VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

Thi Dinh

GAME BOY EMULATOR WRITTEN IN RUST

School of Technology
2023

ABSTRACT

Author	Thi Dinh
Title	Game Boy Emulator Written in Rust
Year	2023
Language	English
Pages	43
Name of Supervisor	Mikael Jakas

The main object of this thesis is to present the design and implementation of an original Game Boy handheld console for retro gaming enthusiasts. Emulation is the process of recreating the functionality and behavior of the hardware using modern software, allowing Game Boy games to be able to run on modern hardware.

The project lifecycle of this thesis is the waterfall model, further detail about this model is mentioned in the next part. The project process was divided into two main phases: Research the Game Boy's architecture and Implementation of the emulator. The research is needed to outline the structure of the project and to understand how each of the components behaves. The implementation part will show how those behaviors can be emulated using modern day programming language and hardware.

The thesis aims to emulate the original Game Boy as closely as possible. However, sound functionality and communication between consoles are not supported due to time constraints.

Overall, this thesis provides a comprehensive overview of the challenges and solutions involving developing and emulating a gaming console.

Keywords Emulation, programming, and software development

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVATIONS

APU	Audio Processing Unit
BCD	Binary-Coded Decimal
CGB	Color Game Boy
CPU	Central Processing Unit
DMG	Dot Matrix Game
EXRAM	Extra Random Access Memory
HRAM	High Random Access Memory
JSON	JavaScript Object Notation
MBC	Memory Bank Control
MMU	Memory Management Unit
OAM	Object Attribute Memory
PEG	Parsing Expression Grammars
PPU	Picture Processing Unit
RAM	Random Access Memory
ROM	Read-only Memory
RTC	Real Time Clock
SDL	Simple DirectMedia Layer
VRAM	Video Random Access Memory
WRAM	Work Random Access Memory

1 INTRODUCTION

The Game Boy is a handheld console developed and distributed by Nintendo, released in Japan back in 1989. The Game Boy was one of the bestselling gaming consoles of its time and is still popular among retro gaming enthusiasts till this day.

In this project, the aim is to create an emulator for the original DMG Game Boy and to provide an accurate and faithful gaming experience. The research methods, implementation and various challenges that arose during the process are also outlined in this thesis report.

On the part of the author, this thesis helps to widen my knowledge regarding embedded software as well as Rust development and ecosystem. Time management, resource searching, and logic implementation will also be improved throughout the study.

The result of the project is a workable program that can faithfully emulate the Game Boy's behavior, from the CPU, the display screen, sound system to the connection between consoles. It must be able to read the game cartridge (ROM), render the game to the display and allows the user to interact with the game using the keyboard. Games with save files and internal clock should also be supported.

2 METHODOLOGY

This chapter will describe an overview of the pre-implementation phase of the thesis: the research method and the technology overview.

2.1 Research Design

This thesis adopts the waterfall model for the development cycle of the console emulator. With the help of this model, it was possible to execute the project with a planned and methodical approach to development, making sure that each stage was finished before moving on to the next. The waterfall model includes 5 main stages: Requirements, Design, Development, Testing and Maintenance.

2.1.1 Requirements

In this stage the features, mandatory and stretched goals are determined.

Mandatory features:

- Working CPU, with all 256 opcodes supported
- Working internal Timer
- Accept keyboard input as convert that to Joystick and button registers
- Be able to read ROM cartridge, supporting at the very least MBC0
- Render game sprite data to the screen
- Run on Linux platform

Stretched goals:

- Audio support
- Support other MBCs

- Support RTC for those MBCs that have it
- Support serialized data transfer, which is the link cable in hardware terms
- Support for other platforms
- Application UI instead of command line only

2.1.2 Design

First, the technology that meets the purposes best was chosen. Rust was chosen as the main programming language. For the screen rendering Simple DirectMedia Layer 2 (SDL2) was used. The reason for these decisions will be explained more thoroughly in part 2.2: Technology Overview.

Secondly, the structure and architecture of the Gameboy console and its game cartridge needed to be analyzed. With that knowledge it is possible to design the structure of the codebase and plan out the timetable for the whole development cycle.

2.2 Technology Overview

In this part certain technologies for this project are explained and why they were chosen.

2.2.1 Rust

Rust is a multi-paradigm, high-level, general-purpose programming language that emphasizes performance, type safety, and concurrency.

Rust was chosen for the following reasons:

- Performance: Rust is a systems programming language that emphasizes performance and efficiency. It provides low-level control over system resources, enabling developers to write high-performance code that runs efficiently on modern hardware. As an emulator is a computationally intensive task that requires high-performance code, Rust is an ideal choice for developing an emulator.
- Memory safety: Rust's memory safety features to enable common-memory-error-free codebase, such as buffer overflow or null pointer dereferences.
- Ecosystem: Rust has a growing ecosystem of robust community libraries (crates) and tools that can make development more efficient. Couple with its excellent package manager (cargo) and Vscode extension (rust-analyzer) make a great developer experience.

2.2.2 SDL2

Simple DirectMedia Layer 2 (SDL2) is a cross-platform development library designed to provide low level access to audio, keyboard, mouse, joystick, and graphics hardware via OpenGL and Direct3D. (1)

SDL2 offers a simple and consistent API for accessing hardware features across different platforms, including Windows, macOS, Linux. It provides functions for handling events, such as mouse clicks and keyboard presses, as well as rendering graphics and playing audio.

Rust has a crate called rust-sdl2 that wraps SDL2's low-level C component in Rust code to make them more idiomatic and make inappropriate manual memory management abstract. (2)

3 GAME BOY ARCHITECTURE

In this part we will go through all the important components of the Gameboy internal excluding the Audio Processing Unit (APU) and the Link Cable system (Serial Data Transfer).

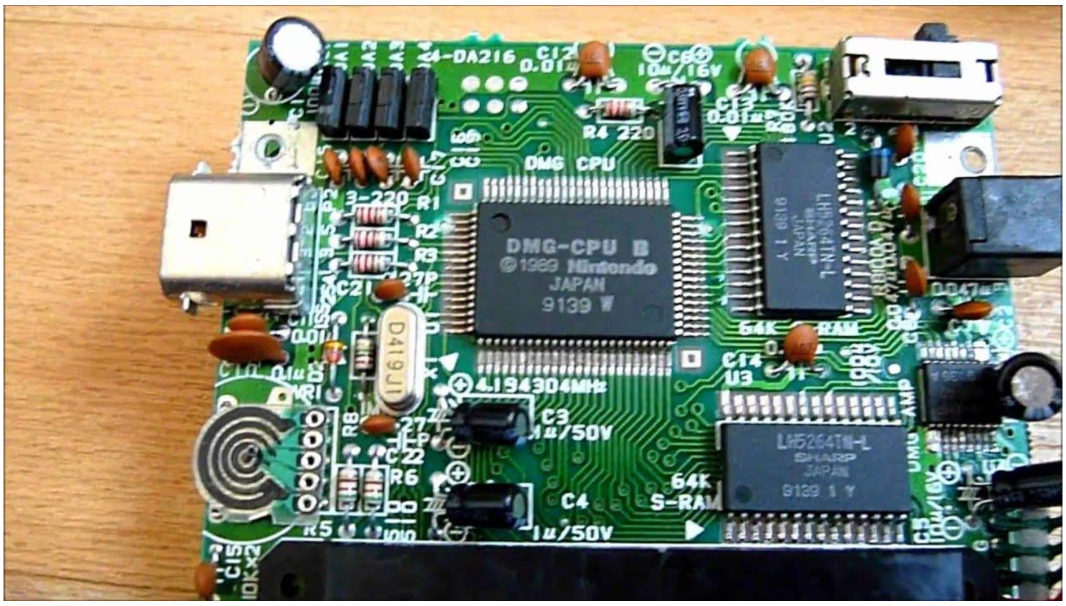


Figure 1. DMG Game Boy's motherboard

3.1 CPU

The Game Boy uses the Sharp LR35902 CPU, which is a custom mix of the Zilog Z80 and Intel 8080. The Zilog Z80 is more powerful and modern compared to the 8080, and Z80 also has backward support as it also supports all of 8080’s instructions. The Sharp LR35902 meanwhile supports almost all 8080’s instructions, supports some of the new Z80’s newly added instructions and at the same time adds some more original instructions of its own. (3)

3.1.1 Registers

Sharp LR35902 contains 8 8-bit registers A, B, C, D, E and F. However, 2 8-bit registers can be combined into a 16-bit one thus there are an extra 4 16-bit registers AF, BC, DE, and HL in addition to the program counter and the stack pointer.

Table 1. CPU Registers

16-bit	Hi	Lo	Name
AF	A	-	Accumulator and Flags
BC	B	C	BC
DE	D	E	DE
HL	H	L	HL
SP	-	-	Stack Pointer
PC	-	-	Program counter

Some more information on the registers:

- The A or Accumulator is a special one since arithmetic and logical operations can only be done on this and not the other registers.
- The F or Flags is used to store the state of various processor flags. These flags can be used to indicate whether certain operations have been completed successfully, or whether certain conditions are true. Its first 4 bits is always set to 0.
    - Z – Zero flag: Indicates whether the result of the last operation is equal to 0

- N – Negative flag: Indicates whether the last operation is a subtraction operation (only use for the BCD operation DAA)
- H – Half-carry flag: this flag is set if the result of the operation caused a carry or borrow in the lower 4 bits (only use for the BCD operation DAA)
- C – Carry flag: this flag is set if the result of an operation exceeds 8 bits and can be used to detect overflows in addition and subtraction.

Table 2. F Register

Bit	Symbol	Name
7	Z	Zero
6	N	Negative
5	H	Half-carry
4	C	Carry

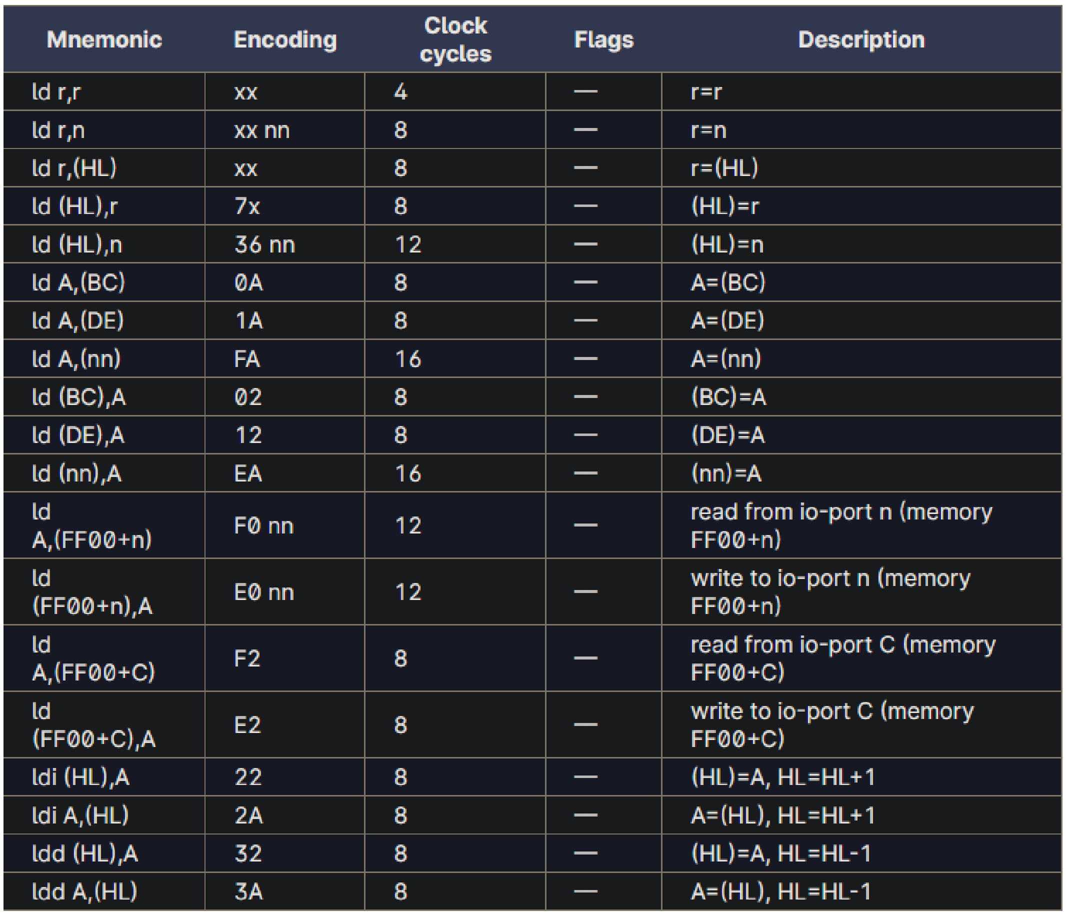
- The other 8-bit registers are used to store 8-bit data and are interchangeable in terms of functionality.
- The program counter holds the memory address of the next instruction to be executed and it will get updated after each instruction.
- The stack pointer holds the position of the top of the stack, or HRAM. It will get updated after each POP and PUSH instructions.
- The other 16-bit registers are used to store 16-bit data, usually addresses.

