Emulation

Takanen Edoardo

Abstract

(still to be placed)
images from:

1. https://www.pastraiser.com/cpu/gameboy/gameboy_opcodes.html

Contents

1 Introduction

As a kid, I used to play with some Nintendo© consoles like the Wii or the DS and I have always been keen about the games they make. This passion for videogames grew on me so that I got interested in the making process of them, leading to game development. I never asked myself one question, though, until this year, which is how are these games able to run onto this consoles? and how can people make emulators so that I could play on my personal computer? Thus, I decided to embrace the unknown world of emulation, because I have always been fascinated by it but always took it for granted.
Emulation is not well explained on the Internet. Mainly, the results you will find if you search for it are "the program pretends to be the console" or "you will be able to play old titles". Unfortunately I was not satisfied with these responses and I wanted to know more. Thus, my emulation journey started with looking for a full definition of this process.
I will try to give my own definition of emulation, so that I can lay a starting point to a general knowledge that will be then deepened during the paper. With this being said, to "make an emulator" means to develop the software that will do exactly what the hardware of the console does, so that when plugging the game data, the program will know how to read and handle it.
Emulation can only happen when the machine in which we run the software is more powerful than the hardware we want to emulate. For example, if our console has 2Kb of memory, for sure we are not able to emulate it on a computer that has 2Kb or less, since we also have to consider that the host computer will have an operating system running (which uses some of the RAM). I chose to make a Game Boy emulator, because while looking for the retro consoles, it seemed the least difficult when talking about the hardware structure complexity, meaning a good way to start tackling this topic.

2 Premises

This emulator project is purely for understanding the concepts and the theory about how a machine like a console (or similarly a computer) is made (and also for fun). There are many better-developed Game Boy emulators online, and making one that could compete with the other popular ones is nowhere near my goals. In addition, I could not achieve the level of knowledge I want to reach if I just looked at other people's codes, I wanted to **fully** understand the subject. Obviously though I have to start somewhere, I do not have the skills to reverse-engineer a real Game Boy (although it would be an extremely interesting challenge), for this reason I will only consult theory guides made by many passionate developers and hackers that already did the work of studying the Game Boy from scratch for us. For a better understanding, I used **two** sources for this project, in order to have a dual perspective on the study.
For anyone who would like to dig into this challenge too, the guides are GBDev and GBEDG.

What this paper is not?
This paper is not and was not intended as a guide, I previously attached some real references. This document is a report of my journey throughtout the development of the emulator, made to understand the fundamentals of what is around us, from personal computers to smartphones. It could also be a way for readers to get passionate about this topic and an inspiration for them to make their own emulators (or even better, their own consoles!).

3 General Structure

The first thing I want to cover is in what way we want to structure our emulator.

```
int main() {
    // Hardware components definition
    Memory mem;
    CPU cpu;
    // ...

    // Components initialization
    mem.init();
    cpu.init();
    cpu.load_boot();
    // ...

    while (true) {
        cpu.execute(mem); // Executing an operation
        // emulate all the other components
    }

    return 0;
}
```

Actually, when we look at the circuit inside the Game Boy, all the components are, on one side, all on their own, they all execute at the same time. The CPU could be executing a simple addition, while the PPU could be rendering graphics onto the screen, all of these things happen simultaneously. This **can** be done with software, but would mean more complexity. Hence we will pick a less complicated path, and decide to execute the components one at a time.

```
while (true) {
    // Returns how many clock cycles the instruction took
    int cycles = cpu.execute();

    // Updating all the other components
    timers.update(cycles);
    ppu.update(cycles);
    // ...
}
```

The real hardware is driven by the clock, while my implementation will be driven by how many clock cycles an instruction took. This may cause some bugs and imprecisions in the emulator (and that was my main concern), but in the end it worked just fine.

4 Memory

Before looking at the main components that shape the Game Boy hardware, I would like to focus on how memory is subdivided inside the console.

4.1 Memory Mapping

The address bus had 16 bits, meaning there could be 65'536 unique addresses (64Kb).
Since the Game Boy did not have a flash memory, those 64Kb were all the console could access (this includes all the different RAMs, the cartridge data, and the registers made to control various components). Internally inside the Game Boy, there is some logic that specifies what component will be activated based on the requesting address, but we do not have to worry about it since we are not dealing with actual hardware this time.
These are the regions into which the memory is split, along with a brief description of their use. Notice that some areas are marked as "Prohibited", though Nintendo has not provided an explanation for this.

Start	End	Description
0000	3FFF	16Kb cartridge ROM
4000	7FFF	16Kb cartridge ROM*
8000	9FFF	8Kb VRAM
A000	BFFF	8Kb External RAM
C000	CFFF	4Kb Work RAM
D000	DFFF	4Kb Work RAM
E000	FDFF	Prohibited area
FE00	FE9F	OAM
FEA0	FEFF	Prohibited area
FF00	FF7F	I/O Registers
FF80	FFFE	High RAM
FFFF	FFFF	IE register

*switchable

Table 1: Game Boy memory mapping

Most of these regions will be discussed later, based on the components that use them.

4.2 Choices for This Project

For simplicity, I decided not to break down all these areas into different regions of memory in the emulator, but I opted for an easier solution, which is to create an array of 65'536 bytes, since the data bus was 8 bits-long, so every address

will have exactly one byte of data.
There is a trade-off in choosing this approach tough. On one hand, it makes things simpler to manage, we can have all the memory in one place and it really helps when debugging, but on the other hand, it is not entirely correct. Each region of memory has different restrictions, cartridge memory should be read-only, some areas might not be fully readable and writable sometimes (we will see an example when implementing the PPU).
By choosing this option, we are making all the memory readable and writable for everyone, so tecnically, the game could edit its own code (and this actually happened when I was emulating Tetris!).

4.3 Implementation

```
struct Memory {
private:
    static constexpr u32 MAX_MEM = 64 * 1024;
    // Array of bytes to emulate all the Gameboy's addresses
    Byte Data[MAX_MEM] = {};
public:
    void init();

    /**
     * Functions used for setting and accessing memory as
     * mem[addr] = value          to set
     * Byte value = mem[addr]     to access
     * */
    Byte operator[](u32 addr) const;
    Byte& operator[](u32 addr);

    /**
     * Dumps all the memory in a file,
     * used for debugging purposes
     * */
    void dump(const char* filename);
};
```

This is the entire memory structure. Every component we create will have access to this structure in order to read from and write to memory using the two operator[] functions. I have also added a **dump** function that writes all the bytes to a binary file at the moment the function is called, for easier debugging.
The **init** function just initializes the array by setting all values to 0. This is **not** actually done in a real Game Boy, as the memory tipically contains random values when powered on. However, I decided to initialize it with all zeros to make debugging easier by allowing me to see if any memory has changed.

5 CPU

The first component we likely want to implement is the Central Processing Unit (CPU). This component is the most important in our circuit, and is the one that coordinates the other components, executes the program we give to it etc. Thus, the first thing I had to implement were all the instructions that the processor could run.

5.1 Architecture and Considerations

The Game Boy CPU is a custom-made by Sharp Corporation (which had a close relationship with Nintendo at that time), it is often referred to as **DMG-CPU** or **Sharp SM83** and runs at around 4.19 MHz. When making the processor a lot of inspiration was taken from the Zilog 80 and the Intel 8080. Personally, I recently had the possibility to work with a real Z80, and over the past few months, I have gained hands-on experience with its architecture. Specifically, when studying the Zilog, I noticed some differences and similarities with the Game Boy processor.
For example, the Nintendo processor lacks the IX and IY registers, which in the Zilog were used to set a base address that could be offset with the $LD(IX+d), r$ and $LD(IY+d), r$ to save instruction bytes. Instead, the DMG-CPU introduced a brand new load instruction, LDH (load from high memory), which always offsets from address **FF00**, pointing to **High-RAM** and the **I/O registers**.
Custom-making their own CPU was the perfect choice for Nintendo, as it allowed them to implement changes like these to better suit their needs, save instruction bytes, and increase performace.
CPUs are the main core of every computer and what they do most of the time is execute instructions defined in some memory. In the next section, we will give a brief summary of the different types of instructions. But what is most important for now is to understand that the majority of these operate on internal registers and external memory.

8-bit registers		16-bit pairs
A*	F**	AF
B	C	BC
D	E	DE
H	L	HL

16-bit register	Description
SP	Stack Pointer
PC	Program Counter

*Accumulator
**Flags

Table 2: Game Boy registers

Registers are the fastest memory to access because it is already inside the processor. However their size is very limited, so we cannot have everything in them. That is why the CPU has a set of instructions for loading data to and from larger memory.

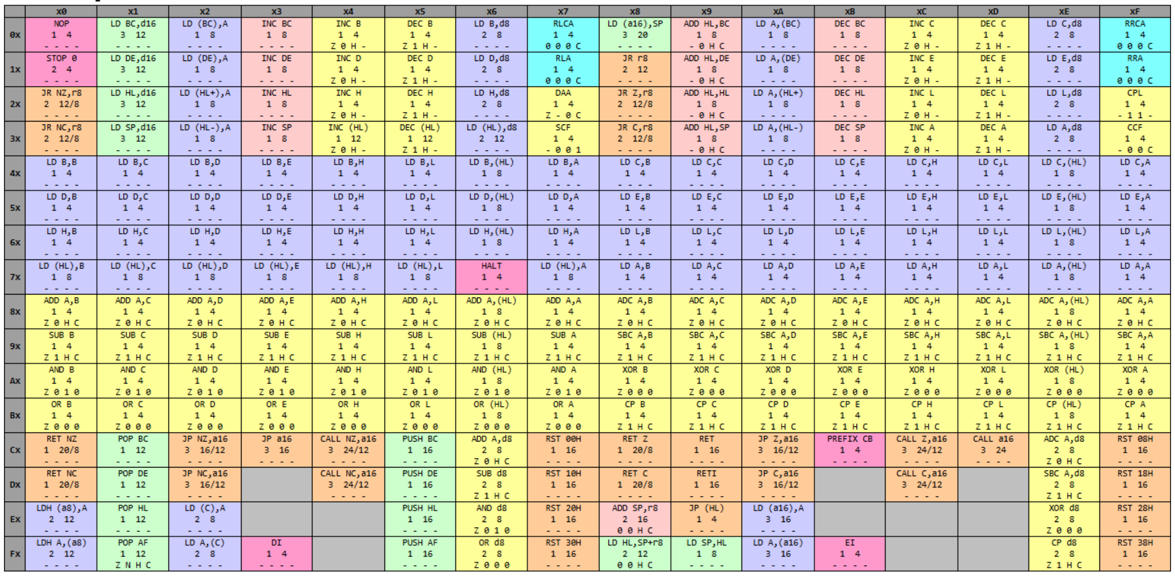
5.2 The Op-Tables

We can arrange all the instructions in tables, called opcode-tables, based on their identifier byte.
The Game Boy has 2 op-tables which are shown in Figure 1. Instructions with similar behaviors have been marked with the same color. Since a single byte (one op-table) was insufficient to cover all instructions, an additional table was made, which however uses 2-byte instructions, with the first one always being *0xCB* in hexadecimal (acting as a prefix), covering all bit operations.
By briefly examining the different instructions, you can see that most of them perform the following operations:

