6CCS3PRJ
Emmy, The Game Boy Emulator

Final Project Report

Author: Oscar Sjöstedt

Supervisor: Ian Kenny

Student ID: K20040078

Programme of Study: Computer Science MSci

April 5, 2023

Abstract

This project aims to create a Game Boy emulator web-application, in other words a program capable of receiving Game Boy 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).

Originality Avowal

I verify that I am the sole author of this report, except where explicitly stated to the contrary. I grant the right to King's College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to Turnitin or another trusted plagiarism detection service. I confirm this report does not exceed 25,000 words.

Oscar Sjöstedt

April 5, 2023

Contents

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 do not 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 Game Boy 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 Game Boy and Game Boy 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 Game Boy (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 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 in the 1980s, 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].

Emulation is also vital for preservation: as transistors and motherboards age, old systems become unusable, and with them the software they ran. Companies may also stop producing the hardware to run this software on. Emulating these systems is often the only future-proof and sustainable way to keep this software usable [4].

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. In the case where the architectures of the guest and the host are the same, we call this *virtualisation* [5]. Virtualisation is nowadays extremely common, with tools like Docker[1], that allow multiple virtualised instances

[1] See https://www.docker.com/

of computer systems running on the same host. This typically allows for great portability, as the same infrastructure can be copied and ran anywhere.

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 the only legally available source of information on these consoles comes from research and reverse-engineering done by hobbyists, and does not come from an official source, as it is proprietary hardware.

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 [6], and were able to run some NES games. The first Gameboy emulators were in the late 90s, with the Virtual GameBoy[2] in 1995 and NO$GMB[3] in 1997 (although its history page[4] seems to indicate development started in 1993) [7].

The original game files and assembled code for video games are copyrighted material, and are 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[5] host wide collections of legal ROMs.

Official emulators also exist, and are developed by the console manufacturers. These usually only allow playing from a selection of games, limiting options. The reason for this is that they are built to emulate these specific games, rather than the console as a whole, meaning games outside of the catalogue will often not work. They are thus usually less accurate than unofficial emulators – this is the case of the Virtual Console, the official emulator of the Nintendo 3DS, that fails many Game Boy test ROMs [8, Test ROMs].

2.3 Game Boy, Game Boy Color

The Game Boy (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

[2] See http://fms.komkon.org/VGB/
[3] See https://problemkaputt.de/gmb.htm
[4] See https://problemkaputt.de/gmbhist.htm
[5] See https://www.retroveteran.com/category/nintendo-game-boy-color/

at 4.19MHz [9, Specifications]. In 1998, the Game Boy 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 colour). 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 [10, STAT IRQ glitches].

From an emulation perspective, the Game Boy 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 Game Boy and the Game Boy Color, as they are very similar. Dot Matrix Game (DMG) refers exclusively to the original Game Boy model.

2.4 Existing Literature

2.4.1 Gameboy Documentation

The Game Boy is one of the best documented consoles for emulation, and a large array of resources exist documenting it. Some useful resources explaining its behaviour are:

- Pandocs[6] 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[7] is a table containing all instructions its 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[8] (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[9] 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

[6] See https://gbdev.io/pandocs/
[7] See https://meganesu.github.io/generate-gb-opcodes/
[8] See https://gekkio.fi/files/gb-docs/gbctr.pdf
[9] See https://gbdev.gg8.se/wiki/

description of the Audio Processing Unit (APU).

2.4.2 Existing Emulators

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

- Game Boy Crust[10] is a simple GB emulator written in Rust. It is quite incomplete but has a comprehensive structure, so its a good project to first figure out how emulators work.
- AccurateBoy[11] is a highly accurate emulator, in particular for its Picture Processing Unit (PPU) that has pixel-perfect accuracy.
- oxideboy[12] is another GB emulator written in Rust, that is much more complete and helpful for some edge cases.
- SameBoy[13] 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 used as a reference when developing this project.
- Mooneye GB[14] 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[15] is a high-accuracy JavaScript emulator, that is particularly useful to understand how to interface the emulator with the browser (notably for the APU).
- Gameboy.js[16] is another JavaScript emulator. It is fairly simple and inaccurate, but is easily hackable, making it useful when starting a new emulator to compare execution traces.
- rboy[17] is an emulator written in Rust that was used when developing the APU, as it passes some complex test ROMs with quite simple code.

The 8th of February 2023, Nintendo announced the release of a Game Boy and Game Boy Color emulator on the Nintendo Switch, via the Nintendo Switch Online subscription [11]. The recent release of official emulators as well as public enthusiasm for the latter prove the relevance

[10] See https://github.com/mattbruv/Gameboy-Crust
[11] See https://github.com/Atem2069/accurateboy
[12] See https://github.com/samcday/oxideboy
[13] See https://github.com/LIJI32/SameBoy
[14] See https://github.com/Gekkio/mooneye-gb
[15] See https://github.com/taisel/GameBoy-Online/
[16] See https://github.com/juchi/gameboy.js/
[17] See https://github.com/mvdnes/rboy

of this kind of emulator.

2.4.3 Gameboy Test ROMs

A core set of resources to develop an emulator are test ROMs. These are ROMs which instead of playing a game will run a series 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 in order to understand what they expect of the console.

An other advantage of using test ROMs is that they tend to re-use the same framework across a given test suite to report results. This means testing can easily be automated over multiple tests by inspecting specific registers/memory addresses, rather than having to store an "expected result" image for each test.

The test ROMs used for this project are:

- Blaarg test ROMs[18] are some of the most well-known and used GB test ROMs. They include tests for the CPU, the timings of instructions, hardware bugs and the APU.
- Mooneye test ROMs[19] 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), timings of the APU, Direct Memory Access (DMA), PPU and timer.
- Acid Test (DMG[20], GBC[21]) is a test that verifies that the PPU of the GB displays data properly (to line-rendering accuracy), for both Game Boy and Game Boy Color displays.
- SameSuite[22] 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).

[18] See https://github.com/retrio/gb-test-roms/
[19] See https://github.com/Gekkio/mooneye-test-suite
[20] See https://github.com/mattcurrie/dmg-acid2
[21] See https://github.com/mattcurrie/cgb-acid2
[22] See https://github.com/LIJI32/SameSuite/

Chapter 3

Requirements and Specification

3.1 Requirements

3.1.1 User Requirements

U1. Run Game Boy games to a satisfiable fidelity, with proper rendering and controls emulation.

U2. Run Game Boy Color games to a satisfiable fidelity, with proper rendering and controls emulation.

U3. Allow the user to run GB and GBC games on both Game Boy and a Game Boy Color.

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, Scale2x, 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 Game Boy 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

Code	Specification	Importance
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[1]).	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 GB via a dropdown button.	Low
F1	A ROM file can be uploaded, is transformed into an `UInt8Array` (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
F3	A `<canvas />` 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[2] framework is used to handle the UI of the web-app.	High

[1] See https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API
[2] See https://preactjs.com/

Code	Specification	Importance
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 do not fail.	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. The emulator's name will be "Emmy", short for emulator[1].

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

In emulators, the different components are usually split into three groups:

- 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.

4.1 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

[1]The emulator was named before the realisation that an emulator-related project called "Emmy" already exists [12]. These two projects are unrelated.

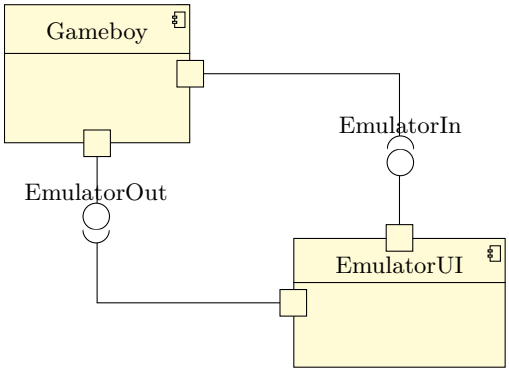

Figure 4.1: Split of emulator core and frontend

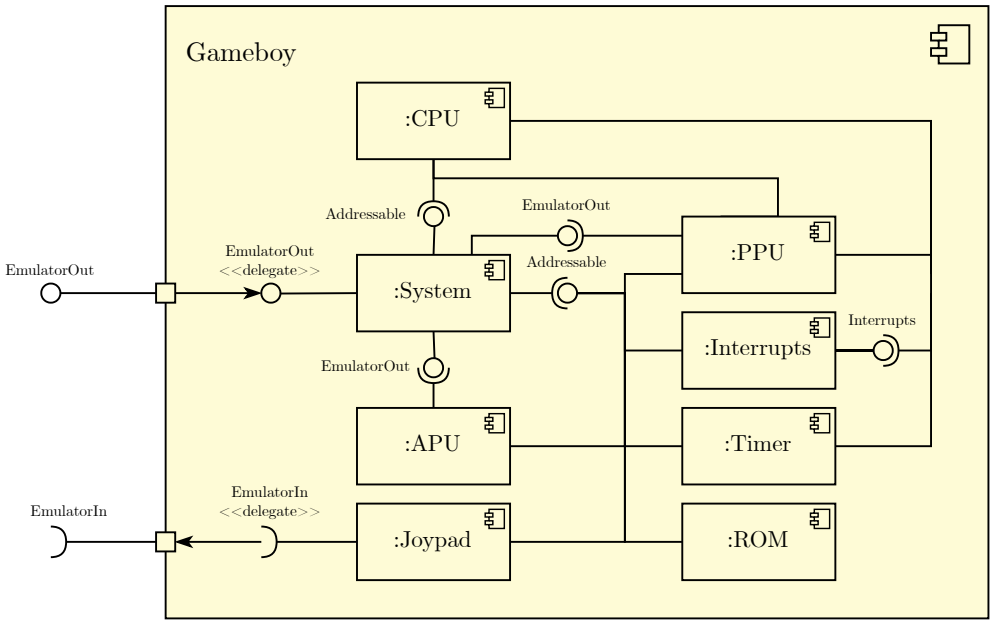

Figure 4.2: Simplified design of the emulator's core
Note utility classes have been excluded for readability purposes

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 and 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 and output, or changes in the running of the emulator.

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

4.2 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 Sharp LR35902, which is a hybrid of the Intel 8080 chip and the Zilog Z80 [13].

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 – it 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 its 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 [9, CPU Registers and Flags].

- `AF`: the accumulator-flag register. Most arithmetic operations can be done on `A`. `F` holds the CPU's flags, and can only be altered indirectly, as a consequence of 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.

4.3 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 colour 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) Direct Memory Access (DMA) – a feature allowing transfer of data from an arbitrary location in memory [9, OAM DMA Transfer].

The behaviour of the PPU changes significantly between the DMG and the GBC: it can, for the latter, support colours, and it also supports Video RAM (VRAM) bank switching, a feature the DMG lacks [9, CGB Registers]. It also has an Object Attribute Memory (OAM) DMA mechanism that is quite complex. As such, to make maintenance easier, the PPU is split into different smaller classes. This is not visible to the rest of the system, as only the PPU is exposed to it. See Figure 4.3 for a simplified class diagram of its structure.

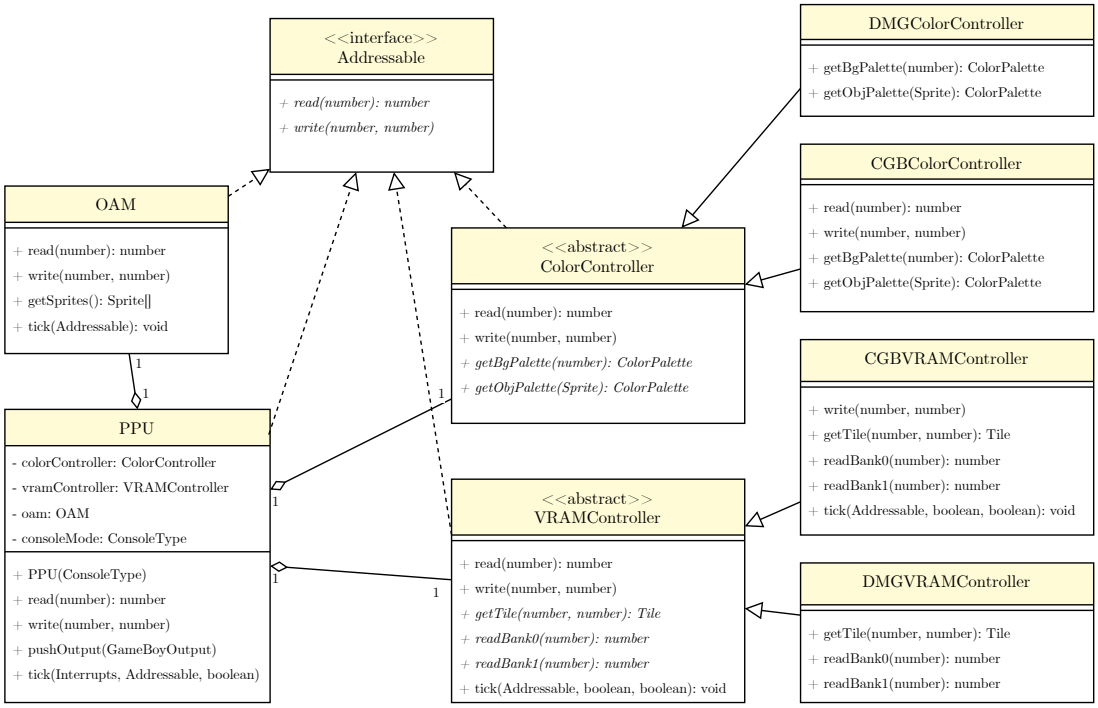

Figure 4.3: Simplified class diagram of the PPU
Note most private fields and methods have been excluded for readability purposes

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. Its timings are controlled by the timer, which can be accessed via a memory read of the system.

Its sound output is derived from four separate channels: two square wave channels, a custom wave channel, and a noise channel [9, Audio]. These channels are here modelled as separate entities, but are purely internal (see Figure 4.4).

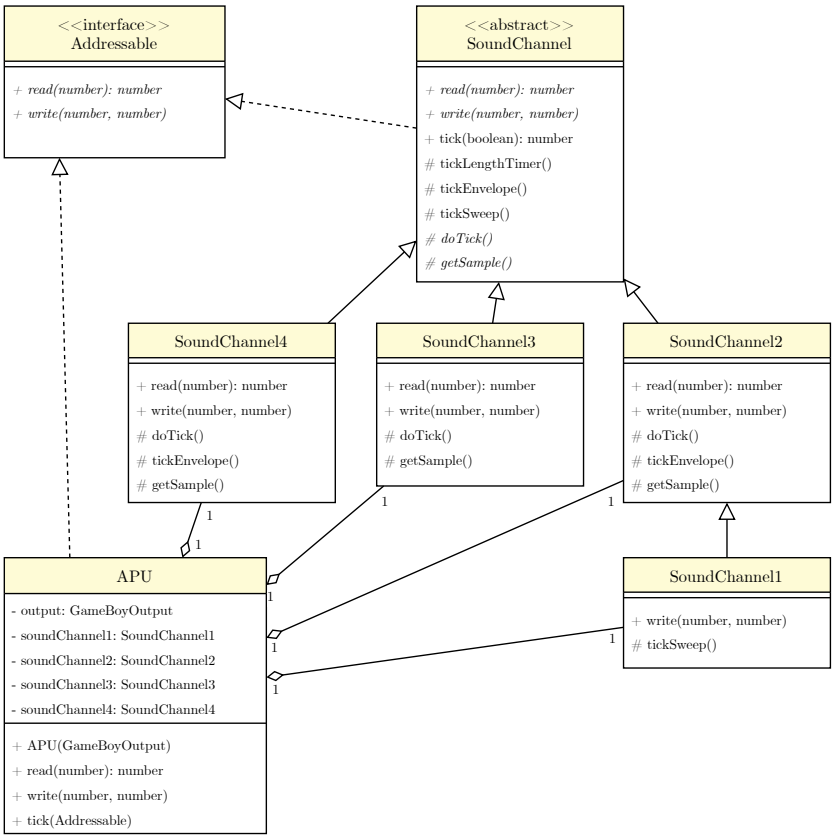

Figure 4.4: Simplified class diagram of the APU
Note most private fields and methods have been excluded for readability purposes

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 re-usable.

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 Figure 4.5). 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 4.5: Picture of a Gameboy, with the screen and joypad visible

4.4 System 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 "system 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.

4.5 Other Components

Aside from the aforementioned components, the GB has a few components that allow it to run but are not 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.

4.5.1 Timer

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

4.5.2 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 [9, Interrupts].

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

4.5.3 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 [9, 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 [9, 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.

4.6 Useful Classes

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

4.6.1 Addressable

The `Addressable` interface (see Figure 4.6) 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 are communicating with is. Aside from the CPU, all components of the emulator implement this method.

```
1 interface Addressable {
2     read(pos: number): number;
3     write(pos: number, data: number): void;
4 }
```

Figure 4.6: `Addressable` interface

4.6.2 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 `Register`s, to provide support for 16-bit registers, used for instance in the CPU.

Chapter 5

Implementation

The project uses Preact[1], a light-weight alternative to the more popular React[2] framework. Preact was chosen because the front-end of the web-app is extremely lightweight, so a smaller framework with less features is enough. This also avoids bloating the app with a heavy framework such as React: its GZipped and minified size is around 31.8Kb, while Preact is only 4Kb (87% less) [14].

The language used for the project is TypeScript[3], a typed version of JavaScript. This is essential for the project, as ensuring the correctness of code can be extremely hard without proper typing constraints, in particular when the codebase becomes larger and more complex.

The project is divided into two parts:

- The `frontend/` directory contains the UI for the web-app. The main logic to create the emulator and run it is contained in `app.tsx`.
- 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 contains the `GameBoyColor` class, the emulator. Instantiating the emulator creates all the necessary sub-components, and calling the `drawFrame()` method runs the emulator for one frame (0.16 seconds).
  - `GameBoyOutput.ts` contains a simple interface, with optional methods to receive any output produced by the emulator (see Figure 5.1). The two main methods of this are `receiveGraphics` and `receiveSound`, which use the output of the actual console.

[1] See https://preactjs.com/
[2] See https://reactjs.org/
[3] See https://www.typescriptlang.org/

– `GameBoyInput.ts` contains a simple interface with a required `read()` method that returns an object with the current inputs for the console (see Figure 5.2).

```ts
interface GameBoyOutput {
    receiveGraphics?(data: Uint32Array): void;
    receiveSound?(data: Float32Array): void;

    // Debugging methods:
    debugBackground?(data: Uint32Array): void;
    debugTileset?(data: Uint32Array): void;
    serialOut?(data: number): void;
}
```

Figure 5.1: `GameBoyOutput` interface methods

```ts
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 for the creation of a simple, fast and lightweight UI to control it. It contains the emulator title, the control buttons and the emulator video output (see Figure 5.3). Along the left of the screen is a sidebar with more options, allowing the user to customise the emulator to their needs and debug the state of the console if needed.

5.1.1 Main Controls

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

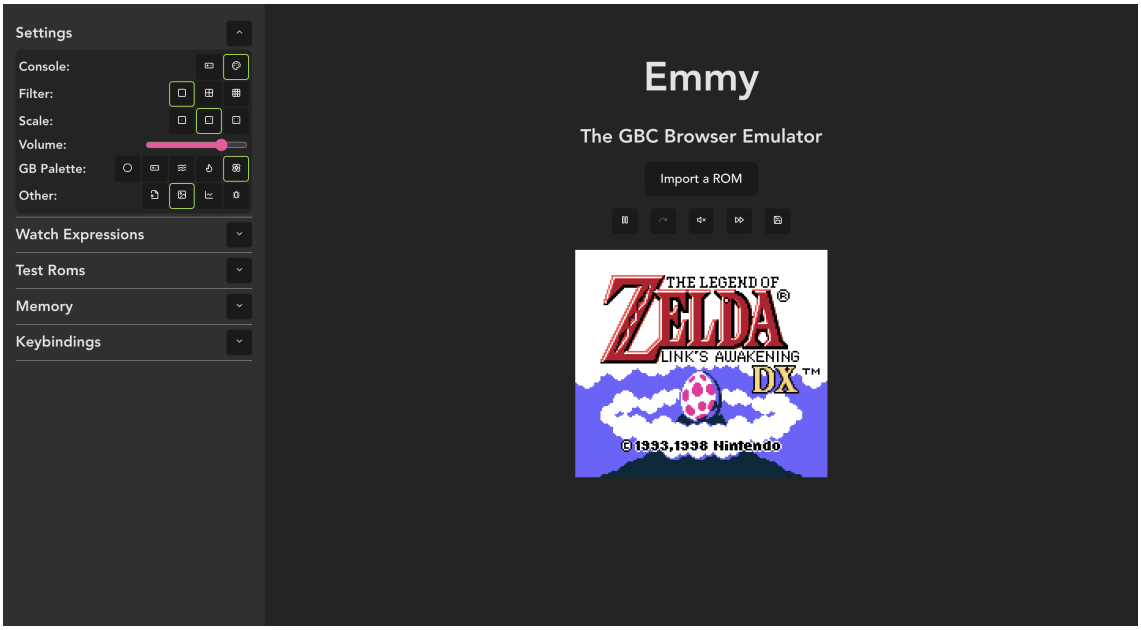

Figure 5.3: Home page of the emulator

- Pause/play: pauses or resumes the emulator.
- Step by one CPU instruction: this is a feature useful for debugging the emulator, when precise information of the state of the emulator is needed.
- Sound toggle: allows enabling or disabling the sound of the emulator. By default the sound is turned off, because modern browsers do noat permit websites playing any sound before the user interacts with the website [15].
- Triple speed toggle: a toggle button that speeds up the emulator to emulate the GB thrice as fast as usual. This is a common feature found in most emulators.
- Save state: saves the current state of the cartridge in the browser's storage, allowing the user to resume playing the game later. Note this does not save the full state of the emulator, but that of the cartridge, making it equivalent to a real-life save on the GB where only the battery-backed storage of the cartridge persists through power-offs.

5.1.2 Settings

In the side drawer, the first tab is "Settings". It contains general settings for the emulator, such as:

- the console used by the emulator. This may either be the DMG or the GBC.
- an extra filter to be applied to the output. This increases the resolution of the screen,

using the Scale2x or Scale4x[4] algorithms (see Figure 5.4).

- the size of the emulator's screen, allowing resizing to two or four times larger.
- a slider to change the volume of the emulator's audio output.
- a palette selector, to change the four hues of the DMG's screen.
- two buttons to upload the boot ROMs of the DMG and GBC. This is needed for users that wish to play with the full start-up screen of the console, as these two ROMs are copyrighted and cannot be distributed. If they are not provided, the emulator simulates their effect on the system.
- miscellaneous toggle-able settings, grouped together:
  - an option to play with or without the initial boot ROM of the emulator (a boot ROM must have been uploaded).
  - a toggle to enable frame-blending, meaning for every new frame the output is mixed with the previous frame. This is a nice addition to have, because certain games made some objects flicker on screen to make them appear translucent (since the flicker was not visible to the eye).
  - a button to show the performance statistics of the emulator. This is mainly useful when developing the emulator to make sure it is still efficient.
  - a toggle to enable the debug view of the emulator, where the currently loaded tileset and background map are displayed (see Figure 5.5).

Figure 5.4: Output of the GBC with, from left to right: no filter, Scale2x and Scale4x

5.1.3 Watch Expressions

The second drawer allows the user to define custom JavaScript functions to inspect the state of the emulator regularly (see Figure A.4). This requires knowledge of the inner structure of the emulator, as the field names need to be used, but is quite useful when needing to inspect parts

[4]See https://www.scale2x.it/

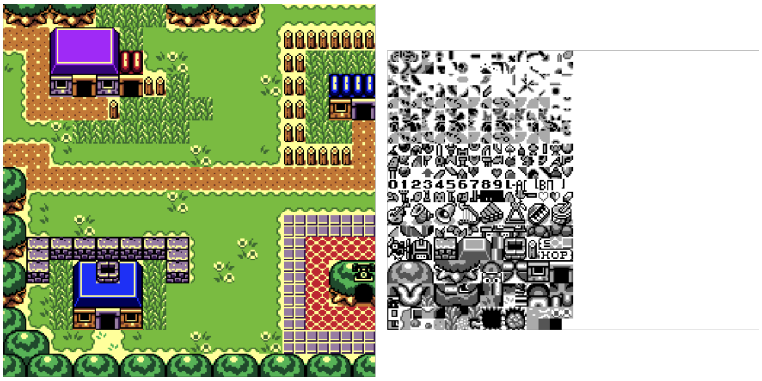

Figure 5.5: Debug view of the emulator

of the console that are not accessible via memory, like internal counters used for components, or register values.

To implement this, the `Function`[5] constructor is used, which takes in a string with the function's code. This allows the user to dynamically change the expression, and the component will simply update the function, without needing to reload the whole application. These expressions are also automatically saved to `localStorage`[6], meaning they will be kept between sessions.

The user-defined function is then repeatedly invoked, and the result output below the expression, allowing for a live-status of the emulator. If an error is thrown by the function (due to a null value, or invalid expression), the error is caught and '`Error`' is displayed.

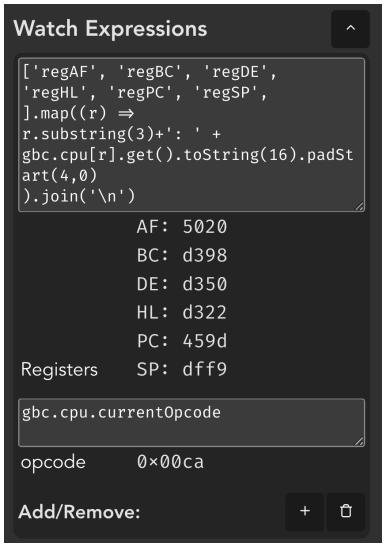

Figure 5.6: Watch expressions drawer

[5] See https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/Function

[6] See https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage

5.1.4 Test ROMs

To ensure emulators work properly, a variety of test ROMs have been made, that test most aspects of the GB (see Gameboy Test ROMs). The front-end of the emulator supports running a large number of them in an automated way (see Figure 5.7). The user can select the group of tests desired by ticking the associated checkbox, or select and unselect everything by pressing the top right button. They can then run them by pressing the "Test" button, making all selected tests run internally (without receiving any input our outputting anything directly). The status of each individual test is displayed, and the user can click on the test name to run it on the main emulator, for further debugging or inspection.

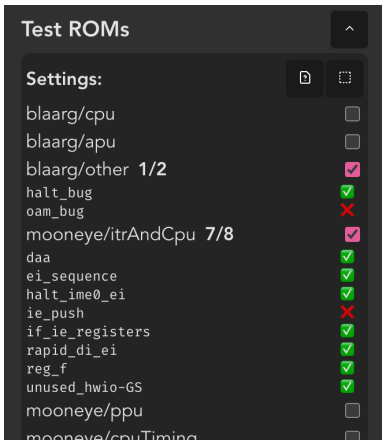

Figure 5.7: Test ROMs drawer

To run the tests, an emulator instance is created, with the test ROM as the input data and "spy" objects provided for input and output – these spies store the output of the emulator, to evaluate the state of the test. To stop execution, the emulator's state is inspected regularly, and the outcome of the test is checked (with execution terminating if the test takes too long to end).

To detect the outcome of each test, these are grouped according to what test suite they belong to. Each test suite is then associated to a *success function*, that can either return `"success"`, `"failure"` or `null` if no outcome has been reached yet. As such, all tests in a test suite have a similar way of reporting success and failure.

The currently automated test suites are:

- The Blaarg test ROMs[7]. The success function of this suite is quite complex, as this suite does not have a standard way of outputting the result. As such multiple parts of the

[7] See https://github.com/retrio/gb-test-roms

emulator are inspected simultaneously:

  - `"Passed"`[8] and `"Failed"`[9] may be output to the console's serial port.
  - If the memory at `0xa001-0xa003` is equal to `0xdeb061`, then the byte stored at `0xa000` is the status of the test[10].
  - For the `halt_bug` test, there does not seem to be anywhere where the result is output, so the graphical output of the emulator is verified.

- The Mooneye test ROMs[11] and SameSuite[12], of which all tests are verified the same way. In case of success, the `B`, `C`, `D`, `E`, `H` and `L` registers hold the values 3, 5, 8, 13, 21 and 34 respectively (Fibonacci's sequence). If they instead all hold the value `0x42`, the test failed[13].
- The Acid test ROMs (DMG[14], GBC[15]). These need to be tested graphically, as their purpose is verifying the actual output of the emulator rather than its behaviour.

Thanks to this, the emulator's frontend supports a total of 191 automated test ROMs, that verify the behaviour of most of the emulator. Of these 191 tests, 101 pass.

5.1.5 Memory Inspect

When debugging an emulator, being able to inspect its memory is essential, as some bugs may be caused by the wrong mapping of components, or a fault when writing data. To help debugging this, the frontend comes with a basic memory inspection tool, that can show the entirety of the *addressable* data of the emulator (see Figure A.6). This means data that is unaccessible by the CPU (for instance, because the appropriate ROM bank is not selected) cannot be inspected here. A simple offset can also be indicated, to limit the data to a certain area.

5.1.6 Keybindings

The user can also customise their keybindings for the emulator, by mapping each input to a separate key (see Figure A.3). This is only relevant on keyboard-equipped devices.

[8] See lines 50-54 of `mem_timing/source/common/testing.s` in https://github.com/retrio/gb-test-roms
[9] See lines 112-139 of `mem_timing/source/common/runtime.s` in https://github.com/retrio/gb-test-roms
[10] See 'Output to memory' in `dmg_sound/readme.txt` of https://github.com/retrio/gb-test-roms
[11] See https://github.com/Gekkio/mooneye-test-suite
[12] See https://github.com/LIJI32/SameSuite/
[13] See https://github.com/Gekkio/mooneye-test-suite/#passfail-reporting for the Mooneye test suite, see lines 265-281 of `include/base.inc` in https://github.com/LIJI32/SameSuite/ for the SameSuite
[14] See https://github.com/mattcurrie/dmg-acid2
[15] See https://github.com/mattcurrie/cgb-acid2

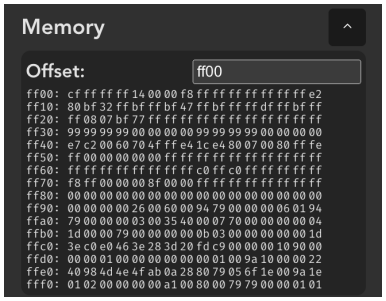

Figure 5.8: The memory inspection drawer

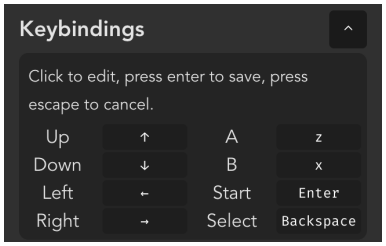

Figure 5.9: The keybindings settings

5.2 Emulator-Frontend Interfaces

To allow the emulator to communicate with the frontend, three components were created, each responsible for part of the output/input.

5.2.1 Screen output handler

To allow easily displaying graphics data from the emulator on the frontend, a `Screen` component was created. This highly configurable screen takes multiple parameters, to allow configuring the screen's size, scale, if it has any upscaling filters applied, if it has any colour palettes, and if it needs to blend frames together (see Figure 5.10).

```
type VideoReceiver = (data: Uint32Array) => void;
type ScreenProps = {
    inputRef: MutableRef<VideoReceiver | undefined>;
    width?: number;
    height?: number;
    scale?: number;
    Filter?: ImageFilter;
    blending?: boolean;
    id?: string;
    palette?: Partial<Record<number, number>>;
};
```

Figure 5.10: Parameters of the `Screen` component

All of the parameters are optional, and use sensible defaults when not specified. The only required parameter is `inputRef`, a modifiable "pointer" to the screen input function. This allows passing data to the screen without re-rendering the whole website. Because pointers do not exist in TypeScript, what is passed is actually an object with a modifiable `value` field.

Whenever any parameter changes, a new *render function* is generated, and is set in the given "pointer". This function takes as input the graphics data (a `Uint32Array`[16], containing ARGB values), and applies the required transformations to it. The function also holds a reference to two backing data buffers, for the previous and the current frame. This allows for blending between frames, and avoids creating new arrays repeatedly, lowering the memory consumption of the screen. The transformations it applies are:

1. Roll the buffers, setting the previous frame to the current frame, and the current frame to the newly received frame.
2. If needed, apply the colour palette to the input. The palette is a simple map, from source colour to target colour.
3. If it is the first frame to be drawn, also set the previous frame to the newly received frame (this avoid blending the new frame with a black frame, since the buffers are initialised with `0x000000` values).
4. If required, blend the previous frame and the current frame.
5. Apply the upscale filter to the image (this can be an identity filter, that simply returns the image, or upscale filters like Scale2x and Scale4x).
6. The image data is created from the `Uint32Array`, and drawn on the `<canvas />`.

This simple pipeline of functions makes the code really easy to understand and potentially modify, and allows for easy extension, as any filter could be given to the screen and it would work seamlessly. It is also of interest to note that the order in which the operations are applied maximises efficiency. For instance, the filter is applied last, to avoid having to do previous operations like blending on much larger images.

5.2.2 Audio output handler

To output the sound generated by the emulator, a utility class `AudioPlayer` was made. It is a simple class, that creates an `AudioContext`[17] instance on creation. This class allows playing

[16] See https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Uint32Array

[17] See https://developer.mozilla.org/en-US/docs/Web/API/AudioContext

sound output from the browser.

A difficulty with playing audio data stems from the fact that audio samples are quite short (they are output every frame, so every 16.6 milliseconds), and they must be constantly output, with tolerance for small delays in the output. If there happens to be even a small gap between two consecutive samples, a audible pop might play, which results in a very unpleasant experience.

To avoid this, `AudioPlayer` actually adds a small delay before starting to play audio. This ensures that by the time the first received sample is done playing, a new sample has already been received and queued for output, meaning there are never gaps between samples (except if there is an exceptionally long delay between samples, which should not happen).

A drawback from this solution however is that if samples are received too fast, a delay may create itself, as too many samples are enqueued, and new samples need to wait for all the previous ones to be played before being played itself. To avoid having this delay grow too large, a parameter of `AudioPlayer` allows setting the max size of the enqueued audio samples (the frontend uses a value of 8) – any samples that would make the queue exceed this size are ignored, meaning the max delay of the playback is $8 * \frac{1}{60} = 0.13$ seconds, a reasonable delay.

5.2.3 Input handler

To provide a simple way of handling inputs whether the device has a keyboard or uses touch controls, a `GameInput` component was created. It handles keyboard inputs, via a helper hook[18], `useKeys`. To handle touch screen inputs, this component creates buttons mimicking those of the GB. These buttons all have the `mobile-only` class, which is defined in `main.css` to only be visible when no fine pointer is available on the device (ie. no mouse is detected), as can be seen on Figure 5.11.

```
@media (any-pointer: fine) {
    .mobile-only {
        display: none;
    }
}
```

Figure 5.11: `mobile-only` CSS class

This means that if the user is on a mobile devices, the buttons will show, allowing for input without the need for a keyboard (see Figure 5.12).

[18] See https://reactjs.org/docs/hooks-intro.html

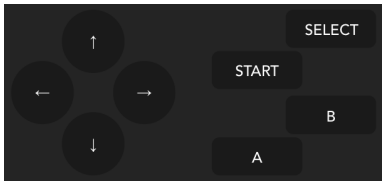

Figure 5.12: Buttons for mobile devices

The component itself then takes as a parameter a callback, that is called with the input function created by this component and returns an up to date object with the currently read inputs `GameBoyInputRead` (see Figure 5.2).

5.3 CPU

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

The Game Boy's clock ticks at $2^{22} \approx 4.19$MHz, with each of these clock cycles being commonly referred to as T-cycles or T-states [9, CPU Instruction Set]. However, because all of its components tick every four T-cycles, timings of the GB are divided by four and called M-cycles. In other words, four T-cycles make for one M-cycle, and M-cycles have a frequency of $\frac{2^{22}}{4} = 2^{20} \approx 1.05$Mhz [13].

5.3.1 Instruction Set Decoding

Because the Game Boy 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 'unused' opcodes, that will lock the console when used[9, CPU Comparison with Z80], 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 possible operations. This is the most straightforward option, but can result in quite large switch-statements, especially for CPUs with more opcodes. The GB has comparatively few opcodes as it is an 8-bit system; the Gameboy Advance, the console that followed it, had 32-bit opcodes: way too many for a switch statement to be appropriate.
- Decode the operation by reading specific parts of the opcode, and generate the instructions

dynamically. This method is what is done for larger instruction sets, where opcodes can be split into separate parts to describe the operation's behaviour [16, ARM CPU Reference]. It however comes with the cost of extra processing for each instruction, as it needs to be decoded. This method is applicable to the GB, as many instructions follow a pattern – see, for instance the `LD` instructions, that all follow the same order of registers: `B`, `C`, `D`, `E`, `H`, `L`, `(HL)` (the byte at address `HL`) and `A` [17].

- Create a dispatch table: a map that associates each opcode to a function to execute [18]. This technique is also only viable for small numbers of opcodes, as the map can result quite large. An advantage of this method is that the map does not have to be explicitly written out entirely – generators, or macros, can be used to populate some chunks of it, making it a hybrid of the two previous solutions: there is very little overhead to execute an instruction, as each opcode is associated to a function, but there is also no need to write out every instruction separately, as the map can be generated in parts or in its entirety (inducing a light setup cost).

Initially, the emulator had a large map, with all the opcodes as keys. The functions associated would then execute the instruction and return the number of cycles taken by the instruction (see Figure 5.13). This however proved quite repetitive and prone to errors. To solve this, a *generator function* was created. To use it, two parameters must be given: a map of opcodes to values (of an arbitrary type), and a helper function that for each arbitrary value returns a function that executes the instruction (see Figure 5.14). This allows generating repetitive instructions more easily, by only specifying what opcodes and objects are used, and not what the whole body of the instruction is (see Figure 5.15). This method is used in other emulators – for instance, SameBoy has a map that has an opcode-function mapping for all opcodes, and uses macros to generate the different functions[19].

```
protected instructionSet: Partial<Record<number, InstructionMethod>> = {
    // 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; },
    ...
}
```

Figure 5.13: Initial instruction set implementation

[19] See https://github.com/LIJI32/SameBoy/blob/master/Core/sm83_cpu.c

```
protected generateOperation<K extends number, T>(
    items: Record<K, T>,
    execute: (r: T) => InstructionMethod
): Record<K, InstructionMethod> {
    const obj: Record<K, InstructionMethod> = {};
    for (const [opcode, item] of Object.entries(items)) {
        obj[opcode] = execute(item);
    }
    return obj;
}
```

Figure 5.14: Generator function used for the CPU

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

Figure 5.15: Improved instruction set implementation

5.3.2 A cycle accurate CPU

The Gameboy is a memory-bound system, meaning that it is limited by its memory accesses. The CPU can only execute either one read or one write per M-cycle [19, CPU core timing]. It also needs to retrieve the opcode for each instruction, which takes an additional cycle, meaning an instruction performing no memory accesses lasts one cycle, and an instruction performing n memory accesses lasts at least $n + 1$ cycles. Finally, the GB overlaps the last cycle of the execution with the fetching cycle for the next opcode – this will thus be of importance when implementing the fetching of the opcode. See Figure 5.16 for the breakdown of an instruction.

The current implementation is not M-cycle accurate: the CPU instruction is executed as one monolithic block, rather than in different smaller parts. This becomes crucial when the timer, OAM and PPU are involved, as they run in parallel with the CPU, so memory accesses

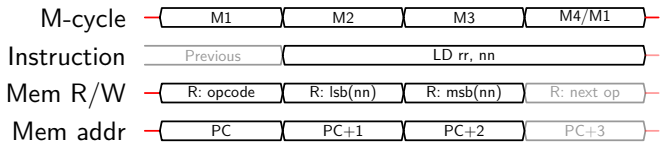

Figure 5.16: Timing of the `LD rr, nn` instruction
Taken from GBCTR [19, Sharp SM83 instruction set]

to these components may return different values depending on the M-cycle.

In all emulators that were found when researching for this project, M-cycle accuracy is reached by making the emulator "CPU-driven". What this means is that inside each instruction, between each 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 5.17).

Some emulator where this method was found:

- In Mooneye GB, this can be seen by the usage of the `CPUContext.tick_cycle()` method when appropriate[20].
- In accurateboy, the `Bus.tick()` method is called from within the instructions (or it is done implicitly by methods such as `Bus.read()`)[21].
- In Sameboy, the CPU uses functions like `cycle_read` or `cycle_no_access` to tick the rest of the system[22].

```
1 protected ld_bc_d16(system: System) { // LD BC, d16
2     const lower = system.read(this.regPC.inc());
3     system.tick();
4     const upper = system.read(this.regPC.inc());
5     system.tick();
6     this.regBC.set(upper << 8 | lower);
7 }
```

Figure 5.17: CPU-driven `LD BC, (HL)`

This solution however comes with the cost of coupling the CPU to the system, or at least to a way of ticking said system. A particularity of this emulator is that the CPU is almost autonomous, in that it does not interact with any other functionality of the system directly, except the interrupts. The only other interface it requires is an `Addressable`, to allow access to memory (and thus the other components). This also means the emulator is no longer "CPU-

[20] See https://github.com/Gekkio/mooneye-gb/blob/master/core/src/cpu.rs
[21] See https://github.com/Atem2069/accurateboy/blob/master/accurateboy/CPU.cpp
[22] See https://github.com/LIJI32/SameBoy/blob/master/Core/sm83_cpu.c

driven", since the CPU *cannot* tick the rest of the system. It is instead up to the root system to tick all of the components (including the CPU). This is thus a code quality improvement, that is not present in other emulators.

This was done by splitting all the instructions into their respective steps, and have each step return the next step (or `null` if it is the last step). The CPU now must simply store whatever the instruction returns: if it is `null` then it needs to fetch an instruction to prepare for the next cycle, otherwise it is a function and must be executed (and its result stored for the next step). See Figure 5.18 and Figure 5.19 for an example of this.

In the final version of the CPU's tick method, a method `loadNextOp` is implemented. It not only handles getting an instruction associated with the desired opcode, but it also is responsible for checking if an interrupt was raised (in which case the interrupt handling procedure occurs [9, Interrupts]), and for ensuring the CPU is not halted before running an instruction.

```
protected ld_bc_d16(system: Addressable) { // LD BC, d16
    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 5.18: System-driven `LD BC, d16`

```
step(system: Addressable, interrupts: Interrupts) {
    if (this.nextStep === null) {
        // looks up opcode in instruction table
        const nextStep = this.loadNextOp(system, interrupts);
        if (nextStep === "halted") return;
        this.nextStep = nextStep;
    }
    this.nextStep = this.nextStep(system, interrupts);
    if (this.nextStep === null) {
        // opcode is fetched on the last cycle of execution
        this.currentOpcode = system.read(this.regPC.inc());
    }
}
```

Figure 5.19: System-driven step of the CPU
Note how the opcode is fetched directly when the next instruction is over, to emulate the overlap between the fetch and the execute steps.

5.4 System

The `System` class implements the system bus, as well as the ticking of all other components. It handles the ticking of the PPU, timer, APU and interrupt logic. 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 implements `Addressable` (see Figure 4.6), and internally has a `getAddress` method that returns the `Addressable` at this specific address, to be accessed in the `read` and `write` methods. This ensures that the logic used to determine what component is accessed depending on the address is not duplicated.

All components of the Game Boy exist is the same address space. The *memory map* is what describes the allocation of different components to different areas of memory.

5.4.1 `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: for a game like "The Legend of Zelda: Link's Awakening DX", the method is called around 650,000 times per second. For a more complex and resource-intensive game such as "Alone in the Dark: The New Nightmare", around 1,900,000 calls are made. This intensity in usage can easily be explained, as the GB is a memory-bound system: almost all interactions between components occur through memory reads and write. `getAddress` must thus be optimised as much as possible, as it is one of the main bottle-necks of the emulator.

An initial implementation of `getAddress` used the combination of a list of if-statements for different ranges of the memory map, as well as a map where all register addresses where mapped to the component responsible for them. The code would first check if the key exists in this map, and if not it would then go through a series of if-conditions (see Figure 5.20). It would return a tuple, containing both `Addressable` to use and the address within in – this was needed as some components had a particular mapping to memory. This is the case for the Work RAM (WRAM), who's memory starts at `0xC000` and ends at `0xFDFF` (spanning over 15.5KB bytes) despite it only being 8KB long, because the last 7.5KB of its address range map back to the beginning of it (this is called the Echo RAM [20]).

This proved quite costly:

- This code creates a new map every time it is called, when checking for the registers'

```
protected getAddress(pos: number): AddressData {
    const register = {
        0xff00: this.joypad,
        ...
        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 5.20: Initial implementation of `getAddress`

addresses.

- Having to return both an `Addressable` and an address is quite costly, as a new array with two items must be created every time. It also adds unnecessary complexity to the method, as the system bus should not be responsible for handling the details of address mapping, and instead simply delegate the task to the appropriate component.
- The if-conditions for the ranges of the biggest areas of memory (everything between `0x0000` and `0xEFFF`) happened after the register checks, which delayed response for these reads and writes. This is all the more important that this range represents $\frac{\texttt{0xEFFF}}{\texttt{0xFFFF}+1} \approx 93\%$ of the total address space.
- Chaining if-conditions is inefficient, 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 been handled.

To fix these issues, we may first move the fine-grained addressing logic to sub-components like the WRAM. This removes the need to return an address from the method: only an `Addressable` is enough, as that component will then be responsible for decoding the address further.

The last two points may then be addressed, by removing the use of if-conditions, and moving this part of the code to the beginning of the function.

With the memory map of the console (see Table 5.1) we can notice how the largest chunks

of memory all end with the last three nibbles of the address as `0xFFF`. This means that the component mapped to an address can be determined by simply looking at the first nibble of said address. This is probably done to simplify the circuitry responsible for addressing, as only the most significant 4 bits need to be compared. The emulator can take advantage of this.

Start	End	Description
0000	3FFF	16 KB ROM bank 00
4000	7FFF	16 KB ROM Bank 01∼NN
8000	9FFF	8 KB Video RAM (VRAM)
A000	BFFF	8 KB External RAM
C000	CFFF	4 KB Work RAM (WRAM)
D000	DFFF	4 KB Work RAM (WRAM)
E000	FDFF	Mirror of 0xC000∼0xDDFF

Table 5.1: Memory map for the largest chunks of memory [20]

For this area of memory (`0x000-0xEFFF`) the system bus may simply isolate the most significant nibble, and then use a map to associate each address block to a component. Only if the address corresponds to `0xFF--` do we try matching it to a register address. This removes the need for the if-conditions, and is also faster as it is evaluated much earlier on (see Figure 5.21). The map can be created on instantiation and kept for later use, to avoid unnecessary memory allocations.