3.1.2 Opcodes

An opcode, or operation code, is the instruction for a machine, dictating which instruction to perform. The Game Boy's CPU has more than 500 opcodes, separated into 8 groups. (4)

For an example, this is all the opcodes in the "8-bit Load instructions" from Pandocs' CPU Instruction Set (4):

Table 3. 8-bit Load Instructions



Mnemonic	Encoding	Clock cycles	Flags	Description
ld r,r	xx	4	—	r=r
ld r,n	xx nn	8	—	r=n
ld r,(HL)	xx	8	—	r=(HL)
ld (HL),r	7x	8	—	(HL)=r
ld (HL),n	36 nn	12	—	(HL)=n
ld A,(BC)	0A	8	—	A=(BC)
ld A,(DE)	1A	8	—	A=(DE)
ld A,(nn)	FA	16	—	A=(nn)
ld (BC),A	02	8	—	(BC)=A
ld (DE),A	12	8	—	(DE)=A
ld (nn),A	EA	16	—	(nn)=A
ld A,(FF00+n)	F0 nn	12	—	read from io-port n (memory FF00+n)
ld (FF00+n),A	E0 nn	12	—	write to io-port n (memory FF00+n)
ld A,(FF00+C)	F2	8	—	read from io-port C (memory FF00+C)
ld (FF00+C),A	E2	8	—	write to io-port C (memory FF00+C)
ldi (HL),A	22	8	—	(HL)=A, HL=HL+1
ldi A,(HL)	2A	8	—	A=(HL), HL=HL+1
ldd (HL),A	32	8	—	(HL)=A, HL=HL-1
ldd A,(HL)	3A	8	—	A=(HL), HL=HL-1

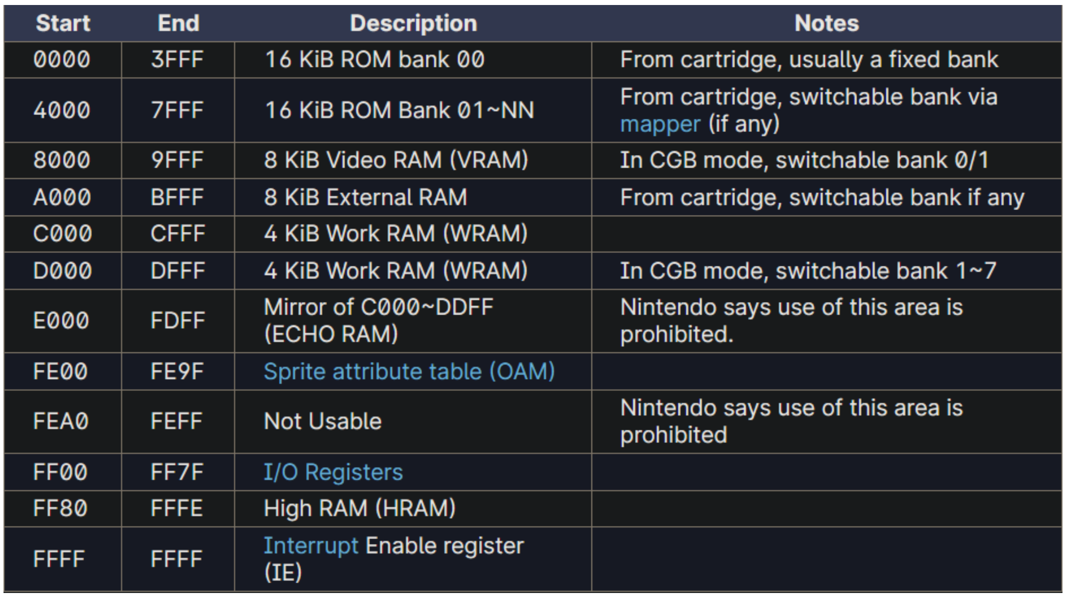
A more detailed table that lists all the opcodes and their instruction code can be found in izik1's gbops, an accurate opcode table for the Game Boy. (5)

Or if a more in depth explanation of what each opcode does is desired, refer to the Game Boy CPU Manual. (6)

3.1.3 Memory Map

To address ROM, RAM, and I/O ports, the Game Boy uses a 16-bit address bus. The Game Boy's CPU is 64KB and mapped into 12 regions:

Table 4. CPU's Memory Map



Start	End	Description	Notes
0000	3FFF	16 KiB ROM bank 00	From cartridge, usually a fixed bank
4000	7FFF	16 KiB ROM Bank 01~NN	From cartridge, switchable bank via mapper (if any)
8000	9FFF	8 KiB Video RAM (VRAM)	In CGB mode, switchable bank 0/1
A000	BFFF	8 KiB External RAM	From cartridge, switchable bank if any
C000	CFFF	4 KiB Work RAM (WRAM)	
D000	DFFF	4 KiB Work RAM (WRAM)	In CGB mode, switchable bank 1~7
E000	FDFF	Mirror of C000~DDFF (ECHO RAM)	Nintendo says use of this area is prohibited.
FE00	FE9F	Sprite attribute table (OAM)	
FEA0	FEFF	Not Usable	Nintendo says use of this area is prohibited
FF00	FF7F	I/O Registers	
FF80	FFFE	High RAM (HRAM)	
FFFF	FFFF	Interrupt Enable register (IE)	

A short explanation of each memory region: (4)

- 0000 – 07FF: This read-only region is the **ROM** copied from the cartridge. From the CGB version onward, the MBC will dictate the content of 4000-7FFF. The MBC will be discussed more in depth in chapter 3.3: Cartridge and MBC.
- 8000 – 9FFF: This is the **VRAM**, sprites for characters and background are stored here.

- A000 – BFFF: This is the **EXRAM**, copied from the game cartridge, used to store game save files.
- C000 – DFFF: This is the **WRAM** of the Game Boy, games can read or write freely in this region, but will be wiped on shut-down.
- E000 – FDFF: Prohibited area as stated by Nintendo, this region is ignored in this implementation.
- FE00 – FE9F: This is the **OAM**; this region is the logic for how the sprites and the background should be rendered.
- FEA0 – FEFF: Prohibited area as stated by Nintendo, this region is ignored in this implementation.
- FF00 – FF7F: **I/O Registers**. Registers for the joystick and buttons as well as serial transfer, audio and more are stored here.
- FF80 – FF7F: **HRAM**, also known as the stack, is stored here.
- FFFF: the **Interrupt Enable register** is stored here.

3.2 Picture Processing Unit

The Game Boy outputs graphics to a 160x144 pixel LCD. The component controlling the rendering process is the Picture Processing Unit (PPU). The PPU consists of 2 main sub-components: VRAM and OAM. (5)

3.2.1 Video Random Access Memory

The Video Random Access Memory (VRAM) is where the Game Boy stores its tile data and tile map.

Tile data represents the actual pixel patterns used to form graphics on the screen. Tile data is stored in area 0x8000-0x97FF, each taking 16 bytes. So, the whole region which is 0x17FF bytes can store 284 tiles. In CGB mode, cartridge can provide another switchable VRAM bank, so the number of tiles stored can go up to 768. In each tile, every 2 bytes make up a line, 2 bits of each byte makes 1 color ID for 1 pixel of the tile, the first byte is the least significant bit while the second byte is the most significant bit of the color ID. (4)

An example of a line in a tile makes out of 0x0 and 0xAC (or 0b1001_0000 and 0b1010_1100):

Table 5. Example of tile data

Index	7	6	5	4	3	2	1	0
Byte 1	1	0	0	1	0	0	0	0
Byte 2	1	0	1	0	1	1	0	0
A line of tile data	11	00	10	01	10	10	00	00

Tile map defines the layout of tiles on the screen. It specifies which tile from the tile data should be rendered at different positions on the screen. The VRAM contains two 32x32 tile maps in memory location 0x9800-0x9BFF and 0x9C00-0x9FFF. Any of those maps can be used to display the Background or the Window. (4)

An example of a tile map is given in Figure 2. (7)

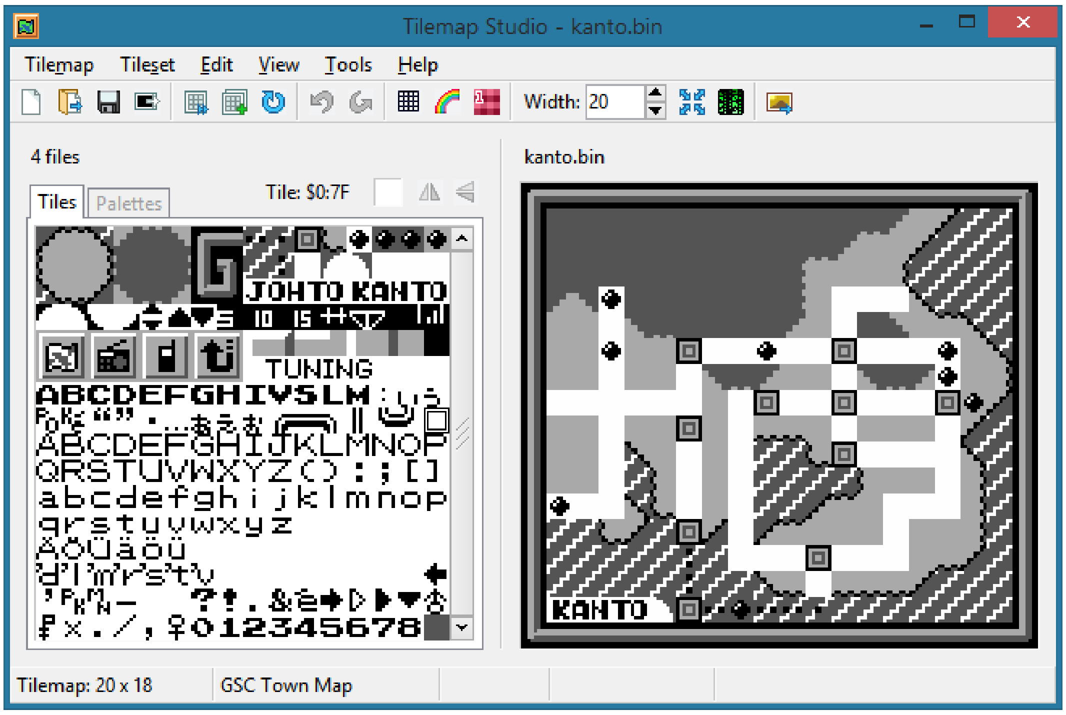


Figure 2. Tile map used in a Pokémon game.

3.2.2 Object Attribute Memory

The Object Attribute Memory (OAM) stores the sprite attributes, which will dictate the behavior of each sprite. The OAM's memory region starts at 0xFE00 and ends at 0xFE9F, which is 160 bytes, it can store 40 entries of sprite attributes, each takes up 4 bytes. This is the structure of each sprite attribute entry:

Table 6. Sprite attribution table

Byte index	Name	Short description
0	Y Position	Y = Sprite's vertical position on the screen + 16.

1	X Position	X = Sprite's horizontal position on the screen + 8.
2	Tile index	Position of the sprite in the tile data at 0x8000-0x8FFF.
3	Attributes/Flags	Some other information: if sprite is flipped, which VRAM bank to use, color palette, if BG and Window over sprite.

3.3 Cartridge and Memory Bank Control

A ROM cartridge (or just cartridge) is a replaceable part designed to connect to the back of a Game Boy. It stores the game data, including program instructions and game sprites. (8)

Since our emulator can only read digital ROMs, we will focus on them. This type of ROM consists of two main parts: the header (from address place 0x0100 to 0x014F) and the instructions (the rest of the cartridge) which will be referred to as ROM for the rest of the thesis. (4)

Figure 3. Pokémon Red and Blue Cartridge

3.3.1 Cartridge Header

The cartridge header provides all the meta data about the cartridge and the expected hardware it expects to run on.

Table 7. Cartridge Header's structure

Start	End	Description	Note
0100	0103	Entry point	The first instructions the Game Boy's boot ROM jump to at start-up, which points to the actual first instruction.
0104	0133	Nintendo logo	Needs to be the Nintendo logo as the Game Boy does check it. A way for Nintendo to control piracy.
0134	0143	Title	The game's title in uppercase ASCII.

	(013E in CGB mode)		
013F	0142	Manufacturer code	Only in CGB mode. Unknown purpose.
0143	0143	CGB flag	Determines between Color mode (CGB Mode) or monochrome compatibility mode (non-CGB Mode)
0144	0145	New licensee code	Indicating the game’s publisher.
0146	0146	SGB flag	Determines if the cartridge supports SGB functions.
0147	0147	Cartridge type	Determines the MBC of the cartridge.
0148	0148	ROM size	Size of the ROM and number of ROM banks.
0149	0149	RAM size	Size of the RAM and number of RAM banks.
014A	014A	Destination code	Specifies whether this version of the game is in-tended to be sold in Japan or elsewhere.
014B	014B	Old licensee code	This byte is used in older (pre-SGB) cartridges to specify the game’s publisher. If equals to 0x33, use the new licensee code instead.
014C	014C	Mask ROM version number	Specifies the version number of the game. It is usu-ally 00.
014D	014D	Header checksum	This byte contains an 8-bit checksum computed from the cartridge header bytes 0134 to 014C.
014E	014F	Global checksum	These bytes contain a 16-bit (big-endian) checksum simply computed as the sum of all the bytes of the cartridge ROM (except these two checksum bytes).

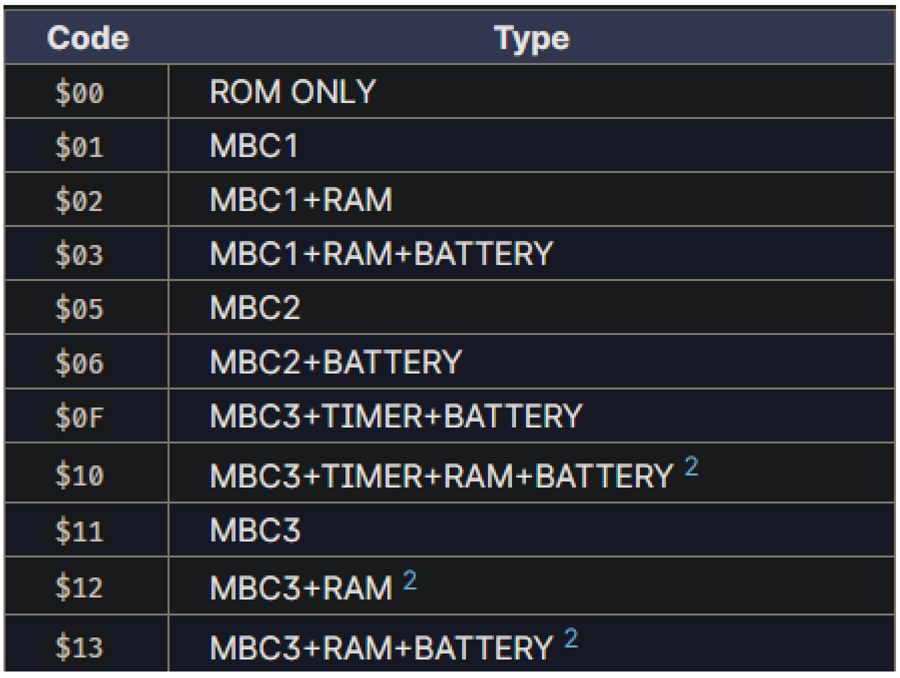
3.3.2 Memory Bank Control

The Memory Bank Control (MBC) is a separate chip not included in the Game Boy console but inside the physical cartridge itself. It was created to solve a big issue regarding ROM size limitation of the Game Boy console.