1. Loading values into registers
2. Adding and subtracting values between registers
3. Reading from and writing to memory
4. Comparing values and manipulating individual bits in registers

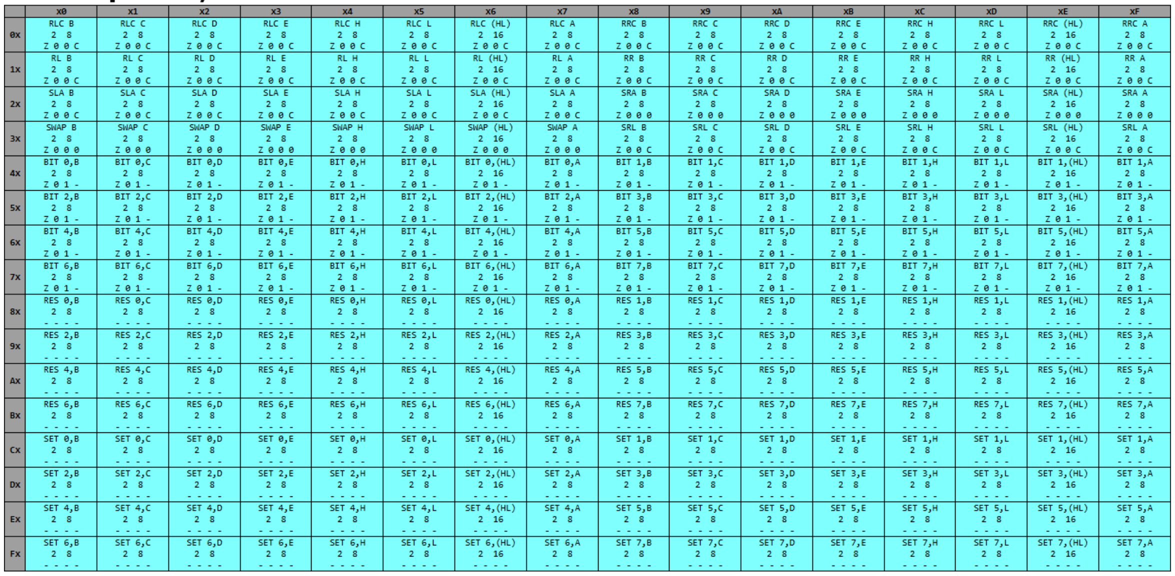
Obviously other instructions also do other kinds of operations but, as I have said above, most of them operate on the CPU registers.

8-bit opcodes



	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	NOP 1 4 - - - -	LD BC,d16 3 12 - - - -	LD (BC),A 1 8 - - - -	INC BC 1 8 - - - -	INC B 1 4 Z 0 H -	DEC B 1 4 Z 1 H -	LD B,d8 2 8 - - - -	RLCA 1 4 0 0 0 C	LD (a16),SP 3 20 - - - -	ADD HL,BC 1 8 - 0 H C	LD A,(BC) 1 8 - - - -	DEC BC 1 8 - - - -	INC C 1 4 Z 0 H -	DEC C 1 4 Z 1 H -	LD C,d8 2 8 - - - -	RRCA 1 4 0 0 0 C
1x	STOP 0 2 4 - - - -	LD DE,d16 3 12 - - - -	LD (DE),A 1 8 - - - -	INC DE 1 8 - - - -	INC D 1 4 Z 0 H -	DEC D 1 4 Z 1 H -	LD D,d8 2 8 - - - -	RLA 1 4 0 0 0 C	JR r8 2 12 - - - -	ADD HL,DE 1 8 - 0 H C	LD A,(DE) 1 8 - - - -	DEC DE 1 8 - - - -	INC E 1 4 Z 0 H -	DEC E 1 4 Z 1 H -	LD E,d8 2 8 - - - -	RRA 1 4 0 0 0 C
2x	JR NZ,r8 2 12/8 - - - -	LD HL,d16 3 12 - - - -	LD (HL+),A 1 8 - - - -	INC HL 1 8 - - - -	INC H 1 4 Z 0 H -	DEC H 1 4 Z 1 H -	LD H,d8 2 8 - - - -	DAA 1 4 Z - 0 C	JR Z,r8 2 12/8 - - - -	ADD HL,HL 1 8 - 0 H C	LD A,(HL+) 1 8 - - - -	DEC HL 1 8 - - - -	INC L 1 4 Z 0 H -	DEC L 1 4 Z 1 H -	LD L,d8 2 8 - - - -	CPL 1 4 - 1 1 -
3x	JR NC,r8 2 12/8 - - - -	LD SP,d16 3 12 - - - -	LD (HL-),A 1 8 - - - -	INC SP 1 8 - - - -	INC (HL) 1 12 Z 0 H -	DEC (HL) 1 12 Z 1 H -	LD (HL),d8 2 12 - - - -	SCF 1 4 - 0 0 1	JR C,r8 2 12/8 - - - -	ADD HL,SP 1 8 - 0 H C	LD A,(HL-) 1 8 - - - -	DEC SP 1 8 - - - -	INC A 1 4 Z 0 H -	DEC A 1 4 Z 1 H -	LD A,d8 2 8 - - - -	CCF 1 4 - 0 0 C
4x	LD B,B 1 4 - - - -	LD B,C 1 4 - - - -	LD B,D 1 4 - - - -	LD B,E 1 4 - - - -	LD B,H 1 4 - - - -	LD B,L 1 4 - - - -	LD B,(HL) 1 8 - - - -	LD B,A 1 4 - - - -	LD C,B 1 4 - - - -	LD C,C 1 4 - - - -	LD C,D 1 4 - - - -	LD C,E 1 4 - - - -	LD C,H 1 4 - - - -	LD C,L 1 4 - - - -	LD C,(HL) 1 8 - - - -	LD C,A 1 4 - - - -
5x	LD D,B 1 4 - - - -	LD D,C 1 4 - - - -	LD D,D 1 4 - - - -	LD D,E 1 4 - - - -	LD D,H 1 4 - - - -	LD D,L 1 4 - - - -	LD D,(HL) 1 8 - - - -	LD D,A 1 4 - - - -	LD E,B 1 4 - - - -	LD E,C 1 4 - - - -	LD E,D 1 4 - - - -	LD E,E 1 4 - - - -	LD E,H 1 4 - - - -	LD E,L 1 4 - - - -	LD E,(HL) 1 8 - - - -	LD E,A 1 4 - - - -
6x	LD H,B 1 4 - - - -	LD H,C 1 4 - - - -	LD H,D 1 4 - - - -	LD H,E 1 4 - - - -	LD H,H 1 4 - - - -	LD H,L 1 4 - - - -	LD H,(HL) 1 8 - - - -	LD H,A 1 4 - - - -	LD L,B 1 4 - - - -	LD L,C 1 4 - - - -	LD L,D 1 4 - - - -	LD L,E 1 4 - - - -	LD L,H 1 4 - - - -	LD L,L 1 4 - - - -	LD L,(HL) 1 8 - - - -	LD L,A 1 4 - - - -
7x	LD (HL),B 1 8 - - - -	LD (HL),C 1 8 - - - -	LD (HL),D 1 8 - - - -	LD (HL),E 1 8 - - - -	LD (HL),H 1 8 - - - -	LD (HL),L 1 8 - - - -	HALT 1 4 - - - -	LD (HL),A 1 8 - - - -	LD A,B 1 4 - - - -	LD A,C 1 4 - - - -	LD A,D 1 4 - - - -	LD A,E 1 4 - - - -	LD A,H 1 4 - - - -	LD A,L 1 4 - - - -	LD A,(HL) 1 8 - - - -	LD A,A 1 4 - - - -
8x	ADD A,B 1 4 Z 0 H C	ADD A,C 1 4 Z 0 H C	ADD A,D 1 4 Z 0 H C	ADD A,E 1 4 Z 0 H C	ADD A,H 1 4 Z 0 H C	ADD A,L 1 4 Z 0 H C	ADD A,(HL) 1 8 Z 0 H C	ADD A,A 1 4 Z 0 H C	ADC A,B 1 4 Z 0 H C	ADC A,C 1 4 Z 0 H C	ADC A,D 1 4 Z 0 H C	ADC A,E 1 4 Z 0 H C	ADC A,H 1 4 Z 0 H C	ADC A,L 1 4 Z 0 H C	ADC A,(HL) 1 8 Z 0 H C	ADC A,A 1 4 Z 0 H C
9x	SUB B 1 4 Z 1 H C	SUB C 1 4 Z 1 H C	SUB D 1 4 Z 1 H C	SUB E 1 4 Z 1 H C	SUB H 1 4 Z 1 H C	SUB L 1 4 Z 1 H C	SUB (HL) 1 8 Z 1 H C	SUB A 1 4 Z 1 H C	SBC A,B 1 4 Z 1 H C	SBC A,C 1 4 Z 1 H C	SBC A,D 1 4 Z 1 H C	SBC A,E 1 4 Z 1 H C	SBC A,H 1 4 Z 1 H C	SBC A,L 1 4 Z 1 H C	SBC A,(HL) 1 8 Z 1 H C	SBC A,A 1 4 Z 1 H C
Ax	AND B 1 4 Z 0 1 0	AND C 1 4 Z 0 1 0	AND D 1 4 Z 0 1 0	AND E 1 4 Z 0 1 0	AND H 1 4 Z 0 1 0	AND L 1 4 Z 0 1 0	AND (HL) 1 8 Z 0 1 0	AND A 1 4 Z 0 1 0	XOR B 1 4 Z 0 0 0	XOR C 1 4 Z 0 0 0	XOR D 1 4 Z 0 0 0	XOR E 1 4 Z 0 0 0	XOR H 1 4 Z 0 0 0	XOR L 1 4 Z 0 0 0	XOR (HL) 1 8 Z 0 0 0	XOR A 1 4 Z 0 0 0
Bx	OR B 1 4 Z 0 0 0	OR C 1 4 Z 0 0 0	OR D 1 4 Z 0 0 0	OR E 1 4 Z 0 0 0	OR H 1 4 Z 0 0 0	OR L 1 4 Z 0 0 0	OR (HL) 1 8 Z 0 0 0	OR A 1 4 Z 0 0 0	CP B 1 4 Z 1 H C	CP C 1 4 Z 1 H C	CP D 1 4 Z 1 H C	CP E 1 4 Z 1 H C	CP H 1 4 Z 1 H C	CP L 1 4 Z 1 H C	CP (HL) 1 8 Z 1 H C	CP A 1 4 Z 1 H C
Cx	RET NZ 1 20/8 - - - -	POP BC 1 12 - - - -	JP NZ,a16 3 16/12 - - - -	JP a16 3 16 - - - -	CALL NZ,a16 3 24/12 - - - -	PUSH BC 1 16 - - - -	ADD A,d8 2 8 Z 0 H C	RST 00H 1 16 - - - -	RET Z 1 20/8 - - - -	RET 1 16 - - - -	JP Z,a16 3 16/12 - - - -	PREFIX CB 1 4 - - - -	CALL Z,a16 3 24/12 - - - -	CALL a16 3 24 - - - -	ADC A,d8 2 8 Z 0 H C	RST 08H 1 16 - - - -
Dx	RET NC 1 20/8 - - - -	POP DE 1 12 - - - -	JP NC,a16 3 16/12 - - - -		CALL NC,a16 3 24/12 - - - -	PUSH DE 1 16 - - - -	SUB d8 2 8 Z 1 H C	RST 10H 1 16 - - - -	RET C 1 20/8 - - - -	RETI 1 16 - - - -	JP C,a16 3 16/12 - - - -		CALL C,a16 3 24/12 - - - -		SBC A,d8 2 8 Z 1 H C	RST 18H 1 16 - - - -
Ex	LDH (a8),A 2 12 - - - -	POP HL 1 12 - - - -	LD (C),A 2 8 - - - -			PUSH HL 1 16 - - - -	AND d8 2 8 Z 0 1 0	RST 20H 1 16 - - - -	ADD SP,r8 2 16 0 0 H C	JP (HL) 1 4 - - - -	LD (a16),A 3 16 - - - -				XOR d8 2 8 Z 0 0 0	RST 28H 1 16 - - - -
Fx	LDH A,(a8) 2 12 - - - -	POP AF 1 12 Z N H C	LD A,(C) 2 8 - - - -	DI 1 4 - - - -		PUSH AF 1 16 - - - -	OR d8 2 8 Z 0 0 0	RST 30H 1 16 - - - -	LD HL,SP+r8 2 12 0 0 H C	LD SP,HL 1 8 - - - -	LD A,(a16) 3 16 - - - -	EI 1 4 - - - -			CP d8 2 8 Z 1 H C	RST 38H 1 16 - - - -