```
protected getAddress(pos: number): Addressable {
    // Checking last nibble
    let addressable = this.addressesLastNibble[pos >> 12];
    if (addressable) return addressable;

    // Registers
    if((address & 0xff00) === 0xff00) {
        addressable = this.addressesRegisters[pos & 0xff];
        if (addressable) return addressable;
    }

    if (pos <= 0xfdff) return this.wram;  // Echo RAM
    if (pos <= 0xfe9f) return this.ppu;   // OAM
    if (pos <= 0xfeff) return Register00; // Illegal Area

    return RegisterFF; // fake register
}
```

Figure 5.21: Optimised implementation of `getAddress`

To ensure placing the "main block" first was the best choice, measurements have been taken of four different GB games. The emulator would log all memory accesses by groups of $100\,000\,000 = 10^8$, and categorise them by address "blocks": the main block, the OAM block, the illegal block (called like this because this area of memory is restricted by Nintendo, and

only returns `0x00`) and the register block. This separation is justified by the fact these are the five chunks of memory that must be checked separately, due to their irregular boundaries. The *second* set of 10^8 accesses was then used to gather the statistics of what blocks are used the most. The first 10^8 accesses are not used, as they may include setup operations that only happen when the game loads, and as such do not represent what the average execution will look like. See Table 5.2 for the results.

Name	Memory Area	Game 1	Game 2	Game 3	Game 4
Main Block	`0x0000-0xEFFF`	88.869%	95.679%	84.289%	95.740%
Echo RAM	`0xF000-0xFDFF`	0.000%	0.000%	0.000%	0.000%
OAM Block	`0xFE00-0xFE9F`	0.001%	0.000%	0.000%	0.000%
Illegal Block	`0xFEA0-0xFEFF`	0.001%	0.000%	0.000%	0.000%
Register Block	`0xFF00-0xFFFF`	11.129%	4.321%	15.711%	4.260%

Table 5.2: Access rate of memory blocks
Sample taken from the second set of 10,000,000 accesses of 4 different games.
Games are, respectively, "Alone in the Dark: The New Nightmare", "The Legend of Zelda: Link's Awakening DX", "Tetris" and "Pokémon Silver".

As we can see, the vast majority of memory accesses go to the main part of memory, with almost all of the rest going to the "register area", the last 256 bytes of the address space. This can easily be explained by the fact all registers that allow interaction with other components (the PPU, the APU, the timer, etc.) are in this narrow range, so it is bound to have a high usage. It is also quite interesting to note that neither the Echo RAM or the OAM block are used at all, except very rarely for one of the tested games. It is thus safe to assume that the conditions responsible for mapping these areas can be left at the end of the function, as they will rarely match an address.

5.4.2 Evaluating the `getAddress` optimisation

A simple experiment was then run, to verify the performance improvement. The first 25 million instructions of the `cpu_instrs`[23] test ROM were run. This sample was chosen because it is considerably large and because the test itself requires around 25 million instructions to complete. For the measurement, `window.performance.now()`[24] was used before and after each drawn frame, and the values were then summed.

The result was the following: 33 955.9ms before the change, and 20 039.1ms after the change. The relative difference is thus $\frac{20\,039.1-33\,955.9}{33\,955.9} = -0.4098$, thus reducing time taken by 40.98%. By measuring the time spent within `getAddress` for these 25 million instructions, we get a total

[23] See https://github.com/retrio/gb-test-roms/tree/master/cpu_instrs
[24] See https://developer.mozilla.org/en-US/docs/Web/API/Performance/now

spent time in the method of 0.000 151ms on average, showing that this method is no longer a bottleneck for the system, as its impact on performance is minimal.

5.5 PPU

The Picture Processing Unit (PPU) is the component responsible for rendering the game onto the Gameboy's screen. It is one of the most complex components of the GB, with intricate timings, and a behaviour that changes between the DMG and the GBC, due to the addition of colour-support, as well as Video RAM (VRAM) banking.

5.5.1 Presentation

The screen rendering is divided intro three layers (see Figure 5.22), drawn on top of each other. From bottom to top, these are:

- The background, a 256×256 image loaded into memory that support scrolling on both axis.
- The window, similar to the background, is a 256×256 image that can be moved around the screen, and is toggle-able. It however does not support scrolling.
- The objects, that are drawn at the very top, are smaller 8×8 tiles. These can be moved freely, and support some transformations, such as horizontal or vertical flipping. Their data is stored in the Object Attribute Memory (OAM).

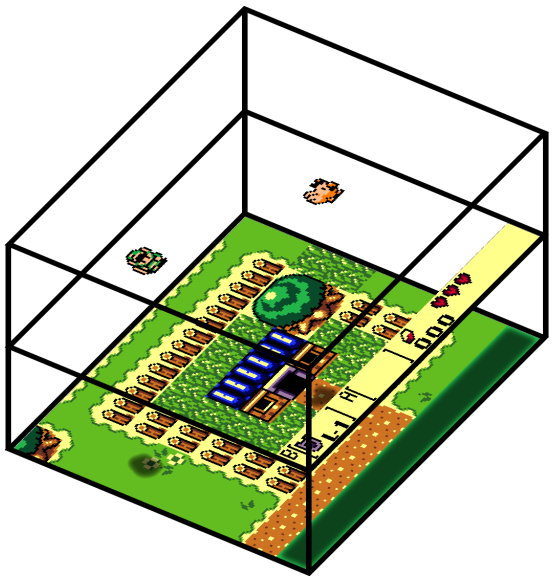

Figure 5.22: Layers of the PPU
From top to bottom, the objects, the window and the background

It renders onto the screen line by line, meaning it will draw a 1-pixel wide line (the scanline) from left to right, and then proceed to the line below. Drawing an entire line takes 114 M-cycles, and there is an 1140 M-cycle delay when the bottom of the screen is reached called the VBlank. The advantage of having per-line rendering is that by modifying the position of the background between each drawn line, complex visual effects can be obtained quite easily [21], see Figure 5.23 for an example.

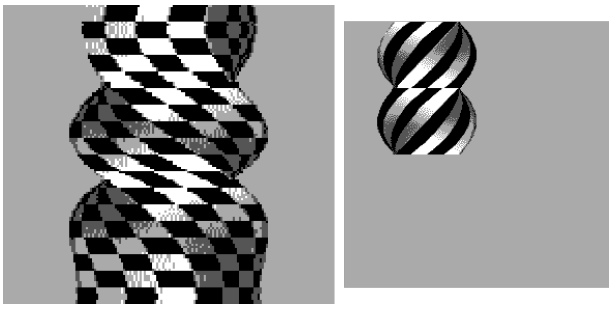

Figure 5.23: Visual effects made by scrolling the background
Gameboy output (right), and loaded background data (right)

One of the main characteristics of the PPU is that it operates in modes. During each mode, it perform a different set of operations, and can only be interacted with in certain way [9, LCD Status Registers]. These four modes are, in order:

- Mode 2: the PPU looks through the OAM, to look for all the sprites to draw. During this time, the OAM is not accessible.
- Mode 3: the PPU reads both the VRAM and OAM, and draws the line. None of the video data is accessible here.
- Mode 0: the PPU does nothing for a short length of time after each line. An interrupt is raised at the start of this period, allowing the CPU to be notified that now is the time to update what's rendered if necessary. This is called the horizontal blank, or HBlank.
- Mode 1: the PPU does nothing for a long length of time after the bottom of the screen is loaded. This is usually where the bulk of the graphics update go, as it lasts a considerable amount of time (1140 M-cycles). An interrupt is also raised here, to notify the CPU. This is called the vertical blank, or VBlank.

Rendering of the line is thus done during mode 3, when no graphics data is accessible, using a FIFO queue (First In First Out) of pixels that can be pushed in and out as the data is read. Some of the registers are still writable, but most games do not use them during this mode because they would require very precise timing. An example of a game relying on this is "Prehistorik Man", that changes the colour palette registers during the scanline in its

intro-scene [8, Tricky-to-emulate games].

Since this is only the case for a minority of games, we can go past this inaccuracy. We will thus take advantage of this to simplify greatly the rendering logic and make the renderer "scan-line based", drawing an entire line all at once. This shortcut is commonly used by emulators that are not too accuracy-oriented.

To handle each mode separately, we have four distinct "mode" objects of type `PPUMode`, that hold some basic information on the operation of the mode: its length, its flag for the `STAT` register and the name of the method in `PPU` responsible for ticking said mode (see Figure 5.24). Note `KeyForType<T, V>` is the union of all keys `k` of `T` such that `T[k]` is of type `V`.

```
type PPUMode = {
    doTick: KeyForType<PPU, (interrupts: Interrupts) => void>;
    flag: number;
    cycles: number;
};
```

Figure 5.24: Type definition of `PPUMode`

5.5.2 Rendering Logic

The first step needed to render the game is to select and order all the sprites to be shown on screen. One of the limitations of the GB is that despite having enough space in memory for 40 sprites, only 10 may be rendered at a time in a scanline [9, OAM]. The sprites are selected during mode 2 – as such, during the last cycle of mode 2, the emulator will look through the available sprites in the OAM, and select the appropriate ones. This can be done quite elegantly in a functional manner, using an array (see Figure 5.25).

```
this.readSprites = this.oam
    .getSprites()
    .filter((sprite) => sprite.y <= y && y < sprite.y + objHeight)
    .slice(0, 10)
    .map((sprite, index) => [sprite, index])
    .sort(objPrioritySort)
    .map(([sprite]) => sprite);
```

Figure 5.25: Retrieve the selected sprites for the scanline

First, we retrieve the sprites from the OAM. Internally, sprite data is cached between scanlines and invalidated when written to – `getSprites()` updates the dirty tiles in the cache and returns them. This avoids decoding the sprite every line if that tile has not received any changes, while also ensuring excess decoding is not done if the sprite is modified twice between

scanlines. We then select the first 10 sprites to be part of the scanline – this selection is done based on index: the PPU looks through the OAM sequentially to find matching sprites [9, OAM]. Finally, a re-ordering of the sprites needs to be done, to determine which sprite goes above which – this may depend on the sprite's position in the OAM, or on its X coordinate. This is handled in `objPrioritySort`. To allow sorting based on both attributes, the sprites must be briefly remapped to a tuple with the sprite and its index (as JavaScript does not expose the indices of elements when ordering them). These sprites are then stored in the PPU object, to be used when rendering at the end of mode 3.

The PPU uses a form of indirect addressing to handle data (see Figure 5.26). It has an area in VRAM that stores tile data, a tile being an 8×8 image. Whenever one of these tiles needs to be used, for the background, or for an object, the identifier of the tile needs to be used, this identifier being derived from the tile's address. This allows for the very simple re-use of tiles, which is particularly relevant for backgrounds where they may be repeated a lot. This background data is stored in a *tile map*, a 256×256 map of 1-byte indices to the actual tile data. The GB has two such maps, and both the background and window can display either of them – this is controlled via a flag in the `LCDC` register.

To render the scanline, the layers need to be drawn on top of each other: background, then window, then objects. Due to the similarities between background and window, a `drawLayer` method can be shared to handle the bulk of the work.

This method will loop over all tiles it needs to draw in the current scanline. It first needs to determine the tile's index. This index can be used to the retrieve the tile's address in VRAM. It can also be used to fetch the attributes of the tile; this is a GBC-exclusive feature, that allows transforming tiles (for instance, flipping them, or selecting a different palette for the tile). These attributes are stored in the second bank of VRAM, at the same address, in a single byte of data. The DMG and GBC distinction is however not needed, because the attributes for a `0x00` value match the attributes used by the DMG. This means that instead of having two different methods split on the console versions, we can setup the DMG emulator to always return `0x00` for the second bank of VRAM, and keep the GBC way of handling of attributes.

Once the tile index has been acquired, the PPU may retrieve the tile data address from VRAM. This one-byte address needs to be converted to a valid address in the tile data range of VRAM, as it contains two partially overlapping tile data areas – a flag in the `LCDC` register controls this. Once the address where the tile data (ie. its texture) is obtained, the PPU may finally retrieve said data, and draw it in the scanline.

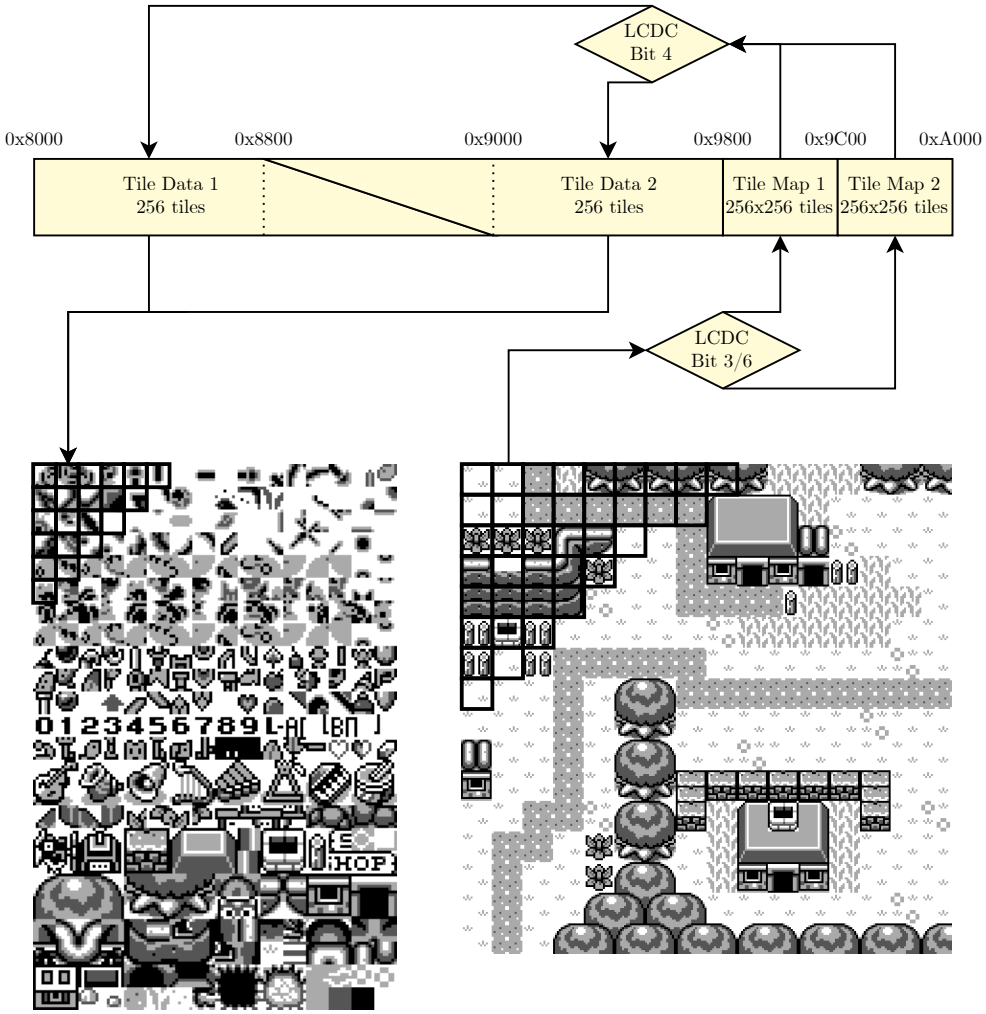

Figure 5.26: PPU logic to obtain tile data
This applies to the DMG – the CGB has two VRAM banks, thus having twice as many tile textures, and an additional tile attribute map.

This tile data, an 8×8 images where each pixel can be one of four shades is encoded in 16 bytes [9, VRAM Tile Data]. Because tile data is rarely changed and the procedure to extract the image data from these 16 bytes is quite complex and requires some bit manipulation, tiles are cached.

5.5.3 Subcomponents

Although it is presented to the system as a unit, the PPU is implemented as multiple smaller components, that handle different parts of the screen logic. This is all the more important that the PPU is extremely complex, so care is needed to ensure the code remains maintainable.

The Object Attribute Memory (OAM) is the memory area in the PPU that stores object

data. This small 160-byte long area has a feature called OAM Direct Memory Access (DMA), allowing for direct transfers from ROM or RAM to the OAM. Because this transfer runs in parallel to the other components, the OAM itself is its own component, contained inside the PPU. This allows for a better separation of concerns between the two: the PPU handles rendering, the OAM handles OAM DMA and sprite storing and decoding.

The colour management of the PPU also requires some extra logic that can be extracted into its own class. Although on the DMG this is limited to splitting a byte into 4 shades of 2 bits (white, light grey, dark grey and black), the GBC has 16 palettes of 4 colours, 8 palettes for the background and window, and 8 for sprites [9, Palettes]. As such, a `ColorController` class was created: it implements `Addressable`, and has a method to retrieve the palette for the background, and a method to retrieve the palette of a sprite. This class is extended by a `DMGColorController`, and a `CGBColorController`. The `PPU` picks one of the two on instantiation, based on the emulated console, and can then use both, without needing to know if the screen is monochrome or coloured.

Similarly, the VRAM behaves slightly different between the DMG and the GBC: the latter's VRAM has two banks it can switch between, via an additional register, and also supports VRAM DMA transfers, similar to what is possible with the OAM. A `VRAMController` class is created: it also implements `Addressable`, and has a few extra methods, to allow getting tiles from it, and reading each bank individually (see Figure 5.27).

```
abstract class VRAMController implements Addressable {
    read(pos: number): number;
    write(address: number, value: number): void;

    tick(system: Addressable, isInHblank: boolean, isLcdOn: boolean): boolean;

    getTile(tileAddress: number, bankId: 0 | 1): Int2[][];
    readBank0(pos: number): number;
    readBank1(pos: number): number;
}
```

Figure 5.27: Interface of `VRAMController`

5.6 APU

The Audio Processing Unit (APU) is the component responsible for producing the sounds and music of the GB. Its output is the combination of four different channels, each with their own properties: two square channels, a wave channel and a noise channel [9, Audio]. These channels

run independently, can be turned on or off individually, and are merged to form the output.

Most features of the channels can be derived to a counter, that ticks down from a value and has an effect when reaching 0. Although they differ in output, all channels have some attributes in common. For example, they all have a *length timer*, that turns off the channel when it reaches 0. Channels 1, 2 and 4 also have an *envelope*, that allows updating the volume of the channel at a set frequency. To have these behaviours work across all channels, an abstract class `SoundChannel` was created. It holds the registers all channels have in common, as well as methods like `tickLengthTimer` to handle common mechanisms across all channels.

To return their output, all channels have a `getSample` method that returns the current output value of the channel, a value between 0 and 15 (4 bits). This output can then be combined, and output to the frontend to handle. It is of interest to note that the GB is always outputting sound, with a resolution of 1 M-cycle, ie. 2^{20}Hz $\approx$ 1.05MHz. This is significantly more than the sample rate computers usually use, 44.1Hz. To fit to this standard, the emulator's APU has an internal counter that ticks down and only produces a sample every $\frac{2^{20}}{44.1*10^3} = 23.8$ cycles. This is not the most accurate way of producing the audio data, but is far simpler and faster than downsampling the audio. An example of very accurate emulator that does downsampling is SameBoy[25].

5.7 MBCs and ROMs

A majority of GB cartridges came shipped with a Memory Bank Controller (MBC). This was done to circumvent the memory limit of the GB: due to its 16-bit addresses, only memory in `0x0000`–`0x7FFF` is mapped to the ROM, limiting its size to 32KB. MBCs provided a way of extending this memory limit, by doing *bank switching* [9, MBCs]. The cartridge holds more data than it can address, and the CPU can modify which part of the memory it is accessing by writing to a register in the MBC. Because ROMs are read-only, the write-operations can be safely re-used to instead write to these built-in registers without losing any functionality.

On top of more memory space, MBCs can sometimes provide additional features, such as external RAM, a battery (to be able to keep the state of the RAM between sessions, which allows saving the game) [9, MBCs], a rumble motor [9, MBC5], or a battery-backed real time clock [9, MBC3], for example. The most common MBCs found were the MBC1, with space for a maximum of 2MB [9, MBC1], and the MBC5 – the only MBC to officially support the GBC's double speed mode [22]. See Table 5.3 for statistics of the usage of different MBCs.

[25] See Accuracy, https://sameboy.github.io/features/

Name	ROM Count	Percentage
No MBC	2150	23.0%
MBC1	4010	43.0%
MBC2	227	2.4%
MBC3	367	3.9%
MBC5	2532	27.1%
Others	53	0.6%

Table 5.3: Statistics of different MBCs [23]

To ensure the emulator can run as many games as possible, the main MBCs have been implemented: MBC1, MBC2, MBC3 and MBC5 (although support for the MBC3's real time clock has not been added). Since the GB interacts with the cartridge in the exact same way whether there is an MBC or not, the choice of making a `MBC` abstract class came quite naturally: its interface is that of `Addressable`, and it also supports a `save` and `load` method, for save files.

To decide which MBC to use, we have a `GameCartridge` class, that wraps around `MBC`. Its role is decoding the cartridge's header, contained in `0x0100-0x014F` [9, The Cartridge Header], to decide on which MBC to use, as well as some of the properties of the cartridge: does it have RAM, is it battery backed (in which case it supports saving), what's the game's title and identifier.

The `read` and `write` methods received an optimisation similar to what was done with `System.getAddress`: because the internal addressing logic of the MBCs only relies on the most significant nibble, a simple switch statement can be made (see Figure 5.28).

Internally, the `MBC`s have a `ROM` instance to store the cartridge data (this is obtained from the ROM uploaded by the user), as well as an optional `RAM` instance. This RAM is what the game can edit. If it is battery backed, memory is kept when the GB is turned off, allowing data like scoreboards and progress to be saved. To replicate this saving behaviour, the emulator exposes a `save` and `load` method, that respectively return and set the data in the RAM. The emulator's core is thus not responsible for handling these saves, and the frontend can decide freely how to store them.

In the implemented frontend, this is done by saving the RAM data in the browser's available storage, using the "localForage" library[26]. When a ROM is loaded, the frontend checks if a save for this game exists, by using the ROM's identifier decoded in the `GameCartridge` class. If it does, the save is then loaded. Similarly, when changing ROMs, closing the window, or pressing the "Save" button, the frontend retrieves the RAM data from the emulator and saves

[26]See https://github.com/localForage/localForage

```
read(pos: number): number {
    switch (pos >> 12) {
        case 0x0: // ROM bank 00
        case 0x1:
        case 0x2:
        case 0x3:
            return this.data[pos & addressMask];
        case 0x4: // ROM bank 01-ff
        case 0x5:
        case 0x6:
        case 0x7: {
            const address =
                (pos & ((1 << 14) - 1)) |
                (this.romBankLower8.get() << 14) |
                (this.romBankUpper1.get() << 22);
            return this.data[address & addressMask];
        }
        case 0xa: // ERAM
        case 0xb: {
            if (this.ramEnable.get() !== RAM_ENABLED) return 0xff;
            const address = this.resolveERAMAddress(pos);
            return this.ram.read(address);
        }
    }
    throw new Error(`Invalid address`);
}
```

Figure 5.28: `read` method of `MBC5` [9, MBC5]

it, by setting the key of the entry in the local storage to the identifier of the ROM.

5.8 Timer

The timer is a component in the GB that ticks regularly. It allows the control of two independent mechanisms [9, Timer and Divider Registers].

First is the *divider counter*, accessed and controlled via the `DIV` register. This counter is internally 16-bits, although only the upper 8-bits can be accessed. It is incremented every clock cycle (ie. increased by 4 every M-cycle), can be read, and writing to the divider resets the counter (see Figure 5.29).

The second, more complex part of the timer is a *customisable timer*. It is made of three registers:

- `TIMA`: the timer counter. Every time a falling edge is detected on one of the bits of `DIV` (ie. the bit goes 1 to 0), this register is incremented. What bit is inspected can be controlled via the `TAC` register, effectively changing the frequency of the timer. When this register

```
protected divider = new DoubleRegister(0xab00);
protected addresses: Record<number, Register> = {
    0xff04: this.divider.h, // we only ever read the upper 8 bits
    ...
};

tick(interrupts: Interrupts): void {
    const newDivider = wrap16(this.divider.get() + 4);
    this.divider.set(newDivider);
    ...
}

write(pos: number, data: number): void {
    if (pos === 0xff04) { // Writing anything to DIV clears it.
        this.divider.set(0);
        return;
    }
    ...
}
```

Figure 5.29: Implementation of divider counter

overflows (is incremented when equal to `0xFF`), an interrupt is raised.

- `TAC`: the timer control register. It allows enabling or disabling the timer, and changing the inspected bit of `DIV`.
- `TMA`: the timer modulo. It defines what value `TIMA` is reset to when overflowing.

Although it is a seemingly simple system made of 3 registers, the behaviour of this system is quite complex, as it has multiple edge cases [9, Timer obscure behaviour]. The code that handles the increase of `TIMA` is thus quite long, and required some re-factoring to be easily readable (see Figure 5.30). As can be seen from this implementation, the `TIMA` register can actually be incremented for two reasons (line 16). The first reason is the regular falling edge on the inspected bit. However, if the timer is disabled then the bit of `DIV` read is going to be 0, regardless of its actual value. This means that if the timer gets disabled while the read bit of `DIV` is 1, the timer detects a falling edge and increases `TIMA`. It is also of interest to note that `TIMA` is not set to the value of `TMA` when overflowing – this is actually done the following cycle (lines 2–9). This logic requires some extra fields to be present, recording the values of the registers on the previous tick. Some additional logic is required in the `write` method, because writes to `TIMA` are ignored when the timer overflows.

```
tick(interrupts: Interrupts): void {
    this.previousTimerOverflowed = false;
    if (this.timerOverflowed) {
        const modulo = this.timerModulo.get();
        this.timerCounter.set(modulo);
        interrupts.requestInterrupt(IFLAG_TIMER);
        this.timerOverflowed = false;
        this.previousTimerOverflowed = true;
    }

    const timerControl = this.timerControl.get();
    const timerIsEnabled = timerControl & TIMER_ENABLE_FLAG;
    const speedMode = (timerControl & 0b11);
    const checkedBit = TIMER_CONTROLS[speedMode];

    const currentBitState = timerIsEnabled && (newDivider >> checkedBit) & 1;

    if (this.previousBitState && !currentBitState) {
        const result = (this.timerCounter.get() + 1) & 0xff;
        this.timerCounter.set(result);
        if (result === 0) this.timerOverflowed = true;
    }

    this.previousBitState = currentBitState;
}
```

Figure 5.30: Code to manage the `TIMA` increments

5.9 Helpful Components

To avoid repeating logic throughout the main components, small classes have been written. One of these is the `Register` class: it holds a single integer, that can be read or written. It also comes with a `flag` method, that returns whether a given flag is set in the register, as this is a frequent operation.

This class is extended by `MaskRegister`, a class to support registers of which all bits may not be used. This is the case for instance for `TAC`: bits 0–2 control the timer, and bits 3–7 are hardwired to 1, and cannot be reset. A `MaskRegister` allows defining this behaviour directly in the register, avoiding the need for additional logical in the class using the register (here, `Timer`). On instantiation, a *mask* is passed as an argument, and is applied whenever the value of the register needs to be changed (see Figure 5.31).

To accommodate the 16-bit registers of the CPU, a `DoubleRegister` class was implemented. It is made of two `Register` instances, and provides methods to get and set its value as if it held a 16-bit number (despite being implemented as two 8-bit numbers). This provides flexibility to the CPU, enabling both 16-bit and 8-bit arithmetic, without having to manage the way the

```
class MaskRegister extends Register {
    protected mask: number;

    constructor(mask: number, value: number = 0) {
        super(value | mask);
        this.mask = mask;
    }

    override set(value: number): void {
        super.set(value | this.mask);
    }
}
```

Figure 5.31: Implementation of `MaskRegister`

register is implemented.

Classes to manage memory were also created, to avoid having high-level components access data structures directly. For instance, the `ROM` class is backed by a `UInt8Array`[27], and implements `Addressable`. It is extended by `RAM`, to allow `write` operations.

To simplify the operation of certain components, a `CircularRAM` class was also created. It extends `RAM`, and must be provided an *offset* on creation. When reading or writing to it, the offset is subtracted from the address, and the modulo of the RAM's length is then applied to it (see Figure 5.32). This allows this part of memory to handle addressing in an independent way: the high-level component simply has to provide it with its address on creation.

```
class CircularRAM extends RAM {
    protected offset: number;

    constructor(size: number, offset: number, data?: Uint8Array) {
        super(size, data);
        this.offset = offset;
    }
    override read(pos: number): number {
        return super.read((pos - this.offset) % this.size);
    }
    override write(pos: number, data: number): void {
        super.write((pos - this.offset) % this.size, data);
    }
}
```

Figure 5.32: Implementation of `CircularRAM`

This can be used to implement the WRAM. It can be addressed from `0xC000` to `0xFDFF`, but `0xE000`–`0xFDFF` maps back to `0xC000`–`DDFF`. As such, a `CircularRAM` with an offset of

[27] See https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Uint8Array

`0xC000` and a length of `0x2000` will properly wrap address in the `0xE000`–`0xFDFF` range back to the beginning of its memory. This avoids handling this logic in high-level components, and provides a generic solution to this kind of memory component. Other places where this class is used include the wave RAM of the APU, and the VRAM of the PPU.

Chapter 6

Evaluation

Evaluation of this emulator consists in verifying it simulates the guest console (the Game Boy) properly. This can be done via test ROMs, that verify the behaviour of parts of the console. In this chapter, we will both look at how the emulator performs by itself, and how it performs compared to other GB emulators. We will then also ensure that it is performant, and that it complies with the specification.

6.1 Absolute Accuracy

To measure the *absolute accuracy* of the emulator (how accurate it is, independently of other emulators), we will run different test ROMs and see what aspects of the emulator pass which tests. This will help us guarantee that specific parts are functional, and can be relied on.

Note that although on a functioning system test ROMs help diagnose issues, if the system is faulty then there may be false positives. Indeed, the test ROM is still a ROM that runs on the system, and relies on it somewhat working. For instance, a faulty CPU might make a test seem like it passes, despite it actually being wrong. As such, results should be taken with a grain of salt. Furthermore, test ROMs can only test known behaviour of the GB: if there exists a bug or behaviour that is unknown of, there is no way of writing a test ROM to verify it.

Using the "Test ROMs" panel from the emulator frontend, we can run a total of 191 tests automatically. Because each test focuses on a specific characteristic of a component, we can group them, to see the *success rate* this particular component has; see Table 6.1.

We first may notice from this table that the accuracy of the CPU is quite good, the only test failing being `ie_push`[1], a test verifying an edge-case of interrupt handling. The success-rate

[1] See https://github.com/Gekkio/mooneye-test-suite/blob/main/acceptance/interrupts/ie_push.s

Component	Category	Test count	Tests passed	Success rate
CPU	Instructions	20	19	95.0%
	Timing	23	23	100.0%
	Sum	43	42	97.7%
PPU	Rendering	2	2	100.0%
	Timing	12	5	41.6%
	DMA	12	7	58.3%
	Sum	26	14	53.8%
Timer		13	13	100.0%
MBCs	MBC1	12	12	100.0%
	MBC2	7	7	100.0%
	MBC5	8	8	100.0%
	Sum	27	27	100.0%
APU	General	17	3	17.6%
	Channel 1	21	0	0.0%
	Channel 2	15	0	0.0%
	Channel 3	16	2	12.5%
	Channel 4	13	0	0.0%
	Sum	82	5	6.1%
All		191	101	52.8%

Table 6.1: Results of the test ROMs on different components

of these tests being high is essential, since of course all tests run on the CPU and rely on it functioning to operate properly.

The PPU's accuracy is good in terms of rendering, but the timings of it are wrong. When analysing the test ROMs further to understand what goes wrong, we can notice that some of the modes of the PPU are one cycle too long. This can be seen by inspecting the test's output (see Figure 6.1), and noticing the error is in the `D` register, that has a value of `0x02` instead of `0x01`. The part of the code responsible for this is in lines 35–40 of the test's source[2]. Although the timing of the emulator could be modified to make this test pass, the change makes other PPU timing tests fail; as such, it is reasonable to suspect the fault is in the logic itself of the `PPU` class and not the number of cycles only. Many modifications have been attempted to improve the accuracy of the timing, to no avail – more research would thus be needed to fix this.

The timer and MBCs have great accuracy, and do not have any problems as far as the tests can tell. This is important, in particular for the timer, because multiple timing tests rely on the timer functioning. As such, having this part of the emulator work reliably is vital to ensuring the other tests work properly.

Finally, the APU performs very poorly. This is firstly due to the complexity of the different channels: they all have several mechanisms working separately, and if only one of them behaves wrong then the whole set of tests will fail. Furthermore, the majority of these tests come from

[2]See https://github.com/Gekkio/mooneye-test-suite/blob/main/acceptance/ppu/intr_2_mode3_timing.s

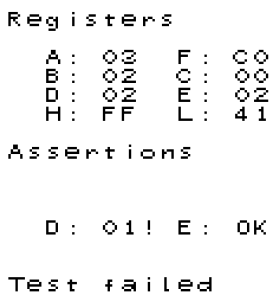

Figure 6.1: Output of PPU timing test `ppu_intr_2_mode3_timing`

the SameSuite[3] test suite. These tests differ from other because they use the `PCM12` and `PCM34` registers of the GBC, to directly inspect the output of each channel [9, Audio Details]. These allows the tests to check with much more precision that the emulator works, making the test significantly more stringent. Although this is an obvious indicator that the APU needs to be reworked and fixed, this does not impact the user experience, or at least not that of a casual user, as the audio of the game does not sound off when being used.

Seeing the total of tests passed is also of interest. The total success rate is 52.8%, but this value is strongly influenced by the fact almost half of the tests are for the faulty APU (82 tests out of 191). If these tests are filtered out, we get a total of 109 tests, of which 96 pass, giving a success rate of 88%. A really good result, given the emulator was written in a short period of time with no prior experience.

6.2 Relative Accuracy

After verifying the accuracy of Emmy by itself, using automated tests, we may also want to compare it to other existing GB emulators, and see how it does in contrast with them.

Thankfully, a tool for this already exists: "GBEmulatorShootout"[4]. It automatically tests GB emulators, and displays their results in a large table, allowing for the comparison of emulators' accuracy.

The way the tool works is that each emulator it supports must have a matching file, to allow downloading the emulator's executable and running it with a given test ROM. The program then regularly screenshots the emulator's window, and compares it with the expected visual output of the test, meaning it expects the emulator's window to only contain the graphical output of the GB.

[3] See https://github.com/LIJI32/SameSuite
[4] See https://github.com/daid/GBEmulatorShootout

This means that adding this emulator to the tool required changes to the tool's source code. Indeed, it has two notable distinctions to other emulators tested by the tool. First and foremost, this emulator is not based on an executable – it is instead *browser based*, meaning that simply running the emulator's file is not possible. The second difference is that the emulator's output does not cover the whole window, as there is an additional UI around it. We must thus instead find a way to extract the emulator's output from the window.

To do this, we will use web development techniques to automate the tests. Selenium[5] is a tool that allows programmatically opening a browser and running actions on it, mimicking those of a user. In this case, these actions would be opening the emulator's website, and uploading the ROM to the emulator, as a user would do. Because the rest of the tool is written in Python[6], we will use the `selenium` Python package[7]. See Figure 6.2 for the code to setup the emulator, and the code to run a ROM on it. Opening the window and accessing the emulator is quite simple (see lines 2–6), as we must just open the window at a given URL. We may note the script also does two additional actions: the first is enabling "triple speed mode" (line 5), to make the tests run faster. We also open the "Settings" tab of the sidebar (line 6), as this is where the console selection (DMG or GBC) is done. This is relevant because some tests are designed for a console in particular, and the only way to switch the console on this emulator is via this UI.

```
1  class Emmy(Emulator):
2      def setup(self):
3          self.driver = webdriver.Chrome("emu/chromedriver_win32/chromedriver.exe")
4          self.driver.get("https://emmy-gbc.vercel.app/")
5          self.driver.find_element(value="emu-speed").click()
6          self.driver.find_element(value="drawer-section-settings").click()
7
8      def startProcess(self, rom, *, model, required_features):
9          model_btn_id = {DMG: "dmg-mode", CGB: "cgb-mode"}.get(model)
10         if model_btn_id is None: # console not supported
11             return None
12         self.driver.find_element(value=model_btn_id).click()
13         rom_path = os.path.abspath(rom)
14         self.driver.find_element(value="rom-input").send_keys(rom_path)
15         try: # if an alert appeared, it means the rom is incompatible
16             self.driver.switch_to.alert.accept()
17             return None
18         except: # no alert, so error thrown, so the rom is compatible
19             return self.driver
```

Figure 6.2: Setup and run code to automate emulator testing

[5] See https://www.selenium.dev/
[6] See https://www.python.org/
[7] See https://pypi.org/project/selenium/

Once the emulator is setup, the test ROM must be handled. First, the desired console model is checked, and the appropriate console is selected in the UI (lines 9–12). This is done my matching the model ID to the ID of the button that selects said console, and clicking the button with that ID. The test ROM may then be uploaded, by pressing the "Import a ROM" button, and entering the ROM's path. We must also check if an alert is raised in the browser (lines 15–19) – this occurs if an error occurs when reading a ROM, usually because the MBC is not supported.

The second part of automated testing requires monitoring the emulator, and verifying if the test succeeded or failed. GBEmulatorShootout requires taking screenshots of the emulator frequently, and comparing the screenshot to an image of the expected result. This method implies storing an additional image for each test ROM, but comes with the advantage that no complex code is required to look into the emulator's state for other indicators of success (like what was done in this project's frontend, see subsection Test ROMs). This makes it the ideal solution for this kind of tool, as the additional code to add an emulator to the test suite is minimal (all emulators will output the Game Boy's screen, whereas not all emulators have the same internal structure).

To do this, we must extract the image from the `<canvas/>` element the emulator's frontend draws in, and convert it to an appropriate Python object (see Figure 6.3). We first fetch the output canvas by its ID, and execute a small JavaScript function to retrieve the image inside the canvas, in a PNG format. This data is then decoded, and used to create an image, that is resized to the correct size (as the emulator's screen may be upscaled by the browser).