The ROM region in the memory map of the Game Boy has 32 KiB which is just right for games such as Tetris, which has exactly 32 KiB ROM and does not have an MBC. However, games can have more, for example, Pokémon Crystal has 976 KiB ROM, and would not fit. Nintendo used the switchable bank approach to solve this issue. The first 16 KiB of the ROM will always stay the same, the rest of the ROM will be mapped into 16 KiB chunks and the MBC will determine which chunk will be accessed. (4)

There are more than 10 types of MBC. In this project only some of the most popular ones are emulated: MBC1, MBC2 and MBC3 and their subtypes. Only switchable ROM banks are common in all MBCs; RAM, battery and RTC are not always present.

Table 8. Supported MBC types



Code	Type
$00	ROM ONLY
$01	MBC1
$02	MBC1+RAM
$03	MBC1+RAM+BATTERY
$05	MBC2
$06	MBC2+BATTERY
$0F	MBC3+TIMER+BATTERY
$10	MBC3+TIMER+RAM+BATTERY [2]
$11	MBC3
$12	MBC3+RAM [2]
$13	MBC3+RAM+BATTERY [2]

MBC has several registers to control its behavior. Each register can be overwritten by attempt to write on various read-only regions of the ROM: (4)

- ROM bank register: specify which ROM bank is being accessed. Always greater than 0. Will change to 1 if set to 0. Max value depends on MBC. Set to 1 by default.
- RAM bank register: specify which RAM bank is being accessed. Max value depends on MBC.
- RAM enable: if false the CPU will not be able to EXRAM. Disabled by default.
- Clock counter registers (0x08 - 0x0C):
    - Registers 0x08 to 0x0A specify the seconds, minutes, and hours of the clock.
    - 0x0B register specifies the lower 8 bits of the day counter (as it is a 9-bit number).
    - 0x0C register specifies the most significant bit of the day counter as well as the timer halt flag and the day counter carry flag.
- Timer enable: if false, disable the timer.
- RTC register select: if true, when CPU accesses a certain part of the RAM, the MBC will instead return the result of the Latch Clock Data register.
- Latch Clock Data: when writing 0x00 and then 0x01 to this register, the current time becomes latched to the RTC registers and will not change until it becomes latch again. This provides a way to read RTC registers while the clock keeps ticking.

4 IMPLEMENTATION

With the Game Boy’s architecture in mind, 3 components are required to be emulated. The CPU is the heart of the console, so it is the most important component. The CPU manages the memory of the console as well as executes the instructions from the cartridge. The PPU oversees taking the sprites, tile map in the VRAM and following the instructions in the OAM to render graphics to the screen. The MBC on the hand, is quite different. The actual physical Game Boy does not contain the MBC, instead the MBC resides in the physical cartridge. However, digital ROMs do not contain the MBC (nor the extra WRAM or EXRAM), just the game data, so the task of implementing the MBC falls on the emulator.

These are my implementations for the most important parts of the Game Boy’s system: CPU, PPU and MBC.

4.1 CPU

The CPU is the most important part of the system, especially the MMU since it controls the read and write memory operations. Therefore, the first component is emulated.

A typical CPU cycle of the emulator should include these following steps:

- Read the next opcode.
- Follow the opcode’s instruction, updating the internal state.
- Increase the program counter.
- Render the changes to the screen if needed.
- Handle the interrupt if needed.
- Repeat.

4.1.1 Registers

The Game Boy CPU has 8 8-bits registers and 6 16-bits registers. Since they will not need to store negative numbers, all of them are made unsigned integers. (4)

Setters and getters are also made for each of them, which will make creating opcodes in the next part much less of a hassle.

4.1.2 Opcodes

The Game Boy has in total 501 opcodes, including both regular and CB prefixed opcodes. (5)

501 opcodes are much to code manually, adds to the fact that many of them share many similarities. Methods for each of them were generated with the help of crates such as “scraper”, “pest” and “tera”. A short description of each crate:

- Scraper: a crate that provides an interface for HTML parsing and querying with CSS selector. (9)
- Pest: pest is a general-purpose parser. It can parse plain text using a PEG file (similar to regex in spirit). (10)
- Tera: tera is a template engine for Rust. It can be used to generate Rust code if provided with an appropriate template. (11)

A separate new Rust project was created for this generator called “gb_opcode_gen”:

First, information was required on every opcode. The HTML was downloaded from gbops’ table of opcode shown in Figure 4. (5)

Then the crate “scraper” could be used to extract the raw string data from the downloaded HTML as shown in Figure 5.

The extracted data for each opcode consists of 4 lines with the following format:

- The first line consists of 2 numbers, separated by a dash, the first one is the opcode number in hexadecimal, the second can be 8, 16, or 0, representing whether the opcode is 8-bit, 16-bit, or a control/branch opcode (e.g., jump instructions).
- The second line is the mnemonic of the code, just for reading comprehension purposes. But this is also where we can extract the operator and the operands of the opcode. (e.g., opcode “LD A, B” has operator “LD” and operands “A” and “B”).
- The third line is the size of the opcode and how many cycles the CPU takes to execute the opcode’s instruction.
- The fourth line is the list of flags that the opcode affects, in order: Z N H C. “-” means unaffected, “1” means it always set the flag, “0” means it always clears the flag, otherwise it depends on the operation.

Figure 6 shows the extracted data saved in a text file.

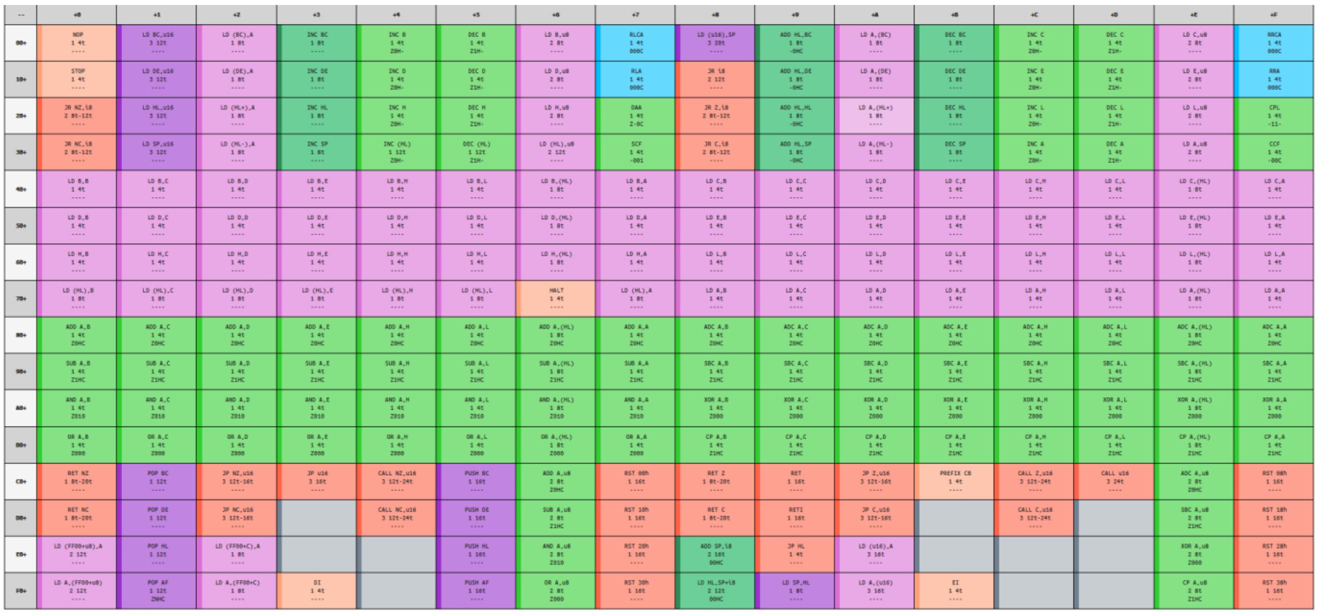
The second step is to parse the raw string into readable Rust data type with the help of the crate “pest”. Figure 7 shows the PEG file that the script uses to parse the text.

The “pest” crate helps parsing the raw data in plain text into a hash map where we can separate the fields to use in the generate template. The hash map was stored in a JSON file shown in Figure 8.

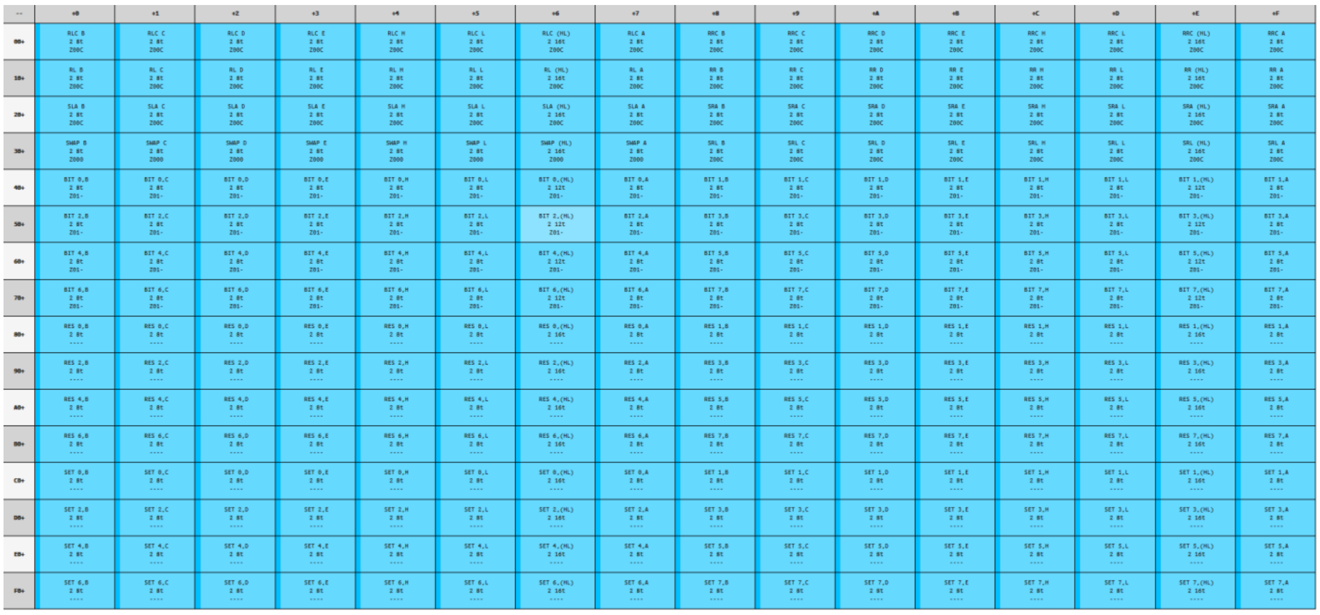
The final step is to create a template and use the “tera” crate to generate the “opcodes.rs” file based on the template. This is the template which the Rust code is generated based on.

Figure 9 is an example of macros used in the template, including macros for opcodes PUSH, POP and ADD.

After running the project with “cargo run”, almost 7300 lines are generated. Figure 10 is the generated Rust file with almost 7300 lines.