16-bit opcodes, where the first 8 bits are 0xCB



	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	RLC B 2 8 Z 0 0 C	RLC C 2 8 Z 0 0 C	RLC D 2 8 Z 0 0 C	RLC E 2 8 Z 0 0 C	RLC H 2 8 Z 0 0 C	RLC L 2 8 Z 0 0 C	RLC (HL) 2 16 Z 0 0 C	RLC A 2 8 Z 0 0 C	RRC B 2 8 Z 0 0 C	RRC C 2 8 Z 0 0 C	RRC D 2 8 Z 0 0 C	RRC E 2 8 Z 0 0 C	RRC H 2 8 Z 0 0 C	RRC L 2 8 Z 0 0 C	RRC (HL) 2 16 Z 0 0 C	RRC A 2 8 Z 0 0 C
1x	RL B 2 8 Z 0 0 C	RL C 2 8 Z 0 0 C	RL D 2 8 Z 0 0 C	RL E 2 8 Z 0 0 C	RL H 2 8 Z 0 0 C	RL L 2 8 Z 0 0 C	RL (HL) 2 16 Z 0 0 C	RL A 2 8 Z 0 0 C	RR B 2 8 Z 0 0 C	RR C 2 8 Z 0 0 C	RR D 2 8 Z 0 0 C	RR E 2 8 Z 0 0 C	RR H 2 8 Z 0 0 C	RR L 2 8 Z 0 0 C	RR (HL) 2 16 Z 0 0 C	RR A 2 8 Z 0 0 C
2x	SLA B 2 8 Z 0 0 C	SLA C 2 8 Z 0 0 C	SLA D 2 8 Z 0 0 C	SLA E 2 8 Z 0 0 C	SLA H 2 8 Z 0 0 C	SLA L 2 8 Z 0 0 C	SLA (HL) 2 16 Z 0 0 C	SLA A 2 8 Z 0 0 C	SRA B 2 8 Z 0 0 0	SRA C 2 8 Z 0 0 0	SRA D 2 8 Z 0 0 0	SRA E 2 8 Z 0 0 0	SRA H 2 8 Z 0 0 0	SRA L 2 8 Z 0 0 0	SRA (HL) 2 16 Z 0 0 0	SRA A 2 8 Z 0 0 0
3x	SWAP B 2 8 Z 0 0 0	SWAP C 2 8 Z 0 0 0	SWAP D 2 8 Z 0 0 0	SWAP E 2 8 Z 0 0 0	SWAP H 2 8 Z 0 0 0	SWAP L 2 8 Z 0 0 0	SWAP (HL) 2 16 Z 0 0 0	SWAP A 2 8 Z 0 0 0	SRL B 2 8 Z 0 0 C	SRL C 2 8 Z 0 0 C	SRL D 2 8 Z 0 0 C	SRL E 2 8 Z 0 0 C	SRL H 2 8 Z 0 0 C	SRL L 2 8 Z 0 0 C	SRL (HL) 2 16 Z 0 0 C	SRL A 2 8 Z 0 0 C
4x	BIT 0,B 2 8 Z 0 1 -	BIT 0,C 2 8 Z 0 1 -	BIT 0,D 2 8 Z 0 1 -	BIT 0,E 2 8 Z 0 1 -	BIT 0,H 2 8 Z 0 1 -	BIT 0,L 2 8 Z 0 1 -	BIT 0,(HL) 2 16 Z 0 1 -	BIT 0,A 2 8 Z 0 1 -	BIT 1,B 2 8 Z 0 1 -	BIT 1,C 2 8 Z 0 1 -	BIT 1,D 2 8 Z 0 1 -	BIT 1,E 2 8 Z 0 1 -	BIT 1,H 2 8 Z 0 1 -	BIT 1,L 2 8 Z 0 1 -	BIT 1,(HL) 2 16 Z 0 1 -	BIT 1,A 2 8 Z 0 1 -
5x	BIT 2,B 2 8 Z 0 1 -	BIT 2,C 2 8 Z 0 1 -	BIT 2,D 2 8 Z 0 1 -	BIT 2,E 2 8 Z 0 1 -	BIT 2,H 2 8 Z 0 1 -	BIT 2,L 2 8 Z 0 1 -	BIT 2,(HL) 2 16 Z 0 1 -	BIT 2,A 2 8 Z 0 1 -	BIT 3,B 2 8 Z 0 1 -	BIT 3,C 2 8 Z 0 1 -	BIT 3,D 2 8 Z 0 1 -	BIT 3,E 2 8 Z 0 1 -	BIT 3,H 2 8 Z 0 1 -	BIT 3,L 2 8 Z 0 1 -	BIT 3,(HL) 2 16 Z 0 1 -	BIT 3,A 2 8 Z 0 1 -
6x	BIT 4,B 2 8 Z 0 1 -	BIT 4,C 2 8 Z 0 1 -	BIT 4,D 2 8 Z 0 1 -	BIT 4,E 2 8 Z 0 1 -	BIT 4,H 2 8 Z 0 1 -	BIT 4,L 2 8 Z 0 1 -	BIT 4,(HL) 2 16 Z 0 1 -	BIT 4,A 2 8 Z 0 1 -	BIT 5,B 2 8 Z 0 1 -	BIT 5,C 2 8 Z 0 1 -	BIT 5,D 2 8 Z 0 1 -	BIT 5,E 2 8 Z 0 1 -	BIT 5,H 2 8 Z 0 1 -	BIT 5,L 2 8 Z 0 1 -	BIT 5,(HL) 2 16 Z 0 1 -	BIT 5,A 2 8 Z 0 1 -
7x	BIT 6,B 2 8 Z 0 1 -	BIT 6,C 2 8 Z 0 1 -	BIT 6,D 2 8 Z 0 1 -	BIT 6,E 2 8 Z 0 1 -	BIT 6,H 2 8 Z 0 1 -	BIT 6,L 2 8 Z 0 1 -	BIT 6,(HL) 2 16 Z 0 1 -	BIT 6,A 2 8 Z 0 1 -	BIT 7,B 2 8 Z 0 1 -	BIT 7,C 2 8 Z 0 1 -	BIT 7,D 2 8 Z 0 1 -	BIT 7,E 2 8 Z 0 1 -	BIT 7,H 2 8 Z 0 1 -	BIT 7,L 2 8 Z 0 1 -	BIT 7,(HL) 2 16 Z 0 1 -	BIT 7,A 2 8 Z 0 1 -
8x	RES 0,B 2 8 - - - -	RES 0,C 2 8 - - - -	RES 0,D 2 8 - - - -	RES 0,E 2 8 - - - -	RES 0,H 2 8 - - - -	RES 0,L 2 8 - - - -	RES 0,(HL) 2 16 - - - -	RES 0,A 2 8 - - - -	RES 1,B 2 8 - - - -	RES 1,C 2 8 - - - -	RES 1,D 2 8 - - - -	RES 1,E 2 8 - - - -	RES 1,H 2 8 - - - -	RES 1,L 2 8 - - - -	RES 1,(HL) 2 16 - - - -	RES 1,A 2 8 - - - -
9x	RES 2,B 2 8 - - - -	RES 2,C 2 8 - - - -	RES 2,D 2 8 - - - -	RES 2,E 2 8 - - - -	RES 2,H 2 8 - - - -	RES 2,L 2 8 - - - -	RES 2,(HL) 2 16 - - - -	RES 2,A 2 8 - - - -	RES 3,B 2 8 - - - -	RES 3,C 2 8 - - - -	RES 3,D 2 8 - - - -	RES 3,E 2 8 - - - -	RES 3,H 2 8 - - - -	RES 3,L 2 8 - - - -	RES 3,(HL) 2 16 - - - -	RES 3,A 2 8 - - - -
Ax	RES 4,B 2 8 - - - -	RES 4,C 2 8 - - - -	RES 4,D 2 8 - - - -	RES 4,E 2 8 - - - -	RES 4,H 2 8 - - - -	RES 4,L 2 8 - - - -	RES 4,(HL) 2 16 - - - -	RES 4,A 2 8 - - - -	RES 5,B 2 8 - - - -	RES 5,C 2 8 - - - -	RES 5,D 2 8 - - - -	RES 5,E 2 8 - - - -	RES 5,H 2 8 - - - -	RES 5,L 2 8 - - - -	RES 5,(HL) 2 16 - - - -	RES 5,A 2 8 - - - -
Bx	RES 6,B 2 8 - - - -	RES 6,C 2 8 - - - -	RES 6,D 2 8 - - - -	RES 6,E 2 8 - - - -	RES 6,H 2 8 - - - -	RES 6,L 2 8 - - - -	RES 6,(HL) 2 16 - - - -	RES 6,A 2 8 - - - -	RES 7,B 2 8 - - - -	RES 7,C 2 8 - - - -	RES 7,D 2 8 - - - -	RES 7,E 2 8 - - - -	RES 7,H 2 8 - - - -	RES 7,L 2 8 - - - -	RES 7,(HL) 2 16 - - - -	RES 7,A 2 8 - - - -
Cx	SET 0,B 2 8 - - - -	SET 0,C 2 8 - - - -	SET 0,D 2 8 - - - -	SET 0,E 2 8 - - - -	SET 0,H 2 8 - - - -	SET 0,L 2 8 - - - -	SET 0,(HL) 2 16 - - - -	SET 0,A 2 8 - - - -	SET 1,B 2 8 - - - -	SET 1,C 2 8 - - - -	SET 1,D 2 8 - - - -	SET 1,E 2 8 - - - -	SET 1,H 2 8 - - - -	SET 1,L 2 8 - - - -	SET 1,(HL) 2 16 - - - -	SET 1,A 2 8 - - - -
Dx	SET 2,B 2 8 - - - -	SET 2,C 2 8 - - - -	SET 2,D 2 8 - - - -	SET 2,E 2 8 - - - -	SET 2,H 2 8 - - - -	SET 2,L 2 8 - - - -	SET 2,(HL) 2 16 - - - -	SET 2,A 2 8 - - - -	SET 3,B 2 8 - - - -	SET 3,C 2 8 - - - -	SET 3,D 2 8 - - - -	SET 3,E 2 8 - - - -	SET 3,H 2 8 - - - -	SET 3,L 2 8 - - - -	SET 3,(HL) 2 16 - - - -	SET 3,A 2 8 - - - -
Ex	SET 4,B 2 8 - - - -	SET 4,C 2 8 - - - -	SET 4,D 2 8 - - - -	SET 4,E 2 8 - - - -	SET 4,H 2 8 - - - -	SET 4,L 2 8 - - - -	SET 4,(HL) 2 16 - - - -	SET 4,A 2 8 - - - -	SET 5,B 2 8 - - - -	SET 5,C 2 8 - - - -	SET 5,D 2 8 - - - -	SET 5,E 2 8 - - - -	SET 5,H 2 8 - - - -	SET 5,L 2 8 - - - -	SET 5,(HL) 2 16 - - - -	SET 5,A 2 8 - - - -
Fx	SET 6,B 2 8 - - - -	SET 6,C 2 8 - - - -	SET 6,D 2 8 - - - -	SET 6,E 2 8 - - - -	SET 6,H 2 8 - - - -	SET 6,L 2 8 - - - -	SET 6,(HL) 2 16 - - - -	SET 6,A 2 8 - - - -	SET 7,B 2 8 - - - -	SET 7,C 2 8 - - - -	SET 7,D 2 8 - - - -	SET 7,E 2 8 - - - -	SET 7,H 2 8 - - - -	SET 7,L 2 8 - - - -	SET 7,(HL) 2 16 - - - -	SET 7,A 2 8 - - - -