```
1  def getScreenshot(self):
2      canvas = self.driver.find_element(value="emulator-frame")
3      canvas_base64 = self.driver.execute_script(
4          "return arguments[0].toDataURL('image/png').substring(21);",
5          canvas
6      )
7      canvas_png = base64.b64decode(canvas_base64)
8      large_image = PIL.Image.open(io.BytesIO(canvas_png))
9      small_image = large_image.resize((160, 144), PIL.Image.NEAREST)
10     return small_image
```

Figure 6.3: Code to get the emulator's output with Selenium

After other minor tweaks to GBEmulatorShootout's code to support web-based emulators, the test script can be run to compile the results and be able to finally compare our emulator to others. The results are accessible online at https://n1ark.github.io/GBEmulatorShootout/ (see Figure 6.4). The website to access the results is made of a large table, with on the X axis

the emulator running the test ROM, and on the Y axis the name of the test ROM. Emulators are ordered from most accurate (most tests passed) to least accurate, from left to right. The tool currently supports 239 test ROMs, for 14 emulators.

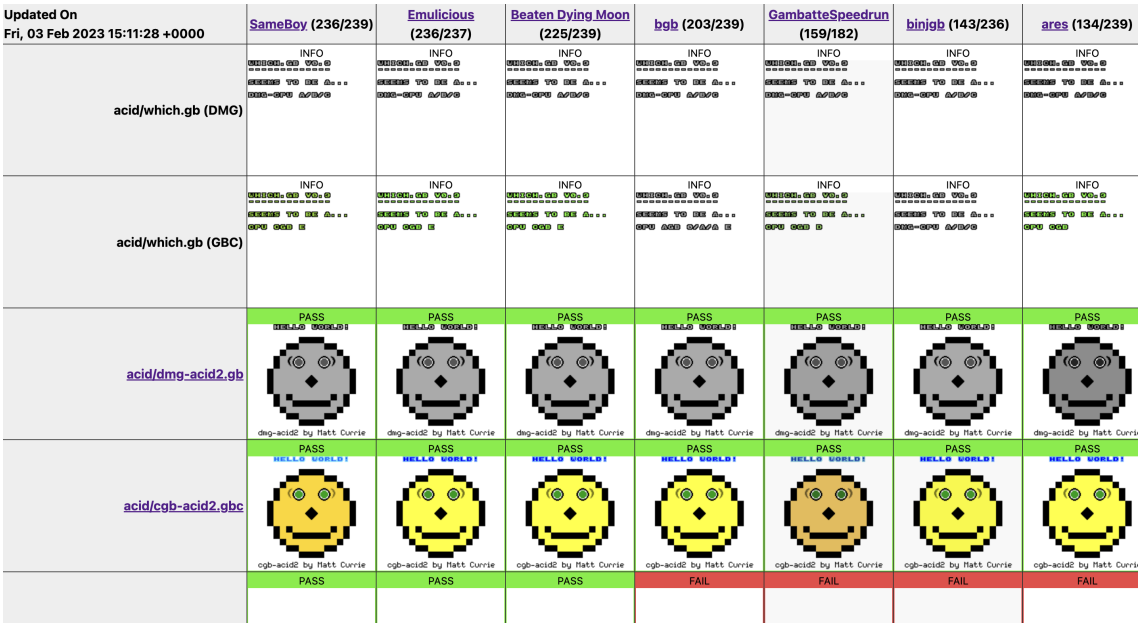

Figure 6.4: Screenshot of the UI of `GBEmulatorShootout`

We may first notice that Emmy passes 118 out of 236 tests (3 tests are skipped because the emulator does not support the "Super Game Boy"), ranking it 9th of the 14 tested emulators. It is relevant here to note that most of these other emulators have existed for several years now, and have been worked on by many people.

We may also use this table to compare the accuracy of this emulator to others, by looking at the success rate for tests of different components (see Table 6.2). Note that if an emulator cannot run a test (because it lacks support for a feature needed by the test), the test is skipped, rather than counted as failed. This explains why "VisualBoyAdvance-M" has a success rate of 57.1% but is ranked lower than Emmy.

This table is a good indicator of where our emulator, Emmy, does and does not perform well, compared to other GB emulators. For instance, it has a really accurate timer, compared to emulators ranked around it. This also emphasises the fact that its CPU is of an accuracy similar to that of the higher accuracy emulators, passing almost 80% of tests. It also helps see that only very high accuracy emulators manage to pass the APU tests. The only outlier seems to be VisualBoyAdvance-M, with an APU success rate of 60.7%. However, when inspecting the test results, we notice that most of the APU tests were not run on it – this is because it lacks

Emulator	Tests Run	CPU	PPU	Timer	MBCs	APU	All
SameBoy	239	100.0%	91.9%	100.0%	100.0%	100.0%	98.7%
Emulicious	237	98.5%	100.0%	100.0%	100.0%	100.0%	99.6%
Beaten Dying Moon	239	100.0%	97.3%	100.0%	97.1%	85.9%	94.1%
bgb	239	74.6%	91.9%	100.0%	97.1%	82.4%	84.9%
GambatteSpeedrun	182	98.5%	81.1%	100.0%	57.1%	100.0%	87.4%
binjgb	236	84.6%	63.9%	92.3%	74.3%	29.4%	60.6%
ares	239	82.1%	51.4%	38.5%	80.0%	29.4%	56.1%
mGBA	233	83.1%	52.8%	92.3%	90.6%	20.0%	57.1%
Emmy	236	76.9%	45.9%	100.0%	79.4%	10.6%	50.0%
VisualBoyAdvance-M	182	53.7%	45.9%	76.9%	62.9%	60.7%	57.1%
PyBoy	143	32.1%	17.2%	38.5%	90.6%	0.0%	40.6%
Goomba	232	30.8%	8.6%	7.7%	59.4%	3.5%	20.7%
no$gmb	236	35.4%	22.2%	0.0%	20.0%	2.4%	17.8%
KiGB	231	29.2%	20.0%	0.0%	12.9%	2.4%	14.7%

Table 6.2: Success rate of emulators for different components, sorted by total passed tests

support for the `PCM12` and `PCM34` registers these tests use. If we add the un-run tests to the percentage, we end with 17 tests passed out of 85, or a 20% success rate, in line with the success rate of similar emulators. This further shows that creating an accurate APU is challenging, especially given the standards of the used test ROMs.

6.3 Performance

Aside from accuracy, we may want to measure how performant the emulator is. This is important, because most emulators offer a "turbo mode", allowing the emulator to run the GB faster. Furthermore, this emulator is designed to run on mobile devices, that may have less computing power – as such, the emulator being efficient with the resources it has is vital for these platforms.

To measure performance, the most relevant metric is frame time: the time needed in milliseconds to draw a frame (this, of course, includes the rendering of the frame but also all the processing that goes before). The emulator's frontend already comes with a measure of frame time – we thus just need to select what ROM the test will be performed on, and how the measure will be taken.

For the ROM, the performance will be measured on 4 distinct ROMs, all outlining different uses of the emulator. These are:

- "The Legend of Zelda: Link's Awakening", a DMG game. It is quite simple and should not be too memory intensive. It will help outline the average performance on the DMG.
- The `cpu_instrs` test ROM. Its advantage is that it is quite long, and its purpose is to

test the entirety of the CPU, meaning it will be very processor-intensive.

- "The Legend of Zelda: Oracle of Ages", a GBC game. It is similar to the previous Zelda ROM, but was built for the GBC, meaning it likely requires more resources to run.
- "Alone in the Dark: The New Nightmare", a GBC game. This is a very complex game released for the GBC, that supports "3D-scenes", and very regularly changes the colour palette registers [8, Tricky-to-emulate games]. Both of these properties combined should make this ROM slower, making it a hypothetical "upper-bound" on frame time.

As for how to measure the performance, the frame time of every frame will be summed and averaged, over the first 10 million cycles, to give a more accurate measure. See Table 6.3 for the results.

Console	**ROM**	**Frame time** (ms)	**Speed increase**
DMG	The Legend of Zelda: Link's Awakening	4.39	279%
	`cpu_instrs`	5.43	207%
GBC	The Legend of Zelda: Oracle of Ages	4.89	241%
	Alone in the Dark: The New Nightmare	5.78	188%

Table 6.3: Average frame time of different ROMs

From this table we can notice two things. First, DMG games run faster than GBC games. This can be due to the double speed mode, that forces the emulator to run twice as many instructions, or to the fact that the PPU logic with colour is more complex and needs to be optimised. Secondly we may look at the average frame times of these ROMs.

The GB runs at 60 frames per second, meaning a frame must last at most $\frac{1}{60} = 16.6$ms. We may calculate the maximum speed increase attainable for a frame time via the formula $\frac{1000}{\text{frame time}*60}$, or $(\frac{1000}{\text{frame time}*60} - 1) * 100$ to get an increase percentage. For instance, for "The Legend of Zelda: Link's Awakening", the maximum speed increase could be of around 279%.

This is a satisfiable value: a speed increase of 200% is good enough for most uses. Many emulators provide speed increases much superior to this, however this is because they are often written in compiled languages like C++ or Rust, and will thus run much faster than an interpreted language like JavaScript.

WebAssembly[8] is a low-level language made for browsers, resembling assembly language. It is designed for efficiency, and would thus be a great pick for a browser-based emulator as it runs faster than JavaScript [24]. Languages like C, C++ and Rust can be compiled to WebAssembly via tools like emscripten[9], allowing for great performance with portable code. Examples of such

[8] See https://webassembly.org/
[9] See https://emscripten.org/

emulators playable from the browser include GBEmu[10], an emulator written in Rust.

This would however require a full rewrite of the code, since TypeScript and C++ have little in common in terms of syntax. Another simpler alternative to convert the emulator's code to a WebAssembly-compilable language is using AssemblyScript[11], a language similar to TypeScript.

Research has been done to convert the project's code to AssemblyScript, and the progress so far can be seen on the `assembly-script` branch[12]. However, because AssemblyScript targets such a low-level language, many TypeScript constructs are not (yet) available in it, such as arrow functions, or dynamic objects. Although the implemented emulator does not use these features extensively and most components were successfully migrated to AssemblyScript, the CPU uses arrow functions for all instructions – a successful conversion of the entire emulator would thus require a lengthy rewrite of the CPU. This project was thus abandoned in favour of additional features in the emulator's core and frontend. An example of GB emulator in AssemblyScript is wasmboy[13].

6.4 Compliance to Specification

The presented emulator fulfils most of the specification, with some specifications of lower importance having been left out. These include:

- 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.

  The user cannot, currently, download the save of their game. The save file is instead stored in the browser. Some research was done to allow saving the emulator's state to the BESS[14] save format, however this required a lot more effort so priority was shifted on other features, as the existing save system was deemed sufficient.
- U7: User can pause the console emulation through a button. They can also input conditions for which the console should break execution.

  The emulator's frontend does not provide a way to add breakpoints to the emulation. This feature was initially supported from the browser's console, but its implementation was unsatisfactory. A possible improvement to the emulator would thus be support for such breakpoints, maybe in a format similar to that of the current "Watch Expressions"

[10] See https://github.com/BlueBlazin/gbemu
[11] See https://www.assemblyscript.org/
[12] See https://github.com/N1ark/gbc-emulator/tree/assembly-script
[13] See https://github.com/torch2424/wasmboy/
[14] See https://github.com/LIJI32/SameBoy/blob/master/BESS.md

menu.

- N3: A downloadable version of the web-app can be used on a computer and provides full functionality, via keyboard and mouse inputs.

  The implemented emulator cannot currently be downloaded as a local app. This could however be added quite easily, using a tool like Electron[15] that allows converting web applications to desktop apps. Another option is converting the web-app into a Progressive Web App (PWA)[16], allowing users to "download" the web-app and use it while offline.

All of these specifications are however of low importance, and are mainly small quality of life features. Other possible improvements could be adding more debugging tools (such as a way to inspect the APU's raw output), or more features added to the emulator's core, like support for other MBCs (like the MBC6, MBC7, or MMM01 [9, MBCs]), the real time clock of the MBC3, additional outputs of the GB like the Game Boy Printer or the Game Boy Camera [9], and support for the Link Cable, allowing multiple Gameboys to play together [9, Serial Data Transfer].

[15] See https://www.electronjs.org/
[16] See https://web.dev/progressive-web-apps/

Chapter 7

Legal, Social, Ethical and Professional Issues

7.1 Privacy

This piece of software is safe to use in terms of privacy – it runs entirely locally, with no data ever being sent from the user to the server. There are no cookies, and the only forms of storage used are localStorage[1] and localForage[2], both of which are local and offline.

localStorage is handled by the browser, so it is the user's responsibility to ensure that they use a secure browser.

localForage is an open source library, allowing for more transparency – one could go through the source code to verify that it is safe as well. Whenever a new update is released, we may simply verify that the modified code is still safe, and then configure the project to use the new version in the `package.json` file.

7.2 Legality

The Game Boy is a copyrighted console, owned by Nintendo, but making an emulator for it is deemed legal as long as it is done following a "clean room" design [25]. Overall, it is legal to make emulators, as long as they follow this method. This was ruled via a series of court appeals between console manufacturers and groups that produce emulators, with the latter consistently winning the appeal. This was the case for the Sony Computer Entertainment America v. Con-

[1] See https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage
[2] See https://github.com/localForage/localForage

nectix Corporation trial, that deemed that the "Virtual Game Station" PlayStation emulator was not a copyright infringement on Sony, and that Connectix's reverse engineering of the original PlayStation's BIOS was fair use [26]. Emmy was developed using online resources about the GB compiled by people who reverse engineered the GB, and as such was developed with a clean room design too.

Although distributing the emulator is legal, distributing the BIOS (Basic Input/Output System) of the console with the emulator is not, as it is still copyrighted software [25]. Because acquiring such BIOS is however not illegal as long as the user owns the console, a user is free to use their own BIOS in the emulator (see subsection Settings). The emulator otherwise simulates the effect of the BIOS on the system, by setting up all the necessary memory and registers.

Distributing copyrighted game ROMs is also not allowed – this is why the user must upload the ROM they want to use themselves. The software does not upload the ROM, and simply stores it in local storage.

7.3 Integrity

This piece of software was developed with integrity, always thinking critically on what was developed, and with no intention to harm others. This report has also been written with honesty, without attempting to withhold information on the resulting software. No acquired data was falsified or modified to fit a narrative.

The produced software is open source, and available at https://github.com/N1ark/gbc-emulator. This means it can be used by others to create similar emulators, or for educational purposes. Other users may also contribute to the project, or fix issues it has.

Overall, this software was developed with the British Computer Society's Code of Conduct & Code of Good Practice[3] in mind, and all of its relevant rules were followed.

[3] See https://www.bcs.org/media/2211/bcs-code-of-conduct.pdf

Chapter 8

Conclusion and Future Work

8.1 Conclusion

This project helped outline the typical structure of a an emulator, and in particular that of a Game Boy emulator. It describes how the console works in detail, what components it is made of, and how they interact with each other. It showcased optimisation methods for emulating different components of an emulator, like the CPU or the system bus. The resulting software is a fully playable Game Boy and Game Boy Color emulator, that has multiple quality of life features to make the experience more pleasant while also providing debugging tools for retro game and emulator developers. This emulator can be played on both computer and mobile devices, and is of good accuracy compared to other existing emulators.

8.2 Future Work

As it stands the emulator has three main properties that could be improved, each independently one of the other.

The first is improving the accuracy of the emulator. As seen in the evaluation of the project, its APU should be reworked. It is currently lacklustre, and represents a part of the system that could be improved without requiring a full re-write, as it is self contained. Other small improvements could be done to improve the accuracy of other components like the MBCs or the PPU, or to add currently unsupported MBCs and accessories. All of these improvements can be done separately and in small increments, and do not entail any breaking changes outside of the emulator's core.

A second important point to be improved in the emulator is its performance. As seen in its evaluation, an average maximum of a 200% speed increase is reachable. To improve the performance of the emulator (which will be needed if more features are to be added), its performance must be improved substantially. Although code optimisations and caching could improve some of the performance issues, a significant bottleneck is the language used itself. If the emulator were to be entirely rewritten using a language such as AssemblyScript, it would likely run much faster. This would be a much more time-consuming change, and would require updating the front-end to work properly with it, but could yield great performance improvements if done properly.

Finally, the third point to improve on this project is its frontend. Although it has a good range of available features, it could still be improved to be up to par with other technical emulators. More and better debugging tools could be provided to the user, as well as more customisation options, such as new screen filters, a full-screen mode or support for input macros.

Acronyms

APU Audio Processing Unit.

CPU Central Processing Unit.

DMA Direct Memory Access.

DMG Dot Matrix Game.

GB Game Boy.

GBC Game Boy Color.

MBC Memory Bank Controller.

OAM Object Attribute Memory.

OS Operating System.

PPU Picture Processing Unit.

ROM Read-Only Memory.

VRAM Video RAM.

WRAM Work RAM.

Bibliography

[1] Merriam-Webster. "Emulator definition and meaning." (Mar. 2022), [Online]. Available: https://www.merriam-webster.com/dictionary/emulator (visited on 03/02/2023).

[2] A. Kaluszka. "Computer emulation, history." (Dec. 2001), [Online]. Available: https://kaluszka.com/vt/emulation/history.html (visited on 03/03/2023).

[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 03/03/2023).

[4] S. Granger, "Emulation as a digital preservation strategy," *D Lib Mag.*, vol. 6, no. 10, Oct. 2000. DOI: 10.1045/october2000-granger. [Online]. Available: https://doi.org/10.1045/october2000-granger (visited on 04/02/2023).

[5] L. Boley. "Emulation or virtualization: What's the difference?" (Mar. 2014), [Online]. Available: https://www.dell.com/en-us/blog/emulation-or-virtualization-what-s-the-difference/ (visited on 04/02/2023).

[6] 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 03/07/2023).

[7] Emulation General Wiki. "History of emulation." (Jan. 2023), [Online]. Available: https://emulation.gametechwiki.com/index.php/History_of_emulation#Game_Boy.2FColor (visited on 03/07/2023).

[8] "Gameboy development wiki." (2023), [Online]. Available: https://gbdev.gg8.se/wiki/ (visited on 03/28/2023).

[9] "Pandocs." (2023), [Online]. Available: https://gbdev.io/pandocs/ (visited on 03/07/2023).

[10] endrift. “"holy grail" bugs in emulation, part 1.” (May 2017), [Online]. Available: https://mgba.io/2017/05/29/holy-grail-bugs/ (visited on 04/03/2023).

[11] T. Phillips, “Game Boy and Game Boy Color titles headed to Nintendo Switch Online,” *Eurogamer*, Feb. 2023. [Online]. Available: https://www.eurogamer.net/game-boy-and-game-boy-color-titles-headed-to-nintendo-switch-online-1 (visited on 04/02/2023).

[12] M. J. Flynn, C. Neuhauser, and R. M. McClure, “EMMY: An emulation system for user microprogramming,” in *Proceedings of the May 19-22, 1975, National Computer Conference and Exposition*, ser. AFIPS ’75, Anaheim, California: Association for Computing Machinery, 1975, pp. 85–89, ISBN: 9781450379199. DOI: 10.1145/1499949.1499968. [Online]. Available: https://doi.org/10.1145/1499949.1499968.

[13] Pan of Anthrox, GABY, M. Fayzullin, *et al.* “Game Boy CPU manual.” (May 2008), [Online]. Available: http://marc.rawer.de/Gameboy/Docs/GBCPUman.pdf (visited on 03/22/2023).

[14] A. Vynogradenko. “Framework-sizes.md.” (2023), [Online]. Available: https://gist.github.com/Restuta/cda69e50a853aa64912d (visited on 04/03/2023).

[15] MDN Web Docs. “Autoplay guide for media and web audio APIs.” (Mar. 2023), [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Media/Autoplay%5C_guide (visited on 03/23/2023).

[16] M. Korth. “GBATEK.” (2014), [Online]. Available: https://problemkaputt.de/gbatek.htm (visited on 03/26/2023).

[17] M. Sullivan. “Game boy CPU intructions.” (Oct. 2022), [Online]. Available: https://meganesu.github.io/generate-gb-opcodes/ (visited on 03/22/2023).

[18] A. Goldfuss. “Function dispatch tables in C.” (Mar. 2019), [Online]. Available: https://blog.alicegoldfuss.com/function-dispatch-tables/ (visited on 03/29/2023).

[19] Joonas “Gekkio” Javanainen. “Game Boy: Complete technical reference.” (Mar. 2023), [Online]. Available: https://gekkio.fi/files/gb-docs/gbctr.pdf (visited on 03/07/2023).

[20] R. Mongenel. “GameBoy memory map.” (2010), [Online]. Available: http://gameboy.mongenel.com/dmg/asmmemmap.html (visited on 03/27/2023).

[21] M. Steil. “The ultimate game boy talk (33c3).” (Dec. 2016), [Online]. Available: https://youtu.be/HyzD8pNlpwI?t=2460 (visited on 03/27/2023).

[22] Great Hierophant. "MBC5 backwards compatibility." (Nov. 2009), [Online]. Available: https://forums.nesdev.org/viewtopic.php?p=52575#post_content52603 (visited on 03/29/2023).

[23] M. Achibet. "Game boy/game boy color roms." (Jul. 2015), [Online]. Available: http://merwanachibet.net/gameboy-roms (visited on 03/29/2023).

[24] B. Velghe. "Is WebAssembly really faster than JavaScript? a hands-on experiment!" (May 2022), [Online]. Available: https://www.tmssoftware.com/site/blog.asp?post=933 (visited on 04/01/2023).

[25] J. Conley, E. Andros, P. Chinai, E. Lipkowitz, and D. Perez, "Use of a game over: Emulation and the video game industry, a white paper," Northwestern Journal of Technology and Intellectual Property, Tech. Rep., 2004. [Online]. Available: https://scholarlycommons.law.northwestern.edu/njtip/vol2/iss2/3/ (visited on 03/31/2023).

[26] *Sony Computer Entertainment America, Inc. v. Connectix Corporation*, 203 F.3d 596, 2000. [Online]. Available: https://web.archive.org/web/20070228070634/http://www.ca9.uscourts.gov/ca9/newopinions.nsf/0/66b3a352ea3371298825695200757 8c2?OpenDocument (visited on 03/31/2023).

Appendix A

User Guide

In this guide, we will walk you through how to use the Emmy emulator, a Game Boy and Game Boy Color emulator for the browser.

A.1 Disclaimer

Emmy is not affiliated with Nintendo in any way. You are responsible for obtaining copies of the ROMs you wish to play yourself, and must ensure you are allowed to have these ROMs. Open source ROMs are available online, on websites like Retro Veteran[1]. We do not condone the usage of illegally obtained ROMs.

One feature of the emulator involves using boot ROMs for the console. These cannot be distributed by Emmy, and must be obtained legally to be used. These can be found and downloaded online. They are not necessary for the usage of the emulator. Open source versions of the boot ROMs can be found online.

A.2 Playing games on Emmy

First, open the emulator from your browser – a hosted version is available at `https://emmy-gbc.vercel.app/`. You will get to the Emmy emulator (see Figure A.1).

The interface is divided into two parts:

- The main area, on the right. This is where the console's screen will appear. It has extra buttons which we will talk about later.

[1] See `https://www.retroveteran.com/category/nintendo-game-boy-color/`

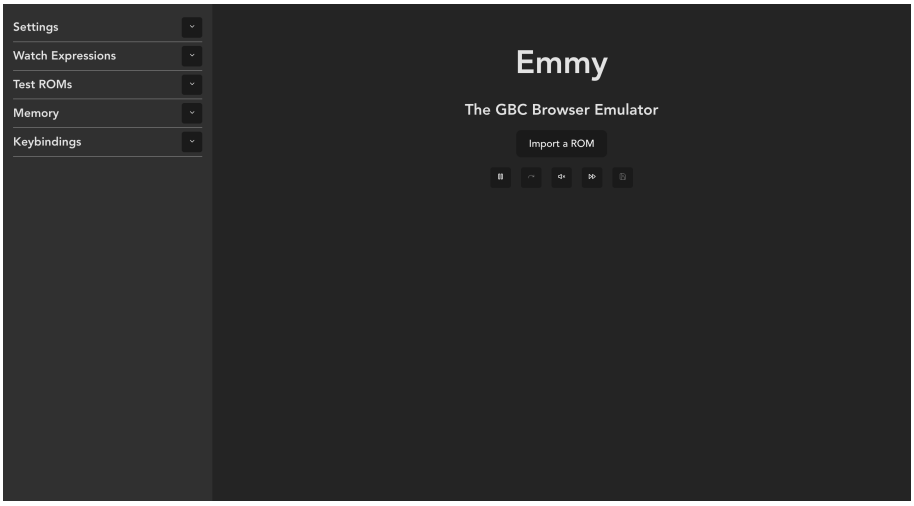

Figure A.1: Start screen of Emmy

- The sidebar, on the left. This is where all of the settings of the emulator are. All submenus can be opened and closed. You can also resize the sidebar by dragging on its right side.

Press the "Import a ROM" button, to select the ROM you wish to use. A file picker dialogue should open – select the ROM, and start playing! The game should directly load into the emulator.

A.3 Settings

The behaviour of Emmy can be customised to your needs. In the main area, you can find buttons to freeze the play-through, or resume it. You can also decide to enable or disable the game's sound, and speed up the emulation, making it go three times as fast. If the game supports save files, press the "Save" button to save the current state of the game. The next time you open the ROM, your save will automatically be loaded.

In the sidebar, you will find more settings to customise the emulator (see Figure A.2).

From top to bottom, these settings are:

- The emulated console. This may either be the classic Game Boy, or the Game Boy Color.
- The upscaling filter chosen. This option modifies the output of the screen. The first option leaves it as it is, without altering it. The two other options, Scale2x and Scale4x[2], allow upscaling the image, by adding more details to it, increasing its resolution by two and four respectively.
- The scale at which the screen is displayed. This allows increasing the size of the screen –

[2]See https://www.scale2x.it/

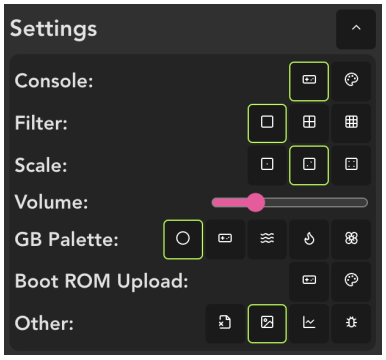

Figure A.2: Settings of Emmy

not its resolution!

- The volume at which the emulator outputs sound (if audio is enabled).
- The palette used for the Game Boy. By default, it will use shades of grey. Options include the classic green hues, a blue palette, a darker red palette, and a flowery pink theme.
- Two buttons to upload the boot ROM of the emulator. This is not needed to play, but can be nice if you want the original startup screen.
- A set of miscellaneous settings, including:
  - If the boot ROM is enabled. For this to work, you must have first uploaded a boot ROM for the current console.
  - Enabling frame blending, meaning frames are blended together. This may cause some slight visual artefacts when the scene is moving around.
  - Displaying the current performance of the emulator.
  - Displaying the tileset and tile data of the emulator. This is intended for debug purposes.

A.4 Keybindings

Emmy comes with default keybindings for the controls, that are common in other emulators (see Table A.1).

Game Boy Button	**Default Key**
Directional arrows	Directional arrows
A	Z
B	X
Start	Enter
Select	Backspace

Table A.1: Default keybindings of Emmy

These can also be customised, in the "Keybindings" submenu (see Figure A.3). To change a keybinding, click on the button you want to remap, and the press the key you want to bind to it. Press enter to confirm, and escape to cancel.

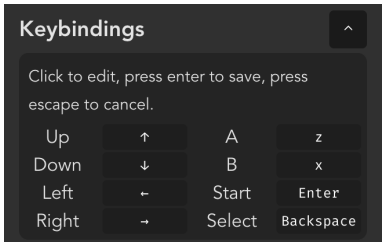

Figure A.3: Keybindings menu

A.5 Debugging

Emmy also provides a set of debugging tools - if you are a retro game developer, an emulator developer or want to study Game Boy ROMs, these tools will be of great help to you.

First, in the sidebar there is a "Watch Expressions" submenu (see Figure A.4). It allows writing expressions and seeing their output directly from the UI. The syntax is the JavaScript syntax. The emulator is defined as the variable `gbc`. To access any data on the emulator, refer yourself to the project's code, available at `https://github.com/N1ark/gbc-emulator`. The emulator is in the `src/emulator` folder.

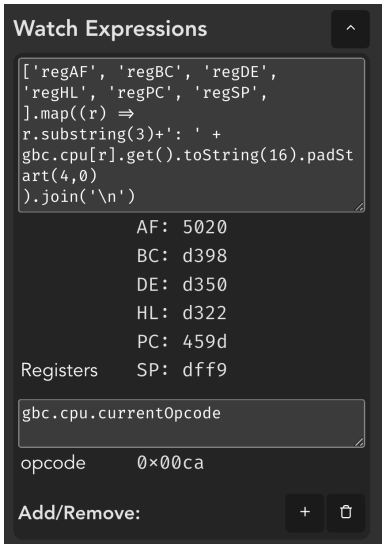

Figure A.4: Settings of Emmy

For instance, to access the `DIV` register of the timer, use the expression `gbc.system.timer.divider`. By default, number are output in hexadecimal. If you want to see the number in a decimal

format, format it into a string: `num.toString(10)`. You can add and remove as many watch expressions as you want! If one of them is not valid, the message "`Error`" will appear.

The emulator also comes with a builtin set of test ROMs, to test the emulator live (see Figure A.5). This allows you to easily see how accurate it is, and what tests the emulator passes and fails - this is very useful if you are working on the emulator's code, to ensure you are not breaking anything as you modify it. You can select the set of tests you wish to run, and there is also a button to select or unselect everything.

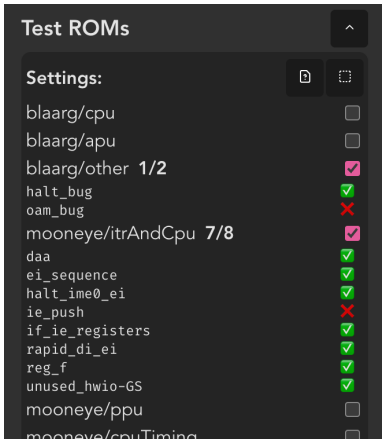

Figure A.5: Automated test ROMs

To inspect the behaviour of the emulator in detail, the emulator can be paused, and ticked instruction by instruction by pressing the "Step" button in the main area. This allows you to see what each instruction does, which is very helpful when a bug arises.

Finally, a memory inspection tool is available, to see the data contained in all of memory. To see it, open the "Memory" submenu (see Figure A.6). It will display every single accessible byte of memory, from `0x0000` to `0xFFFF`. If you are only interested in a part of memory, an offset can be input (for instance, if you only want to see the data from `0xFF00` to `0xFFFF`).

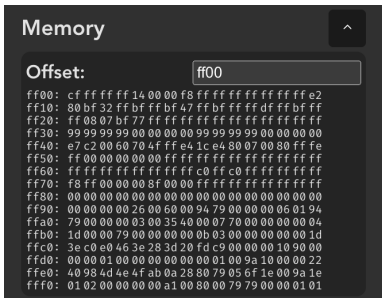

Figure A.6: Memory inspection tool

Appendix B

Source Code

B.1 Emulator Core Code

B.1.1 Emulator Main Interfaces

GameBoyColor (src/emulator/GameBoyColor.ts)

```ts
import { ConsoleType, CYCLES_PER_FRAME, SpeedMode } from "./constants";
import CPU from "./CPU";
import GameBoyInput from "./GameBoyInput";
import System from "./System";
import GameBoyOutput from "./GameBoyOutput";

export type GameBoyColorOptions = {
    bootRom: null | Uint8Array;
};

const DEFAULT_OPTIONS: GameBoyColorOptions = {
    bootRom: null,
};

class GameBoyColor {
    protected options: GameBoyColorOptions;
    protected mode: ConsoleType;

    protected cpu: CPU;
    protected system: System;

    protected cpuIsHalted = false;
    protected cycles: number = 0;
    protected isFullCycle = true; // used for double speed mode

    protected output: GameBoyOutput;

    constructor(
        modeStr: "DMG" | "CGB",
        rom: Uint8Array,
        input: GameBoyInput,
        output: GameBoyOutput,
        options?: Partial<GameBoyColorOptions>
    ) {
        this.mode = modeStr === "DMG" ? ConsoleType.DMG : ConsoleType.CGB;
        this.system = new System(rom, input, output, this.mode);
        this.cpu = new CPU(() => this.system.didStopInstruction());
        this.output = output;
        this.options = { ...DEFAULT_OPTIONS, ...options };

        this.setup();
    }

    protected setup() {
        if (this.options.bootRom !== null) {
            if (this.mode === ConsoleType.DMG && this.options.bootRom.length !== 256) {
                throw new Error("DMG boot ROM must be 256 bytes long");
            }
            if (this.mode === ConsoleType.CGB && this.options.bootRom.length !== 2304) {
                throw new Error("CGB boot ROM must be 2304 bytes long");
            }
            this.system.loadBootRom(this.options.bootRom);
        }

        // Setup registers as if the boot ROM was executed
        else {
            // CPU
            if (this.mode === ConsoleType.DMG) {
                this.cpu["regAF"].set(0x01b0);
```

```
                this.cpu["regBC"].set(0x0013);
                this.cpu["regDE"].set(0x00d8);
                this.cpu["regHL"].set(0x014d);
            } else {
                this.cpu["regAF"].set(0x1180);
                this.cpu["regBC"].set(0x0000);
                this.cpu["regDE"].set(0xff56);
                this.cpu["regHL"].set(0x000d);
            }
            this.cpu["regPC"].set(0x0100);
            this.cpu["regSP"].set(0xfffe);

            // PPU
            this.system["ppu"]["ppu"]["lcdControl"].set(0x91);

            // Emulate GBC compatibility check
            if (this.mode === ConsoleType.CGB) {
                const compat = this.system.read(0x0143);
                if (compat === 0x80 || compat === 0xc0) {
                    this.system.write(0xff4c, compat);
                } else {
                    const colorControl = this.system["ppu"]["ppu"]["colorControl"];
                    // reset palette controls
                    colorControl.write(0xff68, 0x80);
                    colorControl.write(0xff6a, 0x80);

                    // load palettes
                    const bgrPalette = [0xff, 0x7f, 0xef, 0x1b, 0x80, 0x61, 0x00, 0x00];
                    const objPalette = [0xff, 0x7f, 0x1f, 0x42, 0xf2, 0x1c, 0x00, 0x00];
                    bgrPalette.forEach((value) => colorControl.write(0xff69, value)); // bg
                    objPalette.forEach((value) => colorControl.write(0xff6b, value)); // obj0
                    objPalette.forEach((value) => colorControl.write(0xff6b, value)); // obj1

                    this.system.write(0xff4c, 0x04); // change to DMG mode
                    this.system["ppu"]["ppu"]["objPriorityMode"].write(0, 0x01); // OPRI
                }
            }

            // End initialisation
            this.system["bootRomLocked"] = true;
        }
    }

    /**
     * @returns whether the emulator supports saving the current ROM state.
     */
    supportsSaves(): boolean {
        return this.system.supportsSaves();
    }

    /** Saves the current ROM state (null if no save support). */
    save(): Uint8Array | null {
        return this.system.save();
    }

    /** Loads the given ROM data. */
    load(data: Uint8Array): void {
        this.system.load(data);
    }

    /** Returns the title of the current ROM. */
    getTitle(): string {
        return this.system.getTitle();
    }

    /** Returns the identifier of the current ROM. */
    getIdentifier(): string {
        return this.system.getIdentifier();
    }
```

```
129
130     /** The current mode of the emulator. */
131     getMode(): "DMG" | "CGB" {
132         return this.mode === ConsoleType.DMG ? "DMG" : "CGB";
133     }
134
135     /**
136      * Draws a full frame
137      * @param frames number of frames to draw (defaults to 1 to draw every frame - can be used
138      * to speed up emulation).
139      * @param isDebugging whether the emulator is in debugging mode (goes CPU step by step,
140      * prints verbose CPU logs).
141      * @returns the number of T-cycles executed.
142      */
143     drawFrame(frames: number = 1, isDebugging: boolean = false): number {
144         const cycleTarget = CYCLES_PER_FRAME * frames;
145         const interrupts = this.system.getInterrupts();
146         const cyclesStart = this.cycles;
147
148         while (this.cycles < cycleTarget) {
149             const normalSpeedMode = this.system.getSpeedMode() === SpeedMode.Normal;
150             const cycles = normalSpeedMode ? 4 : 2;
151             this.isFullCycle = normalSpeedMode || !this.isFullCycle;
152
153             // one CPU step, convert M-cycles to CPU cycles
154             let cpuIsDone: boolean;
155             if (!this.cpuIsHalted)
156                 cpuIsDone = this.cpu.step(this.system, interrupts, isDebugging);
157             else cpuIsDone = true;
158
159             this.cpuIsHalted = this.system.tick(this.isFullCycle);
160
161             this.cycles += cycles;
162
163             // If instruction finished executing and we're debugging
164             if (cpuIsDone && isDebugging) return this.cycles - cyclesStart;
165         }
166
167         // Keep leftover cycles
168         const cyclesRan = this.cycles - cyclesStart;
169         this.cycles %= cycleTarget;
170
171         // Read input
172         this.system.readInput();
173
174         // Output
175         this.system.pushOutput(this.output);
176
177         return cyclesRan;
178     }
179 }
180
181 export default GameBoyColor;
```

GameBoyInput (src/emulator/GameBoyInput.ts)

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

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

interface GameBoyInput {
    read(): GameBoyInputRead;
}

export default GameBoyInput;
```

GameBoyOutput (src/emulator/GameBoyOutput.ts)

```ts
interface GameBoyOutput {
    /**
     * @param data the GameBoy's screen output, as an array of RGBA values, making up for a
     * 160x144 image.
     */
    receiveGraphics?(data: Uint32Array): void;

    /**
     * @param data a sample of sound. The emulator produces samples at a
     * 44.1Hz rate, and outputs them every 60th of a second (ie. every frame).
     */
    receiveSound?(data: Float32Array): void;

    // Debugging methods:

    /**
     * @param data an array of RGBA values, with the image of the currently loaded background
     * data in 256x256.
     */
    debugBackground?(data: Uint32Array): void;

    /**
     * @param data an array of RGBA values, with the image of the current tileset in 256x192.
     */
    debugTileset?(data: Uint32Array): void;

    /**
     * @param data the serial output of the Gameboy - called everytime a character is pushed.
     */
    serialOut?(data: number): void;
}

export default GameBoyOutput;
```

B.1.2 CPU

CPU (src/emulator/CPU.ts)

```ts
import Interrupts from "./Interrupts";
import { Addressable } from "./Memory";
import { DoubleRegister, Register } from "./Register";
import { asSignedInt8, combine, high, low, wrap16, wrap8 } from "./util";

const FLAG_ZERO = 1 << 7;
const FLAG_SUBSTRACTION = 1 << 6;
const FLAG_HALFCARRY = 1 << 5;
const FLAG_CARRY = 1 << 4;

type InstructionMethod = (system: Addressable, interrupts: Interrupts) => InstructionReturn;
type InstructionReturn = InstructionMethod | null;

/**
 * The CPU of the GBC, responsible for reading the code and executing instructions.
 */
class CPU {
    // All registers are 16 bits long.
    // AF: lower is flags: ZNHC (zero, substraction, half-carry, carry)
    protected regAF = new DoubleRegister();
    protected regBC = new DoubleRegister();
    protected regDE = new DoubleRegister();
    protected regHL = new DoubleRegister();
    protected regPC = new DoubleRegister(); // program counter
    protected regSP = new DoubleRegister(); // stack pointer

    // If the CPU is halted
    protected halted: boolean = false;
    // If the CPU was halted when IME=0
    protected haltBug: boolean = false;

    // Subregisters, for convenience sake
    protected srA = this.regAF.h;
    protected srB = this.regBC.h;
    protected srC = this.regBC.l;
    protected srD = this.regDE.h;
    protected srE = this.regDE.l;
    protected srH = this.regHL.h;
    protected srL = this.regHL.l;

    // Next instruction callable
    protected nextStep: InstructionMethod | null = null;

    // The opcode for the next instruction
    // This is needed because the CPU actually fetches the opcode on the last M-cycle of the
    // previous instruction. This emulator stores the value at the PC at the end of each
    // instruction to use for the next instruction (this opcode is invalidated if an interrupt
    // happens)
    protected currentOpcode: number | null = null;

    // for debug purposes
    protected stepCounter: number = 0;

    // STOP instruction - relies on the rest of the system to stop
    protected stopInstruction: () => void;

    constructor(stopInstruction: () => void) {
        this.stopInstruction = stopInstruction;
    }

    // Returns the next opcode
    protected nextOpCode(system: Addressable): number {
        if (this.currentOpcode === null) {
            return system.read(this.regPC.inc());
```

```
        }
        const op = this.currentOpcode;
        this.currentOpcode = null;
        return op;
    }

    /**
     * Reads the given address and returns it to the receiver.
     * Takes 1 cycle.
     */
    protected readAddress(
        address: number | (() => number),
        receiver: (value: number) => InstructionMethod
    ): InstructionMethod {
        return (system) => {
            const effectiveAddress = typeof address === "number" ? address : address();
            const value = system.read(effectiveAddress);
            return receiver(value);
        };
    }

    /**
     * Reads the next byte from the PC and increases it.
     * Takes 1 cycle.
     */
    protected nextByte(receiver: (value: number) => InstructionMethod): InstructionMethod {
        return (system) => {
            const value = system.read(this.regPC.inc());
            return receiver(value);
        };
    }

    /**
     * Reads the next word (two bytes) from the PC and increases it.
     * Takes 2 cycles.
     */
    protected nextWord(receiver: (value: number) => InstructionMethod): InstructionMethod {
        return this.nextByte((low) => this.nextByte((high) => receiver(combine(high, low))));
    }

    getStepCounts() {
        return this.stepCounter;
    }

    getPC() {
        return this.regPC.get();
    }

    /**
     * Steps through one line of the code, and returns the M-cycles required for the
     * operation
     * @param system The system to execute the instruction on
     * @param verbose If true, prints the executed instruction to the console
     * @returns true if the CPU is in a "set" state (ie. it's halted or just finished an
     * instruction), false if it is mid-instruction.
     */
    step(system: Addressable, interrupts: Interrupts, verbose?: boolean): boolean {
        if (this.nextStep === null) {
            const nextStep = this.loadNextOp(system, interrupts, verbose);
            if (nextStep === "halted") return true;
            this.nextStep = nextStep;
        }

        this.nextStep = this.nextStep(system, interrupts);
        if (this.nextStep === null) {
            this.currentOpcode = system.read(this.regPC.inc());
        }
        return this.nextStep === null;
    }
```

```
134
135     protected loadNextOp(
136         system: Addressable,
137         interrupts: Interrupts,
138         verbose?: boolean
139     ): InstructionMethod | "halted" {
140         // Check if any interrupt is requested. This also stops HALTing.
141         if (interrupts.hasPendingInterrupt) {
142             this.halted = false;
143             if (interrupts.interruptsEnabled) {
144                 const execNext = interrupts.handleNextInterrupt();
145                 // Interrupt handling takes 5 cycles
146                 const nextStep = () => () => this.call(execNext, () => null);
147                 this.currentOpcode = null;
148                 this.regPC.dec(); // undo the read done at the end of the previous instruction
149                 if (verbose)
150                     console.log("[CPU] interrupt execute, goto", execNext.toString(16));
151                 return nextStep;
152             }
153         }
154
155         // Do nothing if halted
156         if (this.halted) {
157             return "halted";
158         }
159
160         // Execute next instruction
161         const opcode = this.nextOpCode(system);
162         ++this.stepCounter;
163         if (verbose)
164             console.log(
165                 `[CPU] ${this.stepCounter} - (0x${(this.regPC.get() - 1).toString(
166                     16
167                 )}) executing op 0x${opcode.toString(16)}`
168             );
169
170         if (this.haltBug) {
171             this.haltBug = false;
172             this.regPC.dec();
173         }
174
175         const instruction = this.instructionSet[opcode];
176         if (instruction === undefined) {
177             throw Error(
178                 `Unrecognized opcode ${opcode?.toString(16)} at address ${(
179                     this.regPC.get() - 1
180                 ).toString(16)}`
181             );
182         }
183         return instruction;
184     }
185
186     /**
187      * A list of all 8-bit opcodes.
188      * Each callable executes the instruction, and returns the number of M-cycles that the
189      * instruction took.
190      * @link https://meganesulli.com/generate-gb-opcodes/
191      * */
192     protected instructionSet: Partial<Record<number, InstructionMethod>> = {
193         // NOP
194         0x00: () => null,
195         // STOP
196         0x10: () => {
197             this.stopInstruction();
198             return null;
199         },
200         // extended instructions
201         0xcb: this.nextByte((opcode) => (system, interrupts) => {
202             const instruction = this.extendedInstructionSet[opcode];
```

```
            if (instruction === undefined) {
                throw Error(
                    `Unrecognized extended opcode ${opcode.toString(16)} at address ${(
                        this.regPC.get() - 1
                    ).toString(16)}`
                );
            }
            return instruction(system, interrupts);
        }),
        // LD BC/DE/HL/SP, d16
        ...this.generateOperation(
            {
                0x01: this.regBC,
                0x11: this.regDE,
                0x21: this.regHL,
                0x31: this.regSP,
            },
            (register) =>
                this.nextWord((value) => () => {
                    register.set(value);
                    return null;
                })
        ),
        // INC BC/DE/HL/SP
        ...this.generateOperation(
            {
                0x03: this.regBC,
                0x13: this.regDE,
                0x23: this.regHL,
                0x33: this.regSP,
            },
            (r) => () => () => {
                r.inc();
                return null;
            }
        ),
        // DEC BC/DE/HL/SP
        ...this.generateOperation(
            {
                0x0b: this.regBC,
                0x1b: this.regDE,
                0x2b: this.regHL,
                0x3b: this.regSP,
            },
            (r) => () => () => {
                r.dec();
                return null;
            }
        ),
        // ADD HL, BC/DE/HL/SP
        ...this.generateOperation(
            {
                0x09: this.regBC,
                0x19: this.regDE,
                0x29: this.regHL,
                0x39: this.regSP,
            },
            (register) => () => () => {
                const hl = this.regHL.get();
                const n = register.get();
                const result = wrap16(hl + n);
                this.regHL.set(result);
                this.setFlag(FLAG_SUBSTRACTION, false);
                this.setFlag(FLAG_HALFCARRY, (((hl & 0xfff) + (n & 0xfff)) & 0x1000) != 0);
                this.setFlag(FLAG_CARRY, hl > 0xffff - n);
                return null;
            }
        ),
        // INC B/D/H/C/E/L/A
```

```
        ...this.generateOperation(
            {
                0x04: this.srB,
                0x0c: this.srC,
                0x14: this.srD,
                0x1c: this.srE,
                0x24: this.srH,
                0x2c: this.srL,
                0x3c: this.srA,
            },
            (r) => () => {
                const result = this.incN(r.get());
                r.set(result);
                return null;
            }
        ),
        // INC (HL)
        0x34: this.readAddress(
            () => this.regHL.get(),
            (value) => (s) => {
                const result = this.incN(value);
                s.write(this.regHL.get(), result);
                return () => null;
            }
        ),
        // DEC B/D/H/C/E/L/A
        ...this.generateOperation(
            {
                0x05: this.srB,
                0x0d: this.srC,
                0x15: this.srD,
                0x1d: this.srE,
                0x25: this.srH,
                0x2d: this.srL,
                0x3d: this.srA,
            },
            (r) => () => {
                const result = this.decN(r.get());
                r.set(result);
                return null;
            }
        ),
        // DEC (HL)
        0x35: this.readAddress(
            () => this.regHL.get(),
            (value) => (s) => {
                const result = this.decN(value);
                s.write(this.regHL.get(), result);
                return () => null;
            }
        ),
        // LD (BC/DE/HL+/HL-), A
        ...this.generateOperation(
            {
                0x02: () => this.regBC.get(),
                0x12: () => this.regDE.get(),
                0x22: () => this.regHL.inc(),
                0x32: () => this.regHL.dec(),
            },
            (getAddress) => (system) => {
                const address = getAddress();
                system.write(address, this.srA.get());
                return () => null;
            }
        ),
        // LD A, (BC/DE/HL+/HL-)
        ...this.generateOperation(
            {
                0x0a: () => this.regBC.get(),
```

```
            0x1a: () => this.regDE.get(),
            0x2a: () => this.regHL.inc(),
            0x3a: () => this.regHL.dec(),
        },
        (getAddress) =>
            this.readAddress(getAddress, (value) => () => {
                this.srA.set(value);
                return null;
            })
    ),
    // LD B/C/D/E/H/L/A, d8
    ...this.generateOperation(
        {
            0x06: this.srB,
            0x0e: this.srC,
            0x16: this.srD,
            0x1e: this.srE,
            0x26: this.srH,
            0x2e: this.srL,
            0x3e: this.srA,
        },
        (r) =>
            this.nextByte((value) => () => {
                r.set(value);
                return null;
            })
    ),
    // LD (HL), d8
    0x36: this.nextByte((value) => (s) => {
        s.write(this.regHL.get(), value);
        return () => null;
    }),
    // LD (a16), SP
    0x08: this.nextWord((value) => (s) => {
        s.write(value, this.regSP.l.get());
        return () => {
            s.write(value + 1, this.regSP.h.get());
            return () => null;
        };
    }),
    // LD B/C/D/E/H/L/A, B/C/D/E/H/L/A
    ...this.generateOperation<number, [Register, Register]>(
        {
            0x40: [this.srB, this.srB],
            0x41: [this.srB, this.srC],
            0x42: [this.srB, this.srD],
            0x43: [this.srB, this.srE],
            0x44: [this.srB, this.srH],
            0x45: [this.srB, this.srL],
            0x47: [this.srB, this.srA],

            0x48: [this.srC, this.srB],
            0x49: [this.srC, this.srC],
            0x4a: [this.srC, this.srD],
            0x4b: [this.srC, this.srE],
            0x4c: [this.srC, this.srH],
            0x4d: [this.srC, this.srL],
            0x4f: [this.srC, this.srA],

            0x50: [this.srD, this.srB],
            0x51: [this.srD, this.srC],
            0x52: [this.srD, this.srD],
            0x53: [this.srD, this.srE],
            0x54: [this.srD, this.srH],
            0x55: [this.srD, this.srL],
            0x57: [this.srD, this.srA],

            0x58: [this.srE, this.srB],
            0x59: [this.srE, this.srC],
```

```
            0x5a: [this.srE, this.srD],
            0x5b: [this.srE, this.srE],
            0x5c: [this.srE, this.srH],
            0x5d: [this.srE, this.srL],
            0x5f: [this.srE, this.srA],

            0x60: [this.srH, this.srB],
            0x61: [this.srH, this.srC],
            0x62: [this.srH, this.srD],
            0x63: [this.srH, this.srE],
            0x64: [this.srH, this.srH],
            0x65: [this.srH, this.srL],
            0x67: [this.srH, this.srA],

            0x68: [this.srL, this.srB],
            0x69: [this.srL, this.srC],
            0x6a: [this.srL, this.srD],
            0x6b: [this.srL, this.srE],
            0x6c: [this.srL, this.srH],
            0x6d: [this.srL, this.srL],
            0x6f: [this.srL, this.srA],

            0x78: [this.srA, this.srB],
            0x79: [this.srA, this.srC],
            0x7a: [this.srA, this.srD],
            0x7b: [this.srA, this.srE],
            0x7c: [this.srA, this.srH],
            0x7d: [this.srA, this.srL],
            0x7f: [this.srA, this.srA],
        },
        ([to, from]) =>
            () => {
                to.set(from.get());
                return null;
            }
    ),
    // LD B/C/D/E/H/L/A (HL)
    ...this.generateOperation(
        {
            0x46: this.srB,
            0x4e: this.srC,
            0x56: this.srD,
            0x5e: this.srE,
            0x66: this.srH,
            0x6e: this.srL,
            0x7e: this.srA,
        },
        (r) =>
            this.readAddress(
                () => this.regHL.get(),
                (value) => {
                    r.set(value);
                    return () => null;
                }
            )
    ),
    // LD (HL), B/C/D/E/H/L/A
    ...this.generateOperation(
        {
            0x70: this.srB,
            0x71: this.srC,
            0x72: this.srD,
            0x73: this.srE,
            0x74: this.srH,
            0x75: this.srL,
            0x77: this.srA,
        },
        (r) => (system) => {
            const address = this.regHL.get();
```

```
                system.write(address, r.get());
                return () => null;
            }
        ),
        // ADD A, B/C/D/E/H/L/A/(HL)/d8
        ...this.generateOperation(
            {
                0x80: this.srB,
                0x81: this.srC,
                0x82: this.srD,
                0x83: this.srE,
                0x84: this.srH,
                0x85: this.srL,
                0x87: this.srA,
            },
            (r) => () => {
                this.addNToA(r.get(), false);
                return null;
            }
        ),
        0x86: this.readAddress(
            () => this.regHL.get(),
            (value) => () => {
                this.addNToA(value, false);
                return null;
            }
        ),
        0xc6: this.nextByte((value) => () => {
            this.addNToA(value, false);
            return null;
        }),
        // ADDC A, B/C/D/E/H/L/A/(HL)/d8
        ...this.generateOperation(
            {
                0x88: this.srB,
                0x89: this.srC,
                0x8a: this.srD,
                0x8b: this.srE,
                0x8c: this.srH,
                0x8d: this.srL,
                0x8f: this.srA,
            },
            (r) => () => {
                this.addNToA(r.get(), true);
                return null;
            }
        ),
        0x8e: this.readAddress(
            () => this.regHL.get(),
            (value) => () => {
                this.addNToA(value, true);
                return null;
            }
        ),
        0xce: this.nextByte((value) => () => {
            this.addNToA(value, true);
            return null;
        }),
        // SUB A, B/C/D/E/H/L/A/(HL)/d8
        ...this.generateOperation(
            {
                0x90: this.srB,
                0x91: this.srC,
                0x92: this.srD,
                0x93: this.srE,
                0x94: this.srH,
                0x95: this.srL,
                0x97: this.srA,
            },
```

```
            (r) => () => {
                this.subNFromA(r.get(), false);
                return null;
            }
        ),
        0x96: this.readAddress(
            () => this.regHL.get(),
            (value) => () => {
                this.subNFromA(value, false);
                return null;
            }
        ),
        0xd6: this.nextByte((value) => () => {
            this.subNFromA(value, false);
            return null;
        }),
        // SBC A, B/C/D/E/H/L/A/(HL)/d8
        ...this.generateOperation(
            {
                0x98: this.srB,
                0x99: this.srC,
                0x9a: this.srD,
                0x9b: this.srE,
                0x9c: this.srH,
                0x9d: this.srL,
                0x9f: this.srA,
            },
            (r) => () => {
                this.subNFromA(r.get(), true);
                return null;
            }
        ),
        0x9e: this.readAddress(
            () => this.regHL.get(),
            (value) => () => {
                this.subNFromA(value, true);
                return null;
            }
        ),
        0xde: this.nextByte((value) => () => {
            this.subNFromA(value, true);
            return null;
        }),
        // AND/XOR/OR B/C/D/E/H/L/A
        ...this.generateOperation<number, [Register, "&" | "|" | "^"]>(
            {
                0xa0: [this.srB, "&"],
                0xa1: [this.srC, "&"],
                0xa2: [this.srD, "&"],
                0xa3: [this.srE, "&"],
                0xa4: [this.srH, "&"],
                0xa5: [this.srL, "&"],
                0xa7: [this.srA, "&"],

                0xa8: [this.srB, "^"],
                0xa9: [this.srC, "^"],
                0xaa: [this.srD, "^"],
                0xab: [this.srE, "^"],
                0xac: [this.srH, "^"],
                0xad: [this.srL, "^"],
                0xaf: [this.srA, "^"],

                0xb0: [this.srB, "|"],
                0xb1: [this.srC, "|"],
                0xb2: [this.srD, "|"],
                0xb3: [this.srE, "|"],
                0xb4: [this.srH, "|"],
                0xb5: [this.srL, "|"],
                0xb7: [this.srA, "|"],
```

```
            },
            ([r, op]) =>
                () => {
                    this.boolNToA(r.get(), op);
                    return null;
                }
        ),
        // AND/XOR/OR (HL)
        ...this.generateOperation(
            {
                0xa6: "&" as const,
                0xae: "^" as const,
                0xb6: "|" as const,
            },
            (op) =>
                this.readAddress(
                    () => this.regHL.get(),
                    (value) => () => {
                        this.boolNToA(value, op);
                        return null;
                    }
                )
        ),
        // AND/XOR/OR d8
        ...this.generateOperation(
            {
                0xe6: "&" as const,
                0xee: "^" as const,
                0xf6: "|" as const,
            },
            (op) =>
                this.nextByte((value) => () => {
                    this.boolNToA(value, op);
                    return null;
                })
        ),
        // CP B/C/D/E/H/L/A/(HL)/d8
        ...this.generateOperation(
            {
                0xb8: this.srB,
                0xb9: this.srC,
                0xba: this.srD,
                0xbb: this.srE,
                0xbc: this.srH,
                0xbd: this.srL,
                0xbf: this.srA,
            },
            (r) => () => {
                this.compNToA(r.get());
                return null;
            }
        ),
        0xbe: this.readAddress(
            () => this.regHL.get(),
            (value) => () => {
                this.compNToA(value);
                return null;
            }
        ),
        0xfe: this.nextByte((value) => (s) => {
            this.compNToA(value);
            return null;
        }),
        // LD (a8), A
        0xe0: this.nextByte((address) => (system) => {
            const value = this.srA.get();
            system.write(0xff00 | address, value);
            return () => null;
        }),
```

```
        // LD A, (a8)
        0xf0: this.nextByte((address) =>
            this.readAddress(0xff00 | address, (data) => () => {
                this.srA.set(data);
                return null;
            })
        ),
        // LD (C), A
        0xe2: (s) => {
            s.write(0xff00 | this.srC.get(), this.srA.get());
            return () => null;
        },
        // LD A, (C)
        0xf2: this.readAddress(
            () => 0xff00 | this.srC.get(),
            (value) => () => {
                this.srA.set(value);
                return null;
            }
        ),
        // LD (a16), A
        0xea: this.nextWord((address) => (system) => {
            const value = this.srA.get();
            system.write(address, value);
            return () => null;
        }),
        // LD A, (a16)
        0xfa: this.nextWord((value) => (s) => {
            const address = s.read(value);
            return () => {
                this.srA.set(address);
                return null;
            };
        }),
        // RST 0/1/2/3/4/5/6/7
        ...this.generateOperation(
            {
                0xc7: 0x00,
                0xcf: 0x08,
                0xd7: 0x10,
                0xdf: 0x18,
                0xe7: 0x20,
                0xef: 0x28,
                0xf7: 0x30,
                0xff: 0x38,
            },
            (jumpAdr) => this.call(jumpAdr, () => () => null)
        ),
        // CALL a16
        0xcd: this.nextWord((value) => this.call(value, () => () => null)),
        // CALL NZ/Z/NC/C a16
        ...this.generateOperation(
            {
                0xc4: () => !this.flag(FLAG_ZERO),
                0xcc: () => this.flag(FLAG_ZERO),
                0xd4: () => !this.flag(FLAG_CARRY),
                0xdc: () => this.flag(FLAG_CARRY),
            },
            (condition) =>
                this.nextWord((value) =>
                    condition() ? this.call(value, () => () => null) : () => null
                )
        ),
        // RET
        0xc9: this.return(() => () => null),
        // RETI
        0xd9: this.return((s, i) => {
            i.enableInterrupts();
            return () => null;
```

```
        }),
        // RET Z/C/NZ/NC
        ...this.generateOperation(
            {
                0xc0: () => !this.flag(FLAG_ZERO),
                0xc8: () => this.flag(FLAG_ZERO),
                0xd0: () => !this.flag(FLAG_CARRY),
                0xd8: () => this.flag(FLAG_CARRY),
            },
            (condition) => () => condition() ? this.return(() => () => null) : () => null
        ),
        // JP a16
        0xc3: this.nextWord((value) => this.jump(value, () => () => null)),
        // JP HL
        0xe9: this.jump(
            () => this.regHL.get(),
            () => null
        ),
        // JP Z/C/NZ/NC, a16
        ...this.generateOperation(
            {
                0xc2: () => !this.flag(FLAG_ZERO),
                0xca: () => this.flag(FLAG_ZERO),
                0xd2: () => !this.flag(FLAG_CARRY),
                0xda: () => this.flag(FLAG_CARRY),
            },
            (condition) =>
                this.nextWord(
                    (value) => () => condition() ? this.jump(value, () => null) : null
                )
        ),
        // JR s8
        0x18: this.nextByte((value) => this.jumpr(asSignedInt8(value), () => () => null)),
        // JR NZ/Z/NC/C, s8
        ...this.generateOperation(
            {
                0x20: () => !this.flag(FLAG_ZERO),
                0x28: () => this.flag(FLAG_ZERO),
                0x30: () => !this.flag(FLAG_CARRY),
                0x38: () => this.flag(FLAG_CARRY),
            },
            (condition) =>
                this.nextByte(
                    (value) => () =>
                        condition() ? this.jumpr(asSignedInt8(value), () => null) : null
                )
        ),
        // POP BC/DE/HL/AF
        ...this.generateOperation(
            {
                0xc1: this.regBC,
                0xd1: this.regDE,
                0xe1: this.regHL,
            },
            (r) =>
                this.pop((value) => () => {
                    r.set(value);
                    return null;
                })
        ),
        // We need to mask lower 4 bits bc hardwired to 0
        0xf1: this.pop((value) => () => {
            this.regAF.set(value & 0xfff0);
            return null;
        }),
        // PUSH BC/DE/HL/AF
        ...this.generateOperation(
            {
                0xc5: this.regBC,
```

```
            0xd5: this.regDE,
            0xe5: this.regHL,
            0xf5: this.regAF,
        },
        (register) =>
            this.push(
                () => register.get(),
                () => () => null
            )
    ),
    // RLCA / RLA / RRCA / RRA
    0x07: () => {
        this.rotateLSr(this.srA, false, false);
        return null;
    },
    0x17: () => {
        this.rotateLSr(this.srA, true, false);
        return null;
    },
    0x0f: () => {
        this.rotateRSr(this.srA, false, false);
        return null;
    },
    0x1f: () => {
        this.rotateRSr(this.srA, true, false);
        return null;
    },
    // ADD SP, s8
    0xe8: this.nextByte((value) => () => {
        const s8 = asSignedInt8(value);
        const sp = this.regSP.get();
        this.regSP.set(this.perfAdd(s8, sp));
        return () => () => null; // 3 cycles (idk the timing yet)
    }),
    // LD HL, SP+s8
    0xf8: this.nextByte((value) => () => {
        const s8 = asSignedInt8(value);
        const sp = this.regSP.get();
        this.regHL.set(this.perfAdd(s8, sp));
        return () => null;
    }),
    // LD SP, HL
    0xf9: () => {
        this.regSP.set(this.regHL.get());
        return () => null;
    },
    // DI / EI
    0xf3: (s, i) => {
        i.disableInterrupts();
        return null;
    },
    0xfb: (s, i) => {
        i.enableInterrupts();
        return null;
    },
    // HALT
    0x76: (s, interrupts) => {
        this.halted = true;
        if (!interrupts.fastEnableInterrupts() && interrupts.hasPendingInterrupt) {
            this.haltBug = true; // halt bug triggered on HALT when IME == 0 & IE&IF != 0
        }
        return null;
    },
    // SCF / CCF
    0x37: () => {
        this.setFlag(FLAG_SUBSTRACTION, false);
        this.setFlag(FLAG_HALFCARRY, false);
        this.setFlag(FLAG_CARRY, true);
        return null;
```

```
        },
        0x3f: () => {
            this.setFlag(FLAG_SUBSTRACTION, false);
            this.setFlag(FLAG_HALFCARRY, false);
            this.setFlag(FLAG_CARRY, !this.flag(FLAG_CARRY));
            return null;
        },
        // DAA
        0x27: () => {
            let a = this.srA.get();
            let adjust = this.flag(FLAG_CARRY) ? 0x60 : 0x00;
            if (this.flag(FLAG_HALFCARRY)) {
                adjust |= 0x06;
            }
            if (!this.flag(FLAG_SUBSTRACTION)) {
                if ((a & 0x0f) > 0x09) adjust |= 0x06;
                if (a > 0x99) adjust |= 0x60;
            }

            a = wrap8(a + (this.flag(FLAG_SUBSTRACTION) ? -adjust : adjust));
            this.srA.set(a);
            this.setFlag(FLAG_CARRY, adjust >= 0x60);
            this.setFlag(FLAG_HALFCARRY, false);
            this.setFlag(FLAG_ZERO, a === 0);
            return null;
        },
        // CPL
        0x2f: () => {
            this.srA.set(~this.srA.get() & 0xff);
            this.setFlag(FLAG_SUBSTRACTION, true);
            this.setFlag(FLAG_HALFCARRY, true);
            return null;
        },
    };

    /**
     * A list of all 16-bit opcodes. Works the same as instructionSet.
     */
    protected extendedInstructionSet: Partial<Record<number, InstructionMethod>> = {
        // RLC ...
        ...this.generateExtendedOperation(0x00, ({ get, set }) =>
            get((value) => set(this.rotateL(value, false, true)))
        ),
        // RRC ...
        ...this.generateExtendedOperation(0x08, ({ get, set }) =>
            get((value) => set(this.rotateR(value, false, true)))
        ),
        // RL ...
        ...this.generateExtendedOperation(0x10, ({ get, set }) =>
            get((value) => set(this.rotateL(value, true, true)))
        ),
        // RC ...
        ...this.generateExtendedOperation(0x18, ({ get, set }) =>
            get((value) => set(this.rotateR(value, true, true)))
        ),
        // SLA ...
        ...this.generateExtendedOperation(0x20, ({ get, set }) =>
            get((value) => {
                const result = (value << 1) & 0xff;
                this.setFlag(FLAG_ZERO, result === 0);
                this.setFlag(FLAG_SUBSTRACTION, false);
                this.setFlag(FLAG_HALFCARRY, false);
                this.setFlag(FLAG_CARRY, ((value >> 7) & 0b1) === 1);
                return set(result);
            })
        ),
        // SRA ...
        ...this.generateExtendedOperation(0x28, ({ get, set }) =>
            get((value) => {
```

```
                const result = ((value >> 1) & 0xff) | (value & (1 << 7)); // bit 7 left unchanged
                this.setFlag(FLAG_ZERO, result === 0);
                this.setFlag(FLAG_SUBSTRACTION, false);
                this.setFlag(FLAG_HALFCARRY, false);
                this.setFlag(FLAG_CARRY, (value & 0b1) === 1);
                return set(result);
            })
        ),
        // SRL ...
        ...this.generateExtendedOperation(0x38, ({ get, set }) =>
            get((value) => {
                const result = (value >> 1) & 0xff;
                this.setFlag(FLAG_ZERO, result === 0);
                this.setFlag(FLAG_SUBSTRACTION, false);
                this.setFlag(FLAG_HALFCARRY, false);
                this.setFlag(FLAG_CARRY, (value & 0b1) === 1);
                return set(result);
            })
        ),
        // SWAP ...
        ...this.generateExtendedOperation(0x30, ({ get, set }) =>
            get((value) => {
                const result = ((value & 0x0f) << 4) | ((value & 0xf0) >> 4);
                this.setFlag(FLAG_ZERO, result === 0);
                this.setFlag(FLAG_SUBSTRACTION, false);
                this.setFlag(FLAG_HALFCARRY, false);
                this.setFlag(FLAG_CARRY, false);
                return set(result);
            })
        ),
        // BIT 0/1/2/.../7, ...
        ...[...new Array(8)].reduce(
            (previous, _, bit) => ({
                ...previous,
                ...this.generateExtendedOperation(0x40 + bit * 8, ({ get }) =>
                    get((value) => {
                        const out = (value >> bit) & 0b1;
                        this.setFlag(FLAG_ZERO, out === 0);
                        this.setFlag(FLAG_SUBSTRACTION, false);
                        this.setFlag(FLAG_HALFCARRY, true);
                        return null;
                    })
                ),
            }),
            {} as Partial<Record<number, InstructionMethod>>
        ),
        // RES 0/1/2/.../7, ...
        ...[...new Array(8)].reduce(
            (previous, _, bit) => ({
                ...previous,
                ...this.generateExtendedOperation(0x80 + bit * 8, ({ get, set }) =>
                    get((value) => {
                        const result = value & ~(1 << bit);
                        return set(result);
                    })
                ),
            }),
            {} as Partial<Record<number, InstructionMethod>>
        ),
        // SET 0/1/2/.../7, ...
        ...[...new Array(8)].reduce(
            (previous, _, bit) => ({
                ...previous,
                ...this.generateExtendedOperation(0xc0 + bit * 8, ({ get, set }) =>
                    get((value) => {
                        const result = value | (1 << bit);
                        return set(result);
                    })
```

```
            ),
        }),
        {} as Partial<Record<number, InstructionMethod>>
    ),
};

// Helper functions for instructions
/** Reads flags */
protected flag(flag: number): boolean {
    return this.regAF.l.flag(flag);
}
/** Sets flags */
protected setFlag(flag: number, state: boolean) {
    this.regAF.l.sflag(flag, state);
}
/** Increments an 8bit value (wrapping), updates flags Z/0/H */
protected incN(n: number): number {
    const result = wrap8(n + 1);
    this.setFlag(FLAG_ZERO, result === 0);
    this.setFlag(FLAG_SUBSTRACTION, false);
    this.setFlag(FLAG_HALFCARRY, (result & 0xf) < (n & 0xf));
    return result;
}

/** Decrements an 8bit value (wrapping), updates flags Z/1/H */
protected decN(n: number): number {
    const result = wrap8(n - 1);
    this.setFlag(FLAG_ZERO, result === 0);
    this.setFlag(FLAG_SUBSTRACTION, true);
    this.setFlag(FLAG_HALFCARRY, (result & 0xf) > (n & 0xf));
    return result;
}

/** Adds a value to subregister A, updates flags Z/0/H/CY */
protected addNToA(n: number, carry: boolean) {
    const a = this.srA.get();
    const carryVal = carry && this.flag(FLAG_CARRY) ? 1 : 0;
    const result = wrap8(a + n + carryVal);
    this.srA.set(result);
    this.setFlag(FLAG_ZERO, result === 0);
    this.setFlag(FLAG_SUBSTRACTION, false);
    this.setFlag(FLAG_HALFCARRY, (a & 0xf) + (n & 0xf) + carryVal > 0xf);
    this.setFlag(FLAG_CARRY, a + n + carryVal > 0xff);
}
/** Adds the two given 16-bit values (updating flags), returns the result */
protected perfAdd(a: number, b: number) {
    const result = wrap16(a + b);
    this.setFlag(FLAG_ZERO, false);
    this.setFlag(FLAG_SUBSTRACTION, false);
    this.setFlag(FLAG_CARRY, (a & 0xff) > 0xff - (b & 0xff));
    this.setFlag(FLAG_HALFCARRY, (a & 0xf) > 0xf - (b & 0xf));
    return result;
}
/** Substracts a value from subregister A, updates flags Z/1/H/CY */
protected subNFromA(n: number, carry: boolean) {
    const a = this.srA.get();
    const carryVal = carry && this.flag(FLAG_CARRY) ? 1 : 0;
    const result = wrap8(a - n - carryVal);
    this.srA.set(result);
    this.setFlag(FLAG_ZERO, result === 0);
    this.setFlag(FLAG_SUBSTRACTION, true);
    this.setFlag(FLAG_HALFCARRY, (a & 0xf) - (n & 0xf) - carryVal < 0);
    this.setFlag(FLAG_CARRY, a - n - carryVal < 0);
}
/** Stores the given boolean operation of A and the given value in A, updates Z/0/H/0 */
protected boolNToA(n: number, op: "&" | "|" | "^") {
    const a = this.srA.get();
    const result = op === "&" ? a & n : op === "|" ? a | n : a ^ n;
    this.srA.set(result);
```

```
        this.setFlag(FLAG_ZERO, result === 0);
        this.setFlag(FLAG_SUBSTRACTION, false);
        this.setFlag(FLAG_HALFCARRY, op === "&");
        this.setFlag(FLAG_CARRY, false);
    }
    /** Compares the given number with the value in A without changing A, updates Z/1/H/CY */
    protected compNToA(n: number) {
        const a = this.srA.get();
        this.subNFromA(n, false);
        this.srA.set(a);
    }
    /**
     * Pushes the given data to the stack pointer's position, and moves it back by two
     * Takes 3 cycles
     */
    protected push(
        data: number | (() => number),
        receiver: () => InstructionReturn
    ): InstructionMethod {
        return () => (system) => {
            const effectiveData = typeof data === "number" ? data : data();
            this.regSP.dec();
            system.write(this.regSP.get(), high(effectiveData));
            return (system) => {
                this.regSP.dec();
                system.write(this.regSP.get(), low(effectiveData));
                return receiver();
            };
        };
    }
    /**
     * Pops a 16bit address from the stack pointer's position, and moves it forward by two.
     * Takes two cycles.
     */
    protected pop(receiver: (value: number) => InstructionMethod): InstructionMethod {
        return (s) => {
            const low = s.read(this.regSP.inc());
            return (s) => {
                const high = s.read(this.regSP.inc());
                return receiver(combine(high, low));
            };
        };
    }
    /**
     * Pushes the current PC to memory, and jump to the given address.
     * Takes 3 cycles.
     */
    protected call(address: number, receiver: () => InstructionReturn): InstructionMethod {
        return this.push(
            () => this.regPC.get(),
            () => {
                this.regPC.set(address);
                return receiver();
            }
        );
    }

    /**
     * Returns the current call (ie. consumes a pointer at SP and sets it to PC).
     * Takes 3 cycles.
     */
    protected return(receiver: InstructionMethod): InstructionMethod {
        return this.pop((value) => (s, i) => {
            this.regPC.set(value);
            return receiver(s, i);
        });
    }
    /**
     * Jumps to the given 16bit address
```

```
     * Takes one cycle
     */
    protected jump(
        n: number | (() => number),
        receiver: () => InstructionReturn
    ): InstructionMethod {
        return () => {
            const address = typeof n === "number" ? n : n();
            this.regPC.set(address);
            return receiver();
        };
    }
    /**
     * Relative-jumps by the given 8-bit value
     * Takes one cycle
     */
    protected jumpr(
        n: number | (() => number),
        receiver: () => InstructionReturn
    ): InstructionMethod {
        return () => {
            const address = typeof n === "number" ? n : n();
            this.regPC.set(wrap16(this.regPC.get() + address));
            return receiver();
        };
    }
    /** Rotates the given number left. Sets flags Z|0/0/0/N7 */
    protected rotateL(n: number, useCarry: boolean, setZero: boolean) {
        const bit7 = (n >> 7) & 0b1;
        const cflag = this.flag(FLAG_CARRY) ? 1 : 0;
        const result = ((n << 1) & 0xff) | (useCarry ? cflag : bit7);
        this.setFlag(FLAG_ZERO, setZero && result === 0);
        this.setFlag(FLAG_SUBSTRACTION, false);
        this.setFlag(FLAG_HALFCARRY, false);
        this.setFlag(FLAG_CARRY, bit7 === 1);
        return result;
    }
    /** Applies rotateL to a subregister. Sets flags Z|0/0/0/Sr7 */
    protected rotateLSr(sr: Register, useCarry: boolean, setZero: boolean) {
        sr.set(this.rotateL(sr.get(), useCarry, setZero));
    }
    /** Rotates the given number right. Sets flags Z|0/0/0/N0 */
    protected rotateR(n: number, useCarry: boolean, setZero: boolean) {
        const bit0 = n & 0b1;
        const cflag = this.flag(FLAG_CARRY) ? 1 : 0;
        const result = ((n >> 1) & 0xff) | ((useCarry ? cflag : bit0) << 7);
        this.setFlag(FLAG_ZERO, setZero && result === 0);
        this.setFlag(FLAG_SUBSTRACTION, false);
        this.setFlag(FLAG_HALFCARRY, false);
        this.setFlag(FLAG_CARRY, bit0 === 1);
        return result;
    }
    /** Applies rotateR to a subregister. Sets flags Z|0/0/0/Sr0 */
    protected rotateRSr(sr: Register, useCarry: boolean, setZero: boolean) {
        sr.set(this.rotateR(sr.get(), useCarry, setZero));
    }

    /**
     * Helper function for instructions that do the same operations for a set of objects.
     * @param items The object the operation runs on, matched to its opcode.
     * @param execute A function that executes the instruction for a given object.
     * @returns An object with the completed instructions
     */
    protected generateOperation<K extends number, T>(
        items: Record<K, T>,
        execute: (r: T) => InstructionMethod
    ): Record<K, InstructionMethod> {
        const obj: Record<K, InstructionMethod> = {} as any;
        for (const [opcode, item] of Object.entries(items) as any as [K, T][]) {
```

```
            obj[opcode] = execute(item);
        }
        return obj;
    }

    /**
     * Helper function for instructions that follow the same B-C-D-E-H-L-(HL)-A pattern
     * @param baseCode The base code of the instruction (e.g. 0x50)
     * @param execute A function that executes the instruction for a given register
     * @returns An object with the completed instructions (e.g. 0x50, 0x51, ..., 0x57)
     */
    protected generateExtendedOperation(
        baseCode: number,
        execute: (r: {
            get: (r: (value: number) => InstructionReturn) => InstructionReturn;
            set: (x: number) => InstructionReturn;
        }) => InstructionReturn
    ): Partial<Record<number, InstructionMethod>> {
        const make = (sr: Register) => ({
            get: (r: (value: number) => InstructionReturn) => {
                const value = sr.get();
                return r(value);
            },
            set: (x: number) => {
                sr.set(x);
                return null;
            },
        });
        const subregisters = {
            b: make(this.srB),
            c: make(this.srC),
            d: make(this.srD),
            e: make(this.srE),
            h: make(this.srH),
            l: make(this.srL),
            a: make(this.srA),
        };
        // order matters: B/C/D/E/H/L/(HL)/A
        return {
            [baseCode + 0]: (s) => execute(subregisters.b),
            [baseCode + 1]: (s) => execute(subregisters.c),
            [baseCode + 2]: (s) => execute(subregisters.d),
            [baseCode + 3]: (s) => execute(subregisters.e),
            [baseCode + 4]: (s) => execute(subregisters.h),
            [baseCode + 5]: (s) => execute(subregisters.l),
            [baseCode + 6]: (s) =>
                execute({
                    get: (r: (value: number) => InstructionReturn) => {
                        const value = s.read(this.regHL.get());
                        return () => r(value);
                    },
                    set: (x: number) => {
                        s.write(this.regHL.get(), x);
                        return () => null;
                    },
                }),
            [baseCode + 7]: (s) => execute(subregisters.a),
        };
    }
}

export default CPU;
```