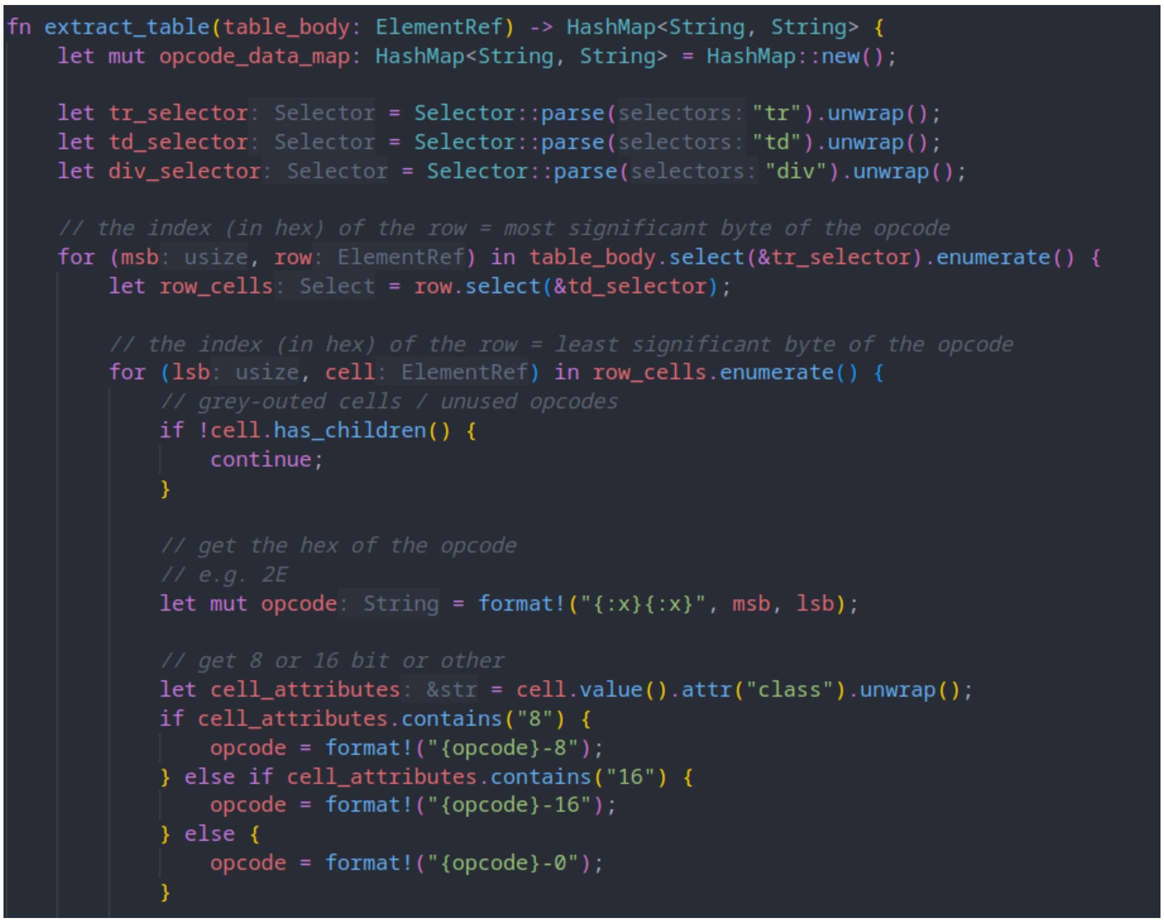


--	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+A	+B	+C	+D	+E	+F
00+	NOP 1 4t ----	LD BC,u16 3 12t ----	LD (BC),A 1 8t ----	INC BC 1 8t ----	INC B 1 4t Z0H-	DEC B 1 4t Z1H-	LD B,u8 2 8t ----	RLCA 1 4t 000C	LD (u16),SP 3 20t ----	ADD HL,BC 1 8t -0HC	LD A,(BC) 1 8t ----	DEC BC 1 8t ----	INC C 1 4t Z0H-	DEC C 1 4t Z1H-	LD C,u8 2 8t ----	RRCA 1 4t 000C
10+	STOP 1 4t ----	LD DE,u16 3 12t ----	LD (DE),A 1 8t ----	INC DE 1 8t ----	INC D 1 4t Z0H-	DEC D 1 4t Z1H-	LD D,u8 2 8t ----	RLA 1 4t 000C	JR i8 2 12t ----	ADD HL,DE 1 8t -0HC	LD A,(DE) 1 8t ----	DEC DE 1 8t ----	INC E 1 4t Z0H-	DEC E 1 4t Z1H-	LD E,u8 2 8t ----	RRA 1 4t 000C
20+	JR NZ,i8 2 8t-12t ----	LD HL,u16 3 12t ----	LD (HL+),A 1 8t ----	INC HL 1 8t ----	INC H 1 4t Z0H-	DEC H 1 4t Z1H-	LD H,u8 2 8t ----	DAA 1 4t Z-0C	JR Z,i8 2 8t-12t ----	ADD HL,HL 1 8t -0HC	LD A,(HL+) 1 8t ----	DEC HL 1 8t ----	INC L 1 4t Z0H-	DEC L 1 4t Z1H-	LD L,u8 2 8t ----	CPL 1 4t -11-
30+	JR NC,i8 2 8t-12t ----	LD SP,u16 3 12t ----	LD (HL-),A 1 8t ----	INC SP 1 8t ----	INC (HL) 1 12t Z0H-	DEC (HL) 1 12t Z1H-	LD (HL),u8 2 12t ----	SCF 1 4t -001	JR C,i8 2 8t-12t ----	ADD HL,SP 1 8t -0HC	LD A,(HL-) 1 8t ----	DEC SP 1 8t ----	INC A 1 4t Z0H-	DEC A 1 4t Z1H-	LD A,u8 2 8t ----	CCF 1 4t -00C
40+	LD B,B 1 4t ----	LD B,C 1 4t ----	LD B,D 1 4t ----	LD B,E 1 4t ----	LD B,H 1 4t ----	LD B,L 1 4t ----	LD B,(HL) 1 8t ----	LD B,A 1 4t ----	LD C,B 1 4t ----	LD C,C 1 4t ----	LD C,D 1 4t ----	LD C,E 1 4t ----	LD C,H 1 4t ----	LD C,L 1 4t ----	LD C,(HL) 1 8t ----	LD C,A 1 4t ----
50+	LD D,B 1 4t ----	LD D,C 1 4t ----	LD D,D 1 4t ----	LD D,E 1 4t ----	LD D,H 1 4t ----	LD D,L 1 4t ----	LD D,(HL) 1 8t ----	LD D,A 1 4t ----	LD E,B 1 4t ----	LD E,C 1 4t ----	LD E,D 1 4t ----	LD E,E 1 4t ----	LD E,H 1 4t ----	LD E,L 1 4t ----	LD E,(HL) 1 8t ----	LD E,A 1 4t ----
60+	LD H,B 1 4t ----	LD H,C 1 4t ----	LD H,D 1 4t ----	LD H,E 1 4t ----	LD H,H 1 4t ----	LD H,L 1 4t ----	LD H,(HL) 1 8t ----	LD H,A 1 4t ----	LD L,B 1 4t ----	LD L,C 1 4t ----	LD L,D 1 4t ----	LD L,E 1 4t ----	LD L,H 1 4t ----	LD L,L 1 4t ----	LD L,(HL) 1 8t ----	LD L,A 1 4t ----
70+	LD (HL),B 1 8t ----	LD (HL),C 1 8t ----	LD (HL),D 1 8t ----	LD (HL),E 1 8t ----	LD (HL),H 1 8t ----	LD (HL),L 1 8t ----	HALT 1 4t ----	LD (HL),A 1 8t ----	LD A,B 1 4t ----	LD A,C 1 4t ----	LD A,D 1 4t ----	LD A,E 1 4t ----	LD A,H 1 4t ----	LD A,L 1 4t ----	LD A,(HL) 1 8t ----	LD A,A 1 4t ----
80+	ADD A,B 1 4t Z0HC	ADD A,C 1 4t Z0HC	ADD A,D 1 4t Z0HC	ADD A,E 1 4t Z0HC	ADD A,H 1 4t Z0HC	ADD A,L 1 4t Z0HC	ADD A,(HL) 1 8t Z0HC	ADD A,A 1 4t Z0HC	ADC A,B 1 4t Z0HC	ADC A,C 1 4t Z0HC	ADC A,D 1 4t Z0HC	ADC A,E 1 4t Z0HC	ADC A,H 1 4t Z0HC	ADC A,L 1 4t Z0HC	ADC A,(HL) 1 8t Z0HC	ADC A,A 1 4t Z0HC
90+	SUB A,B 1 4t Z1HC	SUB A,C 1 4t Z1HC	SUB A,D 1 4t Z1HC	SUB A,E 1 4t Z1HC	SUB A,H 1 4t Z1HC	SUB A,L 1 4t Z1HC	SUB A,(HL) 1 8t Z1HC	SUB A,A 1 4t Z1HC	SBC A,B 1 4t Z1HC	SBC A,C 1 4t Z1HC	SBC A,D 1 4t Z1HC	SBC A,E 1 4t Z1HC	SBC A,H 1 4t Z1HC	SBC A,L 1 4t Z1HC	SBC A,(HL) 1 8t Z1HC	SBC A,A 1 4t Z1HC
A0+	AND A,B 1 4t Z010	AND A,C 1 4t Z010	AND A,D 1 4t Z010	AND A,E 1 4t Z010	AND A,H 1 4t Z010	AND A,L 1 4t Z010	AND A,(HL) 1 8t Z010	AND A,A 1 4t Z010	XOR A,B 1 4t Z000	XOR A,C 1 4t Z000	XOR A,D 1 4t Z000	XOR A,E 1 4t Z000	XOR A,H 1 4t Z000	XOR A,L 1 4t Z000	XOR A,(HL) 1 8t Z000	XOR A,A 1 4t Z000
B0+	OR A,B 1 4t Z000	OR A,C 1 4t Z000	OR A,D 1 4t Z000	OR A,E 1 4t Z000	OR A,H 1 4t Z000	OR A,L 1 4t Z000	OR A,(HL) 1 8t Z000	OR A,A 1 4t Z000	CP A,B 1 4t Z1HC	CP A,C 1 4t Z1HC	CP A,D 1 4t Z1HC	CP A,E 1 4t Z1HC	CP A,H 1 4t Z1HC	CP A,L 1 4t Z1HC	CP A,(HL) 1 8t Z1HC	CP A,A 1 4t Z1HC
C0+	RET NZ 1 8t-20t ----	POP BC 1 12t ----	JP NZ,u16 3 12t-16t ----	JP u16 3 16t ----	CALL NZ,u16 3 12t-24t ----	PUSH BC 1 16t ----	ADD A,u8 2 8t Z0HC	RST 00h 1 16t ----	RET Z 1 8t-20t ----	RET 1 16t ----	JP Z,u16 3 12t-16t ----	PREFIX CB 1 4t ----	CALL Z,u16 3 12t-24t ----	CALL u16 3 24t ----	ADC A,u8 2 8t Z0HC	RST 08h 1 16t ----
D0+	RET NC 1 8t-20t ----	POP DE 1 12t ----	JP NC,u16 3 12t-16t ----		CALL NC,u16 3 12t-24t ----	PUSH DE 1 16t ----	SUB A,u8 2 8t Z1HC	RST 10h 1 16t ----	RET C 1 8t-20t ----	RETI 1 16t ----	JP C,u16 3 12t-16t ----		CALL C,u16 3 12t-24t ----		SBC A,u8 2 8t Z1HC	RST 18h 1 16t ----
E0+	LD (FF00+u8),A 2 12t ----	POP HL 1 12t ----	LD (FF00+C),A 1 8t ----			PUSH HL 1 16t ----	AND A,u8 2 8t Z010	RST 20h 1 16t ----	ADD SP,i8 2 16t 00HC	JP HL 1 4t ----	LD (u16),A 3 16t ----				XOR A,u8 2 8t Z000	RST 28h 1 16t ----
F0+	LD A,(FF00+u8) 2 12t ----	POP AF 1 12t ZNHC	LD A,(FF00+C) 1 8t ----	DI 1 4t ----		PUSH AF 1 16t ----	OR A,u8 2 8t Z000	RST 30h 1 16t ----	LD HL,SP+i8 2 12t 00HC	LD SP,HL 1 8t ----	LD A,(u16) 3 16t ----	EI 1 4t ----			CP A,u8 2 8t Z1HC	RST 38h 1 16t ----