Figure 1: Game Boy opcode-tables

It is important to say that some operations depend of results coming from previous instructions. These results are saved in the so called *flags*. Each flag would be represented by a single bit, which is set to 1 when active, and all the flags are stored together inside the F register. Later, we will see that for simplicity I chose to use a separate variable for each flag, instead of using a single F variable.
These flags are:

Bit*	Name	Description
7	zf	Zero flag
6	n	Add/sub flag
5	h	Half carry flag
4	cy	Carry flag
3-0	-	Not used

*bit position inside the F register

Table 3: DMG-CPU flags

1. **Zero flag**
   Set if the result of an operation is 0
   Used for conditional jumps
2. **Add/sub flag**
   1 if the previous operation was an addition, 0 if it was a subtraction
   Used for DAA instructions only
3. **Half carry flag**
   Set when there is a carry between the lower 4 bits of the operands during an arithmetic operation. It indicates that the lower nibble (4 bits) has overflowed.
4. **Carry flag**
   Set when an arithmetic operation causes a carry beyond the most significant bit of a byte (either the first or the second one in 16-bit operations) in addition, or a borrow when subtracting. Also set when a rotate/shift operation has shifted out a 1.

Instructions also can take different amount of clock cycles to execute.

5.3 Implementation

Thus, the first task I had to do was to implement every single instruction shown above, so that my virtual CPU would imitate the original Game Boy processor behavior.

```cpp
struct CPU {
    Byte A, B, C, D, E, H, L;

    Word SP;
    Word PC;

    // flags
    Byte z, n, h, c, IME;

    Byte fetch_byte(u32& cycles, Memory& mem);
    Word fetch_word(u32& cycles, Memory& mem);
    Byte read_byte(Word addr, u32& cycles, Memory& mem);
    void write_byte(Word addr, Byte data, u32& cycles, Memory& mem);
    void write_word(Word addr, Word data, u32& cycles, Memory& mem);

    // ... Functions to execute different bit manipulations

    // ... List of all instructions written as
    // static constexpr Byte INS_[INSTRUCTION] = [OP-CODE];

    // ... Functions to handle interrups (we will discuss them later)

    // Executes an instruction
    void exec_op(u32&, Memory&);

    // Gets called by the main loop
    u32 execute(Memory& mem);
}
```

This is the CPU structure, as you can see I defined all the registers and flags and I also implemented some utility functions.
The two functions we need to focus on now are the **execute** and the **exec_op** function.

The **execute** function is called by the main loop and, besides executing an instruction, it also handles interrupts.

```cpp
u32 CPU::execute(Memory& mem) {
    u32 cycles = 0;

    handle_interrupts(mem);
    exec_op(cycles, mem);

    // Handling the cartdrige after the boot program is done (we
    will see it later)
    if (PC == 0x100 && is_boot) {
        for (int i = 0; i < 0x100; ++i) {
            mem[i] = rom_first[i];
        }
        is_boot = false;
    }

    return cycles;
}
```

While the **exec_op** is responsible for handling the operations.

```cpp
void CPU::exec_op(u32 &cycles, Memory& mem) {
    switch (Byte ins = fetch_byte(cycles, mem)) {
        case INS_LD_BL: {
            B = L;
            break;
        }
        case INS_LD_BHL: {
            Word addr = L | (H << 8);
            B = read_byte(addr, cycles, mem);
            break;
        }
        case INS_LD_BA: {
            B = A;
            break;
        }
        case INS_LD_BN: {
            B = fetch_byte(cycles, mem);
            break;
        }
        case INS_ADD_AB: {
            n = 0;
            h = ((A & 0xF) + (B & 0xF)) > 0xF;
            c = (u32)((A & 0xFF) + (B & 0xFF)) > 0xFF;
            A += B;
            z = A == 0;
            break;
        }
        // Just some examples of instructions, you can see the
    whole implementation in cpu/cpu_ops.cpp
    }
}
```

6 Debugging the CPU

It was now time to test if my CPU worked, I decided to do so by giving the Game Boy boot program to my emulator and see how it would behave.

6.1 The Boot ROM

The Game Boy has a little program burned inside the CPU that gets executed when the console is powered on and, among other things, shows the Nintendo® logo. This code is exactly **256 bytes** and is stored in the first 256 addresses (from 0000-00FF in hexadecimal).
I decided to download the binary file and start disassemblying by myself and studying from scratch.
(DISASSEMBLY MAYBE)

6.2 Boot Code Analysis

For anyone interested, I will attach my disassembly along with some comments and thoughts I jotted down while studying it.
Anyways, here is what the code does:

1. Resets VRAM
2. Sets the audio to play the famous "ba-ding!" sound
3. Loads the Nintendo logo from the game cartridge into VRAM to display it on screen
4. Scrolls the logo
5. Checks if the Nintendo logo is correct by comparing it with its own version; if not, the Game Boys stops executing.

Note that the Nintendo logo is put in the Background layer–this detail will become clearer once we get to the PPU implementation.
For now, anyway, I would like to take a moment to focus on some peculiar things that are happening in this code that I have not been able to explain. The Game Boys contains the entire Nintendo logo (including the registered trademark), but it only displays the R symbol on screen, while the "Nintendo" text is loaded from the game cartridge. Additionally, the logo is displayed on screen **before** it is checked for correctness.

6.3 The Execution So Far

With this being said, I finally loaded the boot ROM into memory and started executing.

```
void CPU::load_bootup(const char *filename, Memory &mem)  {
    std::ifstream file;
    file.open(filename, std::ios::in | std::ios::binary);

    if (file.is_open()) {
        for (size_t i = 0; i < 0x100; ++i) {
            Byte value = 0;
            file.read((char*)&value, sizeof(char));
            mem[i] = value;
        }
    } else {
        std::cerr << "Failed to open file " << filename << std::
    endl;
    }
    file.close();
    is_boot = true;
}
```

I checked whether the registers that were supposed to be modified had the correct values to verify if my CPU implementation was accurate–and it was!
The only issue now is that execution stops between addresses **0x64** and **0x68**. Looking at my disassembly, I noticed that the code was looping until register **FF44** reached **0x90**. However, after examining the rest of the code, I saw that this register was never modified, meaning it must be a read-only register managed by another component. This component is the PPU (Pixel Processing Unit) which handles rendering on the display. Since the PPU is rather complex and long to implement, I decided to break it down into sections and follow the order in which I implemented it.
Before working on the PPU, though, I first implemented two simpler components.

7 Timers

As the name suggests, timers are in charge of measuring time and execute some code every certain time. One classic application that uses timers is a game where (pseudo) randomness is involved. We can get a random value every time we try to read the DIV register (the core counter) for example, because games execution follows an unpredictable order and because instructions take different amount of clock cycles to complete, the value in the DIV register will likely be at a different value each time.

7.1 Structure

The timer has **four** mapped registers, two of them are for counting, while the other two are for configuring them.

7.1.1 DIV

The DIV register is mapped to address **0xFF04** and is the core of the whole system. Internally, it is a 16-bit counter which is incremented every single clock cycle, although only the upper 8 bits are mapped to memory. The DIV register can be read from at anytime, writing to it will reset the whole 16-bit register to 0.

7.1.2 TIMA

TIMA is a little more complex and gives us the possibility to count at different rates. It is mapped to address **0xFF05** and can be configured using the two registers TMA and TAC.

7.1.3 TAC

This register controls the behavior of TIMA and is mapped to address **0xFF07**.

	7 6 5 4 3	2	1 0
TAC		Enable	Clock select

Table 4: TAC flags

Bit 2 just enables or disables TIMA's counting, while bits 1 and 0 set TIMA's incrementing frequency. Notice that 1 M-Cycle is equal to 4 clock cycles.

Clock select	Frequency
00	256 M-Cycles
01	4 M-Cycles
10	16 M-Cycles
11	64 M-Cycles

Table 5: TAC flags

7.1.4 TMA

TMA is mapped to register **0xFF06** When TIMA overflows, it is reset to the value stored in the TMA register and an interrupt is requested (we will them see later). An example of use can be the following: if TMA is set to 0xFF and the frequency set in TAC is 256 M-Cycles, some piece of code gets executed every 256 M-Cycles.