B.1.3 System

System (src/emulator/System.ts)

```ts
import APU from "./apu/APU";
import { CGBMode, ConsoleType, HRAM_SIZE, SpeedMode } from "./constants";
import GameBoyInput from "./GameBoyInput";
import PPU from "./ppu/PPU";
import JoypadInput from "./JoypadInput";
import { CircularRAM, RAM, Addressable, ROM } from "./Memory";
import { MaskRegister, Register00, RegisterFF, Register } from "./Register";
import GameCartridge from "./GameCartridge";
import Timer from "./Timer";
import GameBoyOutput from "./GameBoyOutput";
import { Int4, rangeObject } from "./util";
import { DMGWRAM, GBCWRAM } from "./WRAM";
import Interrupts from "./Interrupts";

const KEY0_DISABLE_ALL = 1 << 2;
const KEY0_DISABLE_SOME = 1 << 3;

class System implements Addressable {
    // General use
    protected mode: ConsoleType;

    // Devices
    protected timer = new Timer();
    protected apu: APU;
    protected joypad: JoypadInput;
    protected ppu: PPU;
    protected interrupts: Interrupts = new Interrupts();

    // Memory
    protected bootRom: ROM;
    protected cartridge: GameCartridge;
    protected wram: Addressable;
    protected hram: RAM = new CircularRAM(HRAM_SIZE, 0xff80);

    // System registers
    protected bootRomLocked = false;
    protected bootRomRegister: Addressable = {
        read: () => (this.bootRomLocked ? 0xff : 0xfe),
        write: (pos, value) => (this.bootRomLocked ||= (value & 1) === 1),
    };

    // KEY0: CGB features toggle (CGB Register)
    protected cgbMode: CGBMode = CGBMode.CGB;
    protected key0Register: Addressable = {
        read: () => {
            return 0x0;
        },
        write: (_, value) => {
            if (this.bootRomLocked) return; // becomes read-only after boot rom is disabled

            if (value & KEY0_DISABLE_SOME) this.cgbMode = CGBMode.DMGExtended;
            else if (value & KEY0_DISABLE_ALL) this.cgbMode = CGBMode.DMG;
            else this.cgbMode = CGBMode.CGB;

            this.ppu.setCGBMode(this.cgbMode);
            this.addressesRegisters[0x4d] =
                this.cgbMode === CGBMode.CGB ? this.key1Register : undefined;
        },
    };

    // KEY1: Speed switch register (CGB Register)
    protected speedMode: SpeedMode = SpeedMode.Normal;
    protected wantsSpeedModeChange = false;
    protected key1Register: Addressable = {
```

```
65          read: () =>
66              (this.speedMode === SpeedMode.Double ? 1 << 7 : 0) |
67              (this.wantsSpeedModeChange ? 1 << 0 : 0),
68          write: (_, value) => (this.wantsSpeedModeChange = (value & 1) === 1),
69      };
70
71      constructor(
72          rom: Uint8Array,
73          input: GameBoyInput,
74          output: GameBoyOutput,
75          mode: ConsoleType
76      ) {
77          this.mode = mode;
78          this.bootRom = new ROM(mode === ConsoleType.DMG ? 0x100 : 0x900);
79          this.cartridge = new GameCartridge(rom);
80          this.ppu = new PPU(mode);
81          this.wram = mode === ConsoleType.DMG ? new DMGWRAM() : new GBCWRAM();
82          this.joypad = new JoypadInput(input);
83          this.apu = new APU(output);
84
85          const registerSerial: Addressable = {
86              read: () => 0xff,
87              write: (pos, value) => output.serialOut && output.serialOut(value),
88          };
89
90          this.addressesLastNibble = {
91              ...rangeObject(0x0, 0x7, this.cartridge),
92              ...rangeObject(0x8, 0x9, this.ppu),
93              ...rangeObject(0xa, 0xb, this.cartridge),
94              ...rangeObject(0xc, 0xe, this.wram), // wram and echo
95              0xf: undefined, // handled separately
96          };
97
98          this.addressesRegisters = {
99              0x00: this.joypad, // joypad
100             0x01: registerSerial, // SB - serial data
101             0x02: RegisterFF, // CB - serial control
102             ...rangeObject(0x04, 0x07, this.timer), // timer registers
103             0x0f: this.interrupts, // IF
104             ...rangeObject(0x10, 0x26, this.apu), // actual apu registers
105             ...rangeObject(0x30, 0x3f, this.apu), // wave ram
106             ...rangeObject(0x40, 0x4b, this.ppu), // ppu registers
107             0x4c: mode === ConsoleType.CGB ? this.key0Register : undefined, // KEY0 -
108             0x4d: mode === ConsoleType.CGB ? this.key1Register : undefined, // KEY1 - speed
    ↪ switch
109             0x4f: this.ppu, // ppu vram bank register
110             0x50: this.bootRomRegister, // boot rom register
111             ...rangeObject(0x51, 0x55, this.ppu), // ppu vram dma registers
112             ...rangeObject(0x68, 0x6b, this.ppu), // ppu palette registers (CGB only)
113             0x70: mode === ConsoleType.CGB ? this.wram : undefined, // wram bank register
114             0x72: mode === ConsoleType.CGB ? new Register() : undefined, // undocumented register
115             0x73: mode === ConsoleType.CGB ? new Register() : undefined, // undocumented register
116             0x74: mode === ConsoleType.CGB ? new Register() : undefined, // undocumented register
117             0x75: mode === ConsoleType.CGB ? new MaskRegister(0b1000_1111) : undefined, //
    ↪ undocumented register
118             0x76: mode === ConsoleType.CGB ? this.apu : undefined, // PCM12
119             0x77: mode === ConsoleType.CGB ? this.apu : undefined, // PCM34
120             ...rangeObject(0x80, 0xfe, this.hram), // hram
121             0xff: this.interrupts, // IE
122         };
123     }
124
125     /**
126      * Ticks the whole system for one M-cycle.
127      * @param isMCycle if the tick is a regular M-cycle (everything runs) or a double-speed mode
128      * cycle (the APU and PPU don't run).
129      * @returns if the CPU should be halted (because a VRAM-DMA is in progress).
130      */
131     tick(isMCycle: boolean): boolean {
```

```
        const haltCpu = this.ppu.tick(this, this.interrupts, isMCycle);
        this.timer.tick(this.interrupts);
        if (isMCycle) this.apu.tick(this.timer);
        this.interrupts.tick();

        return haltCpu;
    }

    /**
     * Mapping of addressables dependending on the last (most significant) nibble of an address
     * e.g. 0x0 to 0x7 maps to ROM, etc.
     */
    protected addressesLastNibble: Partial<Record<Int4, Addressable>>;
    /**
     * Mapping of addressables depending on the first (most significant) byte of an address -
     * this is only applicable to the 0xff00 to 0xffff range.
     * e.g. 0x00 maps to joypad, 0x01 maps to serial,0x04 to 0x07 maps to timer, etc.
     */
    protected addressesRegisters: Partial<Record<number, Addressable>>;

    /**
     * Responsible for following the memory map.
     * @link https://gbdev.io/pandocs/Memory_Map.html#memory-map
     */
    protected getAddress(pos: number): Addressable {
        if (pos < 0x0000 || pos > 0xffff)
            throw new Error(`Invalid address to read from ${pos.toString(16)}`);

        // Boot ROM
        if (!this.bootRomLocked) {
            if (pos < 0x100) return this.bootRom;
            // (the CGB's boot rom extends to 0x900, but leaves a gap for the header)
            if (this.mode === ConsoleType.CGB && 0x200 <= pos && pos < 0x900)
                return this.bootRom;
        }

        // Checking last nibble
        let addressable = this.addressesLastNibble[(pos >> 12) as Int4];
        if (addressable) return addressable;

        // Registers
        if ((pos & 0xff00) === 0xff00) {
            addressable = this.addressesRegisters[pos & 0xff];
            if (addressable) return addressable;
        }

        // Echo RAM
        if (pos <= 0xfdff) return this.wram;
        // OAM
        if (pos <= 0xfe9f) return this.ppu;
        // Illegal Area
        if (pos <= 0xfeff) return Register00;

        console.debug(
            `Accessed unmapped area ${pos
                .toString(16)
                .padStart(4, "0")}, return a fake 0xff register`
        );
        return RegisterFF;
    }

    /**
     * Reads the data at the given 16-bit address. This method will follow the memory map and
     * return the data belonging to the right component.
     */
    read(pos: number): number {
        return this.getAddress(pos).read(pos);
    }
    /**
```

```
     * Write the 8-bit data at the given 16-bit address. This method will follow the memory map
     * and write the data in the right component.
     */
    write(pos: number, data: number): void {
        this.getAddress(pos).write(pos, data);
    }

    /**
     * When the STOP (0x10) instruction is executed, the system clock will stop. If a speed
     * mode change is requested, this will be applied and the system will continue.
     */
    didStopInstruction() {
        if (this.wantsSpeedModeChange) {
            this.wantsSpeedModeChange = false;
            this.speedMode =
                this.speedMode === SpeedMode.Double ? SpeedMode.Normal : SpeedMode.Double;
        }
    }

    getSpeedMode(): SpeedMode {
        return this.speedMode;
    }

    getInterrupts(): Interrupts {
        return this.interrupts;
    }

    /**
     * Sets the boot ROM data. This will be used on the start up of the system, if the boot ROM
     * is not skipped.
     */
    loadBootRom(data: Uint8Array): void {
        this.bootRom.rawSet(data);
    }

    /**
     * Returns the chain of bytes at the given address, for the given length.
     * @param pos The start position of the inspection
     * @param length The number of inspected bytes
     * @param format The formatting of the values (e.g. 16 for hexadecimal)
     */
    inspect(pos: number, length: number = 16, format: number = 16): string {
        return [...new Array(length)]
            .map((_, index) =>
                this.read(pos + index)
                    .toString(format)
                    .padStart((255).toString(format).length, "0")
            )
            .join(" ");
    }

    /** Reads user input */
    readInput() {
        this.joypad.readInput();
    }

    /** Pushes output data if needed */
    pushOutput(output: GameBoyOutput) {
        this.ppu.pushOutput(output);
    }

    /**
     * @returns if the system supports saving
     */
    supportsSaves(): boolean {
        return this.cartridge.supportsSaves();
    }

    /** Saves the current ROM state (null if no save support). */
```

```
    save(): Uint8Array | null {
        return this.cartridge.save();
    }

    /** Loads the given ROM data. */
    load(data: Uint8Array): void {
        this.cartridge.load(data);
    }

    /** Returns the title of the current ROM. */
    getTitle(): string {
        return this.cartridge.getTitle();
    }

    /** Returns the identifier of the current ROM */
    getIdentifier(): string {
        return this.cartridge.getIdentifier();
    }
}

export default System;
```

B.1.4 PPU

PPU (src/emulator/ppu/PPU.ts)

```ts
import {
    CGBMode,
    ConsoleType,
    IFLAG_LCDC,
    IFLAG_VBLANK,
    SCREEN_HEIGHT,
    SCREEN_WIDTH,
} from "../constants";
import { Addressable } from "../Memory";
import { MaskRegister, RegisterFF, Register } from "../Register";
import { asSignedInt8, Int3, wrap8 } from "../util";
import GameBoyOutput from "../GameBoyOutput";
import OAM, { Sprite } from "./OAM";
import { CGBColorControl, ColorController, DMGColorControl } from "./ColorController";
import { VRAMController, CGBVRAMController, DMGVRAMController } from "./VRAMController";
import Interrupts from "../Interrupts";

type KeyForType<T, V> = NonNullable<
    {
        [k in keyof T]: T[k] extends V ? k : never;
    }[keyof T]
>;

type PPUMode = {
    doTick: KeyForType<PPU, (interrupts: Interrupts) => void>;
    flag: number;
    cycles: number;
};

type PPUModeI = PPUMode & { interrupt: number };

/*
 * All modes, with:
 * - flag: corresponding STAT flag
 * - cycles: cycles until completion (including previous steps)
 * - interrupt?: optional corresponding STAT interrupt flag
 */
const MODE_HBLANK_FIRST: PPUModeI = {
    doTick: "tickHBlankFirst",
    flag: 0b00,
    cycles: 18,
    interrupt: 1 << 3,
};
const MODE_HBLANK: PPUModeI = {
    doTick: "tickHBlank",
    flag: 0b00,
    cycles: 51,
    interrupt: 1 << 3,
};
const MODE_VBLANK: PPUModeI = {
    doTick: "tickVBlank",
    flag: 0b01,
    cycles: 114,
    interrupt: 1 << 4,
};
const MODE_SEARCHING_OAM: PPUModeI = {
    doTick: "tickSearchingOam",
    flag: 0b10,
    cycles: 20,
    interrupt: 1 << 5,
};
const MODE_TRANSFERRING: PPUMode = {
    doTick: "tickTransferring",
    flag: 0b11,
```

```
    cycles: 43,
};

// Helpful constants
const SCREEN_HEIGHT_WOFFSCREEN = 154;

// LCD control flags
const LCDC_BG_WIN_PRIO = 1 << 0;
const LCDC_OBJ_ENABLE = 1 << 1;
const LCDC_OBJ_SIZE = 1 << 2;
const LCDC_BG_TILE_MAP_AREA = 1 << 3;
/** @link https://gbdev.io/pandocs/LCDC.html#lcdc4--bg-and-window-tile-data-area */
const LCDC_BG_WIN_TILE_DATA_AREA = 1 << 4;
const LCDC_WIN_ENABLE = 1 << 5;
const LCDC_WIN_TILE_MAP_AREA = 1 << 6;
const LCDC_LCD_ENABLE = 1 << 7;

// LCD status flags
const STAT_MODE = 0b11;
const STAT_LYC_LY_EQ_FLAG = 1 << 2;
const STAT_LYC_LY_EQ_INT = 1 << 6;

// VRAM2 Attributes
const VRAM2_ATTR_BG_OAM_PRIORITY = 1 << 7;
const VRAM2_ATTR_V_FLIP = 1 << 6;
const VRAM2_ATTR_H_FLIP = 1 << 5;
const VRAM2_ATTR_VRAM_BANK = 1 << 3;
const VRAM2_ATTR_PALETTE = 0b111;

/**
 * The PPU of the GBC, responsible for rendering the current state of the console.
 * @link https://gbdev.io/pandocs/Rendering.html
 */
class PPU implements Addressable {
    // Internal counter for cycles
    cycleCounter: number = 0;
    windowLineCounter: number = 0;
    mode: PPUMode = MODE_VBLANK;

    interruptStateBefore: boolean = false;
    interruptLineState = {
        lycLyMatch: false,
        oamActive: false,
        vblankActive: false,
        hblankActive: false,
    };
    nextInterruptLineUpdate: Partial<typeof this.interruptLineState> | null = null;

    // OAM
    oam = new OAM();
    canReadOam: boolean = true;
    canWriteOam: boolean = true;

    // Variable extra cycles during pixel transfer
    transferExtraCycles: number = 0;

    // Read sprites
    readSprites: Sprite[] = [];

    // Data Store
    vramControl: VRAMController;
    canReadVram: boolean = true;
    canWriteVram: boolean = true;

    // Temporary buffer when drawing line by line
    videoBuffer = new Uint32Array(SCREEN_HEIGHT * SCREEN_WIDTH).fill(0xffffffff);
    // Complete buffer with the last fully drawn frame
    lastVideoOut = new Uint32Array(SCREEN_HEIGHT * SCREEN_WIDTH);
    // Debug video output/storage
```

```
    backgroundVideoBuffer?: Uint32Array;
    tilesetVideoBuffer?: Uint32Array;

    // General use
    /** @link https://gbdev.io/pandocs/LCDC.html */
    lcdControl = new Register(0x00);
    /** @link https://gbdev.io/pandocs/STAT.html */
    lcdStatus = new MaskRegister(0b1000_0000, 0x85);
    /** Only for GBC @link
    ↪ https://gbdev.io/pandocs/CGB_Registers.html#ff6c--opri-cgb-mode-only-object-priority-mode */
    objPriorityMode: Addressable;

    // Positioning
    screenY = new Register(0x00); // these two indicate position of the viewport
    screenX = new Register(0x00); // in the background map

    lcdY = new Register(0x00); // indicates currently drawn horizontal line
    lcdYCompare = new Register(0x00);

    windowY = new Register(0x00); // position of the window
    windowX = new Register(0x00);

    // Color control
    colorControl: ColorController;

    // General use
    consoleMode: ConsoleType;
    cgbMode: CGBMode = CGBMode.DMG; // for cgb only
    isCgbMode: boolean = false;
    protected registerAddresses: Record<number, Addressable> = {};

    constructor(mode: ConsoleType) {
        if (mode === ConsoleType.CGB) {
            this.cgbMode = CGBMode.CGB;
            this.vramControl = new CGBVRAMController(true);
            this.colorControl = new CGBColorControl();
            this.objPriorityMode = new MaskRegister(0b1111_1110);
            this.isCgbMode = true;
        } else {
            this.vramControl = new DMGVRAMController();
            this.colorControl = new DMGColorControl();
            this.objPriorityMode = RegisterFF;
            this.isCgbMode = false;
        }

        this.consoleMode = mode;
        this.updateAddresses();
    }

    updateAddresses(): void {
        this.registerAddresses = {
            0xff40: this.lcdControl,
            0xff41: this.lcdStatus,
            0xff42: this.screenY,
            0xff43: this.screenX,
            0xff44: this.lcdY,
            0xff45: this.lcdYCompare,
            0xff46: this.oam,
            0xff47: this.colorControl, // DMG Palettes
            0xff48: this.colorControl, // |
            0xff49: this.colorControl, // |
            0xff4a: this.windowY,
            0xff4b: this.windowX,
            0xff4f: this.vramControl, // VRAM Bank
            0xff51: this.vramControl, // CGB VRAM DMA
            0xff52: this.vramControl, // |
            0xff53: this.vramControl, // |
            0xff54: this.vramControl, // |
            0xff55: this.vramControl, // |
```

```
            0xff68: this.colorControl, // CGB Palettes
            0xff69: this.colorControl, // |
            0xff6a: this.colorControl, // |
            0xff6b: this.colorControl, // |
            0xff6c: this.objPriorityMode,
        };
    }

    setCGBMode(mode: CGBMode): void {
        this.cgbMode = mode;
        this.isCgbMode = mode !== CGBMode.DMG;

        switch (mode) {
            case CGBMode.CGB:
                this.objPriorityMode = new MaskRegister(0b1111_1110);
                this.vramControl = new CGBVRAMController(true);
                break;

            case CGBMode.DMGExtended:
                this.objPriorityMode = new MaskRegister(0b1111_1110);
                this.vramControl = new CGBVRAMController(false);
                break;

            case CGBMode.DMG:
                this.objPriorityMode = RegisterFF;
                this.vramControl = new DMGVRAMController();
                break;
        }

        this.colorControl.changeCBGMode(mode);

        this.updateAddresses();
    }

    protected haltCpu: boolean = false;

    /**
     * This the PPU, effectively updating the screen-buffer and rendering it if it's done.
     * @param system The system that links all components together
     * @returns Whether the CPU should be halted (a GBC VRAM-DMA is in progress)
     * @link https://gbdev.io/pandocs/pixel_fifo.html
     */
    tick(system: Addressable, interrupts: Interrupts, isMCycle: boolean): boolean {
        const isLcdOn = this.lcdControl.flag(LCDC_LCD_ENABLE);

        this.oam.tick(system);

        this.haltCpu = this.vramControl.tick(
            system,
            this.mode === MODE_HBLANK && this.lcdY.get() < SCREEN_HEIGHT,
            isLcdOn
        );

        if (!isMCycle || !isLcdOn) return this.haltCpu;

        // Update interrupt line from previous write operations?
        if (this.nextInterruptLineUpdate !== null) {
            this.updateInterrupt(interrupts, this.nextInterruptLineUpdate);
            this.nextInterruptLineUpdate = null;
        }

        this.cycleCounter++;

        if (this.cycleCounter === 1) {
            this.setMode(this.mode);
        }

        this[this.mode.doTick](interrupts);

```

```
271         return this.haltCpu;
272     }
273
274     tickHBlankFirst() {
275         if (this.cycleCounter === MODE_HBLANK_FIRST.cycles) {
276             this.cycleCounter = 0;
277             this.mode = MODE_TRANSFERRING;
278         }
279     }
280
281     tickHBlank(interrupts: Interrupts) {
282         if (this.cycleCounter === 1) {
283             this.updateInterrupt(interrupts, { hblankActive: true });
284
285             this.canReadOam = true;
286
287             // For LY <= 1 there is a delay of one extra cycle
288             if (this.lcdY.get() > 1) {
289                 this.canReadVram = true;
290             }
291         }
292
293         if (this.cycleCounter === 2) {
294             this.canReadVram = true;
295             this.canWriteOam = true;
296             this.canWriteVram = true;
297         }
298
299         if (this.cycleCounter === MODE_HBLANK.cycles - this.transferExtraCycles) {
300             this.cycleCounter = 0;
301             const lcdY = this.lcdY.get() + 1;
302             this.lcdY.set(lcdY);
303
304             if (lcdY !== this.lcdYCompare.get()) {
305                 this.updateInterrupt(interrupts, { lycLyMatch: false });
306             }
307
308             if (lcdY === SCREEN_HEIGHT) {
309                 this.mode = MODE_VBLANK;
310             } else {
311                 this.mode = MODE_SEARCHING_OAM;
312             }
313         }
314     }
315
316     tickVBlank(interrupts: Interrupts) {
317         if (this.cycleCounter === 1) {
318             const isVblankStart = this.lcdY.get() === 144;
319             this.updateInterrupt(interrupts, {
320                 lycLyMatch: this.lcdY.get() === this.lcdYCompare.get(),
321                 vblankActive: isVblankStart || this.interruptLineState.vblankActive,
322                 oamActive: isVblankStart || this.interruptLineState.oamActive,
323             });
324
325             if (this.lcdY.get() === 144) {
326                 interrupts.requestInterrupt(IFLAG_VBLANK);
327                 this.lastVideoOut.set(this.videoBuffer);
328             }
329         } else if (this.cycleCounter === 20) {
330             this.updateInterrupt(interrupts, { oamActive: false });
331         } else if (this.cycleCounter === MODE_VBLANK.cycles) {
332             this.cycleCounter = 0;
333             const lcdY = this.lcdY.get() + 1;
334             if (lcdY === SCREEN_HEIGHT_WOFFSCREEN) {
335                 this.lcdY.set(0);
336                 this.windowLineCounter = 0;
337                 this.mode = MODE_SEARCHING_OAM;
338             } else {
339                 this.lcdY.set(lcdY);
```

```
            }
        }
    }

    tickSearchingOam(interrupts: Interrupts) {
        if (this.cycleCounter === 1) {
            this.updateInterrupt(interrupts, {
                oamActive: true,
                hblankActive: false,
                vblankActive: false,
                lycLyMatch: this.lcdY.get() === this.lcdYCompare.get(),
            });
            this.canReadOam = false;
        }

        if (this.cycleCounter === 2) {
            this.canWriteOam = false;
        }

        if (this.cycleCounter === MODE_SEARCHING_OAM.cycles) {
            this.cycleCounter = 0;
            this.mode = MODE_TRANSFERRING;

            // Read the sprite data here ! this should create a copy !!
            const y = this.lcdY.get();
            // Height of objects in pixels
            const objHeight = this.lcdControl.flag(LCDC_OBJ_SIZE) ? 16 : 8;
            // This is only relevant in GBC: priority by position or by index
            let objPrioritySort: (a: [Sprite, number], b: [Sprite, number]) => number;
            if (this.objPriorityMode.read(0) & 1) {
                // priority by X position, then by index
                objPrioritySort = ([spriteA, indexA], [spriteB, indexB]) =>
                    spriteA.x !== spriteB.x ? spriteA.x - spriteB.x : indexA - indexB;
            } else {
                // priority only by index
                objPrioritySort = ([spriteA, indexA], [spriteB, indexB]) => indexA - indexB;
            }
            // We select the sprites the following way:
            // - must be visible
            // - max 10 per line
            // - sorted, first by X position then by index
            this.readSprites = this.oam
                .getSprites()
                // only get selected sprites
                .filter((sprite) => sprite.y <= y && y < sprite.y + objHeight)
                // only 10 sprites per scanline, lower index first
                .slice(0, 10)
                // need to add the index, for sorting
                .map((sprite, index) => [sprite, index] as [Sprite, number])
                .sort(objPrioritySort)
                .map(([sprite]) => sprite);
        }
    }

    tickTransferring() {
        if (this.cycleCounter === 1) {
            this.updateInterrupt(null, { oamActive: false });

            this.canReadOam = false;
            this.canReadVram = false;

            this.transferExtraCycles = 0;

            // Extra cycles are spent during transfer depending on scroll, because the tile is
            // still loaded (the pixels are simply thrown away)
            const offsetX = this.screenX.get() % 8;
            this.transferExtraCycles += Math.ceil(offsetX / 4);

            // When drawing sprites we delay extra 6 cycles per sprite.
```

```
            // We may also delay longer if the sprite if towards the left of the screen, because
            // the PPU must wait for those pixels to be drawn from the background before
            // taking care of the sprites. This delay can thus add up to 5 cycles per
            // X-position.
            // https://gbdev.io/pandocs/STAT.html#properties-of-stat-modes
            if (this.lcdControl.flag(LCDC_OBJ_ENABLE)) {
                let extraSpriteTCycles = 0;
                let lastPenaltyX = NaN;
                let lastPenaltyPaid = false;
                for (let sprite of this.readSprites) {
                    if (lastPenaltyX !== sprite.x || !lastPenaltyPaid) {
                        lastPenaltyX = sprite.x;
                        lastPenaltyPaid = true;
                        extraSpriteTCycles +=
                            5 - Math.min(5, (sprite.x + this.screenX.get()) % 8);
                    }
                    extraSpriteTCycles += 6;
                }
                // Re-convert to M-cycles
                this.transferExtraCycles += Math.floor(extraSpriteTCycles / 4);
            }

            this.updateScanline();
        }

        if (this.cycleCounter === 2) {
            this.canWriteOam = false;
            this.canWriteVram = false;
        }

        if (this.cycleCounter === MODE_TRANSFERRING.cycles + this.transferExtraCycles) {
            this.cycleCounter = 0;
            this.mode = MODE_HBLANK;
        }
    }

    pushOutput(output: GameBoyOutput) {
        if (output.receiveGraphics) {
            output.receiveGraphics(this.lastVideoOut);
        }
        if (output.debugBackground) {
            const backgroundImg = this.debugBackground();
            output.debugBackground(backgroundImg);
        }
        if (output.debugTileset) {
            const tilesetImg = this.debugTileset();
            output.debugTileset(tilesetImg);
        }
    }

    /** Sets the current mode of the PPU, updating the STAT register. */
    setMode(mode: PPUMode) {
        this.lcdStatus.set((this.lcdStatus.get() & ~STAT_MODE) | mode.flag);
    }

    /**
     * Will update the STAT interrupt line, raise an interrupt if there is a high to low
     * transition and the passed in System isn't null (ie. pass null to disable interrupts).
     */
    updateInterrupt(
        interrupts: Interrupts | null,
        data: Partial<typeof this.interruptLineState>
    ) {
        Object.assign(this.interruptLineState, data);
        const lcdStatus = this.lcdStatus.get();
        const interruptState =
            (lcdStatus & STAT_LYC_LY_EQ_INT && this.interruptLineState.lycLyMatch) ||
            (lcdStatus & MODE_HBLANK.interrupt && this.interruptLineState.hblankActive) ||
            (lcdStatus & MODE_VBLANK.interrupt && this.interruptLineState.vblankActive) ||
```

```
            (lcdStatus & MODE_SEARCHING_OAM.interrupt && this.interruptLineState.oamActive);

        this.lcdStatus.sflag(STAT_LYC_LY_EQ_FLAG, this.interruptLineState.lycLyMatch);

        // LCDC Interrupt only happens on rising edges (if allowed)
        if (interrupts && interruptState && !this.interruptStateBefore) {
            interrupts.requestInterrupt(IFLAG_LCDC);
        }
        this.interruptStateBefore = !!interruptState;
    }

    protected bgPriorities = new Uint8Array(SCREEN_WIDTH);

    /** Updates the current scanline, by rendering the background, window and then objects. */
    updateScanline() {
        this.bgPriorities.fill(0);
        // The BG/WIN priority flag acts as a toggle only in DMG
        if (this.isCgbMode || this.lcdControl.flag(LCDC_BG_WIN_PRIO)) {
            this.drawBackground();

            if (this.lcdControl.flag(LCDC_WIN_ENABLE)) {
                this.drawWindow();
            }
        } else {
            this.fillWhite();
        }

        if (this.lcdControl.flag(LCDC_OBJ_ENABLE)) {
            this.drawObjects();
        }
    }

    /** Function to get access to the tile data, ie. the shades of a tile */
    getTileAddress(n: number): number {
        return this.lcdControl.flag(LCDC_BG_WIN_TILE_DATA_AREA)
            ? // Unsigned regular, 0x8000-0x8fff
              0x8000 + n * 16
            : // Signed offset, 0x9000-0x97ff for 0-127 and 0x8800-0x8fff for 128-255
              0x9000 + asSignedInt8(n) * 16;
    }

    debugBackground() {
        const width = 256;
        const height = 256;
        if (this.backgroundVideoBuffer === undefined)
            this.backgroundVideoBuffer = new Uint32Array(width * height);

        // The tilemap used (a map of tile *pointers*)
        const tileMapLoc = this.lcdControl.flag(LCDC_BG_TILE_MAP_AREA) ? 0x9c00 : 0x9800;

        for (let i = 0; i < 1024; i++) {
            // Tile positions (0 <= n < 32)
            const posX = i % 32; // 32 tiles on width
            const posY = Math.floor(i / 32); // 32 tiles on height

            const tileIndex = tileMapLoc + i;

            // On CGB, the attributes of the tile
            // Note we can do this even in DMG mode, because VRAM2 in DMG is just a 00 register,
            // and all the 0 attributes match the normal behaviour of the DMG
            const tileAttributes = this.vramControl.readBank1(tileIndex);
            const flipX = (tileAttributes & VRAM2_ATTR_H_FLIP) !== 0;
            const flipY = (tileAttributes & VRAM2_ATTR_V_FLIP) !== 0;
            const vramBank = (tileAttributes & VRAM2_ATTR_VRAM_BANK) !== 0 ? 1 : 0;
            const tilePalette = (tileAttributes & VRAM2_ATTR_PALETTE) as Int3;

            // Map of colors for each shade
            const palette = this.colorControl.getBgPalette(tilePalette);