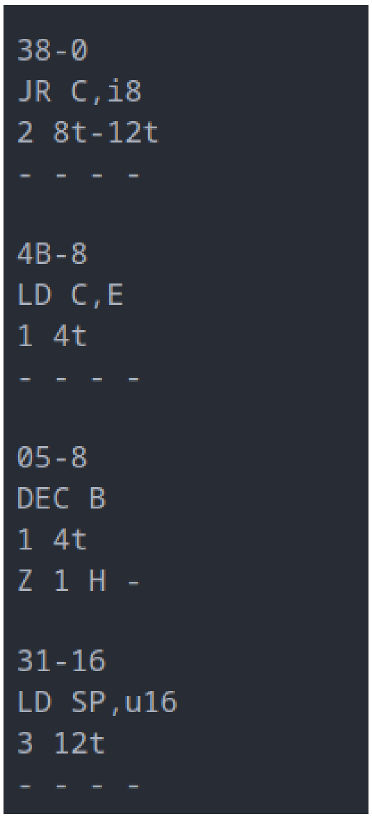
--	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+A	+B	+C	+D	+E	+F
00+	RLC B 2 8t Z00C	RLC C 2 8t Z00C	RLC D 2 8t Z00C	RLC E 2 8t Z00C	RLC H 2 8t Z00C	RLC L 2 8t Z00C	RLC (HL) 2 16t Z00C	RLC A 2 8t Z00C	RRC B 2 8t Z00C	RRC C 2 8t Z00C	RRC D 2 8t Z00C	RRC E 2 8t Z00C	RRC H 2 8t Z00C	RRC L 2 8t Z00C	RRC (HL) 2 16t Z00C	RRC A 2 8t Z00C
10+	RL B 2 8t Z00C	RL C 2 8t Z00C	RL D 2 8t Z00C	RL E 2 8t Z00C	RL H 2 8t Z00C	RL L 2 8t Z00C	RL (HL) 2 16t Z00C	RL A 2 8t Z00C	RR B 2 8t Z00C	RR C 2 8t Z00C	RR D 2 8t Z00C	RR E 2 8t Z00C	RR H 2 8t Z00C	RR L 2 8t Z00C	RR (HL) 2 16t Z00C	RR A 2 8t Z00C
20+	SLA B 2 8t Z00C	SLA C 2 8t Z00C	SLA D 2 8t Z00C	SLA E 2 8t Z00C	SLA H 2 8t Z00C	SLA L 2 8t Z00C	SLA (HL) 2 16t Z00C	SLA A 2 8t Z00C	SRA B 2 8t Z00C	SRA C 2 8t Z00C	SRA D 2 8t Z00C	SRA E 2 8t Z00C	SRA H 2 8t Z00C	SRA L 2 8t Z00C	SRA (HL) 2 16t Z00C	SRA A 2 8t Z00C
30+	SWAP B 2 8t Z000	SWAP C 2 8t Z000	SWAP D 2 8t Z000	SWAP E 2 8t Z000	SWAP H 2 8t Z000	SWAP L 2 8t Z000	SWAP (HL) 2 16t Z000	SWAP A 2 8t Z000	SRL B 2 8t Z00C	SRL C 2 8t Z00C	SRL D 2 8t Z00C	SRL E 2 8t Z00C	SRL H 2 8t Z00C	SRL L 2 8t Z00C	SRL (HL) 2 16t Z00C	SRL A 2 8t Z00C
40+	BIT 0,B 2 8t Z01-	BIT 0,C 2 8t Z01-	BIT 0,D 2 8t Z01-	BIT 0,E 2 8t Z01-	BIT 0,H 2 8t Z01-	BIT 0,L 2 8t Z01-	BIT 0,(HL) 2 12t Z01-	BIT 0,A 2 8t Z01-	BIT 1,B 2 8t Z01-	BIT 1,C 2 8t Z01-	BIT 1,D 2 8t Z01-	BIT 1,E 2 8t Z01-	BIT 1,H 2 8t Z01-	BIT 1,L 2 8t Z01-	BIT 1,(HL) 2 12t Z01-	BIT 1,A 2 8t Z01-
50+	BIT 2,B 2 8t Z01-	BIT 2,C 2 8t Z01-	BIT 2,D 2 8t Z01-	BIT 2,E 2 8t Z01-	BIT 2,H 2 8t Z01-	BIT 2,L 2 8t Z01-	BIT 2,(HL) 2 12t Z01-	BIT 2,A 2 8t Z01-	BIT 3,B 2 8t Z01-	BIT 3,C 2 8t Z01-	BIT 3,D 2 8t Z01-	BIT 3,E 2 8t Z01-	BIT 3,H 2 8t Z01-	BIT 3,L 2 8t Z01-	BIT 3,(HL) 2 12t Z01-	BIT 3,A 2 8t Z01-
60+	BIT 4,B 2 8t Z01-	BIT 4,C 2 8t Z01-	BIT 4,D 2 8t Z01-	BIT 4,E 2 8t Z01-	BIT 4,H 2 8t Z01-	BIT 4,L 2 8t Z01-	BIT 4,(HL) 2 12t Z01-	BIT 4,A 2 8t Z01-	BIT 5,B 2 8t Z01-	BIT 5,C 2 8t Z01-	BIT 5,D 2 8t Z01-	BIT 5,E 2 8t Z01-	BIT 5,H 2 8t Z01-	BIT 5,L 2 8t Z01-	BIT 5,(HL) 2 12t Z01-	BIT 5,A 2 8t Z01-
70+	BIT 6,B 2 8t Z01-	BIT 6,C 2 8t Z01-	BIT 6,D 2 8t Z01-	BIT 6,E 2 8t Z01-	BIT 6,H 2 8t Z01-	BIT 6,L 2 8t Z01-	BIT 6,(HL) 2 12t Z01-	BIT 6,A 2 8t Z01-	BIT 7,B 2 8t Z01-	BIT 7,C 2 8t Z01-	BIT 7,D 2 8t Z01-	BIT 7,E 2 8t Z01-	BIT 7,H 2 8t Z01-	BIT 7,L 2 8t Z01-	BIT 7,(HL) 2 12t Z01-	BIT 7,A 2 8t Z01-
80+	RES 0,B 2 8t ----	RES 0,C 2 8t ----	RES 0,D 2 8t ----	RES 0,E 2 8t ----	RES 0,H 2 8t ----	RES 0,L 2 8t ----	RES 0,(HL) 2 16t ----	RES 0,A 2 8t ----	RES 1,B 2 8t ----	RES 1,C 2 8t ----	RES 1,D 2 8t ----	RES 1,E 2 8t ----	RES 1,H 2 8t ----	RES 1,L 2 8t ----	RES 1,(HL) 2 16t ----	RES 1,A 2 8t ----
90+	RES 2,B 2 8t ----	RES 2,C 2 8t ----	RES 2,D 2 8t ----	RES 2,E 2 8t ----	RES 2,H 2 8t ----	RES 2,L 2 8t ----	RES 2,(HL) 2 16t ----	RES 2,A 2 8t ----	RES 3,B 2 8t ----	RES 3,C 2 8t ----	RES 3,D 2 8t ----	RES 3,E 2 8t ----	RES 3,H 2 8t ----	RES 3,L 2 8t ----	RES 3,(HL) 2 16t ----	RES 3,A 2 8t ----
A0+	RES 4,B 2 8t ----	RES 4,C 2 8t ----	RES 4,D 2 8t ----	RES 4,E 2 8t ----	RES 4,H 2 8t ----	RES 4,L 2 8t ----	RES 4,(HL) 2 16t ----	RES 4,A 2 8t ----	RES 5,B 2 8t ----	RES 5,C 2 8t ----	RES 5,D 2 8t ----	RES 5,E 2 8t ----	RES 5,H 2 8t ----	RES 5,L 2 8t ----	RES 5,(HL) 2 16t ----	RES 5,A 2 8t ----
B0+	RES 6,B 2 8t ----	RES 6,C 2 8t ----	RES 6,D 2 8t ----	RES 6,E 2 8t ----	RES 6,H 2 8t ----	RES 6,L 2 8t ----	RES 6,(HL) 2 16t ----	RES 6,A 2 8t ----	RES 7,B 2 8t ----	RES 7,C 2 8t ----	RES 7,D 2 8t ----	RES 7,E 2 8t ----	RES 7,H 2 8t ----	RES 7,L 2 8t ----	RES 7,(HL) 2 16t ----	RES 7,A 2 8t ----
C0+	SET 0,B 2 8t ----	SET 0,C 2 8t ----	SET 0,D 2 8t ----	SET 0,E 2 8t ----	SET 0,H 2 8t ----	SET 0,L 2 8t ----	SET 0,(HL) 2 16t ----	SET 0,A 2 8t ----	SET 1,B 2 8t ----	SET 1,C 2 8t ----	SET 1,D 2 8t ----	SET 1,E 2 8t ----	SET 1,H 2 8t ----	SET 1,L 2 8t ----	SET 1,(HL) 2 16t ----	SET 1,A 2 8t ----
D0+	SET 2,B 2 8t ----	SET 2,C 2 8t ----	SET 2,D 2 8t ----	SET 2,E 2 8t ----	SET 2,H 2 8t ----	SET 2,L 2 8t ----	SET 2,(HL) 2 16t ----	SET 2,A 2 8t ----	SET 3,B 2 8t ----	SET 3,C 2 8t ----	SET 3,D 2 8t ----	SET 3,E 2 8t ----	SET 3,H 2 8t ----	SET 3,L 2 8t ----	SET 3,(HL) 2 16t ----	SET 3,A 2 8t ----
E0+	SET 4,B 2 8t ----	SET 4,C 2 8t ----	SET 4,D 2 8t ----	SET 4,E 2 8t ----	SET 4,H 2 8t ----	SET 4,L 2 8t ----	SET 4,(HL) 2 16t ----	SET 4,A 2 8t ----	SET 5,B 2 8t ----	SET 5,C 2 8t ----	SET 5,D 2 8t ----	SET 5,E 2 8t ----	SET 5,H 2 8t ----	SET 5,L 2 8t ----	SET 5,(HL) 2 16t ----	SET 5,A 2 8t ----
F0+	SET 6,B 2 8t ----	SET 6,C 2 8t ----	SET 6,D 2 8t ----	SET 6,E 2 8t ----	SET 6,H 2 8t ----	SET 6,L 2 8t ----	SET 6,(HL) 2 16t ----	SET 6,A 2 8t ----	SET 7,B 2 8t ----	SET 7,C 2 8t ----	SET 7,D 2 8t ----	SET 7,E 2 8t ----	SET 7,H 2 8t ----	SET 7,L 2 8t ----	SET 7,(HL) 2 16t ----	SET 7,A 2 8t ----

Figure 4. 501 opcodes



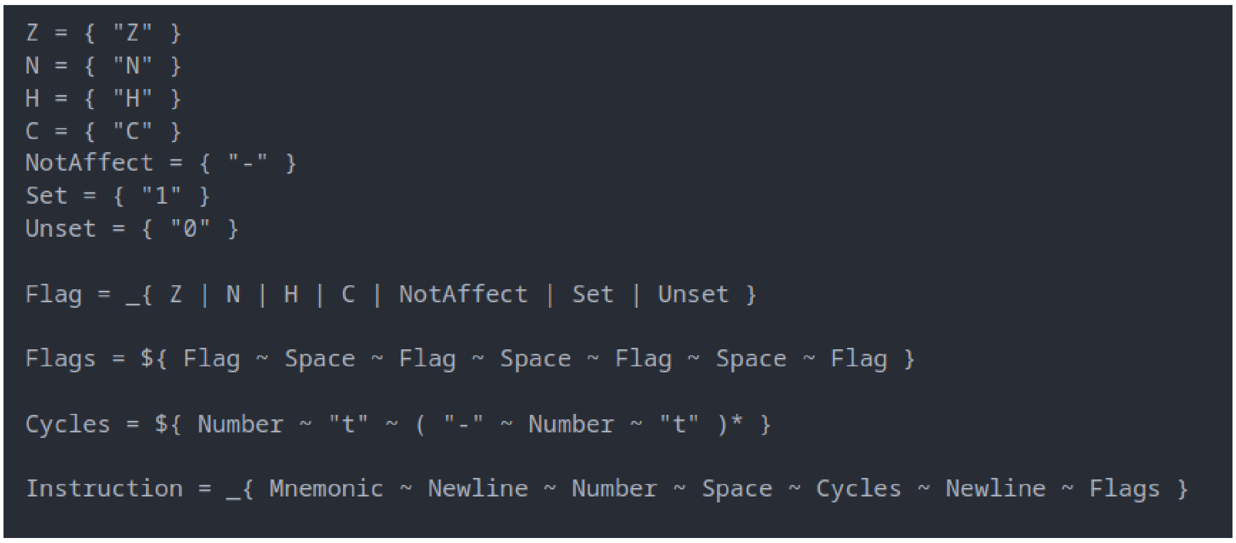
```rust
fn extract_table(table_body: ElementRef) -> HashMap<String, String> {
    let mut opcode_data_map: HashMap<String, String> = HashMap::new();

    let tr_selector: Selector = Selector::parse(selectors: "tr").unwrap();
    let td_selector: Selector = Selector::parse(selectors: "td").unwrap();
    let div_selector: Selector = Selector::parse(selectors: "div").unwrap();

    // the index (in hex) of the row = most significant byte of the opcode
    for (msb: usize, row: ElementRef) in table_body.select(&tr_selector).enumerate() {
        let row_cells: Select = row.select(&td_selector);

        // the index (in hex) of the row = least significant byte of the opcode
        for (lsb: usize, cell: ElementRef) in row_cells.enumerate() {
            // grey-outed cells / unused opcodes
            if !cell.has_children() {
                continue;
            }

            // get the hex of the opcode
            // e.g. 2E
            let mut opcode: String = format!("{:x}{:x}", msb, lsb);

            // get 8 or 16 bit or other
            let cell_attributes: &str = cell.value().attr("class").unwrap();
            if cell_attributes.contains("8") {
                opcode = format!("{opcode}-8");
            } else if cell_attributes.contains("16") {
                opcode = format!("{opcode}-16");
            } else {
                opcode = format!("{opcode}-0");
            }
```

Figure 5. Using scraper crate



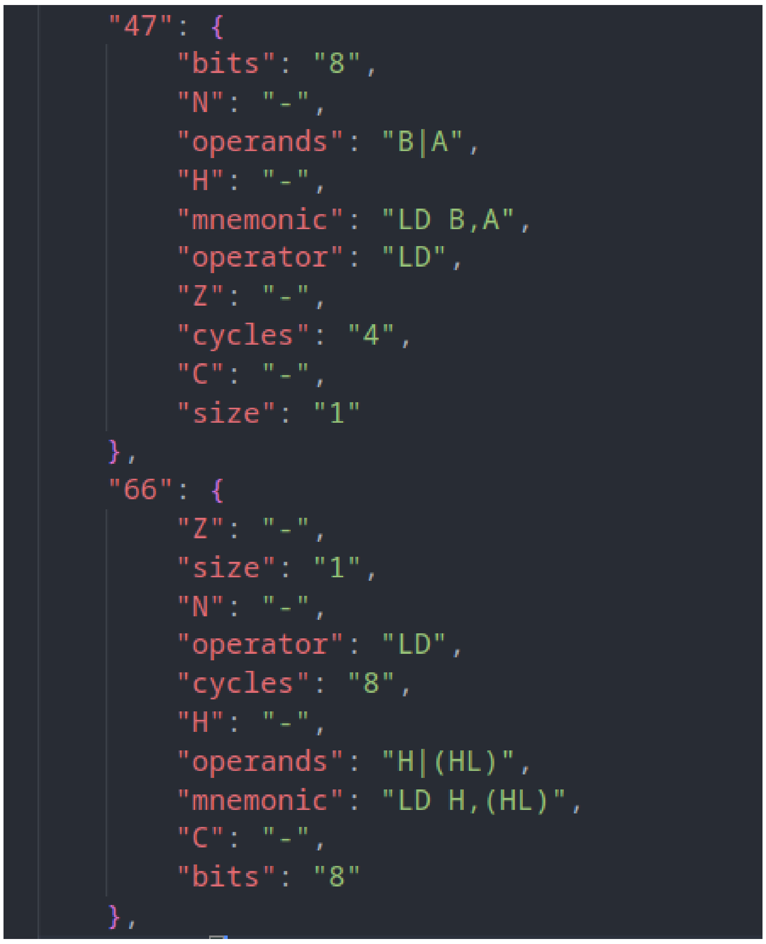
```
38-0
JR C,i8
2 8t-12t
- - - -

4B-8
LD C,E
1 4t
- - - -

05-8
DEC B
1 4t
Z 1 H -

31-16
LD SP,u16
3 12t
- - - -
```

Figure 6. Extracted HTML text



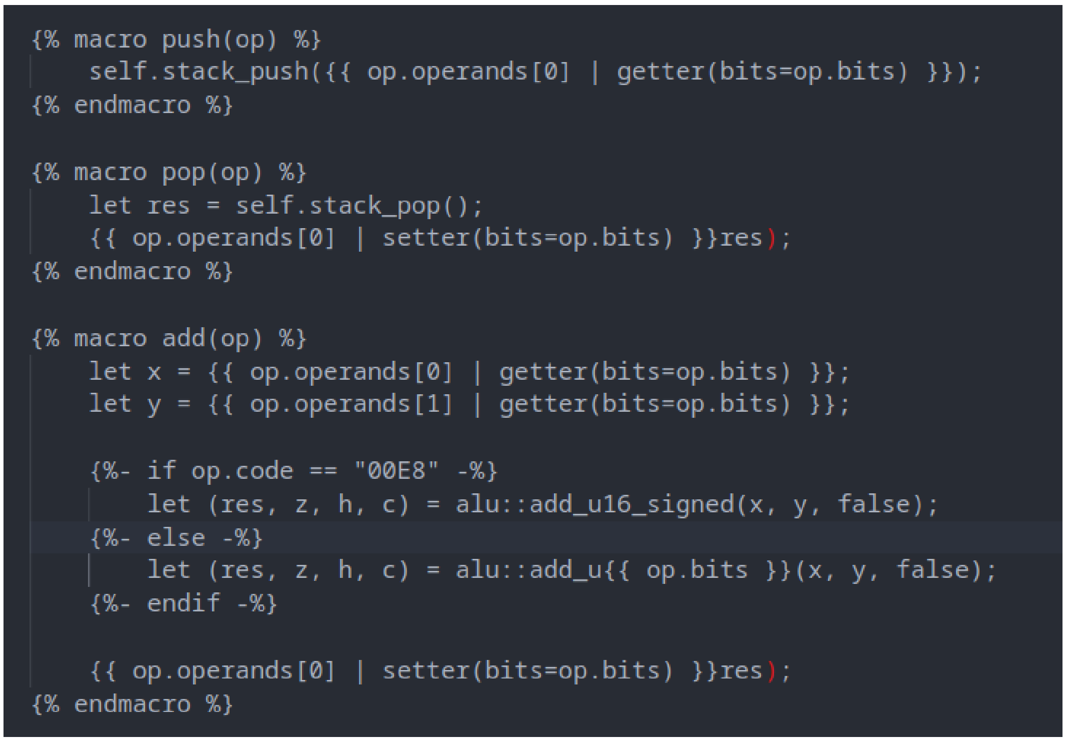
```
Z = { "Z" }
N = { "N" }
H = { "H" }
C = { "C" }
NotAffect = { "-" }
Set = { "1" }
Unset = { "0" }

Flag = _{ Z | N | H | C | NotAffect | Set | Unset }

Flags = ${ Flag ~ Space ~ Flag ~ Space ~ Flag ~ Space ~ Flag }

Cycles = ${ Number ~ "t" ~ ( "-" ~ Number ~ "t" )* }

Instruction = _{ Mnemonic ~ Newline ~ Number ~ Space ~ Cycles ~ Newline ~ Flags }
```