7.1.5 Timing Behaviors

When TIMA overflows, it does not get reset instantly. Instead, it contains a value of zero and waits for a duration of four clock cycles before it is updated. This update can be **aborted** by writing **any** value to TIMA during these four clock cycles. In this case, TIMA keeps the value that was written and an interrupt does **not** get requested. However, if TIMA is written on the **same** clock cycle on which the reload occurs, the write is ignored. While if TMA is written on the same clock cycle on which the reload occurs, TMA is updated **before** its value is loaded into TIMA.

7.2 Implementation

I decided not to implement these oddities, although I **did** implement the TIMA overflow abort.
The **update** function structure is the following.

```cpp
void Timers::update(u32 cycles, Memory& mem) {
    // the cycles parameter is the number of M-Cycles
    // Which then gets multiplied by 4 to get the number of clock
    cycles
    for (u32 i = 0; i < cycles * 4; ++i) {
        // if someone writes into DIV, the register gets reset to 0
        if (mem[DIV_REG] != ((DIV >> 8) & 0xFF))
            DIV = 0;

        // Incrementing DIV
        DIV++;
        mem[DIV_REG] = (DIV >> 8) & 0xFF;

        if (!tima_overflow) {
            // Check if TIMA needs to be incremented

            // ... condition logic

            if (is_increment) {
                const Byte tima_value = ++mem[TIMA_REG];
                if (tima_value == 0) {
                    tima_overflow = true;
                }
            }
        } else {
            // Handle TIMA overflow
            tima_overflow_cycles++;

            if (tima_overflow_cycles == 4) {
                mem[TIMA_REG] = mem[TMA_REG];
                mem[IF_REG] |= 1 << 2; // Calling interrupt
                tima_overflow = false;
                tima_overflow_cycles = 0;
            } else {
                if (mem[TIMA_REG] != 0) {
                    // overflow aborted
                    tima_overflow = false;
                    tima_overflow_cycles = 0;
                }
            }
        }
    }
}
```

8 Interrupts

An Interrupt is a signal sent to the CPU by other components, temporarily pausing the CPU current execution to handle an urgent task. Once the task is completed, the CPU resumes what it was doing. Interrupts are still present in modern processors and are used to notify the Operating System when an event occurrs. They can also be triggered by a subprogram to request OS services. For those familiar with assembly, calling an interrupt is equivalent to making a **syscall**, such as writing output to the terminal.

8.1 How They Work

When a component wants to trigger an interrupt, it sets the CPU interrupt pin to HIGH. If the CPU acknowledges the request, the component is then allowed to place an 8-bit vector on the data bus, specifying the type of interrupt that occurred.
It is important to specify that an interrupt is acknowledged **only** if the IME flag is set. IME (Interrupt Master Enable) is a flag internal to the CPU, it cannot be read in any way and can only be modified by some instructions or events. For example, IME is set to 0 (interrupts are disabled) while an interrupt routine is being executed, preventing new interrupts from being triggered until that routine finishes. A component requests an interrupt by writing to the **IF** (Interrupt Flag) register, where each bit represents a different type of interrupt. Additionally, the **IE** (Interrupt Enable) register, which is configurable by the programmer, specifies which interrupt the CPU should handle.
The CPU executes the interrupt handler **only** if:

1. The corresponding bit is set in both **IF** and **IE** registers.
2. **IME** is set to 1 (interrupts are enabled).

8.2 Types of Interrupts

These are the different types of interrupts, each with a specific address where the handler execution code begins.

8.2.1 V-Blank

This interrupt is requested every time the Game Boy enters the V-Blank mode, we will see it when we will talk about the PPU. It occurs around 59.7 times per second and starts at address **0x0040**.

8.2.2 LCD STAT

This interrupt can be configured using register **0xFF41** to choose when it should be triggered (again, we will talk about it with the PPU). A STAT interrupt will be triggered only if there is a transition from LOW to HIGH on the STAT interrupt line. Its handler address is **0x0048**.

8.2.3 Timer

As described in the Timers section, this interrupt occurs every time TIMA overflows. Its handler address is **0x0050**.

8.2.4 Serial

The serial interrupt is requested when a serial data transfer is completed. Two Game Boy systems could communicate using a link cable, this thing is not implemented in the emulator, thus the interrupt is never requested. Its handler address is **0x0058**.

8.2.5 Joypad

The Joypad interrupt is requested when any of the bits in the Joypad register changes from HIGH to LOW. As we will see later when we will talk about Joypad emulation, this happens when a button is pressed. Its handler address is **0x0060**.

8.3 IF and IE Registers

The IE register is located at **0xFFFF** and controls whether an interrupt handler may be called.

	7 6 5	4	3	2	1	0
IE		Joypad	Serial	Timer	LCD STAT	V-Blank

Table 6: IE flags

While the IF register is located at **0xFF0F** and controls whether an interrupt handle is being requested.

	7 6 5	4	3	2	1	0
IF		Joypad	Serial	Timer	LCD STAT	V-Blank

Table 7: IF flags

In case more than one interrupt is requested at the same time, the one with the highest priority is serviced first. The priorities follow the order of the bits in the IE and IF registers, with bit 0 (V-Blank) having the highest priority and bit 4 having the lowest one.

8.4 Implementation

This is the **handle_interrupts** function that, as we saw before, is called by the CPU **update** function.

```cpp
void CPU::handle_interrupts(Memory& mem) {
    if (IME == 0)
        return;

    Byte IE = mem[IE_REG];
    Byte IF = mem[IF_REG];

    if (is_vblank_int(IE, IF)) {
        ack_int(mem, 0);
        call_int(mem, 0x0040);
        return;
    }

    bool lcd = is_lcd_int(IE, IF);
    if (lcd && !old_lcd_int_flag) {
        ack_int(mem, 1);
        call_int(mem, 0x0040);
        old_lcd_int_flag = lcd;
        return;
    }
    old_lcd_int_flag = lcd;

    if (is_timer_int(IE, IF)) {
        ack_int(mem, 2);
        call_int(mem, 0x0050);
        return;
    }

    if (is_joypad_int(IE, IF)) {
        ack_int(mem, 4);
        call_int(mem, 0x0060);
    }
}
```

9 PPU – Very Basic Start

It was finally time to work on the PPU and ensure the boot ROM could continue executing properly.

9.1 General Introduction

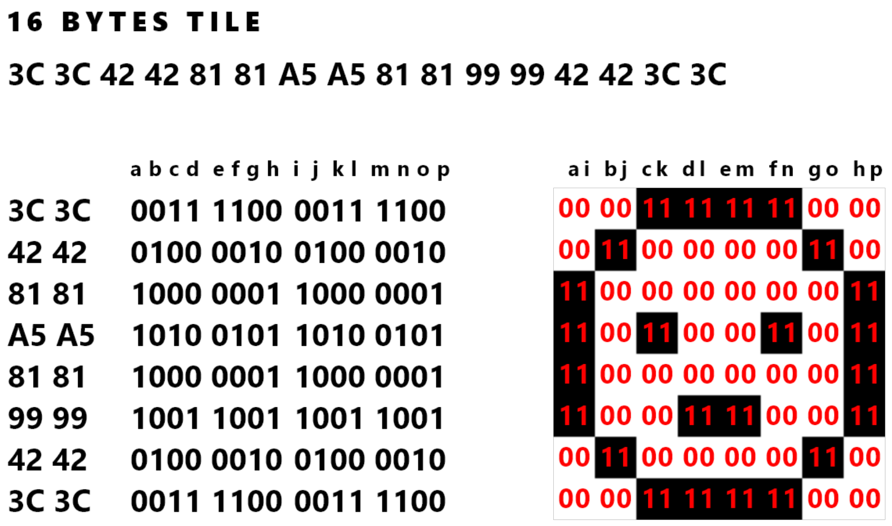
The PPU has been mentioned many times so far because it is the second most important component inside the Game Boy–and also the most complex, even more so than the CPU.
When I first began working on this component I did not aim to render anything on the screen immediately. My initial goal was merely to bypass the code segment that was blocking the boot execution, which involved a specific register. However, we first need some theory and some basic knowledge to understand how the PPU works.

9.1.1 The Screen

The original Game Boy display was a 160x144 pixel LCD (Liquid Crystal Display), which could display up to 4 different shades of gray.

9.1.2 Tiles

However, pixels could not be manipulated individually. Instead, they were grouped into **tiles**–fixed **8x8 pixel textures** that could be placed on the screen within a predefined grid. This resulted in a screen that was 20x18 tiles in size. Since there were only 4 shades of gray, only 2 bits per pixel were required, making each tile **16 bytes** (2 bytes per row). These tiles were stored in VRAM from address **0x8000** to **0x97FF**



16 BYTES TILE

3C 3C 42 42 81 81 A5 A5 81 81 99 99 42 42 3C 3C

	a b c d	e f g h	i j k l	m n o p
3C 3C	0011	1100	0011	1100
42 42	0100	0010	0100	0010
81 81	1000	0001	1000	0001
A5 A5	1010	0101	1010	0101
81 81	1000	0001	1000	0001
99 99	1001	1001	1001	1001
42 42	0100	0010	0100	0010
3C 3C	0011	1100	0011	1100

ai	bj	ck	dl	em	fn	go	hp
00	00	11	11	11	11	00	00
00	11	00	00	00	00	11	00
11	00	00	00	00	00	00	11
11	00	11	00	00	11	00	11
11	00	00	00	00	00	00	11
11	00	00	11	11	00	00	11
00	11	00	00	00	00	11	00
00	00	11	11	11	11	00	00

Figure 2: Tile explanation

9.1.3 Display Layers

But how do we use these tiles? The Game Boy organizes its display into three separate layers: the **Background**, the **Window** and the **Sprites**.

The Background is a 32x32 tile grid. Since the Game Boy can only display a 20x18 tile section, it can be scrolled using **SCX** and **SCY** registers, respectively Scroll X and Scroll Y registers.

The Window is another 32x32 tile grid that acts as an overlay over the Background. the position of this Window is determined using the **WX** and **WY** registers, Window X and Window Y respectively.

Sprites are the objects that move inside the screen, like the player character in a game. It can be either 1 tile or 2 tiles (8x16 pixels) and they are not limited by the Background or the Window grid. Due to memory size, there can be up to 40 sprites and up to 10 sprites per row.

9.1.4 Layers in Memory

But where are all of these layers stored?
The Background and Window are stored in two **Background Maps**, located at memory ranges **0x9800-0x9BFF** and **0x9C00-0x9FFF**. Which map represents the Window and which one the Background is configured using the **LCDC register**. Bytes in these maps are organized row by row, with each byte representing a tile via a **tile number**. The first byte corresponds to the top-left tile,

the next one to the tile to its right, and so on.
A **tile number** is merely an index pointing to a tile in the tile data region. There are **two methods** the Game Boy uses to interpret these tile numbers.