```

```
            // The ID (pointer) of the tile
            const tileAddress = this.vramControl.readBank0(tileIndex);
            // Convert the ID to the actual address
            const tileDataAddress = this.getTileAddress(tileAddress);
            // Get the tile data
            const tileData = this.vramControl.getTile(tileDataAddress, vramBank);

            // Draw the 8 lines of the tile
            for (let tileY = 0; tileY < 8; tileY++) {
                for (let tileX = 0; tileX < 8; tileX++) {
                    const arrayX = flipX ? 7 - tileX : tileX;
                    const arrayY = flipY ? 7 - tileY : tileY;
                    const colorId = tileData[arrayX][arrayY];
                    const index = posX * 8 + posY * width * 8 + tileX + tileY * width;
                    this.backgroundVideoBuffer[index] = palette[colorId];
                }
            }
        }

        return this.backgroundVideoBuffer;
    }

    debugTileset() {
        const width = 256; // 16 * 8 * 2;
        const height = 192; // 24 * 8;
        if (this.tilesetVideoBuffer === undefined)
            this.tilesetVideoBuffer = new Uint32Array(width * height);

        // The colors used
        const palette = {
            0b00: 0xffffffff,
            0b01: 0xffaaaaaa,
            0b10: 0xff555555,
            0b11: 0xff000000,
        };

        for (let i = 0; i < 0x180; i++) {
            const tileAddress = 0x8000 + i * 16;
            // Tile positions (0 <= n < 32)
            const posX = i % 16; // 20 tiles on width
            const posY = Math.floor(i / 16);

            // Get tile data (VRAM 0 and 1)
            const tileData1 = this.vramControl.getTile(tileAddress, 0);
            const tileData2 = this.vramControl.getTile(tileAddress, 1);
            // Draw the 8 lines of the tile
            for (let tileX = 0; tileX < 8; tileX++) {
                for (let tileY = 0; tileY < 8; tileY++) {
                    const colorId1 = tileData1[tileX][tileY];
                    const colorId2 = tileData2[tileX][tileY];

                    const index = posX * 8 + posY * width * 8 + tileX + tileY * width;
                    this.tilesetVideoBuffer[index] = palette[colorId1];
                    this.tilesetVideoBuffer[index + width / 2] = palette[colorId2];
                }
            }
        }
        return this.tilesetVideoBuffer;
    }

    fillWhite() {
        const y = this.lcdY.get();
        const bgPalette = this.colorControl.getBgPalette(0);
        const white = bgPalette[0]; // needed for CGB in DMG-mode with custom palette
        for (let x = 0; x < SCREEN_WIDTH; x++) {
            this.videoBuffer[y * SCREEN_WIDTH + x] = white;
        }
    }

```

```
    /**
     * Handles drawing a single line of the screen, for the background and window.
     * @param startX The X position to start drawing at.
     * @param y The Y position to draw at.
     * @param locationFlag The flag to use to determine the tilemap location.
     * @param scrollOffsetX The X offset to use for scrolling.
     * @param getX The function to use to get the X position of the tile.
     */
    drawLayer(
        startX: number,
        y: number,
        locationFlag: number,
        scrollOffsetX: number,
        getX: (x: number) => number
    ) {
        // Global BG priority bit (CGB only)
        const bgPrioCgb = this.lcdControl.flag(LCDC_BG_WIN_PRIO);
        // The tilemap used (a map of tile *pointers*)
        const tileMapLoc = this.lcdControl.flag(locationFlag) ? 0x9c00 : 0x9800;
        // The currently read Y position of the corresponding tile (one tile is 8 pixels long)
        const tileY = Math.floor(y / 8);
        // The currently read Y position *inside* the tile
        const tileInnerY = y % 8;
        // Start of video buffer for this line
        const bufferStart = this.lcdY.get() * SCREEN_WIDTH;

        for (let i = startX; i < SCREEN_WIDTH + scrollOffsetX; i += 8) {
            // The currently read X pixel of the tile map
            const x = getX(i);
            // The currently read X position of the corresponding tile
            // this determines the tile of the next 8 pixels
            const tileX = Math.floor(x / 8);
            // Index of the tile in the current tile data
            const tileIndex = tileMapLoc + tileX + tileY * 32;

            // On CGB, the attributes of the tile
            // Note we can do this even in DMG mode, because VRAM2 in DMG is filled with 0x00,
            // and all the 0 attributes match the normal behaviour of the DMG
            const tileAttributes = this.vramControl.readBank1(tileIndex);
            const bgToOamPrio = (tileAttributes & VRAM2_ATTR_BG_OAM_PRIORITY) !== 0;
            const flipX = (tileAttributes & VRAM2_ATTR_H_FLIP) !== 0;
            const flipY = (tileAttributes & VRAM2_ATTR_V_FLIP) !== 0;
            const vramBank = (tileAttributes & VRAM2_ATTR_VRAM_BANK) !== 0 ? 1 : 0;
            const tilePalette = (tileAttributes & VRAM2_ATTR_PALETTE) as Int3;

            // Map of colors for each shade
            const palette = this.colorControl.getBgPalette(tilePalette);
            // The ID (pointer) of the tile
            const tileAddress = this.vramControl.readBank0(tileIndex);
            // Convert the ID to the actual address
            const tileDataAddress = this.getTileAddress(tileAddress);
            // Get the tile data
            const tileData = this.vramControl.getTile(tileDataAddress, vramBank);

            for (let innerX = 0; innerX < 8; innerX++) {
                const posX = i + innerX - scrollOffsetX;
                if (posX < 0) continue;

                const arrayX = flipX ? 7 - innerX : innerX;
                const arrayY = flipY ? 7 - tileInnerY : tileInnerY;

                // Get the RGBA color, and draw it!
                const colorId = tileData[arrayX][arrayY];
                this.videoBuffer[bufferStart + posX] = palette[colorId];

                // Update priorities
                this.bgPriorities[posX] = 0;
                if (colorId > 0) {
                    if (!this.isCgbMode) {
```

```
                    this.bgPriorities[posX] = 1;
                } else {
                    if (bgPrioCgb) this.bgPriorities[posX] += 2;
                    if (bgToOamPrio) this.bgPriorities[posX] += 1;
                }
            }
        }
    }
}

drawBackground() {
    // The top-left corner of the 160x144 view area
    const viewX = this.screenX.get();
    const viewY = this.screenY.get();
    // Current Y position in the map
    const y = wrap8(viewY + this.lcdY.get());
    // The offset of the current line in the map
    const scrollOffsetX = viewX % 8;

    this.drawLayer(
        0, // start x
        y, // y
        LCDC_BG_TILE_MAP_AREA, // location flag
        scrollOffsetX, // scroll offset x
        (x) => wrap8(x + viewX) // get x
    );
}

drawWindow() {
    // The top-left corner of the 160x144 view area
    const windowX = this.windowX.get() - 7;
    const windowY = this.windowY.get();

    // If the window is not visible, return
    if (this.lcdY.get() < windowY || windowX >= SCREEN_WIDTH) return;

    // The currently read Y pixel of the window map
    const y = this.windowLineCounter++;

    this.drawLayer(
        windowX, // start x
        y, // y
        LCDC_WIN_TILE_MAP_AREA, // location flag
        0, // scroll offset x
        (x) => wrap8(x - windowX) // get x
    );
}

drawObjects() {
    const y = this.lcdY.get();
    const doubleObjects = this.lcdControl.flag(LCDC_OBJ_SIZE);
    const sprites = this.readSprites;

    for (const sprite of sprites.reverse()) {
        // Get tile id (to get the actual data pointer)
        let tileId = sprite.tileIndex;
        if (doubleObjects) {
            // We ignore bit 0 for 8x16 objects
            tileId &= ~0b1;
            // if below tile and not flipped, or upper tile but flipped
            if (y - sprite.y >= 8 !== sprite.yFlip) tileId += 1;
        }
        // We need to check if we have double height sprites and this is the lower half of
        // the sprite, in which case the actual tile address is the next byte
        const tileAddress = 0x8000 + tileId * 16;
        // The currently read Y position inside the corresponding tile
        let tileY = (y - sprite.y) % 8;
        tileY = sprite.yFlip ? 7 - tileY : tileY;
        // Get the palette for the object
```

```
            const palette = this.colorControl.getObjPalette(sprite);

            // Get tile colors
            const tileData = this.vramControl.getTile(tileAddress, sprite.cgbVramBank);

            for (let innerX = 0; innerX < 8; innerX++) {
                const x = innerX + sprite.x;
                // The X value of the sprite is offset by 8 to the left, so we skip off-screen
                if (x < 0 || x >= SCREEN_WIDTH) continue;
                const tileX = sprite.xFlip ? 7 - innerX : innerX;
                const colorId = tileData[tileX][tileY];

                // if transparent, skip
                // also skip if bg/win should be above, and priority is set
                if (colorId === 0) continue;
                if (this.isCgbMode) {
                    if (this.bgPriorities[x] + (sprite.bgAndWinOverObj ? 1 : 0) > 2) continue;
                } else {
                    if (this.bgPriorities[x] && sprite.bgAndWinOverObj) continue;
                }

                this.videoBuffer[y * SCREEN_WIDTH + x] = palette[colorId];
            }
        }
    }

    protected address(pos: number): Addressable {
        // VRAM
        if (0x8000 <= pos && pos <= 0x9fff) return this.vramControl;
        // OAM
        if (0xfe00 <= pos && pos <= 0xfe9f) return this.oam;
        // Registers
        const register = this.registerAddresses[pos];
        if (register) return register;

        throw new Error(`Invalid address given to PPU: ${pos.toString(16)}`);
    }

    read(address: number): number {
        const component = this.address(address);
        if (component === this.oam && !this.canReadOam) return 0xff;
        if (0x8000 <= address && address <= 0x9fff && !this.canReadVram) return 0xff;
        return component.read(address);
    }

    write(address: number, data: number): void {
        const component = this.address(address);

        if (component === this.oam && !this.canWriteOam) return;
        if (0x8000 <= address && address <= 0x9fff && !this.canWriteVram) return;

        if (component === this.lcdControl) {
            const isEnabled = this.lcdControl.flag(LCDC_LCD_ENABLE);
            const willEnable = (data & LCDC_LCD_ENABLE) === LCDC_LCD_ENABLE;

            // Will disable LCD
            if (isEnabled && !willEnable) {
                // console.warn("disabled LCD");
                this.lcdY.set(0);
                this.setMode(MODE_HBLANK_FIRST);

                this.canWriteOam = true;
                this.canReadOam = true;
                this.canWriteVram = true;
                this.canReadVram = true;
            }
            // Will enable LCD
            else if (!isEnabled && willEnable) {
                // console.warn("enabled LCD");
```

```
                this.lcdY.set(0);
                this.mode = MODE_HBLANK_FIRST;
                this.setMode(MODE_HBLANK_FIRST);
                this.cycleCounter = 0;

                this.nextInterruptLineUpdate = {
                    lycLyMatch: this.lcdY.get() === this.lcdYCompare.get(),
                    vblankActive: false,
                    hblankActive: false,
                    oamActive: false,
                };
            }
        }
        if (component === this.lcdStatus) {
            this.nextInterruptLineUpdate = {};
            // 3 first bits are read-only
            data = (data & 0b1111_1000) | (this.lcdStatus.get() & 0b0000_0111);
        }

        component.write(address, data);

        // Writing to LYC updates interrupt line if screen is on only
        if (component === this.lcdYCompare && this.lcdControl.flag(LCDC_LCD_ENABLE)) {
            this.nextInterruptLineUpdate = {
                lycLyMatch: this.lcdYCompare.get() === this.lcdY.get(),
            };
        }
    }
}

/**
 * To have nice TypeScript types we need to be able to refer to the protected attributes of the
 * PPU class. However this isn't possible. As such we instead create an exported class that
 * has all the public attributes of PPU, and choose to not export the PPU that has all its
 * fields as public.
 * This means other classes can keep using PPU as it was and don't have access to anything else,
 * but inside of the file everything is public and usable.
 */
class PPUExported implements Addressable {
    protected ppu: PPU;

    constructor(mode: ConsoleType) {
        this.ppu = new PPU(mode);
    }

    tick(system: Addressable, interrupts: Interrupts, isMCycle: boolean): boolean {
        return this.ppu.tick(system, interrupts, isMCycle);
    }

    setCGBMode(mode: CGBMode): void {
        this.ppu.setCGBMode(mode);
    }

    pushOutput(output: GameBoyOutput): void {
        this.ppu.pushOutput(output);
    }

    read(pos: number): number {
        return this.ppu.read(pos);
    }

    write(pos: number, data: number): void {
        return this.ppu.write(pos, data);
    }
}

export default PPUExported;
```

OAM (src/emulator/ppu/OAM.ts)

```ts
import { Addressable, RAM } from "../Memory";
import { Register } from "../Register";
import { Int1, Int3 } from "../util";

export type Sprite = {
    x: number;
    y: number;
    tileIndex: number;
    // From attributes:
    xFlip: boolean;
    yFlip: boolean;
    bgAndWinOverObj: boolean;
    // DMG only
    dmgPaletteNumber: Int1;
    // CGB only
    cgbPaletteNumber: Int3;
    cgbVramBank: Int1;
};

const ATTRIB_DMG_PALETTE_NUM_IDX = 4;
const ATTRIB_CGB_PALETTE_NUM = 0b111;
const ATTRIB_CGB_VRAM_BANK_IDX = 3;
const ATTRIB_X_FLIP = 1 << 5;
const ATTRIB_Y_FLIP = 1 << 6;
const ATTRIB_BG_AND_WIN_OVER_OBJ = 1 << 7;

const NOT_TRANSFERRING = -3;
const SHOULD_TRANSFER = -2;
const TRANSFER_END = 160;

/**
 * The OAM (Object Attribute Memory) used to store sprite data. It is the same as RAM, but has
 * an extra method to more easily retrieve sprite data, and to do OAM DMA transfers.
 * @link https://gbdev.io/pandocs/OAM.html
 */
class OAM implements Addressable {
    /**
     * There is a 2-cycle delay before any transfer.
     * -3 = not transferring
     * -2 = transfer starts in 2 cycles
     * -1 = transfer starts in 1 cycle
     * 0-159 = next byte to transfer
     */
    protected transferStep: number = NOT_TRANSFERRING;
    protected transferStart = new Register(0xff);
    protected data = new RAM(160);

    /** @link https://gbdev.io/pandocs/OAM_DMA_Transfer.html */
    tick(system: Addressable) {
        // If we're transferring...
        if (this.transferStep >= 0) {
            const baseAddress = this.transferStart.get() << 8;

            // Copy a byte
            const transferredByte = system.read(baseAddress + this.transferStep);
            this.data.write(this.transferStep, transferredByte);
            this.spriteCache[this.transferStep >> 2].valid = false;
        }

        // Tick the transfer and the start delay
        if (this.transferStep !== NOT_TRANSFERRING) {
            this.transferStep++;
            // Transfer ended
            if (this.transferStep === TRANSFER_END) {
                this.transferStep = NOT_TRANSFERRING;
            }
        }
```

```
    }

    read(pos: number): number {
        if (pos === 0xff46) return this.transferStart.get();
        if (0xfe00 <= pos && pos <= 0xfe9f) {
            if (this.transferStep >= 0) return 0xff; // read disabled
            return this.data.read(pos - 0xfe00);
        }
        throw new Error(`Invalid address for OAM: 0x${pos.toString(16)}`);
    }

    write(pos: number, data: number): void {
        if (pos === 0xff46) {
            this.transferStart.set(data);
            if (this.transferStep === NOT_TRANSFERRING || this.transferStep > 0)
                this.transferStep = SHOULD_TRANSFER;
        } else if (0xfe00 <= pos && pos <= 0xfe9f) {
            if (this.transferStep !== NOT_TRANSFERRING) return;
            const address = pos - 0xfe00;
            this.data.write(address, data);
            this.spriteCache[address >> 2].valid = false;
        }
    }

    protected spriteCache: (Sprite & { valid: boolean })[] = [...new Array(40)].map(() => ({
        y: 0,
        x: 0,
        tileIndex: 0,
        xFlip: false,
        yFlip: false,
        bgAndWinOverObj: false,
        dmgPaletteNumber: 0,
        cgbPaletteNumber: 0,
        cgbVramBank: 0,
        valid: false,
    }));

    getSprites(): Readonly<Sprite>[] {
        this.spriteCache
            .filter((x) => !x.valid)
            .forEach((sprite, index) => {
                const address = index << 2;
                const attribs = this.data.read(address + 3);
                sprite.y = this.data.read(address + 0) - 16;
                sprite.x = this.data.read(address + 1) - 8;
                sprite.tileIndex = this.data.read(address + 2);
                sprite.xFlip = (attribs & ATTRIB_X_FLIP) !== 0;
                sprite.yFlip = (attribs & ATTRIB_Y_FLIP) !== 0;
                sprite.bgAndWinOverObj = (attribs & ATTRIB_BG_AND_WIN_OVER_OBJ) !== 0;
                sprite.dmgPaletteNumber = ((attribs >> ATTRIB_DMG_PALETTE_NUM_IDX) & 1) as Int1;
                sprite.cgbPaletteNumber = (attribs & ATTRIB_CGB_PALETTE_NUM) as Int3;
                sprite.cgbVramBank = ((attribs >> ATTRIB_CGB_VRAM_BANK_IDX) & 1) as Int1;
                sprite.valid = true;
            });
        return this.spriteCache;
    }
}

export default OAM;
```

ColorController (src/emulator/ppu/ColorController.ts)

```ts
import { CGBMode } from "../constants";
import { Addressable, RAM } from "../Memory";
import { Sprite } from "./OAM";
import { Register, MaskRegister } from "../Register";
import { Int3 } from "../util";

export type ColorPalette = [number, number, number, number];

// Palette flags
const PALETTE_AUTO_INCREMENT = 1 << 7;
const PALETTE_INDEX = 0b0011_1111;

abstract class ColorController implements Addressable {
    protected abstract readonly addresses: Record<number, Addressable>;
    abstract getBgPalette(id: Int3): ColorPalette;
    abstract getObjPalette(sprite: Sprite): ColorPalette;

    changeCBGMode(mode: CGBMode): void {}

    read(pos: number): number {
        const component = this.addresses[pos];
        if (!component) return 0xff;
        return component.read(pos);
    }

    write(pos: number, value: number): void {
        const component = this.addresses[pos];
        if (!component) return;
        component.write(pos, value);
    }
}

class DMGColorControl extends ColorController {
    static readonly colorOptions = [
        0xffffffff, // white
        0xffaaaaaa, // light gray
        0xff555555, // dark gray
        0xff000000, // black
    ];

    // Background palette
    protected bgPalette = new Register(0x00);
    // Object palettes
    protected objPalette0 = new Register(0x00);
    protected objPalette1 = new Register(0x00);

    protected addresses = {
        0xff47: this.bgPalette,
        0xff48: this.objPalette0,
        0xff49: this.objPalette1,
    };

    getBgPalette(): ColorPalette {
        const palette = this.bgPalette.get();
        return [
            DMGColorControl.colorOptions[(palette >> 0) & 0b11],
            DMGColorControl.colorOptions[(palette >> 2) & 0b11],
            DMGColorControl.colorOptions[(palette >> 4) & 0b11],
            DMGColorControl.colorOptions[(palette >> 6) & 0b11],
        ];
    }

    getObjPalette(sprite: Sprite): ColorPalette {
        const palette =
            sprite.dmgPaletteNumber === 0 ? this.objPalette0.get() : this.objPalette1.get();
        return [
            0x00000000, // unused, color 0b00 is transparent
```

```
            DMGColorControl.colorOptions[(palette >> 2) & 0b11],
            DMGColorControl.colorOptions[(palette >> 4) & 0b11],
            DMGColorControl.colorOptions[(palette >> 6) & 0b11],
        ];
    }
}

class CGBColorControl extends ColorController {
    // Background palette
    protected bgPaletteOptions = new MaskRegister(0b0100_0000);
    protected bgPaletteData = new RAM(64);
    // Object palettes
    protected objPaletteOptions = new MaskRegister(0b0100_0000);
    protected objPaletteData = new RAM(64);
    // Palette cache
    protected paletteCache: { data: ColorPalette; valid: boolean }[] = [...Array(16)].map(
        () => ({
            data: [0, 0, 0, 0],
            valid: false,
        })
    );
    // Compatibility mode
    protected dmgBgPalette = new Register(0x00);
    protected dmgObj0Palette = new Register(0x00);
    protected dmgObj1Palette = new Register(0x00);
    protected dmgMode = false;

    protected addresses = {
        0xff68: this.bgPaletteOptions,
        0xff69: this.bgPaletteData,
        0xff6a: this.objPaletteOptions,
        0xff6b: this.objPaletteData,

        0xff47: this.dmgBgPalette, // Unused
        0xff48: this.dmgObj0Palette, // Unused
        0xff49: this.dmgObj1Palette, // Unused
    };

    changeCBGMode(mode: CGBMode): void {
        this.dmgMode = mode !== CGBMode.CGB;
        this.paletteCache.forEach((x) => (x.valid = false)); // force-reload color 0 for objects
    }

    override read(pos: number): number {
        if (pos === 0xff69)
            return this.bgPaletteData.read(this.bgPaletteOptions.get() & PALETTE_INDEX);

        if (pos === 0xff6b)
            return this.objPaletteData.read(this.objPaletteOptions.get() & PALETTE_INDEX);

        return super.read(pos);
    }

    override write(pos: number, value: number): void {
        if (pos === 0xff69) {
            const bgPaletteOptions = this.bgPaletteOptions.get();
            const index = bgPaletteOptions & PALETTE_INDEX;
            this.bgPaletteData.write(index, value);
            this.paletteCache[index >> 3].valid = false;
            if (bgPaletteOptions & PALETTE_AUTO_INCREMENT)
                this.bgPaletteOptions.set(
                    (bgPaletteOptions & ~PALETTE_INDEX) | ((index + 1) & PALETTE_INDEX)
                );
        } else if (pos === 0xff6b) {
            const objPaletteOptions = this.objPaletteOptions.get();
            const index = objPaletteOptions & PALETTE_INDEX;
            this.objPaletteData.write(index, value);
            this.paletteCache[(index >> 3) + 8].valid = false;
            if (objPaletteOptions & PALETTE_AUTO_INCREMENT)
```

```
                this.objPaletteOptions.set(
                    (objPaletteOptions & ~PALETTE_INDEX) | ((index + 1) & PALETTE_INDEX)
                );
        } else {
            super.write(pos, value);
        }
    }

    protected decodePalette(data: RAM, id: Int3, offset: 0 | 1, cacheOffset: 0 | 8) {
        const palette = this.paletteCache[id + cacheOffset];
        if (palette.valid) return palette.data;

        for (let colorIdx = offset; colorIdx < 4; colorIdx++) {
            const colorLow = data.read(id * 8 + colorIdx * 2);
            const colorHigh = data.read(id * 8 + colorIdx * 2 + 1);
            const fullColor = (colorHigh << 8) | colorLow;

            const red5 = (fullColor >> 0) & 0b0001_1111;
            const green5 = (fullColor >> 5) & 0b0001_1111;
            const blue5 = (fullColor >> 10) & 0b0001_1111;

            const red8 = (red5 << 3) | (red5 >> 2);
            const green8 = (green5 << 3) | (green5 >> 2);
            const blue8 = (blue5 << 3) | (blue5 >> 2);

            palette.data[colorIdx] = (0xff << 24) | (blue8 << 16) | (green8 << 8) | red8;
        }
        palette.valid = true;

        return palette.data;
    }

    protected applyDmgTransform(
        palette: ColorPalette,
        register: Register,
        isObj: boolean
    ): ColorPalette {
        const dmgPalette = register.get();
        return [
            isObj ? 0 : palette[(dmgPalette >> 0) & 0b11],
            palette[(dmgPalette >> 2) & 0b11],
            palette[(dmgPalette >> 4) & 0b11],
            palette[(dmgPalette >> 6) & 0b11],
        ];
    }

    getBgPalette(id: Int3): ColorPalette {
        if (!this.dmgMode) return this.decodePalette(this.bgPaletteData, id, 0, 0);

        const palette = this.decodePalette(this.bgPaletteData, 0, 0, 0);
        const dmgPalette = this.applyDmgTransform(palette, this.dmgBgPalette, false);
        return dmgPalette;
    }

    getObjPalette(sprite: Sprite): ColorPalette {
        if (!this.dmgMode)
            return this.decodePalette(this.objPaletteData, sprite.cgbPaletteNumber, 1, 8);

        const palette = this.decodePalette(this.objPaletteData, sprite.dmgPaletteNumber, 0, 8);
        const dmgPalette = this.applyDmgTransform(
            palette,
            sprite.dmgPaletteNumber ? this.dmgObj1Palette : this.dmgObj0Palette,
            true
        );
        return dmgPalette;
    }
}

export { ColorController, DMGColorControl, CGBColorControl };
```

VRAMController (src/emulator/ppu/VRAMController.ts)

```ts
import { Addressable, CircularRAM } from "../Memory";
import { MaskRegister, Register } from "../Register";
import { combine, high, Int2, low } from "../util";

type TileCache = Record<number, { valid: boolean; data: Int2[][] }>;

const HDMA5_LENGTH = 0b0111_1111;
const HDMA5_MODE = 0b1000_0000;

abstract class VRAMController implements Addressable {
    protected abstract readonly addresses: Record<number, Addressable>;
    protected abstract get currentBank(): Addressable;
    protected abstract get currentCache(): TileCache;

    protected static makeCache(): TileCache {
        return [...new Array(0x180)].map(() => ({
            valid: false,
            data: Array.from(Array(8), () => new Array(8)),
        }));
    }

    protected _getTile(tileAddress: number, bank: Addressable, cache: TileCache): Int2[][] {
        const cachedTile = cache[(tileAddress >> 4) & 0x1ff];
        if (!cachedTile.valid) {
            // Draw the 8 lines of the tile
            for (let tileY = 0; tileY < 8; tileY++) {
                const tileDataH = bank.read(tileAddress + tileY * 2);
                const tileDataL = bank.read(tileAddress + tileY * 2 + 1);
                for (let tileX = 0; tileX < 8; tileX++) {
                    const shadeL = (tileDataH >> (7 - tileX)) & 0b1;
                    const shadeH = (tileDataL >> (7 - tileX)) & 0b1;
                    const shade = ((shadeH << 1) | shadeL) as Int2;
                    cachedTile.data[tileX][tileY] = shade;
                }
            }
            cachedTile.valid = true;
        }
        return cachedTile.data;
    }

    abstract getTile(tileAddress: number, bankId: 0 | 1): Int2[][];

    abstract readBank0(pos: number): number;
    abstract readBank1(pos: number): number;

    /**
     * Ticks the VRAM. This does nothing on DMG, but ticks the VRAM-DMA on CGB.
     * @param system The system of the Gameboy. Used for the DMA.
     * @param isInHblank If the PPU is in a HBlank and LY=0-143.
     * @param isLcdOn If the LCD is on.
     * @returns If the CPU should be halted (because a DMA is in progress)
     */
    tick(system: Addressable, isInHblank: boolean, isLcdOn: boolean): boolean {
        return false;
    }

    read(pos: number): number {
        if (0x8000 <= pos && pos <= 0x9fff) return this.currentBank.read(pos);
        const component = this.addresses[pos];
        if (component) return component.read(pos);
        return 0xff;
    }

    write(address: number, value: number): void {
        if (0x8000 <= address && address <= 0x9fff) {
            if (
                // if in tile memory, dirty tile
```

```
                0x8000 <= address &&
                address < 0x9800 &&
                value !== this.currentBank.read(address)
            ) {
                this.currentCache[(address >> 4) & 0x1ff].valid = false;
            }
            return this.currentBank.write(address, value);
        }
        const component = this.addresses[address];
        if (component) return component.write(address, value);
    }
}

class DMGVRAMController extends VRAMController {
    protected vram = new CircularRAM(8192, 0x8000);
    protected tileCache = VRAMController.makeCache();
    protected currentBank = this.vram;
    protected currentCache = this.tileCache;
    protected readonly addresses: Record<number, Addressable> = {};

    readBank0(pos: number): number {
        return this.vram.read(pos);
    }
    readBank1(pos: number): number {
        return 0;
    }

    getTile(tileAddress: number): Int2[][] {
        return this._getTile(tileAddress, this.vram, this.tileCache);
    }
}

enum DMAState {
    NONE,
    HBLANK,
    GENERAL,
}

class CGBVRAMController extends VRAMController {
    protected vram0 = new CircularRAM(8192, 0x8000);
    protected vram1 = new CircularRAM(8192, 0x8000);
    protected tileCache0 = VRAMController.makeCache();
    protected tileCache1 = VRAMController.makeCache();
    protected vramBank = new MaskRegister(0b1111_1110);

    protected dmaInProgress: DMAState = DMAState.NONE;
    /** Number of bytes in the current block that were transferred [0-16] */
    protected dmaSubSteps: number = 0;
    protected isFirstDmaBlock: boolean = false;

    protected hdma1 = new Register();
    protected hdma2 = new Register();
    protected hdma3 = new Register();
    protected hdma4 = new Register();
    protected hdma5 = new Register();

    protected get currentBank() {
        return this.vramBank.get() & 0b1 ? this.vram1 : this.vram0;
    }
    protected get currentCache() {
        return this.vramBank.get() & 0b1 ? this.tileCache1 : this.tileCache0;
    }

    protected hdmaEnabled: boolean;
    protected readonly addresses: Record<number, Addressable>;

    constructor(hdmaEnabled: boolean) {
        super();

```

```
        this.hdmaEnabled = hdmaEnabled;
        if (this.hdmaEnabled) {
            this.addresses = {
                0xff4f: this.vramBank,
                0xff51: this.hdma1,
                0xff52: this.hdma2,
                0xff53: this.hdma3,
                0xff54: this.hdma4,
                0xff55: this.hdma5,
            };
        } else {
            this.addresses = {
                0xff4f: this.vramBank,
            };
        }
    }

    override tick(system: Addressable, isInHblank: boolean, isLcdOn: boolean): boolean {
        if (
            (this.dmaInProgress === DMAState.HBLANK && isInHblank) ||
            this.dmaInProgress === DMAState.GENERAL
        ) {
            const source = combine(this.hdma1.get(), this.hdma2.get()) & 0xfff0;
            const dest = combine(this.hdma3.get(), this.hdma4.get()) & 0x1ff0;

            const byte1 = system.read(source + this.dmaSubSteps);
            const byte2 = system.read(source + this.dmaSubSteps + 1);
            this.write(0x8000 + dest + this.dmaSubSteps, byte1);
            this.write(0x8000 + dest + this.dmaSubSteps + 1, byte2);

            this.dmaSubSteps += 2;

            if (this.dmaSubSteps === 16) {
                this.dmaSubSteps = 0;
                this.isFirstDmaBlock = false;

                const newSource = source + 16;
                const newDest = dest + 16;

                this.hdma1.set(high(newSource));
                this.hdma2.set(low(newSource));
                this.hdma3.set(high(newDest));
                this.hdma4.set(low(newDest));

                const length = this.hdma5.get() & HDMA5_LENGTH;
                if (length > 0) {
                    this.hdma5.set(length - 1);
                } else {
                    this.dmaInProgress = DMAState.NONE;
                    this.hdma5.set(0xff);
                }
            }

            return true;
        }
        return false;
    }

    override write(address: number, value: number): void {
        super.write(address, value);

        if (address === 0xff55) {
            // Interrupts the transfer
            if (this.dmaInProgress !== DMAState.NONE) {
                this.dmaInProgress = DMAState.NONE;
                this.hdma5.set(this.hdma5.get() | HDMA5_MODE);
            }
            // Starts the transfer
            else {
```

```
                this.dmaInProgress = value & HDMA5_MODE ? DMAState.HBLANK : DMAState.GENERAL;
                this.dmaSubSteps = 0;
                this.isFirstDmaBlock = true;
                this.hdma5.set(this.hdma5.get() & ~HDMA5_MODE);
            }
        }
    }

    readBank0(pos: number): number {
        return this.vram0.read(pos);
    }

    readBank1(pos: number): number {
        return this.vram1.read(pos);
    }

    getTile(tileAddress: number, bank: 0 | 1): Int2[][] {
        return this._getTile(
            tileAddress,
            bank ? this.vram1 : this.vram0,
            bank ? this.tileCache1 : this.tileCache0
        );
    }
}

export { VRAMController, DMGVRAMController, CGBVRAMController };
```

B.1.5 APU

APU (src/emulator/apu/APU.ts)

```ts
import { Addressable } from "../Memory";
import { CLOCK_SPEED, FRAME_RATE } from "../constants";
import GameBoyOutput from "../GameBoyOutput";
import { MaskRegister, Register } from "../Register";
import { Int4, rangeObject } from "../util";
import SoundChannel1 from "./SoundChannel1";
import SoundChannel2 from "./SoundChannel2";
import SoundChannel3 from "./SoundChannel3";
import SoundChannel4 from "./SoundChannel4";

const SAMPLE_RATE = 44100;

/**
 * Cycles for one full sample at 44.1Hz
 * - We divide clock speed by 4 to get M-cycles
 */
const CYCLES_PER_SAMPLE = CLOCK_SPEED / 4 / SAMPLE_RATE;
/** Number of values in a "frame-wide" sample  */
const SAMPLE_SIZE = Math.floor(SAMPLE_RATE / FRAME_RATE);

const NR52_APU_TOGGLE = 1 << 7;
const NR52_CHAN1_ON = 1 << 0;
const NR52_CHAN2_ON = 1 << 1;
const NR52_CHAN3_ON = 1 << 2;
const NR52_CHAN4_ON = 1 << 3;

const DIV_ADDRESS = 0xff04;
const DIV_TICK_BIT = 1 << 4;

/**
 * Converts a digital value in 0 - F into an analog value in -1 - 1 (negative slope)
 */
function DAC(n: Int4): number {
    return (-n / 0xf) * 2 + 1;
}

/**
 * The APU (Audio Processing Unit) of the Gameboy - it handles producing sound.
 */
export class APU implements Addressable {
    protected channel1 = new SoundChannel1((s) => this.nr52.sflag(NR52_CHAN1_ON, s));
    protected channel2 = new SoundChannel2((s) => this.nr52.sflag(NR52_CHAN2_ON, s));
    protected channel3 = new SoundChannel3((s) => this.nr52.sflag(NR52_CHAN3_ON, s));
    protected channel4 = new SoundChannel4((s) => this.nr52.sflag(NR52_CHAN4_ON, s));

    /** Master voulume and stereo mix control  */
    protected nr50 = new Register(0x77);
    /** Stereo mix control register */
    protected nr51 = new Register(0xf3);
    /** Status and control register */
    protected nr52 = new MaskRegister(0b0111_0000, 0xf1);

    /** PCM registers (CGB only) */
    protected pcm12 = new Register();
    protected pcm34 = new Register();

    /** Ticking control */
    protected oldDivBitState = false;

    /** Audio output */
    protected cyclesForSample: number = 0;
    protected sampleIndex: number = 0;
    protected audioBuffer = new Float32Array(SAMPLE_SIZE);
    protected output: GameBoyOutput;
```

```
65
66      constructor(output: GameBoyOutput) {
67          this.output = output;
68      }
69
70      /**
71       * Ticks the APU system.
72       */
73      tick(system: Addressable): void {
74          // Turned off
75          if (!this.nr52.flag(NR52_APU_TOGGLE)) return;
76
77          const divBitState = (system.read(DIV_ADDRESS) & DIV_TICK_BIT) === DIV_TICK_BIT;
78          const divChanged = !divBitState && this.oldDivBitState;
79          this.oldDivBitState = divBitState;
80
81          const chan1Out = this.channel1.tick(divChanged);
82          const chan2Out = this.channel2.tick(divChanged);
83          const chan3Out = this.channel3.tick(divChanged);
84          const chan4Out = this.channel4.tick(divChanged);
85
86          this.pcm12.set(chan1Out | (chan2Out << 4));
87          this.pcm34.set(chan3Out | (chan4Out << 4));
88
89          if (++this.cyclesForSample >= CYCLES_PER_SAMPLE) {
90              this.cyclesForSample -= CYCLES_PER_SAMPLE;
91
92              // Get all variables for processing audio
93              const nr51 = this.nr51.get();
94              const nr52 = this.nr52.get();
95
96              // Get output from each channel
97              const out1 = DAC(chan1Out);
98              const out2 = DAC(chan2Out);
99              const out3 = DAC(chan3Out);
100             const out4 = DAC(chan4Out);
101
102             // Mix right stereo side, enabling relevant channels
103             const rightAudio =
104                 out1 * ((nr51 >> 0) & 1) +
105                 out2 * ((nr51 >> 1) & 1) +
106                 out3 * ((nr51 >> 2) & 1) +
107                 out4 * ((nr51 >> 3) & 1);
108
109             // Mix left stereo side, enabling relevant channels
110             const leftAudio =
111                 out1 * ((nr51 >> 4) & 1) +
112                 out2 * ((nr51 >> 5) & 1) +
113                 out3 * ((nr51 >> 6) & 1) +
114                 out4 * ((nr51 >> 7) & 1);
115
116             // Get volume for each side in range 1 to 8
117             const rightVolume = ((nr52 >> 0) & 0b111) + 1;
118             const leftVolume = ((nr52 >> 4) & 0b111) + 1;
119
120             // Mix both sides together, by averaging (taking into account each volume)
121             const monoAudio = (rightAudio * rightVolume + leftAudio * leftVolume) / 16;
122
123             // Do some balancing so the level is correct
124             this.audioBuffer[this.sampleIndex] = monoAudio / 16;
125
126             if (++this.sampleIndex === SAMPLE_SIZE) {
127                 this.sampleIndex = 0;
128                 if (this.output.receiveSound) {
129                     this.output.receiveSound(this.audioBuffer);
130                 }
131             }
132         }
133     }
```

```
134
135     protected addresses: Record<number, Addressable> = {
136         ...rangeObject(0xff10, 0xff14, this.channel1),
137         ...rangeObject(0xff15, 0xff19, this.channel2),
138         ...rangeObject(0xff1a, 0xff1e, this.channel3),
139         ...rangeObject(0xff1f, 0xff23, this.channel4),
140         0xff24: this.nr50,
141         0xff25: this.nr51,
142         0xff26: this.nr52,
143         ...rangeObject(0xff30, 0xff3f, this.channel3), // wave RAM
144         0xff76: this.pcm12,
145         0xff77: this.pcm34,
146     };
147
148     read(pos: number): number {
149         return this.addresses[pos].read(pos);
150     }
151
152     write(pos: number, data: number): void {
153         const component = this.addresses[pos];
154
155         if (component === this.pcm12 || component === this.pcm34) return; // read-only
156
157         // ignore writes to channel when turned off (except for NRX1 and wave RAM)
158         if (
159             !this.nr52.flag(NR52_APU_TOGGLE) &&
160             component !== this.nr52 &&
161             !(0xff30 <= pos && pos <= 0xff3f)
162         )
163             return;
164
165         if (component === this.nr52) {
166             data = data & 0xf0; // lower 4 bits (status of channels) are read-only
167             const wasOn = this.nr52.flag(NR52_APU_TOGGLE);
168             const isOn = data & NR52_APU_TOGGLE;
169
170             if (wasOn && !isOn) {
171                 // when turning off, write 0x00 to all registers
172                 for (let address = 0xff10; address <= 0xff25; address++) {
173                     // Except for NR41 for some reason
174                     if (address === 0xff20) continue;
175                     this.write(address, 0x00);
176                 }
177             }
178         }
179
180         component.write(pos, data);
181     }
182 }
183
184 export default APU;
```

SoundChannel (src/emulator/apu/SoundChannel.ts)

```ts
import { Addressable } from "../Memory";
import { Register } from "../Register";
import { Int4 } from "../util";

const FREQUENCY_SWEEP_PACE = 4;
const FREQUENCY_ENVELOPE = 8;
const FREQUENCY_LENGTH_TIMER = 2;

const NRX4_RESTART_CHANNEL = 1 << 7;
const NRX4_LENGTH_TIMER_FLAG = 1 << 6;

abstract class SoundChannel implements Addressable {
    // Channel-dependent
    protected abstract readonly NRX1_LENGTH_TIMER_BITS: number;

    // Common registers
    protected abstract nrX1: Register;
    protected abstract nrX2: Register;
    protected abstract nrX3: Register;
    protected abstract nrX4: Register;

    // State
    protected enabled = false;
    protected onStateChange: (state: boolean) => void;

    // Counters
    protected step: number = 0;

    constructor(onStateChange: (state: boolean) => void) {
        this.onStateChange = onStateChange;
    }

    /**
     * Ticks the whole channel. This method should not be overrided by subclasses - for ticking
     * behavior, override doTick instead.
     * @returns The output of this channel
     */
    tick(divChanged: boolean): Int4 {
        // Ticks even when disabled

        if (divChanged) {
            this.step = (this.step + 1) % 8;

            if (this.step % 2 === 0) {
                this.tickLengthTimer();
            }
            if (this.step % 4 === 2) {
                this.tickSweep();
            }
            if (this.step === 7) {
                this.tickEnvelope();
            }
        }

        if (!this.enabled) return 0;

        this.doTick();

        return this.getSample();
    }

    /** Ticks the length timer. */
    protected tickLengthTimer(): void {
        // Tick length timer
        if (this.nrX4.flag(NRX4_LENGTH_TIMER_FLAG)) {
            const timerBits = this.NRX1_LENGTH_TIMER_BITS;
            const nrx1 = this.nrX1.get();
```

```
            const lengthTimer = ((nrx1 & timerBits) + 1) & timerBits;
            this.nrX1.set((nrx1 & ~timerBits) | lengthTimer);
            // overflowed
            if (lengthTimer === 0) {
                this.stop();
            }
        }
    }

    protected tickSweep(): void {}

    protected tickEnvelope(): void {}

    /**
     * Ticks the channel. This method should be overrided by subclasses for channel-specific
     * behavior.
     */
    protected abstract doTick(): void;

    /**
     * @returns The current value of the channel (value between 0-F).
     */
    protected abstract getSample(): Int4;

    /**
     * @returns The channel's wavelength, using the NRX3 and NRX4 registers. Only relevant for
     * channels 1, 2 and 3.
     */
    protected getWavelength(): number {
        const lower8 = this.nrX3.get();
        const higher3 = this.nrX4.get() & 0b111;
        return (higher3 << 8) | lower8;
    }

    /**
     * @param waveLength The new wavelength, using the NRX3 and NRX4 registers. Only relevant
     * for channels 1, 2 and 3.
     */
    protected setWavelength(waveLength: number): void {
        waveLength &= (1 << 11) - 1; // ensure it fits in 11bits
        const lower8 = waveLength & 0xff;
        const higher3 = (waveLength >> 8) & 0b111;
        this.nrX3.set(lower8);
        this.nrX4.set((this.nrX4.get() & ~0b111) | higher3);
    }

    /**
     * Starts the channel, if it isn't started already.
     */
    protected start(): void {
        if (this.enabled) return;
        if (!this.isDACOn) return;

        this.enabled = true;
        this.onStateChange(true);
        this.onStart();
    }

    /**
     * Stops the channel, if it isn't stopped already.
     */
    protected stop(): void {
        if (!this.enabled) return;

        this.enabled = false;
        this.onStateChange(false);
    }

    protected onStart(): void {}
```

```
137
138     protected abstract get isDACOn(): boolean;
139
140     abstract read(pos: number): number;
141     abstract write(pos: number, data: number): void;
142 }
143
144 export default SoundChannel;
145 export { NRX4_RESTART_CHANNEL, FREQUENCY_SWEEP_PACE, FREQUENCY_ENVELOPE };
```

SoundChannel1 (src/emulator/apu/SoundChannel1.ts)

```ts
import { Addressable } from "../Memory";
import { MaskRegister, Register } from "../Register";
import { Int4 } from "../util";
import SoundChannel2 from "./SoundChannel2";

const NRX0_SWEEP_CHANGE = 1 << 3;
const NRX0_MULTIPlIER = 0b0000_0111;

/**
 * Sound channel 1 generates a pulse signal, with a wavelength sweep.
 * @link
 *  https://gbdev.io/pandocs/Audio_Registers.html#sound-channel-1--pulse-with-wavelength-sweep
 */
class SoundChannel1 extends SoundChannel2 {
    protected nrX0 = new MaskRegister(0b1000_0000, 0x80);
    protected nrX1 = new Register(0xbf);
    protected nrX2 = new Register(0xf3);
    protected nrX3 = new Register(0xff);
    protected nrX4 = new Register(0xbf);

    // Needed because going from negate to positive mode turns off the channel
    protected inNegateMode: boolean = false;
    // Wavelength sweep pace countdown
    protected waveSweepCounter: number = 0;
    // Whether the sweep is enabled at all
    protected sweepEnabled: boolean = false;

    protected override addresses: Record<number, Addressable> = {
        0xff10: this.nrX0,
        0xff11: this.nrX1,
        0xff12: this.nrX2,
        0xff13: this.nrX3,
        0xff14: this.nrX4,
    };

    protected override tickSweep(): void {
        if (this.waveSweepCounter > 1) this.waveSweepCounter--;
        else {
            const nextCounter = (this.nrX0.get() >> 4) & 0b111; // bits 4-6
            if (nextCounter === 0) {
                this.waveSweepCounter = 8; // 0 is treated as 8
            } else {
                this.waveSweepCounter = nextCounter;
                if (this.sweepEnabled) this.applyWavelengthSweep();
            }
        }
    }

    protected applyWavelengthSweep(): void {
        const addOrSub = this.nrX0.flag(NRX0_SWEEP_CHANGE) ? -1 : 1;
        const multiplier = this.nrX0.get() & NRX0_MULTIPlIER; // bits 0-2

        const wave = this.waveLength;
        this.inNegateMode ||= addOrSub === -1;
        const newWave = wave + addOrSub * (wave >> multiplier);

        // On overflow, stop channel
        if (newWave > 0x7ff) this.stop();
        else {
            // Can't underflow, saturate at 0
            if (newWave < 0) this.setWavelength(0);
            // Normal case
            else this.setWavelength(newWave);
        }
    }

    write(pos: number, data: number): void {
```

```
        super.write(pos, data);

        if (pos === 0xff10) {
            const wasNegating = this.inNegateMode;
            const isNegating = this.nrX0.flag(NRX0_SWEEP_CHANGE);
            if (wasNegating && !isNegating) {
                this.stop();
            }
            this.inNegateMode = false;
        }
    }

    override onStart(): void {
        super.onStart();

        const nrX0 = this.nrX0.get();
        const nextCounter = (nrX0 >> 4) & 0b111; // bits 4-6
        this.waveSweepCounter = nextCounter === 0 ? 8 : nextCounter;
        this.sweepEnabled =
            ((nrX0 >> 4) & 0b111) !== 0 || // sweep period != 0
            (nrX0 & NRX0_MULTIPlIER) !== 0; // sweep shift != 0

        // On start, if the shift isn't 0, a sweep overflow check is made
        if ((nrX0 & NRX0_MULTIPlIER) > 0) {
            this.applyWavelengthSweep();
        }
    }
}

export default SoundChannel1;
```