Figure 7. PEG file



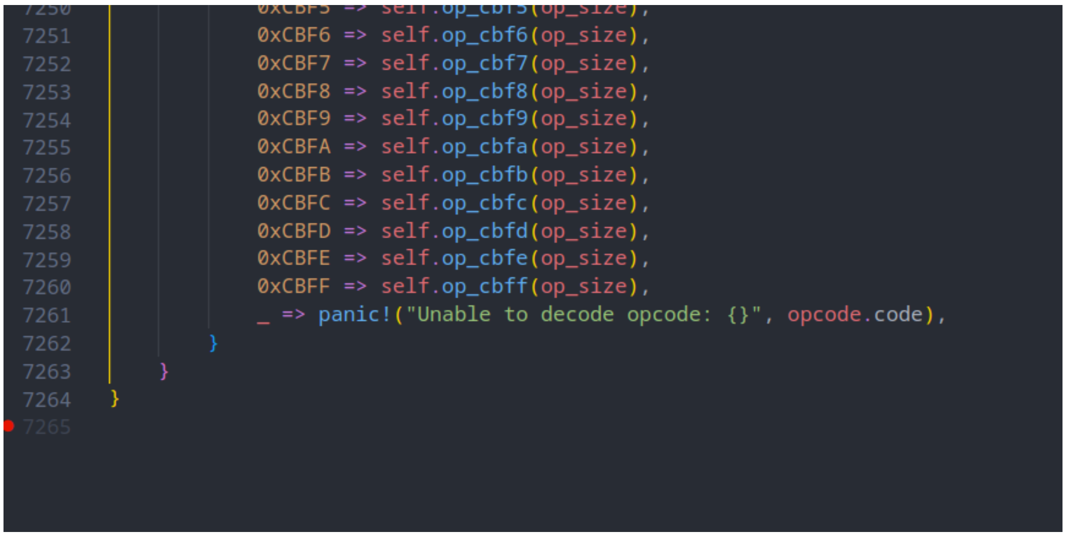
```
    "47": {
        "bits": "8",
        "N": "-",
        "operands": "B|A",
        "H": "-",
        "mnemonic": "LD B,A",
        "operator": "LD",
        "Z": "-",
        "cycles": "4",
        "C": "-",
        "size": "1"
    },
    "66": {
        "Z": "-",
        "size": "1",
        "N": "-",
        "operator": "LD",
        "cycles": "8",
        "H": "-",
        "operands": "H|(HL)",
        "mnemonic": "LD H,(HL)",
        "C": "-",
        "bits": "8"
    },
```

Figure 8. Parsed data



```
{% macro push(op) %}
    self.stack_push({{ op.operands[0] | getter(bits=op.bits) }});
{% endmacro %}

{% macro pop(op) %}
    let res = self.stack_pop();
    {{ op.operands[0] | setter(bits=op.bits) }}res);
{% endmacro %}

{% macro add(op) %}
    let x = {{ op.operands[0] | getter(bits=op.bits) }};
    let y = {{ op.operands[1] | getter(bits=op.bits) }};

    {%- if op.code == "00E8" -%}
        let (res, z, h, c) = alu::add_u16_signed(x, y, false);
    {%- else -%}
        let (res, z, h, c) = alu::add_u{{ op.bits }}(x, y, false);
    {%- endif -%}

    {{ op.operands[0] | setter(bits=op.bits) }}res);
{% endmacro %}
```

Figure 9. Tera template



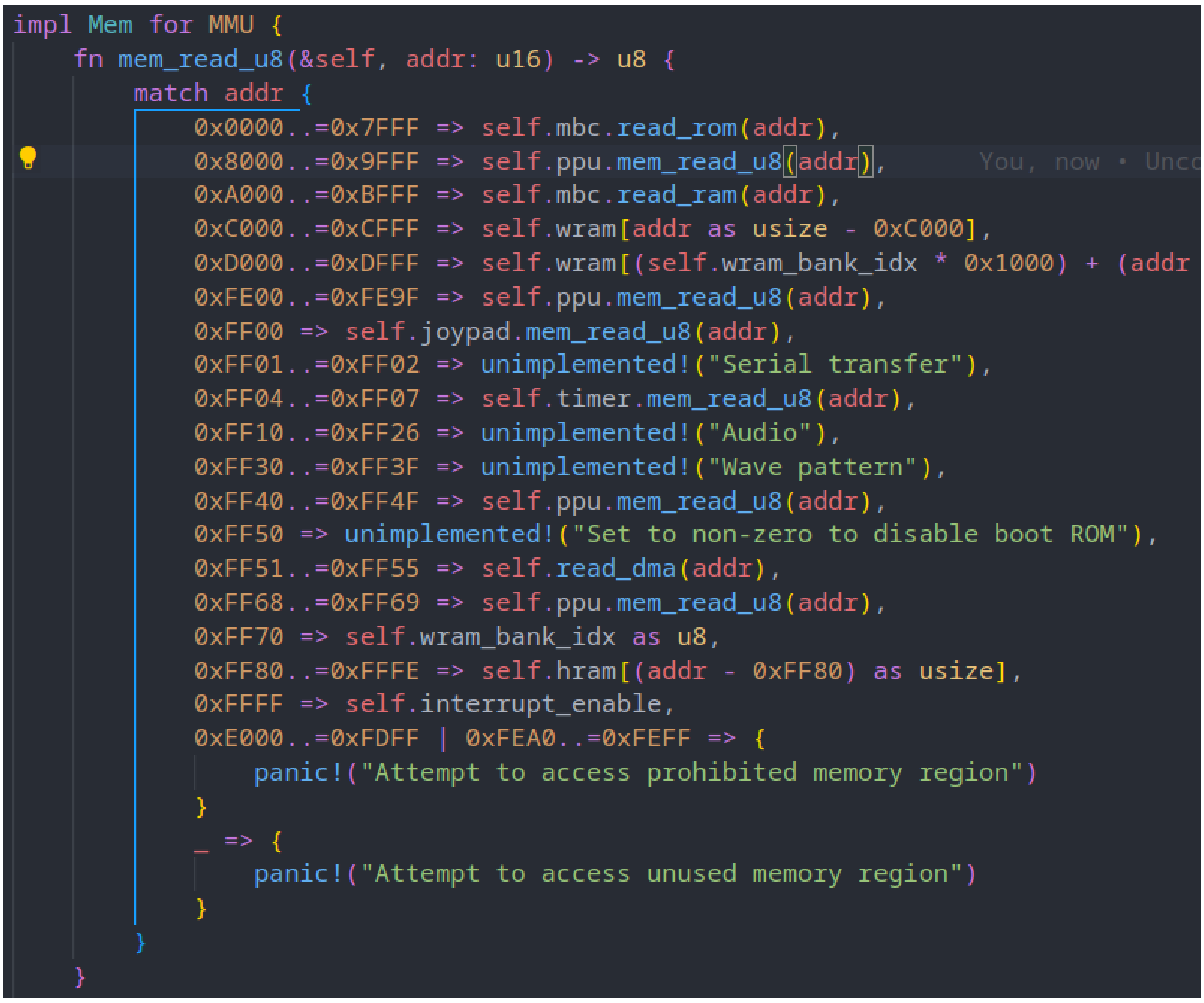
```
            0xCBF5 => self.op_cbf5(op_size),
            0xCBF6 => self.op_cbf6(op_size),
            0xCBF7 => self.op_cbf7(op_size),
            0xCBF8 => self.op_cbf8(op_size),
            0xCBF9 => self.op_cbf9(op_size),
            0xCBFA => self.op_cbfa(op_size),
            0xCBFB => self.op_cbfb(op_size),
            0xCBFC => self.op_cbfc(op_size),
            0xCBFD => self.op_cbfd(op_size),
            0xCBFE => self.op_cbfe(op_size),
            0xCBFF => self.op_cbff(op_size),
            _ => panic!("Unable to decode opcode: {}", opcode.code),
        }
    }
}
```

Figure 10. opcodes.rs

4.1.3 Memory Management Unit

The Memory Management Unit (MMU) is where the memory map of the CPU is emulated. All the read and write operations go through here. The MMU manages which component deals with each read and write operations. For example, if the CPU wants to read 2 bytes (16 bits) from the memory region 0xFF90, which is in the HRAM (also called the stack) region, the MMU will call the HRAM's own "read_u16()" method.

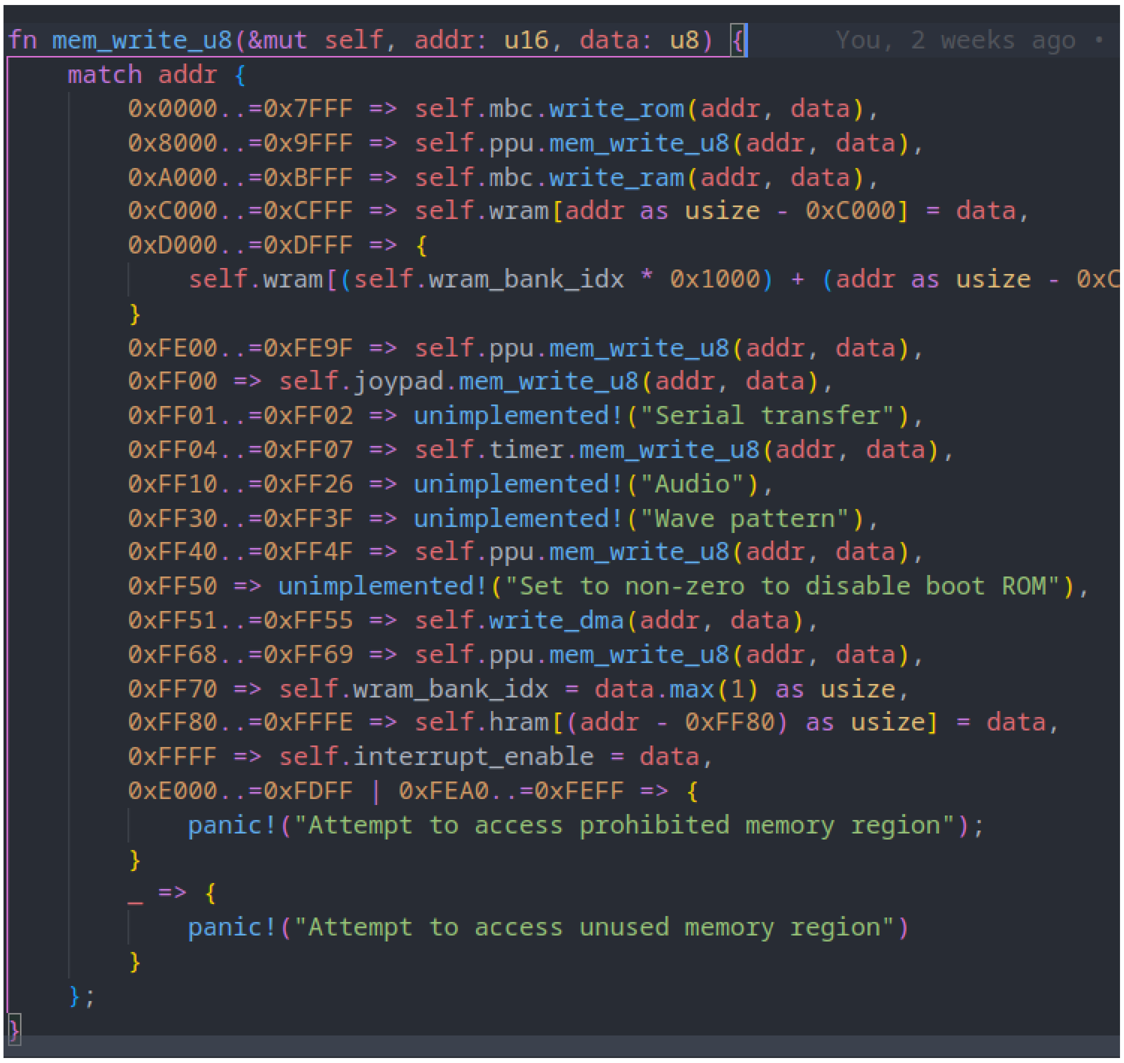
Figure 11 shows how the read 1 byte or "read_u8()" method for the MMU was implemented:



```
impl Mem for MMU {
    fn mem_read_u8(&self, addr: u16) -> u8 {
        match addr {
            0x0000..=0x7FFF => self.mbc.read_rom(addr),
            0x8000..=0x9FFF => self.ppu.mem_read_u8(addr),        You, now • Uncc
            0xA000..=0xBFFF => self.mbc.read_ram(addr),
            0xC000..=0xCFFF => self.wram[addr as usize - 0xC000],
            0xD000..=0xDFFF => self.wram[(self.wram_bank_idx * 0x1000) + (addr
            0xFE00..=0xFE9F => self.ppu.mem_read_u8(addr),
            0xFF00 => self.joypad.mem_read_u8(addr),
            0xFF01..=0xFF02 => unimplemented!("Serial transfer"),
            0xFF04..=0xFF07 => self.timer.mem_read_u8(addr),
            0xFF10..=0xFF26 => unimplemented!("Audio"),
            0xFF30..=0xFF3F => unimplemented!("Wave pattern"),
            0xFF40..=0xFF4F => self.ppu.mem_read_u8(addr),
            0xFF50 => unimplemented!("Set to non-zero to disable boot ROM"),
            0xFF51..=0xFF55 => self.read_dma(addr),
            0xFF68..=0xFF69 => self.ppu.mem_read_u8(addr),
            0xFF70 => self.wram_bank_idx as u8,
            0xFF80..=0xFFFE => self.hram[(addr - 0xFF80) as usize],
            0xFFFF => self.interrupt_enable,
            0xE000..=0xFDFF | 0xFEA0..=0xFEFF => {
                panic!("Attempt to access prohibited memory region")
            }
            _ => {
                panic!("Attempt to access unused memory region")
            }
        }
    }
```