1. The **8000 method** treats the tile number as an **unsigned 8-bit integer** and uses it as an offset from the base address **0x8000**. For example, a tile number of 0 would point to the tile that starts at **0x8000**, a tile number of 1 would point to **0x8010** etc.

2. The **8800 method** instead treats the tile number as a **signed 8-bit integer** and offsets it from the base address **0x9000**. Hence, a tile number of 0 would point to **0x9000** while a tile number of 0xFF (-1) would point to **0x8FF0** etc.

The Background and the Window can be configured to use either the 8000 or 8800 method using the **LCDC register**, while the Sprites always use the 8000 one.
Sprites are stored in the region usually called OAM (Object Attribute Memory). This is because each sprite does not just have to include a tile number but it also needs to supply other information. This OAM section ranges from **0xFE00** to **0xFE9F**, each sprite entry is structured as follows:

1. **Byte 0 - Y-Position**

2. **Byte 1 - X-Position**

3. **Byte 2 - Tile number**

4. **Byte 3 - Sprite flags**

9.2 The Rendering Process

Now that we have a general understanding of how memory is managed by the PPU, we can start discussing the **rendering process**. Keep in mind that for this initial implementation, I did not aim to **actually** create a window and render tiles. Instead, I wanted to **simulate** rendering by updating the registers correctly. The goal was to just **"pretend"** rendering was happening in order to bypass the infinite loop we were stuck in. That is why what I am about to explain will not be exhaustive or complete–for now. More detailed explanations will follow later, in the same order I learned them myself. By the way, this kind of approach mirros how debugging often works in practice: you implement just enough for things to move forward, then come back to refine later.

9.2.1 How Tiles Are Rendered

Although the games organize graphics into tiles, the PPU does not actually set the pixels on a tile-by-tile basis. Instead, frames are rendered row by row—these rows are commonly referred to as **scanlines**. The number of the scanline the PPU is currently working on is stored in the **LY register**, mapped to address

0xFF44–exactly the register that was being checked in our infinite loop! The boot program was waiting for the PPU to process the 90th row.
Every time a scanline is processed, the PPU cycles through different **modes**. I have decided to explain them in the order they are executed during rendering, rather than by their mode identifier.

OAM Scan is entered at the beginning of every scanline (except for V-Blank), the PPU looks for sprites that are on the **current scanline** and stores them in a local buffer which can contain up to 10 sprites (this is why only 10 sprites per row are allowed). A sprite check happens every 2 clock cycles for a total amount of 80 clock cycles, since there can be up to 40 sprites.

Drawing is the mode in which the PPU actually transfers pixels to the LCD. Due to various factors, the duration of this mode is not fixed–it can take as little as 172 clock cycles, in the best case and up to 289 in the worst case.

H-Blank acts as a kind of padding to ensure each scanline takes exactly 456 clock cycles. Nothing significant happens here as the PPU just pauses.

V-Blank is a more interesting mode instead. As mentioned earlier, the Game Boy display has 144 visible pixel rows. However, the PPU processes a total of 154 scanlines. The remaining 10 scanlines are part of the V-Blank period. Each of these takes 456 clock cycles–just like a regular scanline–and, similar to H-Blank, nothing really happens. Notice that V-Blank is the only time when the CPU **can safely access VRAM** (although tecnically it can also be accessed during OAM scan and H-Blank). In later implementations we will see how games use this V-Blank period for **DMA transfers** and much more.

To make everything even clearer, I will include a diagram from the Game Boy Pandocs that visually illustrates the concepts explained above.

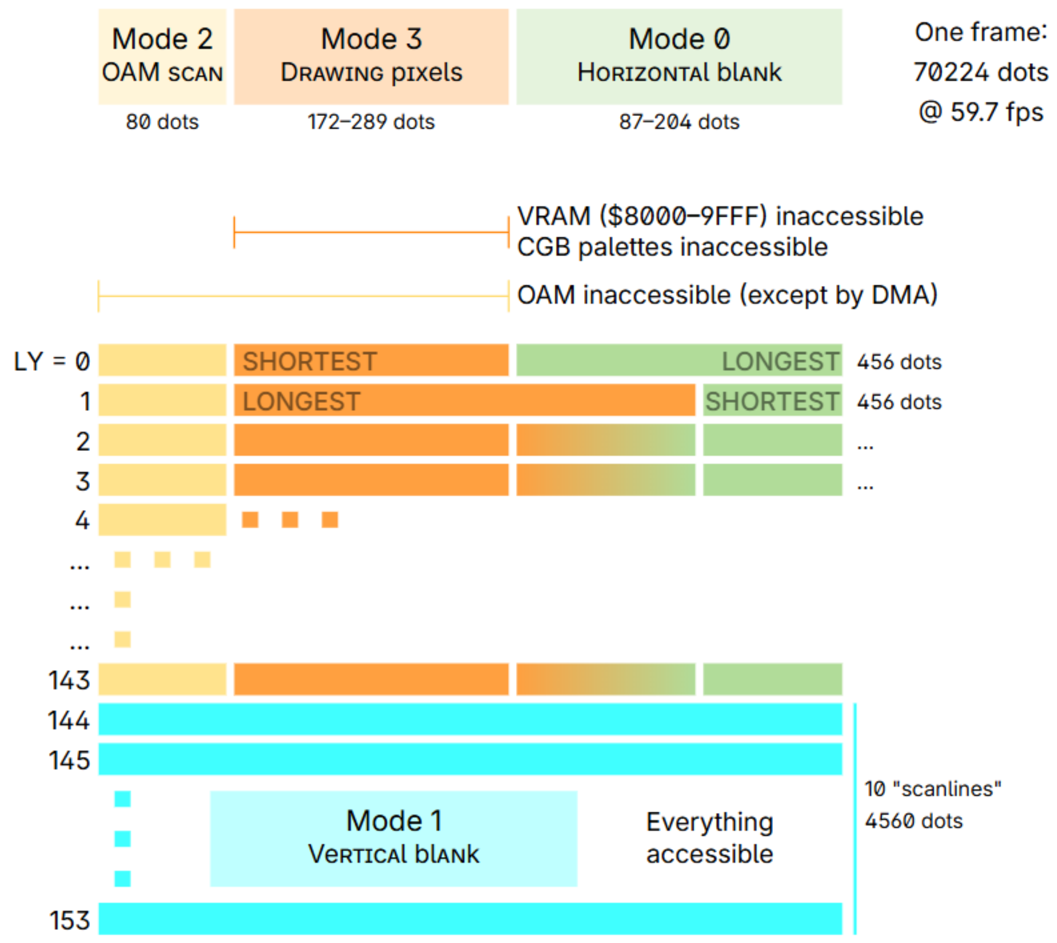


Figure 3: PPU modes timing diagram

9.2.2 PPU Registers

Now let's take a look at the registers that allow the processor to control the PPU.

LCD Control (LCDC) is mapped to address **0xFF40** and contains multiple flags that determine what is displayed and how everything is configured.

Bit*	Name	Description
7	LCD Display enable	Setting this bit to 0 **disables** the PPU and the screen entirely
6	Window Tile Map Select	If set to 1, the Window will use the background map located at **0x9C00-0x9FFF**; otherwise, it will use **0x9800-0x9BFF**.
5	Window Display Enable	Setting this bit to 0 completely hides the Window layer
4	Tile Data Select	Setting this bit to 1 will use the **8000 method**; otherwise, it will use the **8800 method**
3	Background Tile Map Select	Setting this bit to 1 will use the background map located at **0x9C00-0x9FFF**; otherwise, it will use **0x9800-0x9BFF**
2	Sprite Size	By setting this bit to 1, sprites will have a size of 1x2 tiles; otherwise, they will be 1x1 tiles
1	Sprite Enable	Setting this bit to 0 will completely disable sprites
0	Background/Window Enable	Setting this bit to 0 will completely disable the Background **and** the Window

*bit position inside the *LCDC* register

Table 8: LCD Control register flags

LCD Status is mapped to address **0xFF41** and provides both configuration options for the STAT interrupt (previously explained) and information about the current status of the PPU.

Bit*	Name	Description
7	Unused	Always 1
6	LYC=LY Condition	Fires a STAT interrupt whenever LY register value is equal to LYC register value
5	Mode 2 Condition	Fires a STAT interrupt whenever the PPU enters mode 0 (OAM scan)
4	Mode 1 Condition	Fires a STAT interrupt whenever the PPU enters mode 1 (V-Blank)
3	Mode 0 Condition	Fires a STAT interrupt whenever the PPU enters mode 0 (H-Blank)
2	Coincidence Flag	This bit is set 1 if the value in LY register is equal to the value in LYC register
1-0	PPU mode	indicates (in binary) which mode the PPU is currently working on

Table 9: LCD Status register flags

Bits 6-3 are used to enable specific conditions under which a STAT interrupt can be requested, while bits 2-0 reflect the current status of the PPU, allowing the CPU to know what the PPU is doing.

SCY, SCX are located at memory addresses **0xFF42** and **0xFF43**, respectively, and specify the top-left coordinates of the visible area within the Background map.

WY, WX are located at memory addresses **0xFF4A** and **0xFF4B**, respectively, and specify the top-left coordinates of the Window map.

BGP is mapped to address **0xFF47** and assigns gray shades to the color indices of the Background and Window tiles.

	7-6	5-4	3-2	1-0
Colors	**ID 3**	**ID 2**	**ID 1**	**ID 0**

Each of the two-bit identifiers map to a color, with 0 being completely white and 3 being completely black.

OBP0, OBP1 are located at memory addresses **0xFF48** and **0xFF49**, respectively, and contains the two palette data for sprites. These registers work exactly like BGP, except for the lower two bits, which are ignored since color index 0 is always transparent for sprites.

LY as we said earlier is the register that specifies which scanline the PPU is currently working on and is mapped to address **0xFF44**

LYC (LY Compare), as we saw earlier, is the register used to set the textbf-Coincidence flag inside LCDS register and is mapped to **0xFF45**.

