

1.4.3 Gameboy Test ROMs

Finally, some of the most helpful resources I've used are test ROMs, that can be run on an emulator and output if the emulator passes certain tests (these can be CPU, PPU, timer tests, etc.). They help quickly diagnose issues, and also allow me to mark certain components and parts of the emulator as "complete", meaning I can rely on these components and know they are fully-functional. These test ROMs also have the advantage of being open source, meaning I can understand what the test does, and if it fails figure out where and why it goes wrong without having to decompile the ROM.

An other advantage to using test ROM is that they re-use the same framework across a given suite to report results. This means the result of the test is usually logged somewhere in the console, and testing can easily be automated without having to compare the displayed image.

The test ROMs I used 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 much more complete and tougher 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 2

Requirements and Specification

2.1 Requirements

2.1.1 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.

2.1.2 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.

2.1.3 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.

2.2 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 window 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, button 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 3

Design

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 3.1)
 - `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 3.2).

```
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 3.1: 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 3.2: GameBoyInput interface method

3.1 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 3.3).

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 3.4)
- 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 3.5). 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.

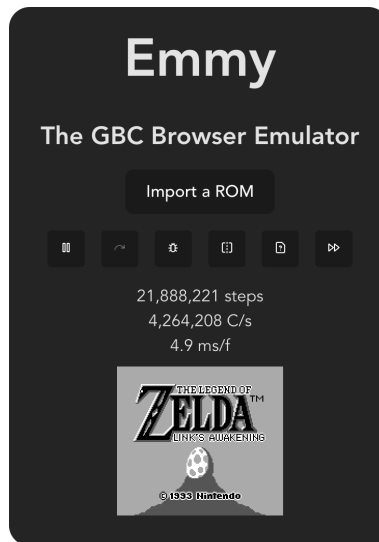


Figure 3.3: Home page of the emulator

3.2 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.

3.2.1 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 3.6). 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.

3.2.2 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

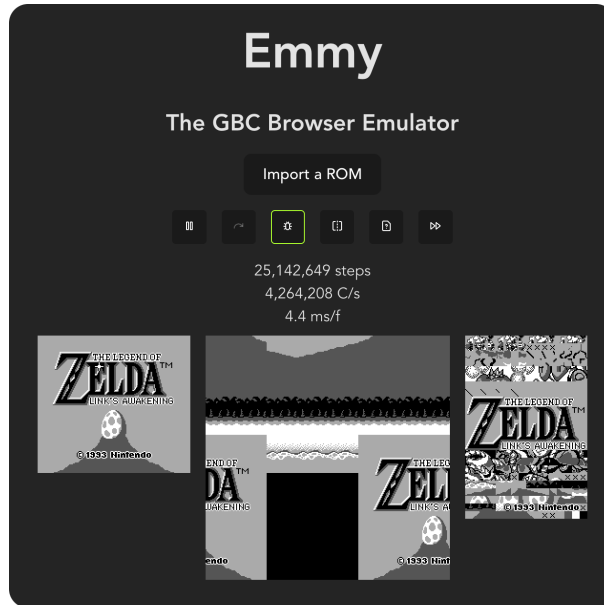


Figure 3.4: Debug view of the emulator

(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 3.5: Snippet of test mode output

through a series of if-conditions (see figure 3.10).

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 3.6: 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 3.7).

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 3.7: 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 3.11).

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 3.8 and 3.9). 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 3.8: 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 3.9: 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 3.10: 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 3.11: Optimised implementation of `getAddress`

3.2.3 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 3.12). 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 3.12: 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 3.13) .

```
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 3.13: 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 3.14 and 3.15 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 3.14: 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 3.15: System-driven step of the CPU

- 3.2.4 PPU**
- 3.2.5 Timer**
- 3.2.6 OAM and OAM DMA**
- 3.2.7 MBCs and ROMs**
- 3.2.8 Audio**

List of Figures

3.1	<code>GameBoyOutput</code> interface methods	11
3.2	<code>GameBoyInput</code> interface method	12
3.3	Home page of the emulator	13
3.4	Debug view of the emulator	14
3.5	Snippet of test mode output	14
3.6	<code>Addressable</code> interface method	15
3.7	Memory map for the largest chunks of memory	15
3.8	Performance measurement of the emulator before <code>getAddress</code> optimisation	16
3.9	Performance measurement of the emulator after <code>getAddress</code> optimisation	16
3.10	Initial implementation of <code>getAddress</code>	16
3.11	Optimised implementation of <code>getAddress</code>	17
3.12	Initial instruction set implementation	18
3.13	CPU-driven LD <code>BC, (HL)</code>	19
3.14	System-driven LD <code>BC, (HL)</code>	19
3.15	System-driven step of the CPU	19