Figure 11. 8-bit read method

“write_u8()” method is implemented in the same way. (Figure 12)

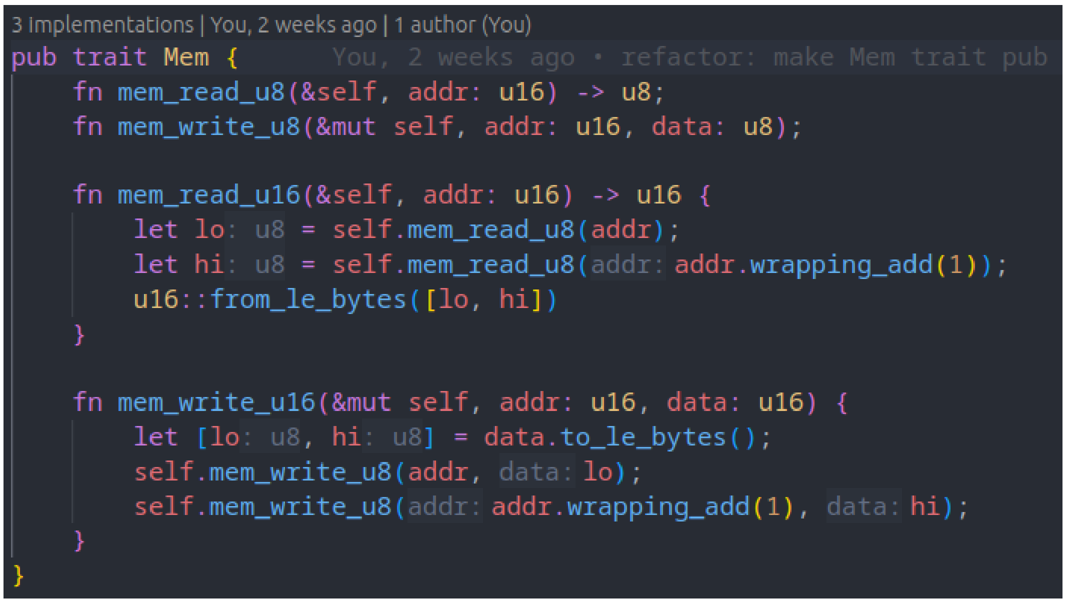


```rust
fn mem_write_u8(&mut self, addr: u16, data: u8) {      You, 2 weeks ago •
    match addr {
        0x0000..=0x7FFF => self.mbc.write_rom(addr, data),
        0x8000..=0x9FFF => self.ppu.mem_write_u8(addr, data),
        0xA000..=0xBFFF => self.mbc.write_ram(addr, data),
        0xC000..=0xCFFF => self.wram[addr as usize - 0xC000] = data,
        0xD000..=0xDFFF => {
            self.wram[(self.wram_bank_idx * 0x1000) + (addr as usize - 0xC
        }
        0xFE00..=0xFE9F => self.ppu.mem_write_u8(addr, data),
        0xFF00 => self.joypad.mem_write_u8(addr, data),
        0xFF01..=0xFF02 => unimplemented!("Serial transfer"),
        0xFF04..=0xFF07 => self.timer.mem_write_u8(addr, data),
        0xFF10..=0xFF26 => unimplemented!("Audio"),
        0xFF30..=0xFF3F => unimplemented!("Wave pattern"),
        0xFF40..=0xFF4F => self.ppu.mem_write_u8(addr, data),
        0xFF50 => unimplemented!("Set to non-zero to disable boot ROM"),
        0xFF51..=0xFF55 => self.write_dma(addr, data),
        0xFF68..=0xFF69 => self.ppu.mem_write_u8(addr, data),
        0xFF70 => self.wram_bank_idx = data.max(1) as usize,
        0xFF80..=0xFFFE => self.hram[(addr - 0xFF80) as usize] = data,
        0xFFFF => self.interrupt_enable = data,
        0xE000..=0xFDFF | 0xFEA0..=0xFEFF => {
            panic!("Attempt to access prohibited memory region");
        }
        _ => {
            panic!("Attempt to access unused memory region")
        }
    };
}
```

Figure 12. 8-bit write method

The unimplemented parts are stretched goals.

The read/write for 16-bit is doing 8-bit read/write twice:

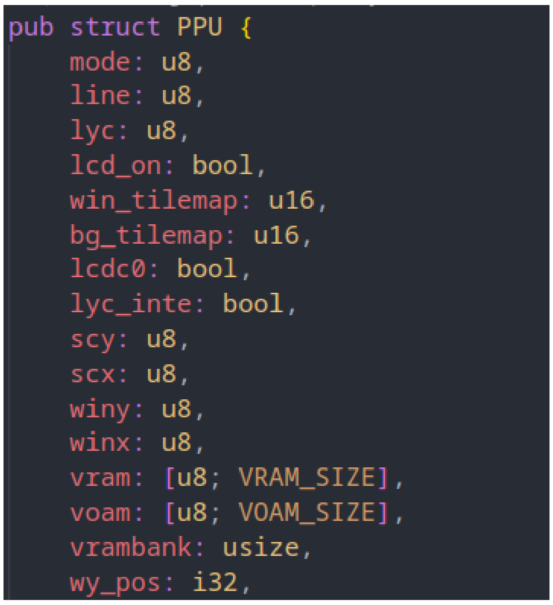


```
3 implementations | You, 2 weeks ago | 1 author (You)
pub trait Mem {        You, 2 weeks ago • refactor: make Mem trait pub
    fn mem_read_u8(&self, addr: u16) -> u8;
    fn mem_write_u8(&mut self, addr: u16, data: u8);

    fn mem_read_u16(&self, addr: u16) -> u16 {
        let lo: u8 = self.mem_read_u8(addr);
        let hi: u8 = self.mem_read_u8(addr: addr.wrapping_add(1));
        u16::from_le_bytes([lo, hi])
    }

    fn mem_write_u16(&mut self, addr: u16, data: u16) {
        let [lo: u8, hi: u8] = data.to_le_bytes();
        self.mem_write_u8(addr, data: lo);
        self.mem_write_u8(addr: addr.wrapping_add(1), data: hi);
    }
}
```

Figure 13. 16-bit read and write methods

4.2 PPU

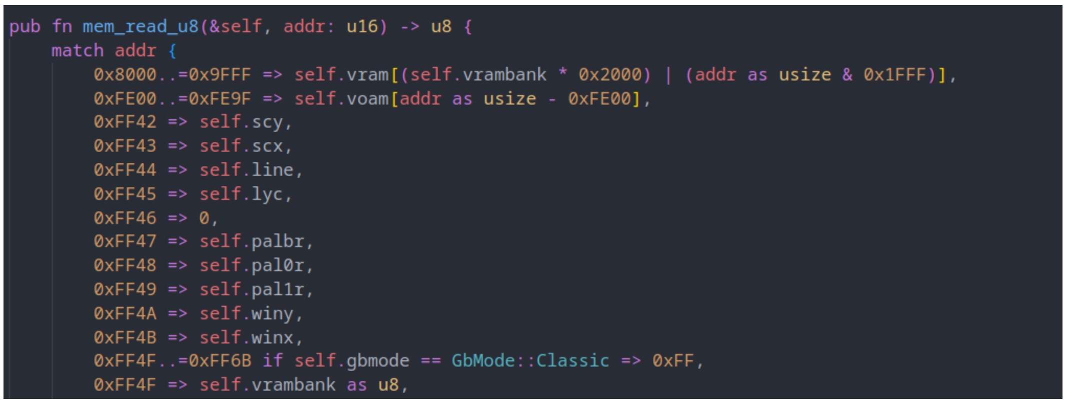
First, a PPU struct was created that stores the tile data, tile map, OAM, and other PPU-related registers.



```
pub struct PPU {
    mode: u8,
    line: u8,
    lyc: u8,
    lcd_on: bool,
    win_tilemap: u16,
    bg_tilemap: u16,
    lcdc0: bool,
    lyc_inte: bool,
    scy: u8,
    scx: u8,
    winy: u8,
    winx: u8,
    vram: [u8; VRAM_SIZE],
    voam: [u8; VOAM_SIZE],
    vrambank: usize,
    wy_pos: i32,
```

Figure 14. PPU struct

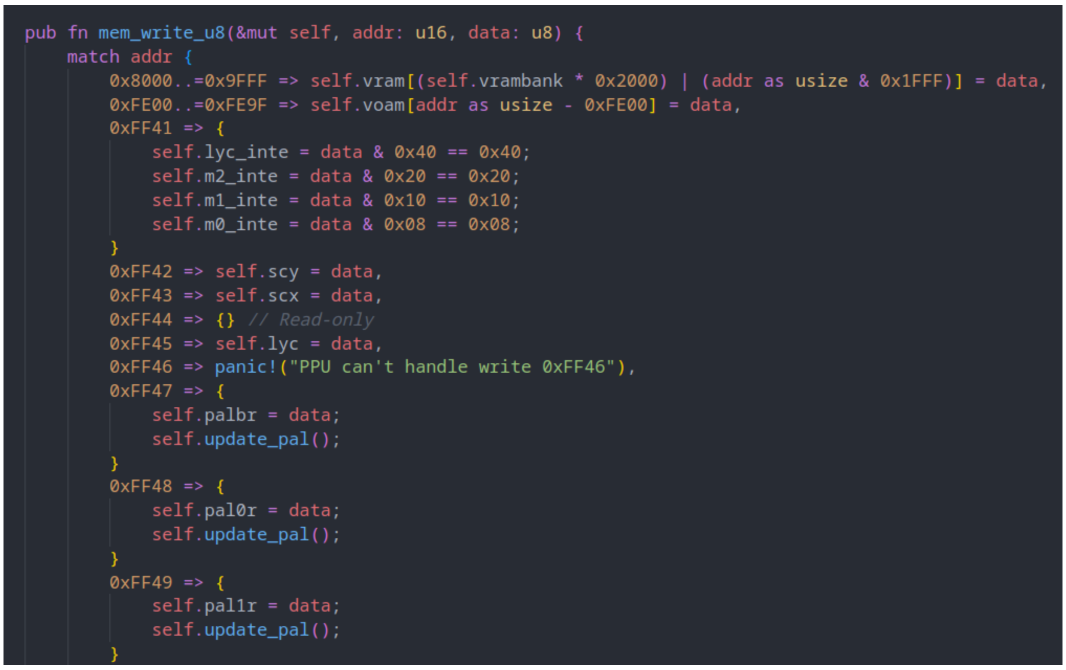
The MMU calls the read method of each of its subcomponent, therefore, “mem_read_u8” method was implemented for the PPU:



```rust
pub fn mem_read_u8(&self, addr: u16) -> u8 {
    match addr {
        0x8000..=0x9FFF => self.vram[(self.vrambank * 0x2000) | (addr as usize & 0x1FFF)],
        0xFE00..=0xFE9F => self.voam[addr as usize - 0xFE00],
        0xFF42 => self.scy,
        0xFF43 => self.scx,
        0xFF44 => self.line,
        0xFF45 => self.lyc,
        0xFF46 => 0,
        0xFF47 => self.palbr,
        0xFF48 => self.pal0r,
        0xFF49 => self.pal1r,
        0xFF4A => self.winy,
        0xFF4B => self.winx,
        0xFF4F..=0xFF6B if self.gbmode == GbMode::Classic => 0xFF,
        0xFF4F => self.vrambank as u8,
```

Figure 15. PPU read

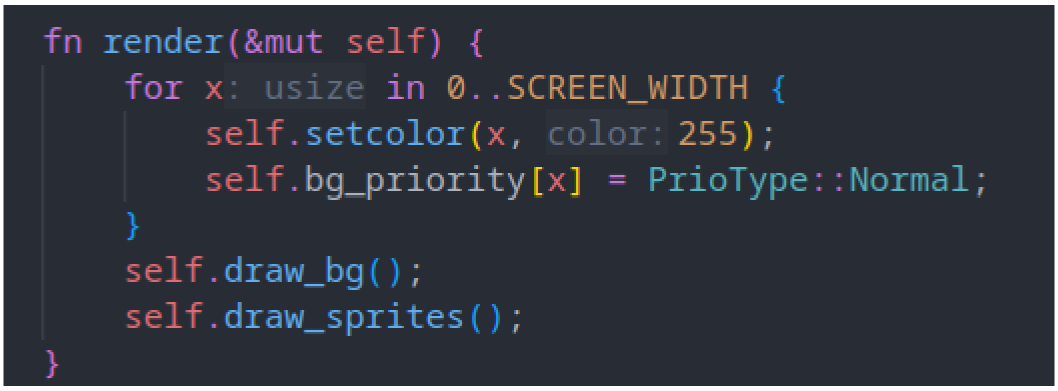
The same implementation was made for the “mem_write_u8” method:



```rust
pub fn mem_write_u8(&mut self, addr: u16, data: u8) {
    match addr {
        0x8000..=0x9FFF => self.vram[(self.vrambank * 0x2000) | (addr as usize & 0x1FFF)] = data,
        0xFE00..=0xFE9F => self.voam[addr as usize - 0xFE00] = data,
        0xFF41 => {
            self.lyc_inte = data & 0x40 == 0x40;
            self.m2_inte = data & 0x20 == 0x20;
            self.m1_inte = data & 0x10 == 0x10;
            self.m0_inte = data & 0x08 == 0x08;
        }
        0xFF42 => self.scy = data,
        0xFF43 => self.scx = data,
        0xFF44 => {} // Read-only
        0xFF45 => self.lyc = data,
        0xFF46 => panic!("PPU can't handle write 0xFF46"),
        0xFF47 => {
            self.palbr = data;
            self.update_pal();
        }
        0xFF48 => {
            self.pal0r = data;
            self.update_pal();
        }
        0xFF49 => {
            self.pal1r = data;
            self.update_pal();
        }
```

Figure 16. PPU write

In each CPU cycle, after the CPU has executed an instruction and updated all the memory regions, the PPU calls the render function to update the screen:



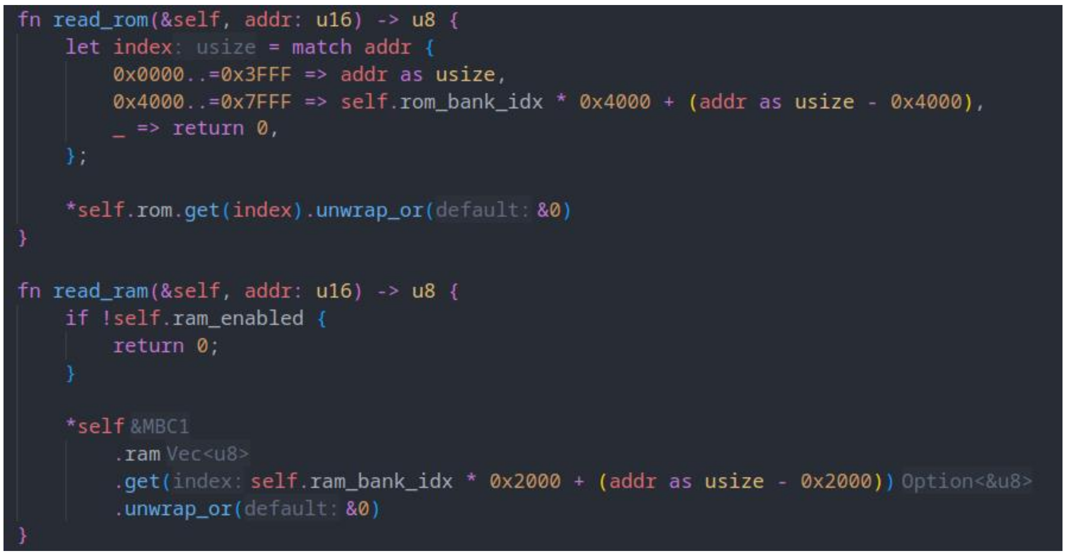
```
fn render(&mut self) {
    for x: usize in 0..SCREEN_WIDTH {
        self.setcolor(x, color: 255);
        self.bg_priority[x] = PrioType::Normal;
    }
    self.draw_bg();
    self.draw_sprites();
}
```

Figure 17. PPU render

4.3 Cartridge and MBC

The first type of cartridge is MBC0, or cartridges without an MBC. These cartridges only contain up to 32KiB of ROM and do not need any special operations.

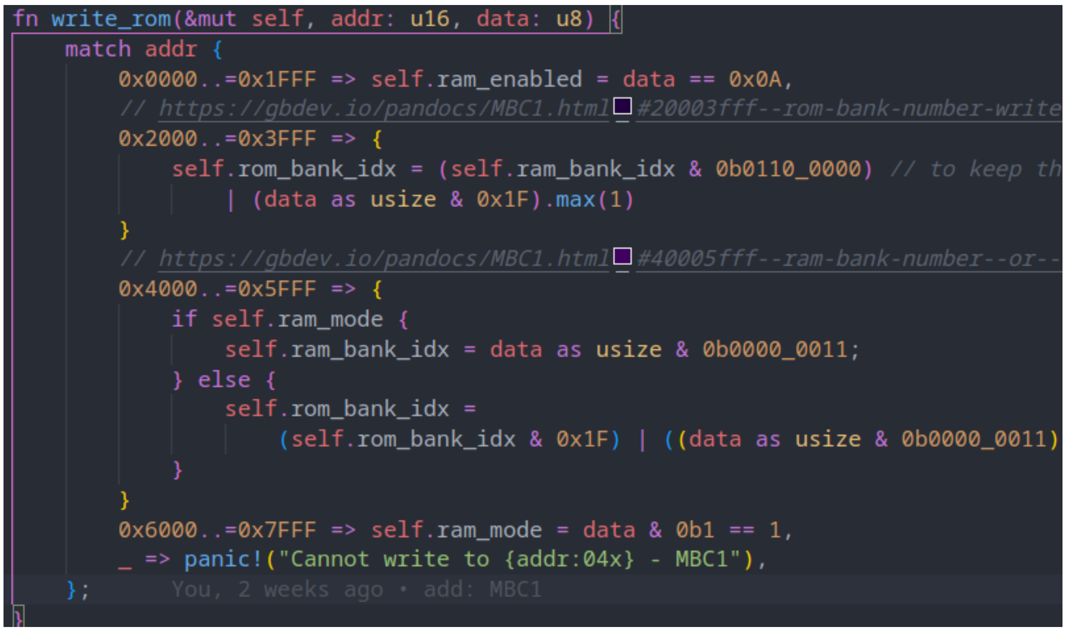
The other MBC types that my emulator supports (MBC1, MBC2 and MBC3) all have multiple extra ROM and RAM banks as well as registers. Since they are quite similar only an example of MBC1 implementation is shown in Figure 18.



```
fn read_rom(&self, addr: u16) -> u8 {
    let index: usize = match addr {
        0x0000..=0x3FFF => addr as usize,
        0x4000..=0x7FFF => self.rom_bank_idx * 0x4000 + (addr as usize - 0x4000),
        _ => return 0,
    };

    *self.rom.get(index).unwrap_or(default: &0)
}

fn read_ram(&self, addr: u16) -> u8 {
    if !self.ram_enabled {
        return 0;
    }

    *self &MBC1
        .ram Vec<u8>
        .get(index: self.ram_bank_idx * 0x2000 + (addr as usize - 0x2000)) Option<&u8>
        .unwrap_or(default: &0)
}
```

Figure 18. Read ROM and RAM in MBC

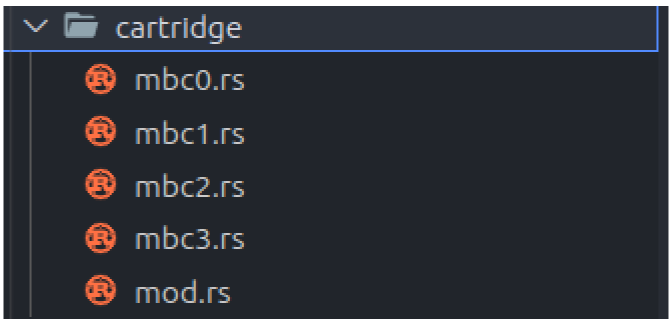
Since MBC has ROM and RAM banks, whose data will be read will depend on which bank is being selected. The bank index can be changed by writing to a certain part of ROM. Writing this way will not change the ROM, but will update the MBC's internal register instead:



```
fn write_rom(&mut self, addr: u16, data: u8) {
    match addr {
        0x0000..=0x1FFF => self.ram_enabled = data == 0x0A,
        // https://gbdev.io/pandocs/MBC1.html#20003fff--rom-bank-number-write
        0x2000..=0x3FFF => {
            self.rom_bank_idx = (self.ram_bank_idx & 0b0110_0000) // to keep th
                | (data as usize & 0x1F).max(1)
        }
        // https://gbdev.io/pandocs/MBC1.html#40005fff--ram-bank-number--or--
        0x4000..=0x5FFF => {
            if self.ram_mode {
                self.ram_bank_idx = data as usize & 0b0000_0011;
            } else {
                self.rom_bank_idx =
                    (self.rom_bank_idx & 0x1F) | ((data as usize & 0b0000_0011)
            }
        }
        0x6000..=0x7FFF => self.ram_mode = data & 0b1 == 1,
        _ => panic!("Cannot write to {addr:04x} - MBC1"),
    };      You, 2 weeks ago • add: MBC1
}
```

Figure 19. Writing to ROM

Expanding those methods to fit MBC 2 and MBC3 we have finished the cartridge mod:



```
cartridge
    mbc0.rs
    mbc1.rs
    mbc2.rs
    mbc3.rs
    mod.rs
```

Figure 20. MBC implementation

With the MBCs covered, we have reached the end of the implementation part.

5 CONCLUSIONS

This chapter will demonstrate how the final product works, evaluate its functionalities, and provide a self-evaluation.

Figure 21 shows the folder structure of the project.

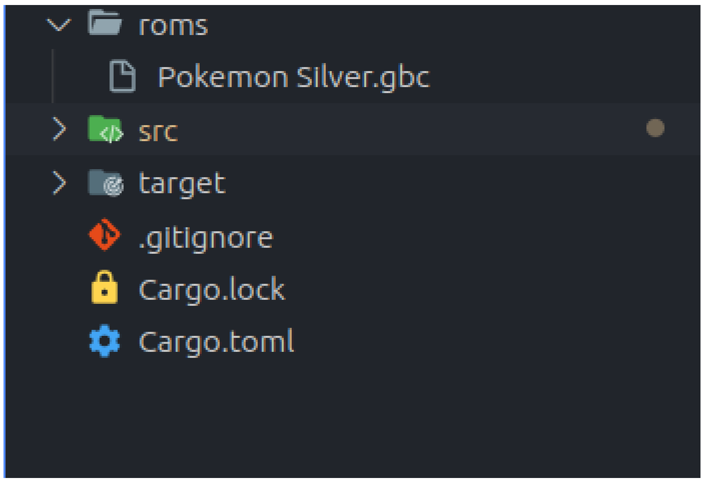


Figure 21. Project structure

To run the game Pokémon Silver for example, we will need to use the command "cargo run roms/Pokemon\ Silver.gbc", and this is the emulator running on my laptop:

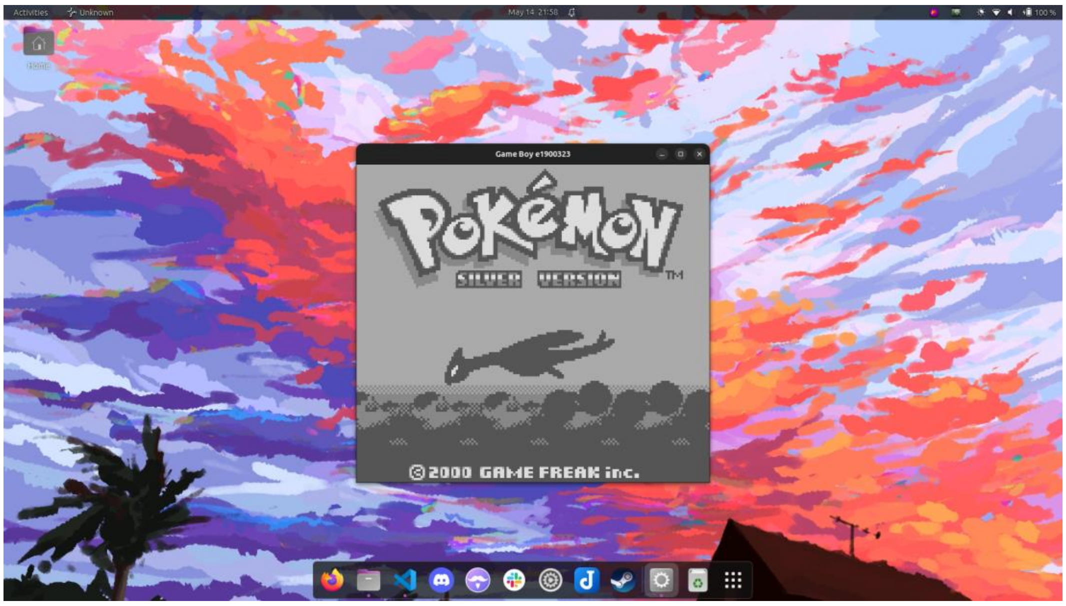


Figure 22. Emulator running

5.1 Product Evaluation

The emulator is functionable, being able to run games like Tetris and Pokémon Silver.

However, it is very incomplete, as it still lacks many features of the original Game Boy. The most notable missing feature is the audio. There was not enough time to implement the Audio Processing Unit. The emulator also cannot run all games, it only supports game ROMs without MBC or MBC 1 to 3. Even among the MBC3 ROMs, the emulator does not support their internal clock.

5.2 Future Development

To make the emulator feature complete, support needs to be added for audio, internal clocks for MBCs and other types of MBCs. There are also many features that modern emulators have that the original consoles did not, such as speed up, save state and export, import save files.

5.3 Self-evaluation

I am a software developer, and have limited knowledge regarding embedded systems, to the point that understanding the Game Boy’s technical reference proves to be the most challenging aspect of the whole process. I wanted to choose a project that is suitable for my experience, not too simple but not so hard that I cannot complete for the thesis, so projects like emulating a CHIP 8 would be too simple and emulating later generations consoles, such as the Nintendo 3DS would be too difficult to complete within a reasonable deadline. With all of that said, I am extremely satisfied with the process, I have gained much knowledge about this field and most importantly, know how to read documentation for something I know nothing about. I really believe that the knowledge gained from this project will serve as a solid foundation for my desired career as an embedded software developer.

REFERENCES

1 SDL Wiki. **Introduction to SDL 2.0**. Accessed 10.05.2023. https://wiki.libsdl.org/SDL2/Introduction

2 Nilsen, S., Aldridge, T., Cobrand. 2022. **Rust-SDL2**. Accessed 10.05.2023. https://crates.io/crates/sdl2

3 Gbdev Wiki. 2020. **CPU Comparison with Z80**. Accessed 10.05.2023. https://gbdev.gg8.se/wiki/articles/CPU_Comparision_with_Z80

4 Pan Docs. Accessed 10.05.2023. https://gbdev.io/pandocs/About.html.

5 izik1. **gbops, an accurate opcode table for the Game Boy**. Accessed 10.05.2023. https://izik1.github.io/gbops

6 Pan of Anthrox et al. **Game Boy CPU Manual**. Accessed 10.05.2023. https://realboyemulator.files.wordpress.com/2013/01/gbcpuman.pdf

7 Rangi. 2019. **Tilemap Studio 4.0.1: a tilemap editor+creator for Nintendo**. Accessed 11.05.2023. https://gbdev.gg8.se/forums/viewtopic.php?id=648

8 Wikipedia. 2023. **ROM cartridge**. Accessed 11.05.2023. https://en.wikipedia.org/wiki/ROM_cartridge

9 June and Carlo Federico Vescovo. 2023. **Scraper**. Accessed 11.05.2023. https://crates.io/crates/scraper

10 pest-parser and Dragoș Tiselice. 2023. **Pest**. Accessed 11.05.2023. https://pest.rs/book/

11 Vincent Prouillet. 2023. **Tera**. Accessed 11.05.2023. https://tera.netlify.app/