9.3 First Implementation

To begin, we will define the structure of our PPU class.

```
class PPU {
    int mode = 0;
    u32 total_cycles = 0;

    // Registers addresses
    static constexpr Word IF_REG = 0xFF0F;
    // ...
public:
    void init();
    static void request_stat_interrupt(Memory&);

    // We will just update stuff for now
    void update(u32, Memory&);

    int get_mode() const { return mode; }
};
```

As the **init** function just initializes some variables, the only important function for now is **update**.

```cpp
// Here the cycles parameter is the number of M-Cycles (4 clock
    cycles)
void update(u32 cycles, Memory& mem) {
    // Check if LCD Control enable flag is set
    Byte LCDC = mem[LCDC_REG];
    if ((LCDC & 0x80) == 0) return;

    switch (mode) {
        case 2: {
            // OAM scan mode
            // Right now, it just increments total_cycles
            oam_scan(cycles, mem);
            break;
        }
        case 3: {
            // Drawing mode
            // Right now, it just increments total_cycles
            draw(cycles, mem);
            break;
        }
        case 0: {
            // H-Blank mode
            for (u32 i = 0; i < cycles / 2; ++i) {
                total_cycles += 2;
                // Doing nothing until we hit 456 clock cycles
                if (total_cycles >= 456) {
                    total_cycles = 0;
                    // Finally incrementing LY
                    mem[LY_REG]++;
                    if (mem[LY_REG] >= 144) {
                        mode = 1;
                    } else {
                        mode = 2;
                    }
                }
            }
            break;
        }
        case 1: {
            //V-Blank mode
            for (u32 i = 0; i < cycles / 2; ++i) {
                total_cycles += 2;
                if (total_cycles >= 456) {
                    total_cycles = 0;
                    mem[LY_REG]++;
                    if (mem[LY_REG] >= 154) {
                        mode = 2;
                    }
                }
            }
            break;
        }
    }


    Byte LY = mem[LY_REG];
    Byte LYC = mem[LYC_REG];

    // Updating LCD Status register
    Byte value = mem[LCDS_REG] & (~0b111);
    value |= (LY == LYC) << 2;
    value |= mode & 0b11;
    mem[LCDS_REG] = value;

    // Firing STAT interrupts
    const Byte status = mem[LCDS_REG];
    if (status & (1 << 6) && LY == LYC) {
        request_stat_interrupt(mem);
    }
    if (status & (1 << 5) && mode == 2) {
        request_stat_interrupt(mem);
    }
    if (status & (1 << 4) && mode == 1) {
        request_stat_interrupt(mem);
    }
    if (status & (1 << 3) && mode == 0) {
        request_stat_interrupt(mem);
    }
}
```

As you can see, at lines 28 and 44, we are finally incrementing LY.

10 The Execution So Far

Running the boot program at this stage successfully executes the Nintendo logo scrolling section. However, it gets stuck in another infinite loop–this time during the logo check. Since we have not yet emulated the game cartridge, the comparison cannot succeed. Though, we still managed to make meaningful progress. That said, there is no time to celebrate just yet–there is still plenty of work ahead.

11 PPU – Further Progress

Finally, after putting it off for quite some time, it is finally time to implement the screen and begin rendering at least the Background layer. To do so, we first need to understand how the PPU pushes the pixels into the screen.

11.1 The Pixel FIFO

We have already established that the PPU constructs each frame on a scanline basis. The final piece to understand is how these pixels are transferred to the LCD. We will see in 11.2 that during the **drawing phase** tiles are collected singularly, and for each of them the PPU transitions through different phases. At the core of this process, though, are two queues that temporarily store the pixel data before it is sent to the screen. I am talking about the **Pixel FIFOs**, FIFO stands for First In First Out, meaning that the first pixel to be pushed into this queue is the first that gets popped out to be rendered. We will see that this

method of rendering is not ideal, since it implies a lot of timings restrictions that must be followed, slowing down the rendering process. Although the hardware limitations of that time could not offer an alternative solution.
There are two pixel FIFOs, one for the Background (or Window) layer, and one for Sprites. Each FIFO can hold up to 16 pixels (2 tile rows) which are only popped out under specific conditions.
Each pixel data chunk keeps track of:

1. Color number
2. Palette (whether OBP0 or OBP1)
3. Background priority (only relevant for sprites)

11.2 Fetching the Background

The process of fetching pixels is divided into 4 steps, each one taking 2 clock cycles.

1. During this first step, the **Pixel Fetcher** fetches and stores the tile number of the corresponding tile to be used, following the configurations in the LCDC register.
2. Using the tile number provided, the first byte of tile data is now fetched and stored internally.
3. The second byte of tile data is fetched in this third step.
4. At this stage, the Pixel Fetcher finally attempts to push the decoded pixels into the LCD. However, this action is only carried out if the **Background FIFO** is **entirely empty**. If not, the operation is deferred and retried each clock cycle until the FIFO is cleared.

Here we encounter the first example of those timing restrictions I mentioned earlier. The final step requires the FIFO to be completely empty. If the FIFO happens to be full at this point, it would take **8 clock cycles** (see 11.3) to clear it, whereas only 6 clock cycles were needed to reach this stage. Meaning this step must be retried at least twice before it can successfully complete.

11.3 Pushing Pixels

During **each clock cycle**, the PPU attempts to push a pixel to the LCD. This can only proceed if **there are elements** present in the **Background FIFO**. Merging pixels involves combining one from the Background FIFO with another from the Sprite FIFO–if available. However, for this initial implementation, only pixels from the Background are processed.

11.4 Second Implementation

Let's now take a look at how I chose to implement everything discussed so far.

11.4.1 The Update Method

In addition to what was happening before in the **update** method, we are now switching through **real modes** that actually do something.

```
void PPU::update(u32 cycles, Memory& mem, LCD& lcd) {
    for (u32 i = 0; i < cycles / 2; ++i) {
        switch (mode) {
            case 2: {
                oam_scan(2, mem);
                break;
            }
            case 3: {
                draw(2, mem);
                break;
            }
            case 0: { // Simulating H-Blank
                total_cycles += 2;
                if (total_cycles >= 456) {
                    sprites_buffer.clear();
                    total_cycles = 0;
                    mem[LY_REG]++;
                    if (mem[LY_REG] >= 144) {
                        mode = 1; // Entering VBlank
                    } else {
                        mode = 2;
                    }
                }
                break;
            }
            case 1: { //Simulating V-Blank
                total_cycles += 2;
                if (total_cycles >= 456) {
                    total_cycles = 0;
                    mem[LY_REG]++;
                    if (mem[LY_REG] >= 154) {
                        mem[LY_REG] = 0;
                        lcd.reset();
                        mode = 2;
                        x_pos_counter = 0;
                        background_fifo.empty();
                        frames_count++;
                    }
                }
                break;
            }
        }
        // Mixing 2 pixels every 2 clock cycles
        pop_and_mix_to_lcd(2, lcd);
    }
}
```

11.4.2 Draw and Push Methods

```cpp
void PPU::background_draw(Memory& mem) {
    switch (drawing_step) {
        case 1: {
            background_fetch_tile_number(mem);
            break;
        }
        case 2: {
            background_fetch_tile_data_low(mem);
            break;
        }
        case 3: {
            background_fetch_tile_data_high(mem);
            break;
        }
        case 4: {
            background_push_tile_data(mem);
            break;
        }
        default: return;
    }
}

void PPU::draw(u32 cycles, Memory& mem) {
    for (u32 i = 0; i < cycles / 2; ++i) {
        background_draw(mem);
    }
}

void PPU::pop_and_mix_to_lcd(u32 cycles, LCD& lcd) {
    if (sprites_drawing_step > 1) return;

    for (u32 i = 0; i < cycles; ++i) {
        // Skipping if Background FIFO is empty
        if (background_fifo.is_empty()) return;

        Pixel pixel = background_fifo.pop();
        lcd.push(pixel);
    }
}
```

11.4.3 The LCD Class

I still have not clearified what the **LCD class**–used throughtout these methods–actually does. Following the structure of the real Game Boy, I wanted to separate the LCD from the PPU. As a result, the LCD has its own class, although it is quite simple and straightforward.

```cpp
class LCD {
    Byte* display = nullptr;

    int pos = 0;
public:
    void init();

    inline Byte* get_display() const { return display; }
    inline bool is_ready() const { return pos >= (DISPLAY_WIDTH *
    DISPLAY_HEIGHT) - 1; }

    void push(Pixel pixel);
    void reset();
};

void LCD::init() {
    display = (Byte*)malloc(sizeof(Byte) * (DISPLAY_WIDTH *
    DISPLAY_HEIGHT));

    for (int y = 0; y < DISPLAY_HEIGHT; y++) {
        for (int x = 0; x < DISPLAY_WIDTH; x++) {
            display[y * DISPLAY_WIDTH + x] = 0;
        }
    }
}

void LCD::push(Pixel pixel) {
    display[pos++] = VideoUtils::palette_to_color(pixel);
}

void LCD::reset() {
    pos = 0;
}
```

As we can see here, we merely keep a **byte array** of pixels, each storing the color of a pixel, ready to be rendered on our emulated window.

12 Creating a Virtual Window

The next step we need to take is displaying these pixels in a window, which will require a graphics library. What I considered at first was using **plain OpenGL**, which would have been a good excuse to finally learn how to use it. However, OpenGL is built around rendering using **vertices and polygons**. What I thought would be a better approach is to edit individual pixels instead, as we are currently doing with the LCD byte array. I found out that **SDL** makes this possible, thanks to a low-level API that gives us direct access to the pixel array that will be rendered on screen. For this project, I am using the latest SDL version at the time I am writing this paper, which is **SDL 3.2.10**.

12.1 CPU Window Implementation

At first, I just needed to check if the copyright symbol was showing up correctly and progressively scrolling from the top to the center of the screen. Thus, for now, a very inefficient, non-hardware-accelerated implementation was good enough.

```
if (!SDL_Init(SDL_INIT_VIDEO)) {
    std::cerr << "SDL_Init failed: " << SDL_GetError() << std::endl;
    return 1;
}

SDL_Window* window = SDL_CreateWindow(
    "GBEmulator",
    160, 144,
    SDL_WINDOW_OPENGL
    );
if (!window) {
    std::cerr << "SDL_CreateWindow failed: " << SDL_GetError() << std::endl;
    SDL_Quit();
    return 1;
}

// SDL_Surface has a "pixels" member, the array of pixels
SDL_Surface* surface = SDL_GetWindowSurface(window);
auto palette = SDL_GetSurfacePalette(surface);
auto format = SDL_GetPixelFormatDetails(surface->format);

bool rendered = false;
```

At the very start of the **main** method we initialize SDL and create a 160x144 window.

Then, in the while loop, we update the window surface when V-Blank mode is entered.

```cpp
if (lcd.is_ready() && !lcd.is_full_empty() && !rendered) {
    // Accessing the array
    auto pixels = static_cast<Uint32*>(surface->pixels);

    for (int y = 0; y < 144; ++y) {
        for (int x = 0; x < 160; ++x) {
            Byte b = lcd.get(x, y);
            // Either setting a red or a black color for now
            Uint32 color = SDL_MapRGB(format, palette, b > 0 ? 255
    : 0, 0, 0);
            pixels[y * surface->w + x] = color;
        }
    }
    rendered = true;

    SDL_UpdateWindowSurface(window);
} else if (!lcd.is_ready()) {
    rendered = false;
}
```

And, as you can see in 4, we got our first window.