SoundChannel2 (src/emulator/apu/SoundChannel2.ts)

```ts
import { Addressable } from "../Memory";
import { RegisterFF, Register } from "../Register";
import { clamp, Int2, Int4 } from "../util";
import SoundChannel, { FREQUENCY_ENVELOPE, NRX4_RESTART_CHANNEL } from "./SoundChannel";

const NRX2_STOP_DAC = 0b1111_1000;
const wavePatterns: (0 | 1)[][] = [
    [1, 1, 1, 1, 1, 1, 1, 0],
    [0, 1, 1, 1, 1, 1, 1, 0],
    [0, 1, 1, 1, 1, 0, 0, 0],
    [1, 0, 0, 0, 0, 0, 0, 1],
];

/**
 * Sound channel 2 is identical to channel 1, except that it doesn't have a wavelength sweep.
 * @link https://gbdev.io/pandocs/Audio_Registers.html#sound-channel-2--pulse
 */
class SoundChannel2 extends SoundChannel {
    protected NRX1_LENGTH_TIMER_BITS = 0b0011_1111;

    protected nrX1 = new Register(0x3f);
    protected nrX2 = new Register(0x00);
    protected nrX3 = new Register(0xff);
    protected nrX4 = new Register(0xbf);

    protected addresses: Record<number, Addressable> = {
        0xff15: RegisterFF,
        0xff16: this.nrX1,
        0xff17: this.nrX2,
        0xff18: this.nrX3,
        0xff19: this.nrX4,
    };

    // Stores a private copy of wave length on trigger
    protected waveLength: number = 0;

    // For output
    protected ticksPerWaveStep: number = 0;
    protected waveStep: number = 0;
    protected waveStepSubsteps: number = 0;

    // NRx2 needs retriggering when changed
    protected cachedNRX2: number = this.nrX2.get();

    // Channel envelope volume
    protected envelopeVolumeSteps: number = 0;
    protected envelopeVolume: Int4 = 0;

    protected override doTick(): void {
        if (this.waveStepSubsteps++ >= this.ticksPerWaveStep) {
            this.waveStepSubsteps = 0;
            this.waveStep = (this.waveStep + 1) % 8;
        }
    }

    protected tickEnvelope(): void {
        if (this.envelopeVolumeSteps > 0) this.envelopeVolumeSteps--;
        else {
            this.envelopeVolumeSteps = this.cachedNRX2 & 0b111;
            if ((this.cachedNRX2 & 0b111) !== 0) {
                const direction = (this.cachedNRX2 & 0b0000_1000) === 0 ? -1 : 1;
                this.envelopeVolume = clamp(this.envelopeVolume + direction, 0x0, 0xf) as Int4;
            }
        }
    }

    protected override getSample(): Int4 {
```

```
68          const dutyCycleType = ((this.nrX1.get() >> 6) & 0b11) as Int2;
69          const wavePattern = wavePatterns[dutyCycleType];
70          // if (this.constructor.name === "SoundChannel1")
71          // console.log(this.waveStep, wavePattern[this.waveStep], this.envelopeVolume);
72          return (wavePattern[this.waveStep] * this.envelopeVolume) as Int4;
73      }
74
75      protected override setWavelength(waveLength: number): void {
76          super.setWavelength(waveLength);
77          this.ticksPerWaveStep = 2048 - waveLength;
78          this.waveLength = waveLength;
79      }
80
81      /* Audio control */
82
83      get isDACOn(): boolean {
84          return (this.nrX2.get() & NRX2_STOP_DAC) !== 0;
85      }
86
87      override onStart(): void {
88          this.waveLength = this.getWavelength();
89
90          this.cachedNRX2 = this.nrX2.get();
91          this.envelopeVolume = (this.cachedNRX2 >> 4) as Int4;
92          this.ticksPerWaveStep = 2048 - this.getWavelength();
93      }
94
95      read(pos: number): number {
96          const component = this.addresses[pos];
97
98          // bits 0-5 are write only
99          if (component === this.nrX1) return component.read(pos) | 0b0011_1111;
100         // register is write only
101         if (component === this.nrX3) return 0xff;
102         // only bit 6 is readable
103         if (component === this.nrX4) return component.read(pos) | 0b1011_1111;
104
105         return component.read(pos);
106     }
107
108     write(pos: number, data: number): void {
109         const component = this.addresses[pos];
110
111         component.write(pos, data);
112
113         // Restart
114         if (component === this.nrX4 && (data & NRX4_RESTART_CHANNEL) !== 0) {
115             this.stop();
116             this.start();
117             data &= ~NRX4_RESTART_CHANNEL; // bit is write-only
118         }
119
120         // Clearing bits 3-7 of NRX2 turns the DAC (and the channel) off
121         if (component === this.nrX2 && (data & NRX2_STOP_DAC) === 0) {
122             this.stop();
123         }
124     }
125 }
126
127 export default SoundChannel2;
```

SoundChannel3 (src/emulator/apu/SoundChannel3.ts)

```ts
import { CircularRAM, Addressable } from "../Memory";
import { MaskRegister, Register } from "../Register";
import { Int2, Int4, rangeObject } from "../util";
import SoundChannel, { NRX4_RESTART_CHANNEL } from "./SoundChannel";

const NRX0_DAC_FLAG = 1 << 7;
const NRX2_OUTPUT_LEVEL = 0b0110_0000;

/** How much to shift the sound to the right */
const VOLUME_LEVELS = [
    4, // = muted
    0, // = full volume
    1, // = 50%
    2, // = 25%
];

/**
 * Sound channel 3 generates a wave that can be customised as needed.
 * @link https://gbdev.io/pandocs/Audio_Registers.html#sound-channel-3--wave-output
 */
class SoundChannel3 extends SoundChannel {
    protected NRX1_LENGTH_TIMER_BITS: number = 0b1111_1111;

    protected nrX0 = new MaskRegister(0b0111_1111);
    protected nrX1 = new Register(0xbf);
    protected nrX2 = new MaskRegister(0b1001_1111, 0xf3);
    protected nrX3 = new Register(0xff);
    protected nrX4 = new MaskRegister(0b0011_1000, 0xbf);
    protected waveData = new CircularRAM(16, 0xff30);

    protected addresses: Record<number, Addressable> = {
        0xff1a: this.nrX0,
        0xff1b: this.nrX1,
        0xff1c: this.nrX2,
        0xff1d: this.nrX3,
        0xff1e: this.nrX4,
        ...rangeObject(0xff30, 0xff3f, this.waveData),
    };

    // For output
    protected ticksNextSample: number = 0;
    protected waveStep: number = 0;
    protected currentSample: Int4 = 0;

    protected lastReadByte: number = 0xff;

    protected override doTick() {
        if (this.ticksNextSample-- <= 0) {
            const frequency = (2048 - this.getWavelength()) >> 1;
            this.ticksNextSample = frequency;

            this.waveStep = (this.waveStep + 1) % 32;

            const waveIndex = this.waveStep >> 1;
            const waveByte = this.waveData.read(waveIndex);
            const waveNibble = this.waveStep & 1 ? waveByte >> 4 : waveByte & 0b1111;
            this.currentSample = waveNibble as Int4;
            this.lastReadByte = waveByte;
        } else {
            this.lastReadByte = 0xff;
        }
    }

    protected override getSample(): Int4 {
        const outputLevel = ((this.nrX2.get() & NRX2_OUTPUT_LEVEL) >> 5) as Int2;
        const volume = VOLUME_LEVELS[outputLevel];
        return (this.currentSample >> volume) as Int4;
```

```
    }

    override onStart(): void {
        const frequency = (2048 - this.getWavelength()) >> 1;
        this.ticksNextSample = frequency;
    }

    get isDACOn(): boolean {
        return this.nrX0.flag(NRX0_DAC_FLAG);
    }

    read(pos: number): number {
        const component = this.addresses[pos];

        // registers are write only
        if (component === this.nrX1 || component === this.nrX3) return 0xff;
        // only bit 6 is readable
        if (component === this.nrX4) return this.nrX4.get() | 0b1011_1111;
        // wave data is offset by 0xff30
        if (component === this.waveData) {
            // if (this.enabled) return this.lastReadByte;
        }
        return component.read(pos);
    }

    write(pos: number, data: number): void {
        const component = this.addresses[pos];

        if (component === this.nrX0) {
            const oldDacState = this.nrX0.flag(NRX0_DAC_FLAG);
            const newDacState = (data & NRX0_DAC_FLAG) === NRX0_DAC_FLAG;
            if (oldDacState && !newDacState) {
                this.stop();
            } else if (!oldDacState && newDacState) {
                this.start();
            }
        }
        if (component === this.nrX4) {
            if (data & NRX4_RESTART_CHANNEL) {
                this.stop();
                this.start();
            }
        }
        component.write(pos, data);
    }
}

export default SoundChannel3;
```

SoundChannel4 (src/emulator/apu/SoundChannel4.ts)

```ts
import { Addressable } from "../Memory";
import { MaskRegister, RegisterFF, Register } from "../Register";
import { clamp, Int4 } from "../util";
import SoundChannel, { FREQUENCY_ENVELOPE, NRX4_RESTART_CHANNEL } from "./SoundChannel";

const NRX2_STOP_DAC = 0b1111_1000;
const NRX3_CLOCK_SHIFT_OFFSET = 4;
const NRX3_LFSR_SHORT_MODE = 0b0000_1000;
const NRX3_CLOCK_DIVIDER = 0b0000_0111;

/**
 * Sound channel 4 generates noise, that can be somewhat customised for softer/harsher noise.
 * @link https://gbdev.io/pandocs/Audio_Registers.html#sound-channel-4--noise
 */
class SoundChannel4 extends SoundChannel {
    protected NRX1_LENGTH_TIMER_BITS: number = 0b0011_1111;

    protected nrX1 = new Register(0xff);
    protected nrX2 = new Register(0x00);
    protected nrX3 = new Register(0x00);
    protected nrX4 = new MaskRegister(0b0011_1111, 0xbf);

    protected addresses: Record<number, Addressable> = {
        0xff1f: RegisterFF,
        0xff20: this.nrX1,
        0xff21: this.nrX2,
        0xff22: this.nrX3,
        0xff23: this.nrX4,
    };

    // NRx2 needs retriggering when changed
    protected cachedNRX2: number = this.nrX2.get();

    // Channel envelope volume
    protected envelopeVolumeSteps: number = 0;
    protected envelopeVolume: Int4 = 0;

    // LFSR
    protected ticksForLfsr: number = 0;
    protected lfsr: number = 0x00;

    protected override doTick() {
        if (--this.ticksForLfsr <= 0) {
            this.refreshLsfrTicks();
            const lfsrBit = ~(this.lfsr ^ (this.lfsr >> 1)) & 1;
            const shortMode = this.nrX3.flag(NRX3_LFSR_SHORT_MODE);
            this.lfsr = (this.lfsr >> 1) | (lfsrBit << 15) | (shortMode ? 0 : lfsrBit << 7);
        }
    }

    protected override tickEnvelope(): void {
        if (this.envelopeVolumeSteps-- <= 0 && (this.cachedNRX2 & 0b111) !== 0) {
            const direction = (this.cachedNRX2 & 0b0000_1000) === 0 ? -1 : 1;
            this.envelopeVolume = clamp(this.envelopeVolume + direction, 0x0, 0xf) as Int4;
            this.envelopeVolumeSteps = this.cachedNRX2 & 0b111;
        }
    }

    protected refreshLsfrTicks(): void {
        const clockDivider = this.nrX3.get() & NRX3_CLOCK_DIVIDER || 0.5; // 0 treated as 0.5
        const clockShift = this.nrX3.get() >> NRX3_CLOCK_SHIFT_OFFSET;
        this.ticksForLfsr = 4 * (clockDivider * (1 << clockShift));
    }

    protected override getSample(): Int4 {
        return ((this.lfsr & 1) * this.envelopeVolume) as Int4;
    }
```

```
68
69      get isDACOn(): boolean {
70          return (this.nrX2.get() & NRX2_STOP_DAC) !== 0;
71      }
72
73      override onStart(): void {
74          this.cachedNRX2 = this.nrX2.get();
75          this.envelopeVolume = (this.cachedNRX2 >> 4) as Int4;
76          this.lfsr = 0;
77          this.refreshLsfrTicks();
78      }
79
80      read(pos: number): number {
81          const component = this.addresses[pos];
82          // register is write only
83          if (component === this.nrX1) return 0xff;
84          // only bit 6 is readable
85          if (component === this.nrX4) return this.nrX4.get() | 0b1011_1111;
86          return component.read(pos);
87      }
88
89      write(pos: number, data: number): void {
90          const component = this.addresses[pos];
91
92          // Restart
93          if (component === this.nrX4 && (data & NRX4_RESTART_CHANNEL) !== 0) {
94              this.stop();
95              this.start();
96              data &= ~NRX4_RESTART_CHANNEL; // bit is write-only
97          }
98          // Turning off channel
99          if (component === this.nrX2 && (data & NRX2_STOP_DAC) === 0) {
100             this.stop();
101         }
102
103         component.write(pos, data);
104     }
105 }
106
107 export default SoundChannel4;
```

B.1.6 MBCs

MBC (src/emulator/mbc/MBC.ts)

```ts
import { Addressable, RAM, ROM } from "../Memory";

abstract class MBC implements Addressable {
    static readonly RAM_SIZES: Partial<Record<number, number>> = {
        0x00: 0,
        0x01: 0, // this is unofficial, only used by homebrew roms
        0x02: 1024 * 8,
        0x03: 1024 * 32,
        0x04: 1024 * 128,
        0x05: 1024 * 64,
    };

    protected size: number;
    protected rom: ROM;
    protected ram: RAM | null = null;
    protected hasSaves: boolean = false;

    constructor(data: Uint8Array, hasSaves: boolean) {
        this.size = data.length;
        this.rom = new ROM(this.size, data);
        this.hasSaves = hasSaves;
    }

    /**
     * Creates a new RAM instance for this MBC. This must be called during construction. It reads
     * the RAM size from the ROM header and creates a new RAM instance with that size.
     * @link https://gbdev.io/pandocs/The_Cartridge_Header.html#0149--ram-size
     */
    protected createRAM(): RAM {
        const ramSizeCode = this.rom.read(0x0149);
        const ramSize = MBC.RAM_SIZES[ramSizeCode];
        if (ramSize === undefined)
            throw new Error(`Invalid RAM size header value: ${ramSizeCode.toString(16)}`);
        return new RAM(ramSize);
    }

    /** Returns true if this ROM supports saves, false otherwise. */
    supportsSaves(): boolean {
        return this.hasSaves;
    }

    /** Returns this ROMs data if it supports saves, null otherwise. */
    save(): Uint8Array | null {
        return this.hasSaves && this.ram ? this.ram.rawData() : null;
    }

    /** Loads a save for this ROM. */
    load(data: Uint8Array): void {
        if (!this.hasSaves || !this.ram) return;

        if (data.length !== this.ram.size)
            throw new Error(
                `[MBC] Save data is not the same size as the RAM! ` +
                    `Got ${data.length} bytes, expected ${this.ram.size} bytes.`
            );

        this.ram.rawSet(data);
    }

    abstract read(pos: number): number;
    abstract write(pos: number, data: number): void;
}

export default MBC;
```

NoMBC (src/emulator/mbc/NoMBC.ts)

```ts
import MBC from "./MBC";

class NoMBC extends MBC {
    constructor(data: Uint8Array) {
        super(data, false);
    }

    read(pos: number): number {
        if (0xa000 <= pos && pos <= 0xbfff) return 0xff; // eram
        return this.rom.read(pos);
    }

    write(pos: number, data: number): void {
        // nothing is writable by default
    }
}

export default NoMBC;
```

MBC1 (src/emulator/mbc/MBC1.ts)

```ts
import { RAM } from "../Memory";
import { Register } from "../Register";
import MBC from "./MBC";

const RAM_ENABLED = 0x0a;

type MBC1Params = {
    hasRam: boolean;
    hasBattery: boolean;
};

/**
 * Implementation of MBC1.
 * @link https://gbdev.io/pandocs/MBC1.html
 */
class MBC1 extends MBC {
    /** @link https://gbdev.io/pandocs/MBC1.html#00001fff--ram-enable-write-only */
    protected ramEnable = new Register(0x00);
    /** @link https://gbdev.io/pandocs/MBC1.html#20003fff--rom-bank-number-write-only */
    protected romBank = new Register(0x01);
    /** @link
    ↪ https://gbdev.io/pandocs/MBC1.html#40005fff--ram-bank-number--or--upper-bits-of-rom-bank-number-write-only
    ↪ */
    protected ramBank = new Register(0x00);
    /** @link https://gbdev.io/pandocs/MBC1.html#60007fff--banking-mode-select-write-only */
    protected bankingModeSelect = new Register(0x00);
    /** The RAM contained in the ROM (ERAM). */
    protected ram: RAM;
    protected hasRam: boolean;

    constructor(data: Uint8Array, { hasRam, hasBattery }: MBC1Params) {
        super(data, hasBattery);
        this.ram = this.createRAM();
        this.hasRam = hasRam && this.ram.size > 0;
    }

    /**
     * Resolves a GameBoy address to an address in the ERAM. This uses the banking mode and
     * the current RAM bank to determine the address.
     */
    protected resolveERAMAddress(pos: number): number {
        const mode = this.bankingModeSelect.get() as 0 | 1;
        const pos12bits = pos & ((1 << 13) - 1);
        const ramAddressMask = this.ram.size - 1; // works for powers of 2
        const address = pos12bits | (mode === 0 ? 0 : this.ramBank.get() << 13);
        return address & ramAddressMask;
    }

    /**
     * Reads from the ROM, taking into account banking and the control ROMs.
     * @link https://gbdev.io/pandocs/MBC1.html#addressing-diagrams
     */
    read(pos: number): number {
        const mode = this.bankingModeSelect.get() as 0 | 1;
        const addressMask = this.size - 1; // works for powers of 2
        switch (pos >> 12) {
            case 0x0:
            case 0x1:
            case 0x2:
            case 0x3: {
                const address = mode === 0 ? pos : (this.ramBank.get() << 19) | pos;
                return this.rom.read(address & addressMask);
            }
            case 0x4:
            case 0x5:
            case 0x6:
            case 0x7: {
```

```
                const address =
                    (pos & ((1 << 14) - 1)) |
                    (this.romBank.get() << 14) |
                    (this.ramBank.get() << 19);
                return this.rom.read(address & addressMask);
            }
            case 0xa:
            case 0xb: {
                // RAM disabled
                if (!this.hasRam || this.ramEnable.get() !== RAM_ENABLED) return 0xff;
                const address = this.resolveERAMAddress(pos);
                return this.ram.read(address);
            }
        }
        throw new Error(`Invalid address to read from MBC1: ${pos.toString(16)}`);
    }

    write(pos: number, data: number): void {
        switch (pos >> 12) {
            case 0x0: // RAM enable
            case 0x1:
                return this.ramEnable.set(data & 0b1111); // 4 bit register

            case 0x2: // ROM Bank Number
            case 0x3:
                const bits5 = data & 0b11111; // 5bit register
                // Can't set ROM bank to 0
                // In reality what happens is that a value of 0 is *interpreted* as 1. However
                // simply overriding the write produces the same effect and is simpler.
                return this.romBank.set(bits5 === 0 ? 1 : bits5);

            case 0x4: // RAM Bank Number
            case 0x5:
                return this.ramBank.set(data & 0b11); // 2bit register

            case 0x6: // Banking Mode Select
            case 0x7:
                return this.bankingModeSelect.set(data & 0b1); // 1bit register

            case 0xa: // ERAM Write
            case 0xb:
                if (this.ramEnable.get() !== RAM_ENABLED) return; // RAM disabled

                const address = this.resolveERAMAddress(pos);
                return this.ram.write(address, data);
        }

        throw new Error(`Invalid address to write to MBC1: ${pos.toString(16)}`);
    }
}

export default MBC1;
```

MBC2 (src/emulator/mbc/MBC2.ts)

```ts
import { RAM } from "../Memory";
import { Register } from "../Register";
import MBC from "./MBC";

const RAM_ENABLED = 0x0a;
const MBC2_ROM_BANK = 1 << 8;

type MBC2Params = {
    hasBattery: boolean;
};

/**
 * Implementation of MBC2.
 * @link https://gbdev.io/pandocs/MBC2.html
 */
class MBC2 extends MBC {
    /** @link https://gbdev.io/pandocs/MBC2.html#00003fff--ram-enable-rom-bank-number-write-only */
    protected ramEnable = new Register(0x00);
    /** @link https://gbdev.io/pandocs/MBC1.html#20003fff--rom-bank-number-write-only */
    protected romBank = new Register(0x01);
    /** The RAM contained in the ROM (ERAM). */
    protected ram: RAM = new RAM(512);

    constructor(data: Uint8Array, { hasBattery }: MBC2Params) {
        super(data, hasBattery);
    }

    /**
     * Reads from the ROM, taking into account banking and the control ROMs.
     * @link https://gbdev.io/pandocs/MBC1.html#addressing-diagrams
     */
    read(pos: number): number {
        switch (pos >> 12) {
            case 0x0:
            case 0x1:
            case 0x2:
            case 0x3: {
                return this.rom.read(pos);
            }
            case 0x4:
            case 0x5:
            case 0x6:
            case 0x7: {
                const addressMask = this.size - 1; // works for powers of 2
                const address = (pos & ((1 << 14) - 1)) | (this.romBank.get() << 14);
                return this.rom.read(address & addressMask);
            }
            case 0xa:
            case 0xb: {
                // RAM disabled
                if (this.ramEnable.get() !== RAM_ENABLED) return 0xff;
                return this.ram.read(pos & 0x1ff);
            }
        }
        throw new Error(`Invalid address to read from MBC1: ${pos.toString(16)}`);
    }

    write(pos: number, data: number): void {
        switch (pos >> 12) {
            case 0x0: // RAM enable / ROM Bank Numbers
            case 0x1:
            case 0x2:
            case 0x3:
                data = data & 0b1111;
                // bit 8 controls the register
                if (pos & MBC2_ROM_BANK) {
```

```
67                      // Can't set ROM bank to 0
68                      // In reality what happens is that a value of 0 is *interpreted* as 1.
    ↪ However
69                      // simply overriding the write produces the same effect and is simpler.
70                      return this.romBank.set(data === 0 ? 1 : data);
71                  } else {
72                      return this.ramEnable.set(data);
73                  }
74
75              case 0xa:
76              case 0xb: {
77                  // RAM disabled
78                  if (this.ramEnable.get() !== RAM_ENABLED) return;
79                  return this.ram.write(pos & 0x1ff, data | 0xf0);
80              }
81          }
82
83          return;
84      }
85  }
86
87  export default MBC2;
```

MBC3 (src/emulator/mbc/MBC3.ts)

```ts
import { RAM } from "../Memory";
import { Register } from "../Register";
import MBC from "./MBC";

const RAM_ENABLED = 0x0a;

type MBC3Params = {
    hasRam: boolean;
    hasTimer: boolean;
    hasBattery: boolean;
};

/**
 * Implementation of MBC3.
 * @link https://gbdev.io/pandocs/MBC3.html
 */
class MBC3 extends MBC {
    /** @link https://gbdev.io/pandocs/MBC3.html#0000-1fff---ram-and-timer-enable-write-only */
    protected ramEnable = new Register(0x00);
    /** @link https://gbdev.io/pandocs/MBC3.html#2000-3fff---rom-bank-number-write-only */
    protected romBank = new Register(0x01);
    /** @link
    ↪ https://gbdev.io/pandocs/MBC3.html#4000-5fff---ram-bank-number---or---rtc-register-select-write-only
    ↪ */
    protected ramBank = new Register(0x00);
    /** The RAM contained in the ROM (ERAM). */
    protected ram: RAM;

    /**
     * RTC registers
     * @link https://gbdev.io/pandocs/MBC3.html#the-clock-counter-registers
     */
    protected rtcS = new Register(0x00);
    protected rtcM = new Register(0x00);
    protected rtcH = new Register(0x00);
    protected rtcDL = new Register(0x00);
    protected rtcDH = new Register(0x00);

    protected rtcRegisters: Partial<Record<number, Register>> = {
        0x08: this.rtcS,
        0x09: this.rtcM,
        0x0a: this.rtcH,
        0x0b: this.rtcDL,
        0x0c: this.rtcDH,
    };

    constructor(data: Uint8Array, { hasRam, hasTimer, hasBattery }: MBC3Params) {
        super(data, hasBattery);
        this.ram = this.createRAM();
    }

    /**
     * Resolves a GameBoy address to an address in the ERAM. This uses the banking mode and
     * the current RAM bank to determine the address.
     */
    protected resolveERAMAddress(pos: number): number {
        const pos12bits = pos & ((1 << 13) - 1);
        const ramAddressMask = this.ram.size - 1; // works for powers of 2
        const address = pos12bits | (this.ramBank.get() << 13);
        return address & ramAddressMask;
    }

    /**
     * Reads from the ROM, taking into account banking and the control ROMs.
     * @link https://gbdev.io/pandocs/MBC3.html#memory
     */
    read(pos: number): number {
```

```
66          const addressMask = this.size - 1; // works for powers of 2
67          switch (pos >> 12) {
68              case 0x0: // ROM Bank 00
69              case 0x1:
70              case 0x2:
71              case 0x3:
72                  return this.rom.read(pos & addressMask);
73
74              case 0x4: // ROM Bank 1-7
75              case 0x5:
76              case 0x6:
77              case 0x7:
78                  const address = (pos & ((1 << 14) - 1)) | (this.romBank.get() << 14);
79                  return this.rom.read(address & addressMask);
80
81              case 0xa: // ERAM
82              case 0xb:
83                  // TODO: check for RTC register ?
84                  // RAM disabled
85                  if (this.ramEnable.get() !== RAM_ENABLED) return 0xff;
86
87                  const ramBank = this.ramBank.get();
88                  if (ramBank > 0x03) {
89                      return this.rtcRegisters[ramBank]!.get();
90                  }
91
92                  const eramAddress = this.resolveERAMAddress(pos);
93                  return this.ram.read(eramAddress);
94          }
95          throw new Error(`Invalid address to read from MBC3: ${pos.toString(16)}`);
96      }
97
98      write(pos: number, data: number): void {
99          switch (pos >> 12) {
100             case 0x0: // RAM and timer enable
101             case 0x1:
102                 return this.ramEnable.set(data & 0b1111); // 4 bit register
103
104             case 0x2: // ROM Bank Number
105             case 0x3:
106                 const bits = data & 0b0111_1111; // 7bit register
107                 // Can't set ROM bank to 0
108                 // In reality what happens is that a value of 0 is *interpreted* as 1. However
109                 // simply overriding the write produces the same effect and is simpler.
110                 return this.romBank.set(bits === 0 ? 1 : bits);
111
112             case 0x4: // RAM Bank Number or RTC Register Select
113             case 0x5:
114                 if (data > 0x0c) {
115                     console.error(
116                         `Invalid RAM bank number in MBC3 (write ignored): ${data.toString(16)}`
117                     );
118                     return;
119                 }
120                 return this.ramBank.set(data);
121
122             case 0x6: // Latch clock data
123             case 0x7:
124                 // TODO: check if writes 0, then 1, then latch
125                 return;
126
127             case 0xa: // ERAM Write
128             case 0xb:
129                 if (this.ramEnable.get() !== RAM_ENABLED) return; // RAM disabled
130                 if (this.ramBank.get() > 0x03) return; // TODO: RTC registers
131                 const address = this.resolveERAMAddress(pos);
132                 return this.ram.write(address, data);
133         }
134
```

```
        throw new Error(`Invalid address to write to MBC3: ${pos.toString(16)}`);
    }
}

export default MBC3;
```

MBC5 (src/emulator/mbc/MBC5.ts)

```ts
import { RAM } from "../Memory";
import { Register } from "../Register";
import MBC from "./MBC";

const RAM_ENABLED = 0x0a;

type MBC5Params = {
    hasRam: boolean;
    hasRumble: boolean;
    hasBattery: boolean;
};

/**
 * Implementation of MBC5.
 * @link https://gbdev.io/pandocs/MBC5.html
 */
class MBC5 extends MBC {
    /** @link https://gbdev.io/pandocs/MBC5.html#0000-1fff---ram-enable-write-only */
    protected ramEnable = new Register(0x00);
    /** @link
    ↪ https://gbdev.io/pandocs/MBC5.html#2000-2fff---8-least-significant-bits-of-rom-bank-number-write-only
    ↪ */
    protected romBankLower8 = new Register(0x01);
    /** @link https://gbdev.io/pandocs/MBC5.html#3000-3fff---9th-bit-of-rom-bank-number-write-only
    ↪ */
    protected romBankUpper1 = new Register(0x00);
    /** @link https://gbdev.io/pandocs/MBC5.html#4000-5fff---ram-bank-number-write-only */
    protected ramBank = new Register(0x00);
    /** The RAM contained in the ROM (ERAM). */
    protected ram: RAM;

    constructor(data: Uint8Array, { hasRam, hasRumble, hasBattery }: MBC5Params) {
        super(data, hasBattery);
        this.ram = this.createRAM();
    }

    /**
     * Resolves a GameBoy address to an address in the ERAM. This uses the current RAM bank to
     * determine the address.
     */
    protected resolveERAMAddress(pos: number): number {
        const pos12bits = pos & ((1 << 13) - 1);
        const ramAddressMask = this.ram.size - 1; // works for powers of 2
        const address = pos12bits | (this.ramBank.get() << 13);
        return address & ramAddressMask;
    }

    /**
     * Reads from the ROM, taking into account banking and the control ROMs.
     * @link https://gbdev.io/pandocs/MBC5.html#memory
     */
    read(pos: number): number {
        const addressMask = this.size - 1; // works for powers of 2
        switch (pos >> 12) {
            case 0x0: // ROM bank 00
            case 0x1:
            case 0x2:
            case 0x3: {
                return this.rom.read(pos & addressMask);
            }
            case 0x4:
            case 0x5:
            case 0x6:
            case 0x7: {
                const address =
                    (pos & ((1 << 14) - 1)) |
                    (this.romBankLower8.get() << 14) |
```

```
                    (this.romBankUpper1.get() << 22);
                return this.rom.read(address & addressMask);
            }
            case 0xa: // ERAM
            case 0xb: {
                if (this.ramEnable.get() !== RAM_ENABLED) return 0xff;
                const address = this.resolveERAMAddress(pos);
                return this.ram.read(address);
            }
        }

        throw new Error(`Invalid address to read from MBC5: ${pos.toString(16)}`);
    }

    write(pos: number, data: number): void {
        switch (pos >> 12) {
            case 0x0: // RAM enable
            case 0x1:
                return this.ramEnable.set(data & 0b1111); // 4 bit register

            case 0x2: // ROM Bank Number (lower 8 bits)
                return this.romBankLower8.set(data);

            case 0x3: // ROM Bank Number (upper 1 bit)
                return this.romBankUpper1.set(data & 0b1); // 1 bit register

            case 0x4: // RAM Bank Number
            case 0x5:
                return this.ramBank.set(data & 0b11); // 2bit register

            case 0x6: // Nothing here
            case 0x7:
                return;

            case 0xa: // ERAM Write
            case 0xb:
                if (this.ramEnable.get() !== RAM_ENABLED) return; // RAM disabled
                const address = this.resolveERAMAddress(pos);
                return this.ram.write(address, data);
        }
        throw new Error(`Invalid address to write to MBC5: ${pos.toString(16)}`);
    }
}

export default MBC5;
```

B.1.7 Other Components

GameCartridge (src/emulator/GameCartridge.ts)

```ts
import MBC from "./mbc/MBC";
import MBC1 from "./mbc/MBC1";
import MBC2 from "./mbc/MBC2";
import MBC3 from "./mbc/MBC3";
import MBC5 from "./mbc/MBC5";
import NoMBC from "./mbc/NoMBC";
import { Addressable } from "./Memory";

const TITLE_START = 0x134;
const TITLE_END = 0x143;

const IDENTIFIER_START = 0x134;
const IDENTIFIER_END = 0x14f;

const CARTRIDGE_TYPE = 0x147;

/**
 * The game cartridge of the game boy. This class is a wrapper around the
 * different Memory Bank Controllers (MBCs), and is responsible for choosing the right MBC and
 * parsing some of the meta data from the cartridge header.
 * @link https://gbdev.io/pandocs/MBCs.html
 * @link https://gbdev.io/pandocs/The_Cartridge_Header.html
 */
class GameCartridge implements Addressable {
    /** The MBC to handle addressing etc. */
    protected mbc: MBC;

    /** The ROM's title (included from 0x134 to 0x143), in ASCII uppercase. */
    protected title: string;

    /**
     * The identifier includes the title and extra header data, to properly identify the ROM.
     * The title can't be used alone, as multiple ROMs can have the same title.
     */
    protected identifier: string;

    constructor(data: Uint8Array) {
        this.title = [...new Array(TITLE_END - TITLE_START)]
            .map((_, i) => String.fromCharCode(data[TITLE_START + i]))
            .reduce((prev, x) => prev + x, "")
            .replaceAll("\u0000", "");

        this.identifier = [...new Array(IDENTIFIER_END - IDENTIFIER_START)]
            .map((_, i) => String.fromCharCode(data[IDENTIFIER_START + i]))
            .reduce((prev, x) => prev + x, "");

        const mbcType = data[CARTRIDGE_TYPE];
        const mbcInstance = {
            // No MBC
            0x00: () => new NoMBC(data),
            // MBC1
            0x01: () => new MBC1(data, { hasRam: false, hasBattery: false }),
            0x02: () => new MBC1(data, { hasRam: true, hasBattery: false }),
            0x03: () => new MBC1(data, { hasRam: true, hasBattery: true }),
            // MBC2
            0x05: () => new MBC2(data, { hasBattery: false }),
            0x06: () => new MBC2(data, { hasBattery: true }),
            // MBC3
            0x0f: () => new MBC3(data, { hasTimer: true, hasRam: false, hasBattery: true }),
            0x10: () => new MBC3(data, { hasTimer: true, hasRam: true, hasBattery: true }),
            0x11: () => new MBC3(data, { hasTimer: false, hasRam: false, hasBattery: false }),
            0x12: () => new MBC3(data, { hasTimer: false, hasRam: true, hasBattery: false }),
            0x13: () => new MBC3(data, { hasTimer: false, hasRam: true, hasBattery: true }),
            // MBC5
```

```
            0x19: () => new MBC5(data, { hasRam: false, hasRumble: false, hasBattery: false }),
            0x1a: () => new MBC5(data, { hasRam: true, hasRumble: false, hasBattery: false }),
            0x1b: () => new MBC5(data, { hasRam: true, hasRumble: false, hasBattery: true }),
            0x1c: () => new MBC5(data, { hasRam: false, hasRumble: true, hasBattery: false }),
            0x1d: () => new MBC5(data, { hasRam: true, hasRumble: true, hasBattery: false }),
            0x1e: () => new MBC5(data, { hasRam: true, hasRumble: true, hasBattery: true }),
        }[mbcType];
        if (mbcInstance === undefined)
            throw new Error(`Invalid cartridge type: ${mbcType.toString(16)}`);
        this.mbc = mbcInstance();
        console.debug(`Saved data for game "${this.title}": `, data);
    }

    write(pos: number, data: number): void {
        this.mbc.write(pos, data);
    }

    read(pos: number): number {
        return this.mbc.read(pos);
    }

    getTitle(): string {
        return this.title;
    }

    getIdentifier(): string {
        return this.identifier;
    }

    supportsSaves(): boolean {
        return this.mbc.supportsSaves();
    }

    save(): Uint8Array | null {
        return this.mbc.save();
    }

    load(data: Uint8Array): void {
        this.mbc.load(data);
    }
}

export default GameCartridge;
```

JoypadInput (src/emulator/JoypadInput.ts)

```ts
import GameBoyInput from "./GameBoyInput";
import { Addressable } from "./Memory";

// Inputs
const BUTTON_A = 1 << 0;
const BUTTON_B = 1 << 1;
const BUTTON_SELECT = 1 << 2;
const BUTTON_START = 1 << 3;
const ARROW_RIGHT = 1 << 0;
const ARROW_LEFT = 1 << 1;
const ARROW_UP = 1 << 2;
const ARROW_DOWN = 1 << 3;

const READ_ARROWS_BIT = 1 << 4;
const READ_BUTTON_BIT = 1 << 5;
const CONTROL_BITS = READ_ARROWS_BIT & READ_BUTTON_BIT;

enum JoypadMode {
    BUTTONS,
    ARROWS,
}

/**
 * The joypad input, that takes care of receiving inputs for the buttons and directional arrows.
 * @see https://gbdev.io/pandocs/Joypad_Input.html
 */
class JoypadInput implements Addressable {
    protected input: GameBoyInput;

    // bits 0-3 are state (button or arrow)
    // bit 4 is to read arrow data
    // bit 5 is to read button data
    protected register: number = 0;

    protected buttonData: number = 0;
    protected arrowsData: number = 0;

    protected currentlyReading: JoypadMode = JoypadMode.BUTTONS;

    constructor(input: GameBoyInput) {
        this.input = input;
    }

    readInput(): void {
        const data = this.input.read();
        this.buttonData =
            (data.a ? 0 : BUTTON_A) |
            (data.b ? 0 : BUTTON_B) |
            (data.start ? 0 : BUTTON_START) |
            (data.select ? 0 : BUTTON_SELECT);
        this.arrowsData =
            (data.up ? 0 : ARROW_UP) |
            (data.down ? 0 : ARROW_DOWN) |
            (data.left ? 0 : ARROW_LEFT) |
            (data.right ? 0 : ARROW_RIGHT);
    }

    read(): number {
        const data =
            this.currentlyReading === JoypadMode.BUTTONS ? this.buttonData : this.arrowsData;
        return 0b1100_0000 | (CONTROL_BITS & this.register) | (~CONTROL_BITS & data);
    }

    write(_: number, data: number): void {
        this.register = data & 0b0011_0000;

        // the switch is done when the bit moves to a LOW state.
```

```
68          if ((this.register & READ_ARROWS_BIT) === 0) this.currentlyReading = JoypadMode.ARROWS;
69          if ((this.register & READ_BUTTON_BIT) === 0) this.currentlyReading = JoypadMode.BUTTONS;
70      }
71  }
72
73  export default JoypadInput;
```

Timer (src/emulator/Timer.ts)

```ts
import { IFLAG_TIMER } from "./constants";
import Interrupts from "./Interrupts";
import { Addressable } from "./Memory";
import { MaskRegister, DoubleRegister, Register } from "./Register";
import { Int2, wrap16 } from "./util";

/**
 * Represents the division applied to the clock speed depending on the value of the
 * first two bits of the TAC.
 * @link https://gbdev.io/pandocs/Timer_and_Divider_Registers.html#ff07--tac-timer-control
 */
const TIMER_CONTROLS = [9, 3, 5, 7];

/**
 * The TIMA counter only runs if this flag is true in the TAC.
 */
const TIMER_ENABLE_FLAG = 1 << 2;

class Timer implements Addressable {
    // DIV - divider register
    protected divider = new DoubleRegister(0xab00);
    // TIMA - timer counter
    protected timerCounter = new Register();
    // TMA - timer modulo
    protected timerModulo = new Register();
    // TAC - timer control
    protected timerControl = new MaskRegister(0b1111_1000);

    protected previousBitState: number = 0;
    protected timerOverflowed: boolean = false;
    protected previousTimerOverflowed: boolean = false;

    /**
     * Ticks the timer system in M-cycles
     * @link https://gbdev.io/pandocs/Timer_and_Divider_Registers.html
     */
    tick(interrupts: Interrupts) {
        // Increase internal counter, update DIV
        const newDivider = wrap16(this.divider.get() + 4);
        this.divider.set(newDivider);

        // Check overflow + interrupt
        this.previousTimerOverflowed = false;
        if (this.timerOverflowed) {
            const modulo = this.timerModulo.get();
            this.timerCounter.set(modulo);
            interrupts.requestInterrupt(IFLAG_TIMER);
            this.timerOverflowed = false;
            this.previousTimerOverflowed = true;
        }

        // Increase TIMA
        // Store bit for TIMA edge-detection
        const timerControl = this.timerControl.get();
        const timerIsEnabled = timerControl & TIMER_ENABLE_FLAG;
        const speedMode = (timerControl & 0b11) as Int2;
        const checkedBit = TIMER_CONTROLS[speedMode];

        // Bit is read if enabled, otherwise it's 0
        const currentBitState = timerIsEnabled && (newDivider >> checkedBit) & 1;

        // Timer increases when *read bit* goes from 1 to 0
        // This meaning turning off the timer when the bit was 1 triggers an increase
        if (this.previousBitState && !currentBitState) {
            const result = (this.timerCounter.get() + 1) & 0xff;
            this.timerCounter.set(result);

```

```
            // overflow, need to warn for reset + interrupt
            if (result === 0) {
                this.timerOverflowed = true;
            }
        }

        this.previousBitState = currentBitState;
    }

    protected addresses: Record<number, Register> = {
        0xff04: this.divider.h, // we only ever read the upper 8 bits of the divider
        0xff05: this.timerCounter,
        0xff06: this.timerModulo,
        0xff07: this.timerControl,
    };

    read(pos: number): number {
        return this.addresses[pos].get();
    }

    write(pos: number, data: number): void {
        // Trying to write anything to DIV clears it.
        if (pos === 0xff04) {
            this.divider.set(0);
            return;
        }

        const register = this.addresses[pos];

        if (register === this.timerCounter) {
            // If overflow (reload) occurred, writes are ignored
            if (this.previousTimerOverflowed) return;
            // Otherwise it negates the overflow flag
            this.timerOverflowed = false;
        }
        // If an overflow (reload) just happened, we update the value to the new modulo
        else if (register === this.timerModulo && this.previousTimerOverflowed) {
            this.timerCounter.set(data);
        }

        register.set(data);
    }
}

export default Timer;
```

Interrupts (src/emulator/Interrupts.ts)

```ts
import { IFLAG_JOYPAD, IFLAG_LCDC, IFLAG_SERIAL, IFLAG_TIMER, IFLAG_VBLANK } from "./constants";
import { Addressable } from "./Memory";
import { MaskRegister, Register } from "./Register";

/** The state of the IME. */
enum IMEType {
    DISABLED,
    WILL_ENABLE,
    WILL_ENABLE2,
    ENABLED,
}

/**
 * Transition for the IME. We need two intermediate, because the system ticks right after the CPU, so if we go
 * straight from WILL_ENABLE to ENABLED the CPU will never tick during a non-enabled state.
 */
const IMETransitions: Record<IMEType, IMEType> = {
    [IMEType.DISABLED]: IMEType.DISABLED,
    [IMEType.WILL_ENABLE]: IMEType.WILL_ENABLE2,
    [IMEType.WILL_ENABLE2]: IMEType.ENABLED,
    [IMEType.ENABLED]: IMEType.ENABLED,
};

const IFLAGS = [IFLAG_VBLANK, IFLAG_LCDC, IFLAG_TIMER, IFLAG_SERIAL, IFLAG_JOYPAD];

/**
 * Mapping of interrupt flags to their corresponding interrupt handler address.
 */
const INTERRUPT_CALLS: Record<number, number> = {
    [IFLAG_VBLANK]: 0x0040,
    [IFLAG_LCDC]: 0x0048,
    [IFLAG_TIMER]: 0x0050,
    [IFLAG_SERIAL]: 0x0058,
    [IFLAG_JOYPAD]: 0x0060,
};

class Interrupts implements Addressable {
    // Interrupts
    protected intMasterEnable: IMEType = IMEType.DISABLED; // IME - master enable flag
    protected intEnable = new Register(0x00); // IE - interrupt enable (handler)
    protected intFlag = new MaskRegister(0b1110_0000, 0xe1); // IF - interrupt flag (requests)

    protected addresses: Record<number, Register> = {
        0xffff: this.intEnable,
        0xff0f: this.intFlag,
    };

    tick(): void {
        // Tick IME
        this.intMasterEnable = IMETransitions[this.intMasterEnable];
    }

    read(address: number): number {
        return this.addresses[address].get();
    }

    write(address: number, value: number): void {
        this.addresses[address].set(value);
    }

    get interruptsEnabled(): boolean {
        return this.intMasterEnable === IMEType.ENABLED;
    }

    /** Enables the master interrupt toggle. */
    enableInterrupts() {
```

```
67          if (this.intMasterEnable === IMEType.DISABLED)
68              this.intMasterEnable = IMEType.WILL_ENABLE;
69      }
70      /** Disables the master interrupt toggle. */
71      disableInterrupts() {
72          this.intMasterEnable = IMEType.DISABLED;
73      }
74
75      /**
76       * Forces the transition to IME = enabled (needed when halting).
77       * Returns the state of the IME: enabled (true) or disabled (false)
78       */
79      fastEnableInterrupts(): boolean {
80          // forces transition
81          if (this.intMasterEnable !== IMEType.DISABLED) {
82              this.intMasterEnable = IMEType.ENABLED;
83          }
84          return this.intMasterEnable === IMEType.ENABLED;
85      }
86
87      /** Requests an interrupt for the given flag type. */
88      requestInterrupt(flag: number) {
89          this.intFlag.sflag(flag, true);
90      }
91
92      /** Returns whether there are any interrupts to handle. (IE & IF) */
93      get hasPendingInterrupt(): boolean {
94          return !!(this.intEnable.get() & this.intFlag.get() & 0b11111);
95      }
96
97      /**
98       * Returns the address for the current interrupt handler. This also disables interrupts, and
99       * clears the interrupt flag.
100      */
101     handleNextInterrupt(): number {
102         for (const flag of IFLAGS) {
103             if (this.intEnable.flag(flag) && this.intFlag.flag(flag)) {
104                 this.intFlag.sflag(flag, false);
105                 this.disableInterrupts();
106                 return INTERRUPT_CALLS[flag];
107             }
108         }
109         throw new Error("Cleared interrupt but nothing was called");
110     }
111 }
112
113 export default Interrupts;
```

WRAM (src/emulator/WRAM.ts)

```ts
import { Addressable, CircularRAM } from "./Memory";
import { MaskRegister } from "./Register";

const WRAM_SIZE = 0x2000;
const WRAM_BANK_INDEX = 0b111;
const WRAM_BANK_SIZE = 0x1000;

class DMGWRAM extends CircularRAM {
    constructor() {
        super(WRAM_SIZE, 0xc000);
    }
}

class GBCWRAM implements Addressable {
    protected bank0 = new CircularRAM(WRAM_BANK_SIZE, 0x0000);
    // For convenience, we create the 7 banks and map them to the 8 possible bank indices, since
    // a bank index of 0 is mapped to bank 1.
    protected banks1To7 = (() => {
        const banks = [...new Array(7)].map(() => new CircularRAM(WRAM_BANK_SIZE, 0x1000));
        return [banks[0], banks[0], banks[1], banks[2], banks[3], banks[4], banks[5], banks[6]];
    })();
    protected wramBank = new MaskRegister(0b1111_1000);

    protected address(address: number): Addressable {
        if (address === 0xff70) return this.wramBank;
        // we need to do modulo, since in reality WRAM is a circular buffer
        return (address - 0xc000) % WRAM_SIZE < WRAM_BANK_SIZE
            ? this.bank0
            : this.banks1To7[this.wramBank.get() & WRAM_BANK_INDEX];
    }

    read(address: number): number {
        return this.address(address).read(address);
    }

    write(address: number, value: number): void {
        this.address(address).write(address, value);
    }
}

export { DMGWRAM, GBCWRAM };
```

B.1.8 Helpers

Memory (src/emulator/Memory.ts)

```ts
/**
 * A basic interface for all addressable objects.
 */
interface Addressable {
    read(pos: number): number;
    write(pos: number, data: number): void;
}

/**
 * Simple read only memory object.
 */
class ROM implements Addressable {
    size: number;
    protected data: Uint8Array;

    constructor(size: number, data?: Uint8Array) {
        this.size = size;
        this.data = data ?? new Uint8Array(size);
    }

    read(pos: number) {
        return this.data[pos];
    }

    write(pos: number, data: number): void {
        throw new Error("Can't write to ROM.");
    }

    /**
     * Returns the raw data of this memory. Shouldn't be used for regular access - useful for
     * backups and save states.
     */
    rawData(): Uint8Array {
        return this.data;
    }

    /**
     * Sets the raw data of this memory. Shouldn't be used for regular access - useful for
    ↪  backups
     * and save states.
     */
    rawSet(data: Uint8Array): void {
        this.data.set(data);
    }
}

/**
 * Live memory, that can be read from and written to.
 */
class RAM extends ROM {
    write(pos: number, data: number) {
        this.data[pos] = data;
    }
}

/**
 * Circular RAM is similar to RAM, but it also stores an offset. Any access on the memory (both
 * reads and writes) occur at the position (X - O) % S, where S is the size of the memory, O the
 * offset, and X the accessed address. This means out of bound exeptions cannot happen.
 */
class CircularRAM extends RAM {
    protected offset: number;

    constructor(size: number, offset: number, data?: Uint8Array) {
```

```
        super(size, data);
        this.offset = offset;
    }

    override read(pos: number): number {
        return super.read((pos - this.offset) % this.size);
    }

    override write(pos: number, data: number): void {
        super.write((pos - this.offset) % this.size, data);
    }
}

export type { Addressable };
export { ROM, RAM, CircularRAM };
```

Register (src/emulator/Register.ts)

```ts
import { Addressable } from "./Memory";
import { wrap16 } from "./util";

/**
 * A register, containing an 8bit value.
 * For convenience's sake, a register implements `Addressable`. Calling `read` will simply
 * return the value (ignoring the position), and calling `write` will simply set the value,
 * ignoring the position too.
 */
class Register implements Addressable {
    protected value: number;

    /** Initialises a subregister with either 0 or a value */
    constructor(value: number = 0) {
        this.value = value;
    }

    /** Sets this register byte to the given value */
    set(value: number) {
        this.value = value;
    }
    /** The value of this register byte */
    get() {
        return this.value;
    }
    /** Sets the given flag to 0/1 according to the boolean */
    sflag(flag: number, state: boolean) {
        this.set(state ? this.value | flag : this.value & ~flag);
    }
    /** Returns if the given flag is set */
    flag(flag: number) {
        return (this.get() & flag) === flag;
    }

    /** Read this subregister. */
    read(): number {
        return this.get();
    }
    /** Write to this subregister. Position's ignored. */
    write(_: number, data: number): void {
        this.set(data);
    }
}

/**
 * A MaskRegister is similar to a Register but it only uses a set given of bits. All
 * other bits are hard-wired to 1, and can't be changed.
 */
class MaskRegister extends Register {
    protected mask: number;

    /**
     * Constructs a PaddedSubRegister, using only the given number of bits
     * @param usedBits Either the number of used bits (the end / most significant will be
     * padded) or an array containing the mask of the subregister.
     * @param value The initial value of the register
     */
    constructor(mask: number, value: number = 0) {
        mask &= 0xff;
        super(value | mask);
        this.mask = mask;
    }

    override set(value: number): void {
        super.set(value | this.mask);
    }
}
```

```
68
69 /**
70  * A register, containing two 8 bit values.
71  */
72 class DoubleRegister {
73     // Most significant byte (0xFF00)
74     public h = new Register();
75     // Least significant byte (0x00FF)
76     public l = new Register();
77
78     constructor(value: number = 0) {
79         this.set(value);
80     }
81
82     /** Sets this register to the given 16bit value. */
83     set(value: number) {
84         this.h.set((value >> 8) & 0xff);
85         this.l.set(value & 0xff);
86     }
87
88     /** Returns the 16bit value in this register */
89     get() {
90         return (this.h.get() << 8) | this.l.get();
91     }
92
93     /** Increments this register's value and returns the previous value (equivalent to r++) */
94     inc() {
95         const temp = this.get();
96         this.set(wrap16(temp + 1));
97         return temp;
98     }
99
100    /** Decrements this register's value and returns the previous value (equivalent to r--) */
101    dec() {
102        const temp = this.get();
103        this.set(wrap16(temp - 1));
104        return temp;
105    }
106 }
107
108 const Register00: Addressable = { read: () => 0x00, write: () => {} };
109 const RegisterFF: Addressable = { read: () => 0xff, write: () => {} };
110
111 export { DoubleRegister, Register, MaskRegister, Register00, RegisterFF };
```

constants (src/emulator/constants.ts)

```ts
// CPU
export const CLOCK_SPEED = 4194304; // 2^22Hz
export const FRAME_RATE = 60;
export const CYCLES_PER_FRAME = Math.floor(CLOCK_SPEED / FRAME_RATE);

// Screen
export const SCREEN_WIDTH = 160;
export const SCREEN_HEIGHT = 144;

// Memory
export const HRAM_SIZE = 352;

// Interrupt flags
export const IFLAG_VBLANK = 1 << 0;
export const IFLAG_LCDC = 1 << 1;
export const IFLAG_TIMER = 1 << 2;
export const IFLAG_SERIAL = 1 << 3;
export const IFLAG_JOYPAD = 1 << 4;

// Types
export enum ConsoleType {
    DMG,
    CGB,
}
export enum SpeedMode {
    Normal,
    Double,
}
export enum CGBMode {
    CGB,
    DMG,
    DMGExtended,
}
```

util (src/emulator/util.ts)

```ts
export const high = (x: number) => (x >> 8) & 0xff;
export const low = (x: number) => x & 0xff;
export const combine = (high: number, low: number) => (high << 8) | low;
export const wrap8 = (x: number) => x & 0xff;
export const wrap16 = (x: number) => x & 0xffff;
export const asSignedInt8 = (x: number) => (x > 127 ? x - 256 : x);
export const clamp = (x: number, min: number, max: number) => Math.min(Math.max(x, min), max);
export const range = (from: number, to: number) =>
    [...new Array(to - from + 1)].map((_, i) => i + from);
export const rangeObject = <T>(from: number, to: number, obj: T) =>
    Object.fromEntries(range(from, to).map((i) => [i, obj]));

export type Int1 = 0 | 1;
export type Int2 = 0 | 1 | 2 | 3;
export type Int3 = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7;
export type Int4 = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15;
```

B.2 Frontend Code

B.2.1 Main App

main (src/frontend/main.tsx)

```tsx
import { render } from "preact";

import App from "@frontend/app";
import { ConfigProvider } from "@helpers/ConfigContext";

import "@frontend/index.css";
import "@frontend/mobile.css";

render(
    <ConfigProvider>
        <App />
    </ConfigProvider>,
    document.getElementById("app") as HTMLElement
);
```

app (src/frontend/app.tsx)

```tsx
import { useSignal } from "@preact/signals";
import localforage from "localforage";
import { FastForward, Pause, Play, Redo, Save, Volume2, VolumeX } from "lucide-preact";
import { FunctionalComponent } from "preact";
import { useCallback, useEffect, useRef, useState } from "preact/hooks";

import AudioPlayer from "@helpers/AudioPlayer";
import { useConfig } from "@helpers/ConfigContext";

import { addAlert, AlertManager } from "@components/Alerts";
import Drawer from "@components/Drawer/Drawer";
import GameInput from "@components/GameInput";
import IconButton from "@components/IconButton";
import RomInput from "@components/RomInput";
import Screen, { VideoReceiver } from "@components/Screen";

import GameBoyColor from "@emulator/GameBoyColor";
import GameBoyInput from "@emulator/GameBoyInput";
import GameBoyOutput from "@emulator/GameBoyOutput";

const CACHE_KEY = "rom";
const SAVE_CACHE_KEY = "save_";

const App: FunctionalComponent = () => {
    // Interaction
    const tripleSpeed = useSignal(false);
    const emulatorRunning = useSignal(true);
    const canStep = useSignal(true);
    const [config, setConfig] = useConfig();
    const configPtr = useSignal(config);
    useEffect(() => {
        configPtr.value = config;
    }, [config]);
    const soundOutput = useSignal<AudioPlayer | undefined>(undefined);
    const joypadInput = useSignal<undefined | GameBoyInput>(undefined);

    // DOM Refs
    const emulatorFrameIn = useRef<VideoReceiver | undefined>(undefined);
    const bgDebugger = useRef<VideoReceiver | undefined>(undefined);
    const tilesetDebugger = useRef<VideoReceiver | undefined>(undefined);

    // Emulator data
    const [loadedGame, setLoadedGame] = useState<Uint8Array>();
    const [gameboy, setGameboy] = useState<GameBoyColor>();

    const effectivePalette = gameboy?.getMode() === "DMG" ? config.gbPalette : undefined;

    // Debug state
    const stepCount = useRef<HTMLDivElement>(null);
    const millisPerFrame = useRef<HTMLDivElement>(null);

    const toggleHasSound = () => {
        const audioEnabled = !config.audioEnabled;
        setConfig({ audioEnabled });
        if (audioEnabled) {
            soundOutput.value = new AudioPlayer({
                get value() {
                    return configPtr.value.volume;
                },
            });
        } else {
            soundOutput.value?.delete();
            delete soundOutput.value;
        }
    };

    const saveGame = useCallback(() => {
```

```
        if (gameboy && gameboy.supportsSaves()) {
            const save = gameboy.save();
            if (!save) return;
            localforage.setItem(SAVE_CACHE_KEY + gameboy.getIdentifier(), save, (err) => {
                if (err)
                    console.error(
                        `Could not save game ${gameboy.getTitle()}
    ↪ (${gameboy.getIdentifier()}):`,
                        err
                    );
                else {
                    console.log(
                        `Saved game ${gameboy.getTitle()} (${gameboy.getIdentifier()})`
                    );
                    addAlert(`Saved game '${gameboy.getTitle()}'`);
                }
            });
        }
    }, [gameboy]);

    /** Backup  */
    useEffect(() => {
        window.addEventListener("beforeunload", saveGame);
        return () => window.removeEventListener("beforeunload", saveGame);
    }, [saveGame]);

    /**
     * Loads a ROM into the gameboy, instantiating it. Also creates the 2nd emulator if needed
     */
    const loadGame = useCallback(
        (rom: Uint8Array) => {
            /** Save previous state, clear variables */
            saveGame();

            /** Setup input (we can't pass the value directly, because the object might change)
    ↪ */
            const gameIn: GameBoyInput = {
                read: () => joypadInput.value!.read(),
            };

            /** Setup output (relies on most things not changing) */
            let serialOutTxt = "";
            const gbOut: GameBoyOutput = {
                get receiveGraphics() {
                    return emulatorFrameIn.current;
                },
                receiveSound: (d) => soundOutput.value?.enqueue(d),
                serialOut: (d) => {
                    serialOutTxt += String.fromCharCode(d);
                    console.log("Serial out > ", serialOutTxt);
                },
                get debugBackground() {
                    return bgDebugger.current;
                },
                get debugTileset() {
                    return tilesetDebugger.current;
                },
            };

            let bootRom: Uint8Array | null = null;
            if (config.bootRom === "real") {
                if (config.console === "dmg") bootRom = config.bootRomDmg;
                else bootRom = config.bootRomCgb;
            }

            /** Create the emulator (ensure it loads correctly.) */
            let gbc: GameBoyColor;
            try {
                gbc = new GameBoyColor(
```

```
                    config.console === "dmg" ? "DMG" : "CGB",
                    rom,
                    gameIn,
                    gbOut,
                    {
                        bootRom,
                    }
                );
            } catch (e) {
                alert("Could not load ROM: " + e);
                return;
            }
            addAlert(`Loaded game '${gbc.getTitle()}'`);

            /** Load a save (if one exists) */
            if (gbc.supportsSaves()) {
                localforage.getItem<Uint8Array>(
                    SAVE_CACHE_KEY + gbc.getIdentifier(),
                    (err, save) => {
                        if (!save) return;
                        try {
                            gbc.load(save);
                            console.log(
                                `Loaded save for ${gbc.getTitle()} (${gbc.getIdentifier()})`
                            );
                            addAlert(`Loaded save for '${gbc.getTitle()}'`);
                        } catch (e) {
                            console.error(
                                `Could not load save for ${gbc.getTitle()}
↪  (${gbc.getIdentifier()}):`,
                                e
                            );
                        }
                    }
                );
            }

            setGameboy(gbc);

            let lastFrame = Date.now();

            /** Run the emulator (this is the "main loop") */
            const runEmulator = () => {
                /**
                 * This is a bit of a hack to ensure that the emulator doesn't run if the
                 * instance has changed. It doesn't update the state, because the instance
                 * remains the same.
                 * This relies on the fact the state setter (setGameboy) doesn't change and is
                 * synchronous.
                 */
                const expectedInstance = gbc;
                let currentInstance: GameBoyColor | undefined = undefined;
                setGameboy((g) => (currentInstance = g));

                // if the instance has changed, stop this loop.
                if (currentInstance !== expectedInstance) return;

                /** Run the emulator */
                const now = Date.now();
                const delta = now - lastFrame;
                lastFrame = now;
                const framesToRun = Math.min(3, delta / (1000 / 60)); // can't catch up more than
↪  3 frames
                const speed = tripleSpeed.value ? 3 : 1;
                const frames = framesToRun * speed;

                const before = configPtr.value.showStats ? performance.now() : 0;
                gbc.drawFrame(frames, !emulatorRunning.value);

```

```
                if (configPtr.value.showStats && stepCount.current && millisPerFrame.current) {
                    const millis = performance.now() - before;
                    const cpuSteps = gbc["cpu"]["stepCounter"];

                    stepCount.current.innerHTML = `${cpuSteps.toLocaleString()} steps`;
                    millisPerFrame.current.innerText = `${millis.toLocaleString()} ms/frame`;
                }

                /** Need to handle wait for a step to be made. */
                if (!emulatorRunning.value) {
                    const waitForStep = () => {
                        if (canStep.value || emulatorRunning.value) {
                            canStep.value = false;
                            lastFrame = Date.now();
                            requestAnimationFrame(runEmulator);
                        } else {
                            requestAnimationFrame(waitForStep);
                        }
                    };
                    requestAnimationFrame(waitForStep);
                } else {
                    requestAnimationFrame(runEmulator);
                }
            };

            requestAnimationFrame(runEmulator);

            // @ts-ignore helpful for debugging :)
            window.gbc = gbc;
            return gbc;
        },
        [gameboy, config]
    );

    /**
     * Utility refresh: gets the caches ROM and plays it.
     */
    useEffect(() => {
        const listener = (e: KeyboardEvent) => {
            if (e.key === "r" && loadedGame) {
                var target = (e.target || e.srcElement) as HTMLElement;
                var targetTagName = target === null ? "null" : target.nodeName.toUpperCase();
                if (/INPUT|SELECT|TEXTAREA/.test(targetTagName)) {
                    return;
                }
                loadGame(loadedGame);
            }
        };
        document.addEventListener("keydown", listener);
        return () => document.removeEventListener("keydown", listener);
    }, [loadGame, setLoadedGame]);

    /**
     * Cache the loaded ROM, and load it in.
     */
    const loadRom = useCallback(
        (rom: ArrayBuffer) => {
            const romArray = new Uint8Array(rom);
            // try caching the rom for reloads / refreshes
            localforage.setItem(
                CACHE_KEY,
                romArray,
                (err) => err && console.warn("Error caching ROM: ", err)
            );
            setLoadedGame(romArray);
            loadGame(romArray);
        },
        [loadGame, setLoadedGame]
    );
```

```
271
272     /**
273      * Hot load: if a ROM is cached, instantly loads it on startup.
274      */
275     useEffect(() => {
276         localforage.getItem<Uint8Array>(CACHE_KEY, (err, value) => {
277             if (!value) return;
278             setLoadedGame(value);
279             loadGame(value);
280         });
281     }, []);
282
283     return (
284         <>
285             <AlertManager />
286             <Drawer loadRom={loadRom} />
287
288             <div id="emulator">
289                 <h1>Emmy</h1>
290                 <h2>The GBC Browser Emulator</h2>
291
292                 <RomInput onLoad={loadRom} />
293
294                 <div id="emu-options">
295                     <IconButton
296                         title="Play/Pause"
297                         Icon={emulatorRunning.value ? Pause : Play}
298                         onClick={() =>
299                             (emulatorRunning.value = canStep.value = !emulatorRunning.value)
300                         }
301                         showTooltip
302                     />
303
304                     <IconButton
305                         title="Step"
306                         Icon={Redo}
307                         onClick={() => (canStep.value = true)}
308                         disabled={emulatorRunning.value}
309                         showTooltip
310                     />
311
312                     <IconButton
313                         title="Sound Enabled"
314                         onClick={toggleHasSound}
315                         Icon={config.audioEnabled ? Volume2 : VolumeX}
316                         showTooltip
317                     />
318
319                     <IconButton
320                         id="emu-speed"
321                         title="Triple Speed"
322                         onClick={() => (tripleSpeed.value = !tripleSpeed.value)}
323                         Icon={FastForward}
324                         toggled={tripleSpeed.value}
325                         showTooltip
326                     />
327
328                     <IconButton
329                         title={
330                             gameboy?.supportsSaves() === false
331                                 ? "Game doesn't support saves "
332                                 : "Save Game"
333                         }
334                         onClick={() => saveGame()}
335                         Icon={Save}
336                         showTooltip
337                         disabled={!gameboy || !gameboy.supportsSaves()}
338                     />
339                 </div>
```

```
340
341                 {gameboy && (
342                     <div id="emu-stack">
343                         {config.showStats && (
344                             <div id="emu-stats">
345                                 <div ref={stepCount} />
346                                 <div ref={millisPerFrame} />
347                             </div>
348                         )}
349                         <Screen
350                             inputRef={emulatorFrameIn}
351                             scale={1 << config.scale}
352                             Filter={config.filter}
353                             blending={config.frameBlending}
354                             palette={effectivePalette}
355                             id="emulator-frame"
356                         />
357                     </div>
358                 )}
359
360                 <GameInput inputHandler={(x) => (joypadInput.value = x)} />
361
362                 {gameboy && config.showDebugScreens && (
363                     <div id="emu-screens">
364                         <Screen width={256} height={256} inputRef={bgDebugger} />
365                         <Screen width={256} height={192} inputRef={tilesetDebugger} />
366                     </div>
367                 )}
368             </div>
369         </>
370     );
371 };
372
373 export default App;
```

B.2.2 Main Styling

index (src/frontend/index.css)

```
:root {
    font-family: Inter, Avenir, Helvetica, Arial, sans-serif;
    font-size: 16px;
    line-height: 24px;
    font-weight: 400;

    color-scheme: light dark;
    color: rgba(255, 255, 255, 0.87);
    background-color: #242424;

    font-synthesis: none;
    text-rendering: optimizeLegibility;
    -webkit-font-smoothing: antialiased;
    -moz-osx-font-smoothing: grayscale;
    -webkit-text-size-adjust: 100%;
}

#app {
    width: 100%;
    height: 100vh;
    display: flex;
    flex-direction: row;
}

#emulator {
    text-align: center;
    padding: 2rem;
    width: 100%;
    overflow-x: auto;
}

a {
    font-weight: 500;
    color: #646cff;
    text-decoration: inherit;
}
a:hover {
    color: #535bf2;
}

body {
    margin: 0;
    display: flex;
    min-width: 320px;
    min-height: 100vh;
}

h1 {
    font-size: 3.2em;
    line-height: 1.1;
}

canvas {
    image-rendering: pixelated;
    max-width: 100%;
}

#emu-stack {
    display: flex;
    flex-direction: column;
    align-items: center;
}

#emu-options {
```

```
    display: flex;
    flex-direction: row;
    justify-content: center;
    margin: 16px 0;
}

#emu-options > * {
    margin: 0 8px;
}

#emu-screens {
    display: flex;
    flex-direction: row;
    align-items: center;
    justify-content: center;
    flex-wrap: wrap;
}

.screen-container {
    margin: 4px;
    margin-bottom: -3px;
}

#emu-stack code {
    padding: 2em;
    white-space: break-spaces;
    overflow-wrap: anywhere;
}

code {
    font-family: "Fira Code", monospace;
    transition: color 0.5s ease;
}

button {
    position: relative;
    border-radius: 8px;
    border: 1px solid transparent;
    padding: 0.6em 1.2em;
    font-size: 1em;
    font-weight: 500;
    font-family: inherit;
    background-color: #1a1a1a;
    transition: border-color 0.25s ease, background-color 0.25s ease;
}
button:not(:disabled) {
    cursor: pointer;
}
button:hover:not(:disabled) {
    border-color: #646cff;
}
button:focus,
button:focus-visible {
    outline: 0;
    background-color: #202020;
    border-color: mediumslateblue;
}

button.icon-button {
    border-radius: 4px;
    padding: 0.2em 0.4em;
    font-size: 0.5em;
    aspect-ratio: 1/1;
}

button.icon-button svg {
    transform: scale(0.5);
}

```

```
button.icon-button.toggled {
    border-color: greenyellow;
}

button > .tooltip {
    position: absolute;
    top: 0;
    left: 50%;
    transform: translate(-50%, -50%);
    text-align: center;
    opacity: 0;
    margin-top: -1.5em;
    width: max-content;
    background-color: #1a1a1a77;
    font-size: 1.3em;
    padding: 2px;
    border-radius: 2px;
    transition: opacity 0.1s ease;
    pointer-events: none;
}

button:hover > .tooltip {
    display: block;
    opacity: 1;
}

@media (prefers-color-scheme: light) {
    :root {
        color: #213547;
        background-color: #ffffff;
    }
    a:hover {
        color: #747bff;
    }
    button {
        background-color: #f9f9f9;
    }
}

@media (any-pointer: fine) {
    .mobile-only {
        display: none;
    }
}
```

mobile (src/frontend/mobile.css)

```
@media (max-width: 700px) {
    #drawer {
        order: 2;
        width: calc(100vw - 2em) !important;
        max-width: calc(100vw - 2em) !important;
        min-width: calc(100vw - 2em) !important;
        overflow-x: hidden;
        overflow-y: unset;
    }

    #drawer .resizer {
        display: none;
    }

    #emulator {
        width: unset;
        padding: 0.5rem;
    }

    #app {
        height: unset;
        flex-direction: column;
    }
}
```

B.2.3 Components

Alerts (src/frontend/components/Alerts.tsx)

```tsx
import { FunctionalComponent } from "preact";
import { useEffect, useState } from "preact/hooks";

import "./Alerts.css";

type Alert = {
    id: number;
    text: string;
    endTime: Date;
    state: "new" | "visible" | "fading";
    fade: number;
};

const alerts: Alert[] = [];
let keyId: number = 0;

function addAlert(text: string, duration: number = 3) {
    alerts.push({
        id: keyId++,
        text,
        endTime: new Date(Date.now() + duration * 1000),
        state: "new",
        fade: 0,
    });
}

const AlertManager: FunctionalComponent = () => {
    const [refresh, setRefresh] = useState<number>(0);

    useEffect(() => {
        const interval = setInterval(() => {
            let changed = false;
            const now = Date.now();
            for (const alert of alerts) {
                if (alert.state === "new") {
                    alert.state = "visible";
                    changed = true;
                }
                if (alert.endTime.getTime() < now) {
                    if (alert.state === "visible") {
                        alert.state = "fading";
                        changed = true;
                    } else {
                        alert.fade++;
                        if (alert.fade > 5) {
                            alerts.splice(alerts.indexOf(alert), 1);
                            changed = true;
                        }
                    }
                }
            }
            if (changed) setRefresh((r) => r + 1);
        }, 100);
        return () => clearInterval(interval);
    }, [setRefresh]);

    return (
        <div id="alert-box">
            {alerts.map((alert) => (
                <div key={alert.id} className={`alert ${alert.state} ${alert.id}`}>
                    {alert.text}
                </div>
            ))}
        </div>
```

```
    );
};

export { addAlert, AlertManager };
```

Alerts (`src/frontend/components/Alerts.css`)

```css
#alert-box {
    position: fixed;
    top: 0;
    left: 0;
    width: 100%;
    height: 100%;
    z-index: 100;
    display: flex;
    flex-direction: column;
    justify-content: end;
    align-items: end;
    pointer-events: none;
}

#alert-box .alert {
    padding: 5px;
    background-color: rgba(0, 0, 0, 0.5);
    color: white;
    border-radius: 5px;
    margin: 5px;
    transition: opacity 0.2s ease;
}

#alert-box .alert.visible {
    opacity: 1;
}

#alert-box .alert.fading {
    opacity: 0;
}
```

GameInput (src/frontend/components/GameInput.tsx)

```tsx
import { FunctionalComponent } from "preact";
import { useCallback, useEffect, useMemo } from "preact/hooks";

import GameBoyInput from "@emulator/GameBoyInput";
import { useConfig } from "@helpers/ConfigContext";
import useKeys from "@helpers/useKeys";

import "./GameInput.css";

type GameBoyInputObj = ReturnType<GameBoyInput["read"]>;

type JoypadButtonProps = {
    name: string;
    objKey: keyof GameBoyInputObj;
    obj: GameBoyInputObj;
};

const JoypadButton: FunctionalComponent<JoypadButtonProps> = ({ name, objKey, obj }) => {
    return (
        <button
            className={`control-button btn-${objKey}`}
            onTouchStart={(e) => {
                obj[objKey] = true;
                if (e.cancelable) e.preventDefault();
            }}
            onTouchEnd={(e) => {
                obj[objKey] = false;
                if (e.cancelable) e.preventDefault();
            }}
        >
            {name}
        </button>
    );
};

const NO_INPUT: GameBoyInputObj = {
    a: false,
    b: false,
    start: false,
    select: false,
    up: false,
    down: false,
    left: false,
    right: false,
};

type GameInputProps = {
    inputHandler: (input: GameBoyInput) => void;
};

const GameInput: FunctionalComponent<GameInputProps> = ({ inputHandler }) => {
    const [
        {
            controlA,
            controlB,
            controlStart,
            controlSelect,
            controlArrowUp,
            controlArrowDown,
            controlArrowLeft,
            controlArrowRight,
        },
    ] = useConfig();

    const pressedKeys = useKeys([
        controlA,
        controlB,
```

```
        controlStart,
        controlSelect,
        controlArrowUp,
        controlArrowDown,
        controlArrowLeft,
        controlArrowRight,
    ]);

    const touchControlStatus = useMemo(() => ({ ...NO_INPUT }), []);

    const inputFn = useMemo(() => {
        const obj = { ...NO_INPUT };
        return () => {
            obj.a = touchControlStatus.a || pressedKeys.includes(controlA);
            obj.b = touchControlStatus.b || pressedKeys.includes(controlB);
            obj.start = touchControlStatus.start || pressedKeys.includes(controlStart);
            obj.select = touchControlStatus.select || pressedKeys.includes(controlSelect);
            obj.up = touchControlStatus.up || pressedKeys.includes(controlArrowUp);
            obj.down = touchControlStatus.down || pressedKeys.includes(controlArrowDown);
            obj.left = touchControlStatus.left || pressedKeys.includes(controlArrowLeft);
            obj.right = touchControlStatus.right || pressedKeys.includes(controlArrowRight);
            return obj;
        };
    }, [
        touchControlStatus,
        pressedKeys,
        controlA,
        controlB,
        controlStart,
        controlSelect,
        controlArrowUp,
        controlArrowDown,
        controlArrowLeft,
        controlArrowRight,
    ]);

    useEffect(() => {
        inputHandler({ read: inputFn });
    }, [inputFn]);

    return (
        <div className="mobile-only joypad-input">
            <div className="arrow-buttons">
                <JoypadButton name="↑" objKey="up" obj={touchControlStatus} />
                <JoypadButton name="←" objKey="left" obj={touchControlStatus} />
                <JoypadButton name="→" objKey="right" obj={touchControlStatus} />
                <JoypadButton name="↓" objKey="down" obj={touchControlStatus} />
            </div>
            <div className="main-buttons">
                <JoypadButton name="A" objKey="a" obj={touchControlStatus} />
                <JoypadButton name="B" objKey="b" obj={touchControlStatus} />
                <JoypadButton name="START" objKey="start" obj={touchControlStatus} />
                <JoypadButton name="SELECT" objKey="select" obj={touchControlStatus} />
            </div>
        </div>
    );
};

export default GameInput;
```

GameInput (src/frontend/components/GameInput.css)

```css
.joypad-input {
    display: flex;
    padding: 5px;
    justify-content: space-between;
}

.joypad-input > * {
    width: 45%;
    position: relative;
    aspect-ratio: 1;
}

.joypad-input .arrow-buttons > * {
    position: absolute;
    border-radius: 100%;
    width: 38%;
    inset: 0;
    aspect-ratio: 1;
    margin: auto;
}

.joypad-input .arrow-buttons .btn-up {
    bottom: unset;
}

.joypad-input .arrow-buttons .btn-left {
    right: unset;
}

.joypad-input .arrow-buttons .btn-right {
    left: unset;
}

.joypad-input .arrow-buttons .btn-down {
    top: unset;
}

.joypad-input .main-buttons > * {
    position: absolute;
    width: 55%;
    height: 23%;
}

.joypad-input .main-buttons .btn-a {
    bottom: 0;
    left: 0;
}

.joypad-input .main-buttons .btn-b {
    bottom: 25%;
    right: 0;
}

.joypad-input .main-buttons .btn-start {
    top: 25%;
    left: 0;
}

.joypad-input .main-buttons .btn-select {
    top: 0;
    right: 0;
}
```

IconButton (src/frontend/components/IconButton.tsx)

```tsx
import { LucideProps } from "lucide-preact";
import { FunctionalComponent, JSX } from "preact";

type IconButtonProps = {
    id?: string;
    title: string;
    Icon: (props: LucideProps) => JSX.Element;
    toggled?: boolean;
    disabled?: boolean;
    onClick?: () => void;
    showTooltip?: boolean;
};

const IconButton: FunctionalComponent<IconButtonProps> = ({
    id,
    title,
    Icon,
    toggled,
    disabled,
    onClick,
    showTooltip,
}) => {
    return (
        <button
            title={title}
            className={`icon-button ${toggled ? "toggled" : ""}`}
            onClick={onClick}
            disabled={disabled}
            id={id}
        >
            {showTooltip && <div className="tooltip">{title}</div>}
            <Icon />
        </button>
    );
};

export default IconButton;
```

KeybindingInput (src/frontend/components/KeybindingInput.tsx)

```tsx
import { FunctionalComponent } from "preact";
import { useCallback, useEffect, useRef, useState } from "preact/hooks";

import "./KeybindingInput.css";

type KeybindingInputProps = {
    value: string;
    onChange: (value: string) => void;
};

const KEY_DISPLAY: Record<string, string | undefined> = {
    ArrowUp: "↑",
    ArrowDown: "↓",
    ArrowLeft: "←",
    ArrowRight: "→",
    " ": "Space",
};

const KeybindingInput: FunctionalComponent<KeybindingInputProps> = ({ value, onChange }) => {
    const [isEditing, setIsEditing] = useState<boolean>(false);
    const [key, setKey] = useState<string | undefined>(undefined);

    const ref = useRef<HTMLButtonElement>(null);

    const onKeyDown = useCallback(
        (e: KeyboardEvent) => {
            if (e.key === "Escape") {
                setIsEditing(false);
                ref.current?.blur();
                return;
            }
            if (e.key === "Enter" && key) {
                setIsEditing(false);
                onChange(key);
                ref.current?.blur();
                return;
            }

            setKey(e.key);
            e.stopPropagation();
            e.preventDefault();
        },
        [setIsEditing, onChange, setKey, key]
    );

    useEffect(() => {
        if (isEditing) {
            document.addEventListener("keydown", onKeyDown);
        }
        return () => {
            document.removeEventListener("keydown", onKeyDown);
        };
    }, [isEditing, onKeyDown]);

    return (
        <button
            ref={ref}
            className={`keybinding-input ${isEditing ? "editing" : ""}`}
            onClick={() => {
                setIsEditing(true);
                setKey(undefined);
            }}
            onBlur={() => {
                setIsEditing(false);
            }}
        >
            {isEditing
```

```
                ? key
                    ? KEY_DISPLAY[key] ?? key
                    : "Press a key"
                : KEY_DISPLAY[value] ?? value}
        </button>
    );
};

export default KeybindingInput;
```

KeybindingInput (src/frontend/components/KeybindingInput.css)

```css
.keybinding-input {
    text-align: center;
    font-family: "Fira Code", monospace;
    font-size: 12px;
    background: #1e1e1e;
    margin: 1px;
    padding: 1px;
    border-radius: 4px;
}

.keybinding-input:focus {
    background: #2d2d2d;
}

/* Animate text pulsating */
.keybinding-input.editing {
    font-weight: bolder;
    animation: pulse 1s infinite;
}

@keyframes pulse {
    0% {
        color: #eee;
    }
    50% {
        color: #888;
    }
    100% {
        color: #eee;
    }
}
```

Resizable (src/frontend/components/Resizable.tsx)

```tsx
import { useSignal } from "@preact/signals";
import { FunctionalComponent, JSX } from "preact";

import "./Resizable.css";

export type ResizableProps = JSX.HTMLAttributes<HTMLDivElement> & { initalWidth: number };

export const Resizable: FunctionalComponent<ResizableProps> = ({
    initalWidth,
    children,
    ...rest
}) => {
    const width = useSignal(initalWidth);

    const mouseDownHandler = (e: MouseEvent) => {
        const startX = e.clientX;
        const startWidth = width.value;

        const onMouseMove = (e: MouseEvent) => {
            width.value = startWidth + (e.clientX - startX);
            e.stopPropagation();
        };

        const onMouseUp = () => {
            document.removeEventListener("mousemove", onMouseMove);
            document.removeEventListener("mouseup", onMouseUp);
        };

        document.addEventListener("mousemove", onMouseMove);
        document.addEventListener("mouseup", onMouseUp);
    };

    return (
        <div {...rest} style={{ minWidth: width.value, maxWidth: width.value }}>
            {children}
            <div className="resizer" onMouseDown={mouseDownHandler} />
        </div>
    );
};
```

Resizable (`src/frontend/components/Resizable.css`)

```
.resizer {
    position: absolute;
    width: 10px;
    height: 100%;
    top: 0;
    right: 0;
    transform: translateX(50%);
    z-index: 1;
    touch-action: none;
    cursor: ew-resize;
}
```

RomInput (src/frontend/components/RomInput.tsx)

```tsx
import { FunctionalComponent } from "preact";
import { useRef } from "preact/hooks";
import { JSXInternal } from "preact/src/jsx";

type RomInputProps = {
    onLoad: (data: ArrayBuffer) => void;
};

const RomInput: FunctionalComponent<RomInputProps> = ({ onLoad }) => {
    const fileInput = useRef<HTMLInputElement>(null);

    const romClick = () => {
        fileInput.current?.click();
    };

    const romChange: JSXInternal.GenericEventHandler<HTMLInputElement> = (e) => {
        if (!e.currentTarget.files?.length) return;
        const uploadedRom = e.currentTarget.files[0];
        const reader = new FileReader();
        reader.readAsArrayBuffer(uploadedRom);
        reader.onload = (e) => {
            if (e.target?.result && typeof e.target.result === "object") {
                onLoad(e.target.result);
            }
        };
        console.log("Uploaded rom", uploadedRom);
    };

    return (
        <>
            <button onClick={romClick}>Import a ROM</button>

            <input
                type="file"
                id="rom-input"
                accept=".gb,.gbc"
                ref={fileInput}
                onChange={romChange}
                style={{ display: "none" }}
            />
        </>
    );
};

export default RomInput;
```

Screen (src/frontend/components/Screen.tsx)

```tsx
import { FunctionalComponent } from "preact";
import { MutableRef, useEffect, useMemo, useRef, useState } from "preact/hooks";

import { SCREEN_HEIGHT, SCREEN_WIDTH } from "@emulator/constants";
import { Identity, ImageFilter } from "@helpers/ImageFilter";

type ScreenProps = {
    width?: number;
    height?: number;
    scale?: number;
    inputRef: MutableRef<VideoReceiver | undefined>;
    Filter?: ImageFilter;
    blending?: boolean;
    id?: string;
    palette?: Partial<Record<number, number>>;
};

export type VideoReceiver = (data: Uint32Array) => void;

function applyPalette(frame: Uint32Array, palette: Partial<Record<number, number>>) {
    for (let index = 0; index < frame.length; index++) {
        frame[index] = palette[frame[index]] ?? frame[index];
    }
}

function mixImages(frame1: Uint32Array, frame2: Uint32Array, target: Uint32Array) {
    for (let index = 0; index < frame1.length; index++) {
        const r1 = (frame1[index] >> 16) & 0xff;
        const g1 = (frame1[index] >> 8) & 0xff;
        const b1 = (frame1[index] >> 0) & 0xff;

        const r2 = (frame2[index] >> 16) & 0xff;
        const g2 = (frame2[index] >> 8) & 0xff;
        const b2 = (frame2[index] >> 0) & 0xff;

        const r = Math.floor((r1 + r2) / 2);
        const g = Math.floor((g1 + g2) / 2);
        const b = Math.floor((b1 + b2) / 2);

        target[index] = (0xff << 24) | (r << 16) | (g << 8) | b;
    }
}

const Screen: FunctionalComponent<ScreenProps> = ({
    inputRef,
    width = SCREEN_WIDTH,
    height = SCREEN_HEIGHT,
    scale = 1,
    Filter = Identity,
    blending = false,
    id = undefined,
    palette = undefined,
}) => {
    const [stateRefresh, setStateRefresh] = useState(0);
    const canvasRef = useRef<HTMLCanvasElement>(null);

    const newFrame = useMemo(() => {
        const canvas = canvasRef.current;
        if (!canvas) {
            setStateRefresh((state) => state + 1);
            return;
        }

        const currentFrame = new Uint32Array(width * height);
        const previousFrame = new Uint32Array(width * height);

        const filterInstance = new Filter(width, height);
```

```
        const dataAsUint8 = new Uint8ClampedArray(filterInstance.output.buffer);
        const imageData = new ImageData(dataAsUint8, ...filterInstance.outputSize);

        const targetWidth = width * scale * window.devicePixelRatio;
        const targetHeight = height * scale * window.devicePixelRatio;

        let firstFrame = true;

        return (data: Uint32Array) => {
            const context = canvas.getContext("2d", { alpha: false });
            if (!context) return;

            context.imageSmoothingEnabled = false;

            previousFrame.set(currentFrame);
            currentFrame.set(data);

            if (palette) {
                // Apply the color palette if needed
                applyPalette(currentFrame, palette);
            }

            if (firstFrame) {
                firstFrame = false;
                // We copy the current frame into the previous one as it is entirely black
                // This avoid a black frame at the beginning
                previousFrame.set(currentFrame);
            }

            if (blending) {
                // We mix both frames into the previous one. This is needed because some games actually
                // flicker entities to display more sprites and have a darker color
                // (example: Link's Awakening chains)
                mixImages(currentFrame, previousFrame, previousFrame);
            } else {
                // We copy the current frame into the previous one to avoid a one frame delay
                previousFrame.set(currentFrame);
            }

            // We apply the filter to the frame
            filterInstance.apply(previousFrame);

            // Actual drawing to the canvas - we scale the image to fit the canvas
            createImageBitmap(imageData).then((bitmap) => {
                context.drawImage(bitmap, 0, 0, targetWidth, targetHeight);
                bitmap.close();
            });
        };
    }, [stateRefresh, canvasRef.current, width, height, scale, Filter, blending, palette]);

    useEffect(() => {
        inputRef.current = newFrame;
        return () => (inputRef.current = undefined);
    }, [inputRef, newFrame]);

    useEffect(() => {
        const canvas = canvasRef.current;
        if (!canvas) return;

        let currentImage: ImageData | null = null;

        const oldContext = canvas.getContext("2d", { alpha: false });
        if (oldContext) {
            currentImage = oldContext.getImageData(0, 0, canvas.width, canvas.height);
        }

        canvas.width = width * scale * window.devicePixelRatio;
        canvas.height = height * scale * window.devicePixelRatio;
```

```
        canvas.style.width = `${width * scale}px`;
        canvas.style.aspectRatio = `${width} / ${height}`;

        if (currentImage) {
            const newContext = canvas.getContext("2d", { alpha: false });
            if (!newContext) return;
            newContext.imageSmoothingEnabled = false;
            // Actual drawing to the canvas - we scale the image to fit the canvas
            createImageBitmap(currentImage).then((bitmap) => {
                newContext.drawImage(bitmap, 0, 0, canvas.width, canvas.height);
                bitmap.close();
            });
        }
    }, [width, height, scale, canvasRef.current]);

    return (
        <div className="screen-container">
            <canvas id={id} ref={canvasRef} />
        </div>
    );
};

export default Screen;
```

B.2.4 Drawer Components

Drawer (`src/frontend/components/Drawer/Drawer.tsx`)

```tsx
import { FunctionalComponent } from "preact";

import { Resizable } from "@components/Resizable";

import DrawerSection from "./DrawerSection";
import ExpressionDrawer from "./ExpressionDrawer";
import KeysDrawer from "./KeysDrawer";
import MemoryDrawer from "./MemoryDrawer";
import SettingsDrawer from "./SettingsDrawer";
import TestDrawer from "./TestDrawer";

import "./Drawer.css";

type DrawerProps = {
    loadRom: (rom: Uint8Array) => void;
};

const Drawer: FunctionalComponent<DrawerProps> = ({ loadRom }) => (
    <Resizable initalWidth={300} id="drawer">
        <DrawerSection title="settings">
            <SettingsDrawer />
        </DrawerSection>
        <DrawerSection title="watch expressions">
            <ExpressionDrawer />
        </DrawerSection>
        <DrawerSection title="test ROMs">
            <TestDrawer loadRom={loadRom} />
        </DrawerSection>
        <DrawerSection title="memory">
            <MemoryDrawer />
        </DrawerSection>
        <DrawerSection title="keybindings">
            <KeysDrawer />
        </DrawerSection>
    </Resizable>
);

export default Drawer;
```

Drawer (src/frontend/components/Drawer/Drawer.css)

```css
/* General drawer */

#drawer {
    position: relative;
    max-height: calc(100% - 2em);
    background-color: #303030;
    padding: 1em;
    overflow-y: auto;
    overflow-x: hidden;
}

.drawer-section {
    border-bottom: solid gray 1px;
    padding: 2px 0;
}

.drawer-title {
    display: flex;
    flex-direction: row;
    margin: 4px 0;
    align-items: center;
}

.drawer-title > h3 {
    font-weight: bolder;
    text-transform: capitalize;
    width: 100%;
    text-align: start;
    margin: 0;
}

.drawer-section > .drawer-content > div {
    background-color: #242424;
    border-radius: 8px;
    padding: 4px;
    margin-bottom: 2px;
}

.drawer-section-title {
    display: flex;
    align-items: center;
    margin: 2px 0;
    flex-wrap: wrap;
}

.drawer-section-title > div:first-child {
    margin-right: auto;
    text-align: start;
    font-weight: bold;
}

.drawer-section-title > button {
    margin-left: 2px;
}

.drawer-section-description {
    font-size: 14px;
    padding: 4px;
}

/* Test Drawer */

.test-drawer label {
    width: 100%;
    display: flex;
    flex-direction: row;
    justify-content: space-between;
```

```
}

.group-label > strong {
    margin-left: 8px;
}

.test-result {
    all: inherit;
    width: 100%;
    margin: -4px 0;
    display: flex;
    flex-direction: row;
    justify-content: space-between;
}

.test-result .test-name {
    font-family: "Fira Code", monospace;
    font-size: 12px;
}

.test-result .test-state {
    font-size: 12px;
    margin-right: 8px;
}

/* Expression drawer */

.exp-watch {
    display: flex;
    flex-direction: column;
    text-align: left;
}

.exp-drawer-control {
    display: flex;
    width: 100%;
    justify-content: flex-end;
}

.exp-drawer-control > button {
    margin-right: 8px;
}

.exp-watch > * {
    font-family: "Fira Code", monospace;
}

.exp-watch > textarea {
    resize: vertical;
}

.exp-watch > div {
    transition: color 0.5s ease;
    margin-bottom: 12px;
}

.exp-watch > div > .label {
    width: 30%;
    color: white;
    margin-right: 8px;
    background: none;
    border: none;
    font-size: inherit;
    font-family: Inter, Avenir, Helvetica, Arial, sans-serif;
}

.exp-watch > div.error {
    color: orangered;
}
```

```
137
138 .exp-watch-output {
139     display: inline-block;
140     white-space: pre-wrap;
141     word-break: break-word;
142 }
143
144 /* Memory drawer */
145
146 .memory-output {
147     font-family: "Fira Code", monospace;
148     font-size: 10px;
149     white-space: pre-wrap;
150     text-align: start;
151     line-height: 110%;
152 }
153
154 /* Keys drawer */
155
156 .keys-drawer {
157     display: flex;
158     flex-direction: row;
159     flex-wrap: wrap;
160     justify-content: space-between;
161 }
162
163 .keys-drawer > div {
164     width: 20%;
165     text-align: center;
166 }
167
168 .keys-drawer > button {
169     width: 25%;
170 }
```

DrawerSection (src/frontend/components/Drawer/DrawerSection.tsx)

```tsx
import { useSignal } from "@preact/signals";
import { ChevronDown, ChevronUp } from "lucide-preact";
import { ComponentChildren, FunctionalComponent } from "preact";
import { useEffect, useRef } from "preact/hooks";

import IconButton from "@components/IconButton";

type DrawerSectionProps = {
    title: string;
    children: ComponentChildren;
};

const localStorageKey = "drawer-section-status";

const DrawerSection: FunctionalComponent<DrawerSectionProps> = ({ title, children }) => {
    const contentRef = useRef<HTMLDivElement>(null);
    const isOpen = useSignal<boolean>(false);

    const id = `drawer-section-${title.toLowerCase().replace(/ /g, "-")}`;

    useEffect(() => {
        isOpen.value = localStorage.getItem(`${localStorageKey}-${title}`) === "1";
    }, []);
    useEffect(() => {
        localStorage.setItem(`${localStorageKey}-${title}`, isOpen.value ? "1" : "0");
    }, [isOpen.value]);

    return (
        <div className="drawer-section">
            <div className="drawer-title">
                <h3>{title}</h3>
                <IconButton
                    id={id}
                    title="Open/Close Tab"
                    onClick={() => (isOpen.value = !isOpen.value)}
                    Icon={isOpen.value ? ChevronUp : ChevronDown}
                />
            </div>
            <div className="drawer-content">
                {isOpen.value && <div ref={contentRef}>{children}</div>}
            </div>
        </div>
    );
};

export default DrawerSection;
```

ExpressionDrawer (src/frontend/components/Drawer/ExpressionDrawer.tsx)

```tsx
import { useSignal } from "@preact/signals";
import { Plus, Trash } from "lucide-preact";
import { FunctionalComponent } from "preact";
import { useEffect, useState } from "preact/hooks";

import IconButton from "@components/IconButton";
import ExpressionWatch from "./ExpressionWatch";

const localStorageKey = "exp-drawer-list";

const ExpressionDrawer: FunctionalComponent = () => {
    const refresh = useState(0)[1];
    const expressionList = useSignal<[string, string][]>([]);
    useEffect(
        () =>
            (expressionList.value = JSON.parse(localStorage.getItem(localStorageKey) ?? "[]")),
        []
    );
    const saveToLocalStorage = () =>
        localStorage.setItem(localStorageKey, JSON.stringify(expressionList.value));
    useEffect(saveToLocalStorage, [expressionList.value]);

    return (
        <div className="exp-drawer">
            {expressionList.value.map(([exp, label], i) => (
                <ExpressionWatch
                    key={i}
                    expression={exp}
                    onChange={(e) => {
                        expressionList.value[i][0] = e;
                        saveToLocalStorage();
                        refresh((r) => r + 1);
                    }}
                    label={label}
                    onLabelChange={(l) => {
                        expressionList.value[i][1] = l;
                        saveToLocalStorage();
                        refresh((r) => r + 1);
                    }}
                />
            ))}
            <div className="drawer-section-title">
                <div>Add/Remove:</div>
                <IconButton
                    title="Add"
                    Icon={Plus}
                    onClick={() => (expressionList.value = [...expressionList.value, ["", ""]])}
                />
                {expressionList.value.length > 0 && (
                    <IconButton
                        title="Delete"
                        Icon={Trash}
                        onClick={() =>
                            (expressionList.value = expressionList.value.slice(0, -1))
                        }
                    />
                )}
            </div>
        </div>
    );
};

export default ExpressionDrawer;
```

ExpressionWatch (src/frontend/components/Drawer/ExpressionWatch.tsx)

```tsx
import { FunctionalComponent } from "preact";
import { useEffect, useState } from "preact/hooks";

type ExpressionWatchProps = {
    expression: string;
    onChange: (expression: string) => void;
    label: string;
    onLabelChange: (label: string) => void;
};

const handleValue = (
    value: Object | { get: () => number } | number | string | null
): string => {
    if (value === null) return "null";
    if (value === undefined) return "undefined";

    if (typeof value === "object" && "get" in value) value = value.get.call(value);

    if (value === null) return "null";
    if (value === undefined) return "undefined";
    if (typeof value === "number") return `0x${value.toString(16).padStart(4, "0")}`;
    return value.toString();
};

const ExpressionWatch: FunctionalComponent<ExpressionWatchProps> = ({
    expression,
    onChange,
    label,
    onLabelChange,
}) => {
    const [func, setFunction] = useState<Function | null>(null);
    const [output, setOutput] = useState<string | null>(null);

    useEffect(() => {
        try {
            setFunction(() => new Function(`return ${expression}`));
        } catch {
            setFunction(null);
        }
    }, [expression]);

    useEffect(() => {
        const callbackId = setInterval(() => {
            let value: typeof output;
            try {
                if (func === null) value = null;
                else value = handleValue(func());
            } catch {
                value = null;
            }
            setOutput(value);
        }, 100);
        return () => clearInterval(callbackId);
    }, [func]);

    return (
        <div className="exp-watch">
            <textarea
                placeholder="gbc.cpu.regAF"
                spellCheck={false}
                value={expression}
                onInput={(e) => onChange(e.currentTarget.value)}
            />
            <div className={output === null ? "error" : undefined}>
                <input
                    placeholder="label"
                    type="text"
```

```
                    className="label"
                    spellcheck={false}
                    value={label}
                    onInput={(e) => onLabelChange(e.currentTarget.value)}
                />
                <span className="exp-watch-output">{output === null ? "Error" : output}</span>
            </div>
        </div>
    );
};

export default ExpressionWatch;
```

Grid2x (src/frontend/components/Drawer/Grid2x.tsx)

```tsx
import { LucideProps } from "lucide-preact";
import { ComponentType, h, toChildArray } from "preact";
import { JSX } from "preact/jsx-runtime";

const defaultAttributes = {
    xmlns: "http://www.w3.org/2000/svg",
    width: 24,
    height: 24,
    viewBox: "0 0 24 24",
    fill: "none",
    stroke: "currentColor",
    "stroke-width": 2,
    "stroke-linecap": "round",
    "stroke-linejoin": "round",
};

type IconNode = [elementName: keyof JSX.IntrinsicElements, attrs: Record<string, string>][];

export const toKebabCase = (string: string) =>
    string.replace(/([a-z0-9])([A-Z])/g, "$1-$2").toLowerCase();

/**
 * Taken from Preact library directly:
 * @link
 ↪  https://github.com/lucide-icons/lucide/blob/main/packages/lucide-preact/src/createPreactComponent.ts
 */
const createPreactComponent = (
    iconName: string,
    iconNode: IconNode
): ((props: LucideProps) => JSX.Element) => {
    const Component = ({
        color = "currentColor",
        size = 24,
        strokeWidth = 2,
        children,
        ref, // ignore ref
        ...rest
    }: LucideProps) =>
        h(
            "svg" as unknown as ComponentType<
                Partial<JSX.SVGAttributes<SVGElement> & { "stroke-width": number | string }>
            >,
            {
                ...defaultAttributes,
                width: String(size),
                height: size,
                stroke: color,
                "stroke-width": strokeWidth,
                class: `lucide lucide-${toKebabCase(iconName)}`,
                ...rest,
            },
            [...iconNode.map(([tag, attrs]) => h(tag, attrs)), ...toChildArray(children)]
        );

    Component.displayName = `${iconName}`;

    return Component;
};
const Grid2x = createPreactComponent("Grid2x", [
    [
        "rect",
        {
            x: "3",
            y: "3",
            width: "18",
            height: "18",
            rx: "2",
```

```
            ry: "2",
            key: "maln0c",
        },
    ],
    [
        "line",
        {
            x1: "3",
            y1: "12",
            x2: "21",
            y2: "12",
            key: "1uch6j",
        },
    ],
    [
        "line",
        {
            x1: "12",
            y1: "3",
            x2: "12",
            y2: "21",
            key: "nvcl17",
        },
    ],
]);

export default Grid2x;
```

KeysDrawer (src/frontend/components/Drawer/KeysDrawer.tsx)

```tsx
import { FunctionalComponent } from "preact";

import KeybindingInput from "@components/KeybindingInput";
import { useConfig } from "@helpers/ConfigContext";

const KeysDrawer: FunctionalComponent = () => {
    const [
        {
            controlArrowUp,
            controlArrowDown,
            controlArrowLeft,
            controlArrowRight,
            controlA,
            controlB,
            controlStart,
            controlSelect,
        },
        setConfig,
    ] = useConfig();
    return (
        <div>
            <div className="drawer-section-description">
                Click to edit, press enter to save, press escape to cancel.
            </div>
            <div className="keys-drawer">
                <div>Up</div>
                <KeybindingInput
                    value={controlArrowUp}
                    onChange={(v) => setConfig({ controlArrowUp: v })}
                />
                <div>A</div>
                <KeybindingInput
                    value={controlA}
                    onChange={(v) => setConfig({ controlA: v })}
                />
                <div>Down</div>
                <KeybindingInput
                    value={controlArrowDown}
                    onChange={(v) => setConfig({ controlArrowDown: v })}
                />
                <div>B</div>
                <KeybindingInput
                    value={controlB}
                    onChange={(v) => setConfig({ controlB: v })}
                />
                <div>Left</div>
                <KeybindingInput
                    value={controlArrowLeft}
                    onChange={(v) => setConfig({ controlArrowLeft: v })}
                />
                <div>Start</div>
                <KeybindingInput
                    value={controlStart}
                    onChange={(v) => setConfig({ controlStart: v })}
                />
                <div>Right</div>
                <KeybindingInput
                    value={controlArrowRight}
                    onChange={(v) => setConfig({ controlArrowRight: v })}
                />

                <div>Select</div>
                <KeybindingInput
                    value={controlSelect}
                    onChange={(v) => setConfig({ controlSelect: v })}
                />
            </div>
```

```
68          </div>
69      );
70  };
71
72  export default KeysDrawer;
```

MemoryDrawer (src/frontend/components/Drawer/MemoryDrawer.tsx)

```tsx
import { useSignal } from "@preact/signals";
import { FunctionalComponent } from "preact";
import { useEffect } from "preact/hooks";

import GameBoyColor from "@emulator/GameBoyColor";

const refreshMemory = (offset: number) => {
    // @ts-ignore
    const gbc: GameBoyColor = window.gbc;
    if (!gbc) return "Something went wrong";

    let memory = "";
    for (let address = offset; address < 0x10000; address += 16) {
        memory += address.toString(16).padStart(4, "0") + ": ";
        for (let i = 0; i < 16; i++) {
            memory +=
                gbc["system"]
                    .read(address + i)
                    .toString(16)
                    .padStart(2, "0") + "\u2009"; // thin space character
        }
        memory += "\n";
    }
    return memory;
};

const MEMORY_DRAWER_LOCAL_STORAGE_KEY = "memory-drawer-offset";

const MemoryDrawer: FunctionalComponent = () => {
    const memory = useSignal<string>("");
    const offset = useSignal<number>(0);

    useEffect(() => {
        const value = localStorage.getItem(MEMORY_DRAWER_LOCAL_STORAGE_KEY);
        if (value) offset.value = +value;
        memory.value = refreshMemory(offset.value);
    }, []);

    useEffect(() => {
        const callbackId = setInterval(() => {
            memory.value = refreshMemory(offset.value);
        }, 500);
        return () => clearInterval(callbackId);
    }, [offset]);

    return (
        <div>
            <div className="drawer-section-title">
                <div>Offset:</div>
                <input
                    type="text"
                    value={offset.value.toString(16).padStart(4, "0")}
                    onChange={(e) => {
                        offset.value = Number(`0x${e.currentTarget.value}`);
                        localStorage.setItem(
                            MEMORY_DRAWER_LOCAL_STORAGE_KEY,
                            offset.value.toString()
                        );
                    }}
                />
            </div>
            <div className="memory-output">{memory}</div>
        </div>
    );
};

export default MemoryDrawer;
```

SettingsDrawer (src/frontend/components/Drawer/SettingsDrawer.tsx)

```tsx
import {
    Bug,
    Circle,
    Dice1,
    Dice2,
    Dice4,
    FileDigit,
    FileX2,
    Flame,
    Flower,
    Gamepad,
    Grid,
    Image,
    ImageOff,
    LineChart,
    Palette,
    Square,
    Waves,
} from "lucide-preact";
import { FunctionalComponent } from "preact";

import IconButton from "@components/IconButton";
import { useConfig } from "@helpers/ConfigContext";
import { Identity, Scale2x, Scale4x } from "@helpers/ImageFilter";

import Grid2x from "./Grid2x";

const availableFilters = [
    {
        name: "identity",
        filter: Identity,
        icon: Square,
    },
    {
        name: "scale2x",
        filter: Scale2x,
        icon: Grid2x,
    },
    {
        name: "scale4x",
        filter: Scale4x,
        icon: Grid,
    },
];

type DMGPalette = Partial<Record<number, number>> | undefined;

const palette = (white: number, lightGray: number, darkGray: number, black: number) => ({
    0xffffffff: white | 0xff000000,
    0xffaaaaaa: lightGray | 0xff000000,
    0xff555555: darkGray | 0xff000000,
    0xff000000: black | 0xff000000,
});

const isPaletteEquivalent = (palette1: DMGPalette, palette2: DMGPalette) => {
    if (!palette1 && !palette2) return true; // both undefined
    if (!palette1 || !palette2) return false; // only one undefined
    if (Object.keys(palette1).length !== Object.keys(palette2).length) return false;
    for (const key of Object.keys(palette1)) {
        // @ts-ignore
        if (palette1[key] !== palette2[key]) return false;
    }
    return true;
};

function loadFileWithInput(id: string, callback: (data: Uint8Array) => void): void {
    const input = document.getElementById(id) as HTMLInputElement;
```

```
    input.click();
    input.onchange = () => {
        if (input.files) {
            const file = input.files[0];
            const reader = new FileReader();
            reader.onload = () => {
                console.log("reader", reader);
                if (reader.result) {
                    const buffer = reader.result as ArrayBuffer;
                    callback(new Uint8Array(buffer));
                }
            };
            reader.readAsArrayBuffer(file);
        }
    };
}

const availablePalettes = [
    {
        name: "monochrome",
        values: undefined, // no transform
        icon: Circle,
    },
    {
        name: "classic",
        values: palette(0x95ddca, 0x6aa48b, 0x3d6042, 0x11180c),
        icon: Gamepad,
    },
    {
        name: "ocean",
        values: palette(0xace2b9, 0x8a9965, 0x67582c, 0x35250c),
        icon: Waves,
    },
    {
        name: "magma",
        values: palette(0x9ed4e5, 0x645ab0, 0x451f7b, 0x3c0112),
        icon: Flame,
    },
    {
        name: "sakura",
        values: palette(0xe1dee9, 0x9377cd, 0x623cb5, 0x2b1449),
        icon: Flower,
    },
];

const SettingsDrawer: FunctionalComponent = () => {
    const [
        {
            filter: currentFilter,
            frameBlending,
            scale,
            bootRom,
            console,
            gbPalette,
            volume,
            showStats,
            showDebugScreens,
            bootRomDmg,
            bootRomCgb,
        },
        setConfig,
    ] = useConfig();

    return (
        <>
            <div className="drawer-section-title">
                <div>Console:</div>
                <IconButton
                    id="dmg-mode"
```

```
                    title="Classic (DMG)"
                    showTooltip
                    toggled={console === "dmg"}
                    Icon={Gamepad}
                    onClick={() => setConfig({ console: "dmg" })}
                />
                <IconButton
                    id="cgb-mode"
                    title="Color (CGB)"
                    showTooltip
                    toggled={console === "cgb"}
                    Icon={Palette}
                    onClick={() => setConfig({ console: "cgb" })}
                />
            </div>

            <div className="drawer-section-title">
                <div>Filter:</div>
                {availableFilters.map(({ name, filter, icon }) => (
                    <IconButton
                        title={name}
                        showTooltip
                        toggled={filter === currentFilter}
                        Icon={icon}
                        onClick={() => setConfig({ filter })}
                    />
                ))}
            </div>

            <div className="drawer-section-title">
                <div>Scale:</div>
                <IconButton
                    title="100%"
                    showTooltip
                    toggled={scale === 0}
                    Icon={Dice1}
                    onClick={() => setConfig({ scale: 0 })}
                />
                <IconButton
                    title="200%"
                    showTooltip
                    toggled={scale === 1}
                    Icon={Dice2}
                    onClick={() => setConfig({ scale: 1 })}
                />
                <IconButton
                    title="400%"
                    showTooltip
                    toggled={scale === 2}
                    Icon={Dice4}
                    onClick={() => setConfig({ scale: 2 })}
                />
            </div>

            <div className="drawer-section-title">
                <div>Volume:</div>

                <input
                    type="range"
                    min="0"
                    max="2"
                    step="0.02"
                    value={volume}
                    onChange={(e) => setConfig({ volume: +e.currentTarget.value })}
                />
            </div>

            <div className="drawer-section-title">
                <div>GB Palette:</div>
```

```
206
207                 {availablePalettes.map(({ name, icon, values }) => (
208                     <IconButton
209                         title={name}
210                         Icon={icon}
211                         onClick={() => setConfig({ gbPalette: values })}
212                         toggled={isPaletteEquivalent(gbPalette, values)}
213                         showTooltip
214                     />
215                 ))}
216             </div>
217
218             <div className="drawer-section-title">
219                 <div>Boot ROM Upload:</div>
220
221                 <input type="file" id="upload-boot-rom" style="display: none" />
222                 <IconButton
223                     title="DMG"
224                     Icon={Gamepad}
225                     onClick={() =>
226                         loadFileWithInput("upload-boot-rom", (bootRomDmg) => {
227                             if (bootRomDmg.length !== 0x0100)
228                                 alert(`Boot ROM of the DMG must be ${0x100} bytes long!`);
229                             else setConfig({ bootRomDmg });
230                         })
231                     }
232                     toggled={bootRomDmg !== null}
233                     showTooltip
234                 />
235                 <IconButton
236                     title="CGB"
237                     Icon={Palette}
238                     onClick={() =>
239                         loadFileWithInput("upload-boot-rom", (bootRomCgb) => {
240                             window.console.log(bootRomCgb);
241                             if (bootRomCgb.length !== 0x0900)
242                                 alert(`Boot ROM of the DMG must be ${0x900} bytes long!`);
243                             else setConfig({ bootRomCgb });
244                         })
245                     }
246                     toggled={bootRomCgb !== null}
247                     showTooltip
248                 />
249             </div>
250
251             <div className="drawer-section-title">
252                 <div>Other:</div>
253                 <IconButton
254                     title="Boot ROM"
255                     Icon={bootRom === "real" ? FileDigit : FileX2}
256                     onClick={() => setConfig({ bootRom: bootRom === "none" ? "real" : "none" })}
257                     toggled={bootRom === "real"}
258                     showTooltip
259                 />
260
261                 <IconButton
262                     title="Toggle blending"
263                     Icon={frameBlending ? Image : ImageOff}
264                     onClick={() => setConfig({ frameBlending: !frameBlending })}
265                     toggled={frameBlending}
266                     showTooltip
267                 />
268
269                 <IconButton
270                     title="Show Stats"
271                     Icon={LineChart}
272                     onClick={() => setConfig({ showStats: !showStats })}
273                     toggled={showStats}
274                     showTooltip
```

```
                />

                <IconButton
                    title="Debug"
                    Icon={Bug}
                    onClick={() => setConfig({ showDebugScreens: !showDebugScreens })}
                    toggled={showDebugScreens}
                    showTooltip
                />
            </div>
        </>
    );
};

export default SettingsDrawer;
```

TestDrawer (src/frontend/components/Drawer/TestDrawer.tsx)

```tsx
import { useSignal } from "@preact/signals";
import { BoxSelect, FileQuestion } from "lucide-preact";
import { Fragment, FunctionalComponent } from "preact";
import { useEffect } from "preact/hooks";

import GameBoyColor from "@emulator/GameBoyColor";
import GameBoyInput from "@emulator/GameBoyInput";
import GameBoyOutput from "@emulator/GameBoyOutput";

import IconButton from "@components/IconButton";
import tests, { Test } from "@frontend/testConfig";

const TEST_PASS = "\u2705"; // tick emoji
const TEST_FAIL = "\u274C"; // cross emoji
const TEST_TIMEOUT = "\u231B"; // hourglass emoji
const TEST_CRASH = "\ud83e\udea6"; // skull emoji

type TestOutput = typeof TEST_PASS | typeof TEST_FAIL | typeof TEST_TIMEOUT | typeof TEST_CRASH;

const makeGameboy = (
    type: "DMG" | "CGB",
    rom: Uint8Array,
    videoOut: (d: Uint32Array) => void,
    serialOut: (s: string) => void
) => {
    const gameIn: GameBoyInput = {
        read: () => ({
            up: false,
            down: false,
            left: false,
            right: false,
            a: false,
            b: false,
            start: false,
            select: false,
        }),
    };

    const gbOut: GameBoyOutput = {
        receiveGraphics: videoOut,
        serialOut: (d) => serialOut(String.fromCharCode(d)),
    };

    return new GameBoyColor(type, rom, gameIn, gbOut);
};

type TestResult = [Test, TestOutput][];

const loadTestRom = async (testType: string, fileName: string) => {
    const romResponse = await fetch(`/tests/${testType}/${fileName}.gb`);
    const romBlob = await romResponse.blob();
    return new Uint8Array(await romBlob.arrayBuffer());
};

const runTests = async (validGroups: string[] = [], results: (r: TestResult) => void) => {
    const localResults: TestResult = [];

    for (const test of tests) {
        const { testType, subTestType, file, consoleType, check } = test;
        if (!validGroups.includes(`${testType}/${subTestType}`)) continue;

        console.log(`Running test ${testType}/${subTestType} -> ${file}`);

        const romArray = await loadTestRom(testType, file);

        let videoOut: Uint32Array = new Uint32Array(160 * 144);
        let serialOut: string = "";
```

```
        let state: TestOutput;

        try {
            const gbc = makeGameboy(
                consoleType,
                romArray,
                (v) => videoOut.set(v),
                (s) => (serialOut += s)
            );

            let prevSteps = 0;
            while (true) {
                gbc.drawFrame();

                const newState = await check(gbc, serialOut, videoOut, file);
                if (newState !== null) {
                    state = newState === "failure" ? TEST_FAIL : TEST_PASS;
                    break;
                }

                const steps = gbc["cpu"]["stepCounter"];
                if (steps > 10_000_000 || steps === prevSteps) {
                    state = TEST_TIMEOUT;
                    break;
                }
                prevSteps = steps;
            }
        } catch (e) {
            console.error("Caught error, skipping test", e);
            state = TEST_CRASH;
        }

        localResults.push([test, state]);
        results(localResults);
    }

    const passedTests = localResults.filter((t) => t[1] === TEST_PASS).length;
    const totalTests = localResults.length;
    console.log(`Finished running tests! Passed ${passedTests}/${totalTests} tests.`);
};

const testGroups = tests
    .map((t) => `${t.testType}/${t.subTestType}`)
    .filter((v, i, a) => a.indexOf(v) === i); // Unique

const localStorageKey = "test-drawer-groups";

type TestDrawerProps = {
    loadRom: (rom: Uint8Array) => void;
};

const TestDrawer: FunctionalComponent<TestDrawerProps> = ({ loadRom }) => {
    const testsRunning = useSignal<boolean>(false);
    const testResults = useSignal<TestResult>([]);
    const keptTests = useSignal<string[]>(testGroups);

    // Loading
    useEffect(
        () => (keptTests.value = JSON.parse(localStorage.getItem(localStorageKey) ?? "[]")),
        []
    );
    // Saving
    useEffect(
        () => localStorage.setItem(localStorageKey, JSON.stringify(keptTests.value)),
        [keptTests.value]
    );

    return (
        <div className="test-drawer">
```

```
            <div className="drawer-section-title">
                <div>Settings:</div>
                <IconButton
                    title="Test"
                    showTooltip
                    Icon={FileQuestion}
                    disabled={testsRunning.value}
                    onClick={() => {
                        testResults.value = [];
                        testsRunning.value = true;
                        runTests(keptTests.value, (r) => (testResults.value = [...r])).then(
                            () => (testsRunning.value = false)
                        );
                    }}
                />
                <IconButton
                    title="Select/Unselect All"
                    showTooltip
                    Icon={BoxSelect}
                    disabled={testsRunning.value}
                    onClick={() =>
                        (keptTests.value = keptTests.value.length === 0 ? testGroups : [])
                    }
                />
            </div>
            <div
                style={{
                    display: "flex",
                    flexDirection: "column",
                    alignItems: "flex-start",
                }}
            >
                {testGroups.map((group) => {
                    const selected = keptTests.value.includes(group);
                    const matchingTests = tests
                        .filter((t) => `${t.testType}/${t.subTestType}` === group)
                        .map(
                            (t) => [t, testResults.value.find((r) => r[0] === t)?.[1]] as const
                        );

                    const passedTests = matchingTests.filter((v) => v[1] === TEST_PASS).length;
                    const totalTests = matchingTests.length;

                    return (
                        <Fragment key={group}>
                            <label>
                                <span className="group-label">
                                    {group}
                                    {selected && matchingTests.length > 0 && (
                                        <strong>
                                            {passedTests}/{totalTests}
                                        </strong>
                                    )}
                                </span>
                                <input
                                    type="checkbox"
                                    checked={selected}
                                    disabled={testsRunning.value}
                                    onChange={(e) =>
                                        (keptTests.value = e.currentTarget.checked
                                            ? [...keptTests.value, group]
                                            : keptTests.value.filter((v) => v !== group))
                                    }
                                />
                            </label>
                            {selected &&
                                matchingTests.map(([test, state]) => (
                                    <button
                                        key={test.file}
```

```
                                        className="test-result"
                                        onClick={() =>
                                            loadTestRom(test.testType, test.file).then(loadRom)
                                        }
                                    >
                                        <span className="test-name">{test.file}</span>
                                        <span className="test-state">{state}</span>
                                    </button>
                                ))}
                            </Fragment>
                        );
                    })}
                </div>
            </div>
        );
};

export default TestDrawer;
```

B.2.5 Helpers

AudioPlayer (src/frontend/helpers/AudioPlayer.ts)

```ts
/**
 * A non-gameboy related class, that handles playing the received sound data.
 * Largely inspired by:
 * @link https://github.com/denislins/gameboy/blob/master/emulator/apu/Player.js
 */
class AudioPlayer {
    protected volume: { value: number };
    protected context: AudioContext | undefined;
    protected lastPlayEnd: number | undefined;
    protected enqued: number = 0;

    protected maxQueueSize: number;

    protected windowBlurListener = () => this.context?.suspend();
    protected windowFocusListener = () => this.context?.resume();

    constructor(volume: { value: number } = { value: 1 }, maxQueueSize = 8) {
        this.maxQueueSize = maxQueueSize;
        this.volume = volume;
        this.context = new AudioContext();
        this.context.resume();
        window.addEventListener("blur", this.windowBlurListener, false);
        window.addEventListener("focus", this.windowFocusListener, false);
    }

    delete() {
        this.context?.close();
        delete this.context;
        window.removeEventListener("blur", this.windowBlurListener, false);
        window.removeEventListener("focus", this.windowFocusListener, false);
    }

    enqueue(sample: Float32Array) {
        // Not allowed to have more than 8 samples in the queue, to avoid delay
        if (this.enqued > this.maxQueueSize || !this.context) {
            return;
        }

        this.enqued++;

        const sampleDuration = sample.length / 44100;
        const startTime =
            this.lastPlayEnd && this.lastPlayEnd >= this.context.currentTime
                ? this.lastPlayEnd + sampleDuration
                : this.context.currentTime + sampleDuration; // add a delay to start
        this.lastPlayEnd = startTime;

        // Create the buffer
        const buffer = this.context.createBuffer(1, sample.length, 44100);
        const bufferContent = buffer.getChannelData(0);
        for (let i = 0; i < sample.length; i++)
            bufferContent[i] = sample[i] * this.volume.value;

        const source = this.context.createBufferSource();
        source.buffer = buffer;
        source.onended = () => this.enqued--;

        source.connect(this.context.destination);
        source.start(startTime);
    }
}

export default AudioPlayer;
```

ConfigContext (src/frontend/helpers/ConfigContext.tsx)

```tsx
import { ComponentChildren, createContext, FunctionalComponent } from "preact";
import { useCallback, useContext, useState } from "preact/hooks";

import { filterByName, Identity, ImageFilter } from "@helpers/ImageFilter";

export type Configuration = {
    scale: 0 | 1 | 2;
    filter: ImageFilter;
    audioEnabled: boolean;
    frameBlending: boolean;
    bootRom: "none" | "real";
    console: "dmg" | "cgb";
    gbPalette: undefined | Partial<Record<number, number>>;
    volume: number;
    showStats: boolean;
    showDebugScreens: boolean;

    bootRomDmg: Uint8Array | null;
    bootRomCgb: Uint8Array | null;

    controlArrowUp: string;
    controlArrowDown: string;
    controlArrowLeft: string;
    controlArrowRight: string;
    controlA: string;
    controlB: string;
    controlStart: string;
    controlSelect: string;
};

type ConfigLoader<T> = {
    to: (v: T) => string;
    from: (v: string) => T;
};

const IdentitySave: ConfigLoader<any> = { to: (v) => v, from: (v) => v };

/** Not very memory efficient, but the sizes of the boot ROMs are small enough that it's
↪  acceptable */
const Uint8ArrayStringSave: ConfigLoader<Uint8Array | null> = {
    to: (v) =>
        v === null
            ? "0"
            : Array.from(v)
                  .map((x) => x.toString(16).padStart(2, "0"))
                  .join(""),
    from: (v) =>
        v === "0"
            ? null
            : new Uint8Array(v.length / 2).map((_, i) =>
                  Number.parseInt(v.substring(i * 2, i * 2 + 2), 16)
              ),
};

const configLoaders: {
    [k in keyof Configuration]: null | ConfigLoader<Configuration[k]>;
} = {
    scale: IdentitySave,
    filter: {
        to: (v: ImageFilter) => v.name,
        from: (v: string) => filterByName(v) ?? Identity,
    },
    audioEnabled: null,
    frameBlending: IdentitySave,
    bootRom: IdentitySave,
    console: IdentitySave,
    gbPalette: IdentitySave,
```

```
    volume: IdentitySave,
    showStats: IdentitySave,
    showDebugScreens: IdentitySave,

    bootRomDmg: Uint8ArrayStringSave,
    bootRomCgb: Uint8ArrayStringSave,

    controlArrowUp: IdentitySave,
    controlArrowDown: IdentitySave,
    controlArrowLeft: IdentitySave,
    controlArrowRight: IdentitySave,
    controlA: IdentitySave,
    controlB: IdentitySave,
    controlStart: IdentitySave,
    controlSelect: IdentitySave,
};

const defaultConfig: Configuration = {
    scale: 1,
    filter: Identity,
    audioEnabled: false,
    frameBlending: true,
    bootRom: "none",
    console: "dmg",
    gbPalette: undefined,
    volume: 0.5,
    showStats: false,
    showDebugScreens: false,

    bootRomDmg: null,
    bootRomCgb: null,

    controlArrowUp: "ArrowUp",
    controlArrowDown: "ArrowDown",
    controlArrowLeft: "ArrowLeft",
    controlArrowRight: "ArrowRight",
    controlA: "z",
    controlB: "x",
    controlStart: "Enter",
    controlSelect: "Backspace",
};

const configToString = (config: Configuration): string =>
    JSON.stringify(
        Object.fromEntries(
            Object.entries(config) // @ts-ignore
                .filter(([k, v]) => configLoaders[k] !== null) // @ts-ignore
                .map(([k, v]) => [k, configLoaders[k].to(v)])
        )
    );

const configFromString = (configString: string): Configuration => {
    const rawConfig = JSON.parse(configString);
    // Create a partial config with only objects that are defined and part of the default config
    const loadedConfig: Partial<Configuration> = Object.fromEntries(
        Object.entries(rawConfig)
            .filter(([k, v]) => k in configLoaders && v !== undefined) // @ts-ignore
            .map(([k, v]) => [k, configLoaders[k].from(v)])
    );
    return { ...defaultConfig, ...loadedConfig };
};

const ConfigContext = createContext<
    [Configuration, (newConfig: Partial<Configuration>) => void]
>([defaultConfig, () => {}]);

export const useConfig = () => useContext(ConfigContext);

const localStorageKey = "config";
```

```
136
137 export const ConfigProvider: FunctionalComponent<ComponentChildren> = ({ children }) => {
138     const [config, setConfig] = useState<Configuration>(() => {
139         const savedConfig = localStorage.getItem(localStorageKey);
140         if (savedConfig) {
141             const config = configFromString(savedConfig);
142             return { ...defaultConfig, ...config };
143         }
144         return defaultConfig;
145     });
146     const configUpdater = useCallback(
147         (newConfig: Partial<Configuration>) => {
148             let fullNewConfig: Configuration;
149             setConfig((c) => (fullNewConfig = { ...c, ...newConfig }));
150             localStorage.setItem(localStorageKey, configToString(fullNewConfig!));
151         },
152         [setConfig]
153     );
154
155     return (
156         <ConfigContext.Provider value={[config, configUpdater]}>
157             {children}
158         </ConfigContext.Provider>
159     );
160 };
```

useKeys (src/frontend/helpers/useKeys.ts)

```ts
import { useEffect, useRef } from "preact/hooks";

const useKeys = (codes: string[] = []) => {
    const pressedKeys = useRef<string[]>([]);

    useEffect(() => {
        const codesLower = codes.map((code) => code.toLowerCase());

        const keyDownListener = (e: KeyboardEvent) => {
            if (!codesLower.includes(e.key.toLowerCase())) return;

            var target = (e.target || e.srcElement) as HTMLElement;
            var targetTagName = target === null ? "null" : target.nodeName.toUpperCase();
            if (/INPUT|SELECT|TEXTAREA/.test(targetTagName)) {
                return;
            }

            e.preventDefault();
            const index = pressedKeys.current.indexOf(e.key);
            if (index === -1) {
                pressedKeys.current.push(e.key);
            }
        };
        const keyUpListener = (e: KeyboardEvent) => {
            if (!codesLower.includes(e.key.toLowerCase())) return;

            const index = pressedKeys.current.indexOf(e.key);
            if (index !== -1) {
                pressedKeys.current.splice(index, 1);
            }
        };
        document.addEventListener("keydown", keyDownListener);
        document.addEventListener("keyup", keyUpListener);

        return () => {
            document.removeEventListener("keydown", keyDownListener);
            document.removeEventListener("keyup", keyUpListener);
        };
    }, [codes]);

    return pressedKeys.current;
};

export default useKeys;
```

B.2.6 Image Filters

index (src/frontend/helpers/ImageFilter/index.ts)

```ts
import ImageFilterConcrete from "./Base";
import Identity from "./Identity";
import Scale2x from "./Scale2x";
import Scale4x from "./Scale4x";

export type ImageFilter = typeof ImageFilterConcrete;
export const filterByName = (name: string): ImageFilter | undefined =>
    [Identity, Scale2x, Scale4x].find((f) => f.name === name);
export { Identity, Scale2x, Scale4x };
```

Base (`src/frontend/helpers/ImageFilter/Base.ts`)

```ts
class ImageFilter {
    protected readonly width: number;
    protected readonly height: number;

    readonly output: Uint32Array;

    constructor(width: number, height: number) {
        this.width = width;
        this.height = height;

        const outSize = this.outputSize;
        this.output = new Uint32Array(outSize[0] * outSize[1]);
    }

    get outputSize(): [number, number] {
        throw new Error("Not implemented");
    }

    apply(image: Uint32Array): void {
        throw new Error("Not implemented");
    }
}

export default ImageFilter;
```

Identity (src/frontend/helpers/ImageFilter/Identity.ts)

```ts
import ImageFilter from "./Base";

/**
 * An identity filter that doesn't alted the image.
 */
class Identity extends ImageFilter {
    override get outputSize(): [number, number] {
        return [this.width, this.height];
    }
    override apply(image: Uint32Array): void {
        this.output.set(image);
    }
}

export default Identity;
```

Scale2x (src/frontend/helpers/ImageFilter/Scale2x.ts)

```ts
import ImageFilter from "./Base";

/**
 * Filter that uses the Scale2x algorithm
 * @link https://www.scale2x.it/algorithm
 */
class Scale2x extends ImageFilter {
    override get outputSize(): [number, number] {
        return [this.width * 2, this.height * 2];
    }

    override apply(image: Uint32Array): void {
        for (let y = 0; y < this.height; y++) {
            let value = image[y * this.width];
            let w = value;
            let e = image[y * this.width + 1];
            for (let x = 0; x < this.width; x++) {
                const n = y - 1 < 0 ? value : image[x + (y - 1) * this.width];
                const s = y + 1 >= this.height ? value : image[x + (y + 1) * this.width];
                if (n !== s && w !== e) {
                    this.output[x * 2 + y * 2 * this.width * 2] = w === n ? w : value;
                    this.output[x * 2 + 1 + y * 2 * this.width * 2] = n === e ? e : value;
                    this.output[x * 2 + (y * 2 + 1) * this.width * 2] = w === s ? w : value;
                    this.output[x * 2 + 1 + (y * 2 + 1) * this.width * 2] = s === e ? e : value;
                } else {
                    this.output[x * 2 + y * 2 * this.width * 2] = value;
                    this.output[x * 2 + 1 + y * 2 * this.width * 2] = value;
                    this.output[x * 2 + (y * 2 + 1) * this.width * 2] = value;
                    this.output[x * 2 + 1 + (y * 2 + 1) * this.width * 2] = value;
                }

                w = value;
                value = e;
                e = x + 2 >= this.width ? value : image[x + 2 + y * this.width];
            }
        }
    }
}

export default Scale2x;
```

Scale4x (src/frontend/helpers/ImageFilter/Scale4x.ts)

```typescript
import Scale2x from "./Scale2x";

class Scale4x extends Scale2x {
    protected subFilter: Scale2x;

    constructor(width: number, height: number) {
        super(width * 2, height * 2);
        this.subFilter = new Scale2x(width, height);
    }

    override apply(image: Uint32Array): void {
        this.subFilter.apply(image);
        super.apply(this.subFilter.output);
    }
}

export default Scale4x;
```

B.2.7 Automated Test Config

testConfig (src/frontend/testConfig.ts)

```ts
import GameBoyColor from "@emulator/GameBoyColor";

type MiniTest = {
    file: string;
    consoleType: "DMG" | "CGB";
};

const mkTests: (consoleType: "DMG" | "CGB", ...names: string[]) => MiniTest[] = (
    consoleType,
    ...names
) =>
    names.map((n) => ({
        file: n,
        consoleType,
    }));

const dmgTests: (...names: string[]) => MiniTest[] = (...names) => mkTests("DMG", ...names);
const cgbTests: (...names: string[]) => MiniTest[] = (...names) => mkTests("CGB", ...names);

/** The list of files to run for tests, per "test group" */
const rawTestFiles = {
    blaarg: {
        cpu: dmgTests(
            "cpu-01-special",
            "cpu-02-interrupts",
            "cpu-03-op sp,hl",
            "cpu-04-op r,imm",
            "cpu-05-op rp",
            "cpu-06-ld r,r",
            "cpu-07-jr,jp,call,ret,rst",
            "cpu-08-misc instrs",
            "cpu-09-op r,r",
            "cpu-10-bit ops",
            "cpu-11-op a,(hl)",
            "instr_timing",
            "mem_timing"
        ),
        apu: dmgTests(
            "apu-01-registers",
            "apu-02-len ctr",
            "apu-03-trigger",
            "apu-04-sweep",
            "apu-05-sweep details",
            "apu-06-overflow on trigger",
            "apu-07-len sweep period sync",
            "apu-08-len ctr during power",
            "apu-09-wave read while on",
            "apu-10-wave trigger while on",
            "apu-11-regs after power",
            "apu-12-wave write while on"
        ),
        other: dmgTests("halt_bug", "oam_bug"),
    },
    mooneye: {
        itrAndCpu: dmgTests(
            "daa",
            "ei_sequence",
            "halt_ime0_ei",
            "ie_push",
            "if_ie_registers",
            "rapid_di_ei",
            "reg_f",
            "unused_hwio-GS"
        ),
```

```
        ppu: dmgTests(
            "ppu_hblank_ly_scx_timing-GS",
            "ppu_intr_1_2_timing-GS",
            "ppu_intr_2_0_timing",
            "ppu_intr_2_mode0_timing",
            "ppu_intr_2_mode0_timing_sprites",
            "ppu_intr_2_mode3_timing",
            "ppu_intr_2_oam_ok_timing",
            "ppu_lcdon_timing-GS",
            "ppu_lcdon_write_timing-GS",
            "ppu_stat_irq_blocking",
            "ppu_stat_lyc_onoff",
            "ppu_vblank_stat_intr-GS"
        ),
        cpuTiming: dmgTests(
            "add_sp_e_timing",
            "call_cc_timing",
            "call_cc_timing2",
            "call_timing",
            "call_timing2",
            "di_timing-GS",
            "div_timing",
            "ei_timing",
            "halt_ime0_nointr_timing",
            "halt_ime1_timing",
            "halt_ime1_timing2-GS",
            "intr_timing",
            "jp_cc_timing",
            "jp_timing",
            "ld_hl_sp_e_timing",
            "pop_timing",
            "push_timing",
            "ret_cc_timing",
            "reti_intr_timing",
            "reti_timing",
            "rst_timing"
        ),
        timer: dmgTests(
            "timer_div_write",
            "timer_rapid_toggle",
            "timer_tim00",
            "timer_tim00_div_trigger",
            "timer_tim01",
            "timer_tim01_div_trigger",
            "timer_tim10",
            "timer_tim10_div_trigger",
            "timer_tim11",
            "timer_tim11_div_trigger",
            "timer_tima_reload",
            "timer_tima_write_reloading",
            "timer_tma_write_reloading"
        ),
        mbc1: dmgTests(
            "mbc1_bits_bank1",
            "mbc1_bits_bank2",
            "mbc1_bits_mode",
            "mbc1_bits_ramg",
            "mbc1_ram_256kb",
            "mbc1_ram_64kb",
            "mbc1_rom_16Mb",
            "mbc1_rom_1Mb",
            "mbc1_rom_2Mb",
            "mbc1_rom_4Mb",
            "mbc1_rom_512kb",
            "mbc1_rom_8Mb"
        ),
        mbc2: dmgTests(
            "mbc2_bits_ramg",
            "mbc2_bits_romb",
```

```
            "mbc2_bits_unused",
            "mbc2_ram",
            "mbc2_rom_1Mb",
            "mbc2_rom_2Mb",
            "mbc2_rom_512kb"
        ),
        mbc5: dmgTests(
            "mbc5_rom_16Mb",
            "mbc5_rom_1Mb",
            "mbc5_rom_2Mb",
            "mbc5_rom_32Mb",
            "mbc5_rom_4Mb",
            "mbc5_rom_512kb",
            "mbc5_rom_64Mb",
            "mbc5_rom_8Mb"
        ),
        oam: dmgTests(
            "mem_oam",
            "oam_dma_restart",
            "oam_dma_start",
            "oam_dma_timing",
            "oam_dma_basic",
            "oam_dma_reg_read",
            "oam_dma_sources-GS"
        ),
    },
    acid: {
        acid: [...dmgTests("dmg-acid2"), ...cgbTests("cgb-acid2")],
    },
    samesuite: {
        dma: cgbTests("gbc_dma_cont", "gdma_addr_mask", "hdma_lcd_off", "hdma_mode0"),
        apu: cgbTests(
            "div_trigger_volume_10",
            "div_write_trigger",
            "div_write_trigger_10",
            "div_write_trigger_volume",
            "div_write_trigger_volume_10"
        ),
        channel1: cgbTests(
            "channel_1_align",
            "channel_1_align_cpu",
            "channel_1_delay",
            "channel_1_duty",
            "channel_1_duty_delay",
            "channel_1_extra_length_clocking-cgb0B",
            "channel_1_freq_change",
            "channel_1_freq_change_timing-A",
            "channel_1_freq_change_timing-cgb0BC",
            "channel_1_freq_change_timing-cgbDE",
            "channel_1_nrx2_glitch",
            "channel_1_nrx2_speed_change",
            "channel_1_restart",
            "channel_1_restart_nrx2_glitch",
            "channel_1_stop_div",
            "channel_1_stop_restart",
            "channel_1_sweep",
            "channel_1_sweep_restart",
            "channel_1_sweep_restart_2",
            "channel_1_volume",
            "channel_1_volume_div"
        ),
        channel2: cgbTests(
            "channel_2_align",
            "channel_2_align_cpu",
            "channel_2_delay",
            "channel_2_duty",
            "channel_2_duty_delay",
            "channel_2_extra_length_clocking-cgb0B",
            "channel_2_freq_change",
```

```
            "channel_2_nrx2_glitch",
            "channel_2_nrx2_speed_change",
            "channel_2_restart",
            "channel_2_restart_nrx2_glitch",
            "channel_2_stop_div",
            "channel_2_stop_restart",
            "channel_2_volume",
            "channel_2_volume_div"
        ),
        channel3: cgbTests(
            "channel_3_and_glitch",
            "channel_3_delay",
            "channel_3_extra_length_clocking-cgb0",
            "channel_3_extra_length_clocking-cgbB",
            "channel_3_first_sample",
            "channel_3_freq_change_delay",
            "channel_3_restart_delay",
            "channel_3_restart_during_delay",
            "channel_3_restart_stop_delay",
            "channel_3_shift_delay",
            "channel_3_shift_skip_delay",
            "channel_3_stop_delay",
            "channel_3_stop_div",
            "channel_3_wave_ram_dac_on_rw",
            "channel_3_wave_ram_locked_write",
            "channel_3_wave_ram_sync"
        ),
        channel4: cgbTests(
            "channel_4_align",
            "channel_4_delay",
            "channel_4_equivalent_frequencies",
            "channel_4_extra_length_clocking-cgb0B",
            "channel_4_freq_change",
            "channel_4_frequency_alignment",
            "channel_4_lfsr",
            "channel_4_lfsr15",
            "channel_4_lfsr_15_7",
            "channel_4_lfsr_7_15",
            "channel_4_lfsr_restart",
            "channel_4_lfsr_restart_fast",
            "channel_4_volume_div"
        ),
    },
};

type TestType = keyof typeof rawTestFiles;
type SubTestType = { [k in TestType]: keyof typeof rawTestFiles[k] }[TestType];
type TestChecker = (
    gbc: GameBoyColor,
    out: string,
    vid: Uint32Array,
    testName: string
) => Promise<null | "success" | "failure">;

export type Test = MiniTest & {
    testType: TestType;
    subTestType: SubTestType;
    check: TestChecker;
};

const loadImageData = async (fileName: string): Promise<Uint32Array> => {
    let promiseResolve: (value: Uint32Array) => void;
    let endPromise = new Promise<Uint32Array>((r) => (promiseResolve = r));

    let img = new Image();
    img.onload = function () {
        var canvas = document.createElement("canvas");
        var ctx = canvas.getContext("2d")!;
        ctx.drawImage(img, 0, 0);
```

```
        const imageData = ctx.getImageData(0, 0, 160, 144);
        const imageDataAsUint32 = new Uint32Array(imageData.data.buffer);
        promiseResolve(imageDataAsUint32);
    };
    img.src = `/tests/${fileName}.png`;
    return endPromise;
};

const compareImages = (imgA: Uint32Array, imgB: Uint32Array) => {
    if (imgA.length !== imgB.length) return "failure";
    for (let i = 0; i < imgA.length; i++) {
        if ((imgA[i] & 0xf8f8f8) !== (imgB[i] & 0xf8f8f8)) return "failure";
    }
    return "success";
};

const blaargTestCheck: TestChecker = async (gbc, serialOut, vid, testName) => {
    if (serialOut.toLowerCase().includes("pass")) return "success";
    if (serialOut.toLowerCase().includes("fail")) return "failure";

    // Some blaarg tests 'sign' $a001-$a003 with the string "de b0 61", to then
    // write the status to $a000
    if (gbc["system"].inspect(0xa001, 3) === "de b0 61") {
        const status = gbc["system"].read(0xa000);
        if (status === 0x80) return null;
        return status === 0x00 ? "success" : "failure";
    }

    if (testName === "halt_bug" && gbc["cpu"].getStepCounts() >= 700_000) {
        const expected = await loadImageData("blaarg/reference-halt_bug");
        return compareImages(expected, vid);
    }
    return null;
};

const mooneyeAndSamesuiteTestCheck: TestChecker = async (gbc) => {
    if (
        gbc["cpu"]["srB"].get() === 3 &&
        gbc["cpu"]["srC"].get() === 5 &&
        gbc["cpu"]["srD"].get() === 8 &&
        gbc["cpu"]["srE"].get() === 13 &&
        gbc["cpu"]["srH"].get() === 21 &&
        gbc["cpu"]["srL"].get() === 34
    )
        return "success";
    if (
        gbc["cpu"]["srB"].get() === 0x42 &&
        gbc["cpu"]["srC"].get() === 0x42 &&
        gbc["cpu"]["srD"].get() === 0x42 &&
        gbc["cpu"]["srE"].get() === 0x42 &&
        gbc["cpu"]["srH"].get() === 0x42 &&
        gbc["cpu"]["srL"].get() === 0x42
    )
        return "failure";
    return null;
};

const acidTestCheck: TestChecker = async (gbc, _, vid, test) => {
    if (test === "dmg-acid2" && gbc["cpu"]["stepCounter"] >= 85_000) {
        const imageData = await loadImageData("acid/reference-dmg");
        return compareImages(imageData, vid);
    } else if (test === "cgb-acid2" && gbc["cpu"]["stepCounter"] >= 140_000) {
        const imageData = await loadImageData("acid/reference-cgb");
        console.log("comparing ", { imageData, vid });
        return compareImages(imageData, vid);
    }

    return null;
};
```

```
341
342 /** The list of test categories, with a runnable that says the status of the test  */
343 const testConfig: Record<TestType, TestChecker> = {
344     blaarg: blaargTestCheck,
345     mooneye: mooneyeAndSamesuiteTestCheck,
346     acid: acidTestCheck,
347     samesuite: mooneyeAndSamesuiteTestCheck,
348 };
349
350 const tests: Test[] = (
351     Object.entries(rawTestFiles) as [TestType, Record<SubTestType, MiniTest[]>][]
352 ).flatMap(([testType, subTests]) =>
353     (Object.entries(subTests) as [SubTestType, MiniTest[]][]).flatMap(
354         ([subTestType, miniTests]) =>
355             miniTests.map((test) => ({
356                 ...test,
357                 testType,
358                 subTestType,
359                 check: testConfig[testType],
360             }))
361     )
362 );
363
364 export default tests;
```

Appendix C

GBEmulatorShootout Contribution

As part of the evaluation of this project, some code was contributed to the open source project GBEmulatorShootout, available at https://github.com/daid/GBEmulatorShootout/. The pull request in which the changes were contributed can be found at https://github.com/daid/GBEmulatorShootout/pull/19.

The deployed results of the tool can be found at https://n1ark.github.io/GBEmulatorShootout/. These test results are hosted on my fork of the original project, because an error occurred during the build of the original repository, and I do not have the permissions to re-run the script on the original repository. My fork has the same code as the original repository.

Although multiple files were modified to accomodate the addition of Emmy to the tested emulators, the main addition to the source is the file `emulators.emmy.py`. The other files will not be included, as I am not their sole author. These can be found on the repository previously mentionned.

C.1 Emmy Testing Code

```python
from util import *
from emulator import Emulator
from test import *
from selenium import webdriver
import PIL.Image

class Emmy(Emulator):
    def __init__(self):
        super().__init__("Emmy", "https://emmy-gbc.vercel.app/", startup_time=0.5)

```

```
    def setup(self):
        # download("http://problemkaputt.de/no$gmb.zip", "downloads/no$gmb.zip",
        ↪  fake_headers=True)
        # extract("downloads/no$gmb.zip", "emu/no$gmb")
        # https://chromedriver.storage.googleapis.com/109.0.5414.74/chromedriver_win32.zip

        ↪  download("https://chromedriver.storage.googleapis.com/109.0.5414.74/chromedriver_win32.zip",
        ↪  "downloads/chromedriver_win32.zip")
        extract("downloads/chromedriver_win32.zip", "emu/chromedriver_win32")
        self.driver = webdriver.Chrome("emu/chromedriver_win32/chromedriver.exe")
        self.driver.get("https://emmy-gbc.vercel.app/")
        self.driver.find_element(value="emu-speed").click()
        self.driver.find_element(value="drawer-section-settings").click()

    def isWindowOpen(self):
        return self.driver is not None

    def isProcessAlive(self, p):
        return True

    def processOutput(self, p):
        return None

    def endProcess(self, p):
        pass

    def returncode(self, p):
        return 0

    def undoSetup(self):
        self.driver.quit()

    def startProcess(self, rom, *, model, required_features):
        systemmode = {DMG: "dmg-mode", CGB: "cgb-mode"}.get(model)
        if systemmode is None:
            return None
        self.driver.find_element(value=systemmode).click()
        rom_path = os.path.abspath(rom)
        try:
            self.driver.find_element(value="rom-input").send_keys(rom_path)
            try:
                # if an alert appeared, it means the rom is incompatible
                self.driver.switch_to.alert.accept()
                return None
            except:
                # no alert, so error thrown, so the rom is compatible
                return self.driver
        except Exception as e:
            return None

    # must return a pillow image object
    def getScreenshot(self):
        canvas = self.driver.find_element(value="emulator-frame")
        canvas_base64 = self.driver.execute_script("return
        ↪  arguments[0].toDataURL('image/png').substring(21);", canvas)

        # decode
        canvas_png = base64.b64decode(canvas_base64)
        # by default, 4 canvas pixels = 1 screen pixel
        large_image = PIL.Image.open(io.BytesIO(canvas_png))
        # resize to 1:1
        small_image = large_image.resize((160, 144), PIL.Image.NEAREST)
        return small_image
```