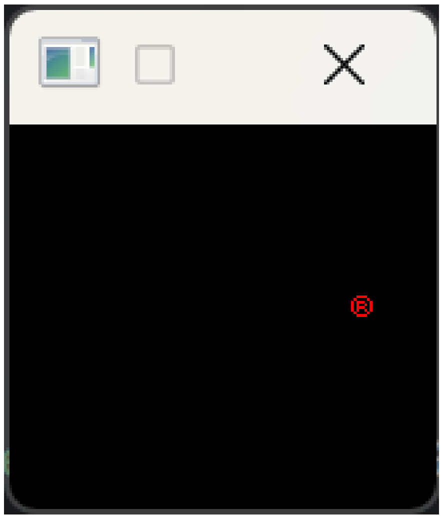


Figure 4: First window

Problematically though, our window (the Game Boy display) is too small for modern displays. On my 1920x1080 display, it was barely even visible. There is no doubt we need to implement a way to **scale** the window up.

12.2 GPU-Accelerated Window

This kind of task sounds like the perfect job for the GPU, though. So, I started working on a fully GPU-based implementation using the **CUDA library**. This library is made by NVIDIA and provides an API that lets us write **kernels**, which will be executed in parallel on the GPU. Of course, CUDA can only compile for NVIDIA graphics card. Luckily, I had already gained some experience with it during my last Physics Simulation project.
Below, I will attach the **final CUDA code**, divided into different sections based on their purpose, along with a brief description of their use.

```
void prepare_pointers(
        Byte** device_video,
        size_t* pitch,
        Uint32** device_scaled_video,
        size_t* scaled_pitch,
        u32 width,
        u32 height,
        int scale
        ) {
    cudaMallocPitch((void**)device_video, pitch, width * sizeof(
    Byte), height);

    u32 new_width = width * scale;
    u32 new_height = height * scale;

    cudaMallocPitch((void**)device_scaled_video, scaled_pitch,
    new_width * sizeof(Uint32), new_height);
}

void free_pointers(Byte* video, Uint32* scaled_video) {
    cudaFree(video);
    cudaFree(scaled_video);
}
```

When setting up the structures for the emulator, we also prepare the CUDA pointers for arrays that reside on the graphics card. Specifically, we allocate memory for **device_video** –which will receive a copy of the byte array from the LCD class–and for **device_scaled_video** –which will eventually be copied into the Uint32 SDL pixel array.

```
__global__ void cuda_build_video(
        Byte* video,
        Uint32* scaled_video,
        int scale
        ) {
    u32 pos = blockIdx.y * DISPLAY_WIDTH + blockIdx.x;
    u32 scaled_pos = (blockIdx.y * scale + threadIdx.y) * (
    DISPLAY_WIDTH * scale) + (blockIdx.x * scale + threadIdx.x);
    scaled_video[scaled_pos] = color_to_rgb(video[pos]);
}

void build_video(
        Byte* device_video,
        Uint32* device_scaled_video,
        Byte* video,
        Uint32* scaled_video,
        u32 width,
        u32 height,
        int scale
) {
    u32 new_width = width * scale;
    u32 new_height = height * scale;


    cudaMemcpy(
            device_video,
            video,
            (width * height) * sizeof(Byte),
            cudaMemcpyHostToDevice
            );

    dim3 video_dim(width, height);
    dim3 scale_dim(scale, scale);
    cuda_build_video<<<video_dim, scale_dim>>>(device_video,
    device_scaled_video, scale);

    cudaMemcpy(
            scaled_video,
            device_scaled_video,
            (new_width * new_height) * sizeof(Uint32),
            cudaMemcpyDeviceToHost
    );
}
```

Of course, these pointers can only be accessed by the GPU. We will create a kernel to manipulate them that is executed $(width \cdot height) \cdot scale^2$ times in parallel.
If you are not familiar with how CUDA works, a kernel is launched by the **build_video** method. Before launching the kernel, we copy the video buffer to the GPU using **cudaMemcpy**, specifying that the copy direction is from host to device with the **cudaMemcpyHostToDevice** flag. The kernel **cuda_build_video** is then launched with $width \times height$ blocks and $scale \times scale$ threads. Each block ID determines the original pixel position in the **video** buffer, while each thread ID determines the scaled position for that pixel. This way, each pixel from the original buffer is expanded into a $scale^2$ region in the output.

A small example is shown in Figure 5, illustrating a scale factor of 2.

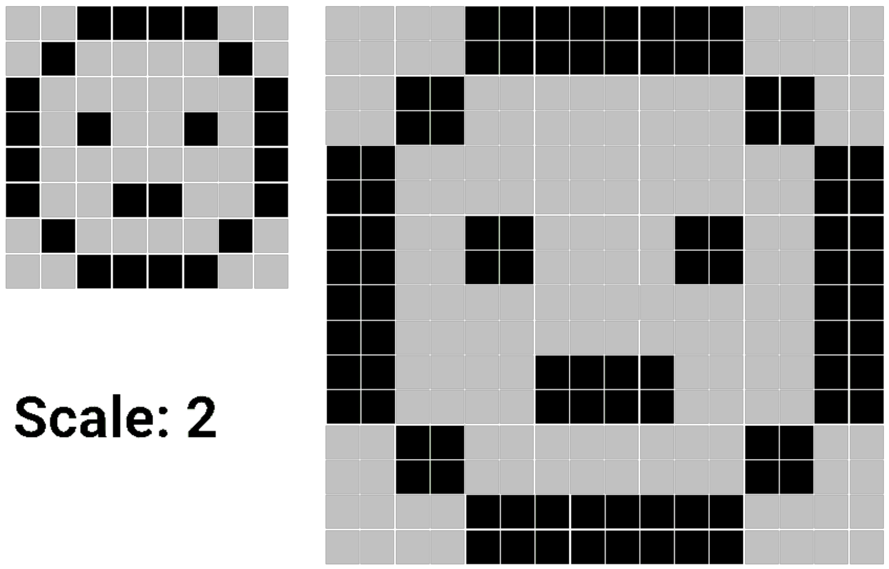


Figure 5: Scale explanation
white pixels are shaded gray for visibility

Finally, the final buffer **device_scaled_video** is copied directly into SDL's pixel buffer, which is done by specifying that the copy direction is from device to host using the **cudaMemcpyDeviceToHost** flag.

```
__device__ unsigned int mapRGB(unsigned char r, unsigned char g,
    unsigned char b) {
    // Assume RGB888 format (8 bits per channel)
    return (r << 16) | (g << 8) | b;
}

__device__ Uint32 color_to_rgb(Byte color) {
    switch (color) {
        case 0: {
            return mapRGB(255, 255, 255);
        }
        case 1: {
            return mapRGB(169, 169, 169);
        }
        case 2: {
            return mapRGB(84, 84, 84);
        }
        case 3: {
            return mapRGB(0, 0, 0);
        }
        default: return mapRGB(255, 255, 255);
    }
}
```

The last thing we have to discuss is the **color_to_rgb** method, which this time returns the original 4 shades of gray, with the help of the **mapRGB** method, which just packs the values **r**, **g** and **b** into a 32 bits value.
And now, the window appears as shown below with a **scale factor of 10**.

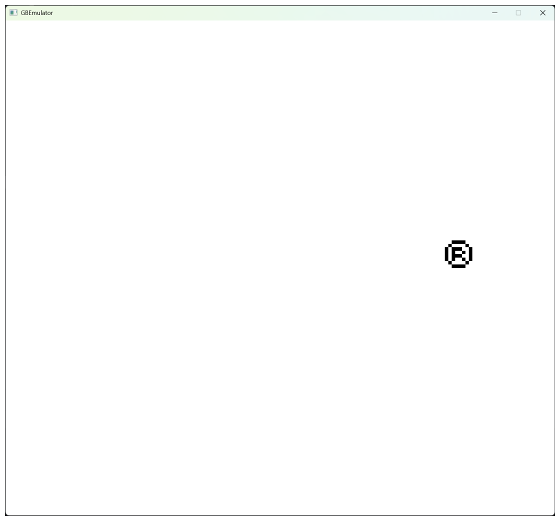


Figure 6: Scaled window

It goes without saying that this GPU-based approach is significantly more efficient than the earlier CPU-only implementation.

13 Emulating the Game Cartridge

Now that have overcome yet another obstacle, it is time to complete our boot ROM emulation. As previously noted in 6.2, the last barrier to the boot process was the "NINTENDO" logo comparison between a version stored inside the boot ROM and one stored inside the **game ROM**. It should be clear by now that, in order to proceed further and eventually run actual games, we must emulate the game cartridge.
Before implementing this, it is necessary to understand how game cartridges are structured and accessed.

13.1 Cartridge

We recall from 4.1 that cartridge data is accessed through memory addresses ranging from **0x0000** to **0x7FFF**. We also remember that this data is split into **two 16Kb banks**, with the second one being marked as **switchable**. It is fascinating to know that some cartridges did not only contain the ROM banks with the game code (as I originally thought), but could also supply additional components, such as SRAMs (Static Random-Access Memory) or a battery to preserve save data. Going back to the **two** banks, due to Game Boy address bus limitations, only **32 Kb** of memory could be accessed at a time. Problematically though, 32 Kb of game memory was not enough for most of the games available on Game Boy, if we think that a game does not only have its code to store, but also graphics tiles, audio tables and other data needed for the game. This problem was addressed using **Memory Bank Controllers** (MBCs)–hardware that enabled switching between different **ROM regions**. However, these additional components are not supported on my emulator, since the original Tetris DMG cartridge just included a 32 Kb ROM.

13.2 Structure

Despite these differences in hardware, all Game Boy cartridges conform to a standardized structure:

1. The region from 0x0000 to 0x00FF typically includes interrupt handlers and other startup routines.
2. The range 0x0100 to 0x014F contains the cartridge header, which stores metadata about the game.
3. The remaining memory contains the actual game code and data.

13.3 Implementation

In my Emulator, file paths are actually hard-coded. At startup, the program specifically looks for files named **dmg_boot.bin** and **tetris.gb**.

```
cpu.load_bootup("dmg_boot.bin", mem);
```

`cpu.load_rom("tetris.gb", mem);`
We have already covered how **load_bootup** works. What remains is the implementation of **load_rom** .

```cpp
void CPU::load_rom(const char *filename, Memory &mem) {
    std::ifstream file;
    file.open(filename, std::ios::in | std::ios::binary);

    if (file.is_open()) {
        uintmax_t size = std::filesystem::file_size(filename);
        std::cout << "ROM size: " << size << std::endl;
        for (size_t i = 0; i < size; ++i) {
            Byte value = 0;
            file.read((char*)&value, sizeof(char));

            if (i < 0x100)
                rom_first[i] = value;
            else
                mem[i] = value;
        }
    } else {
        std::cerr << "Failed to open file " << filename << std::::
    endl;
    }
    file.close();
}
```

One important detail here is that we cannot just read the **.gb** file and copy its content inside the **Memory** structure. Doing so would overwrite the boot ROM, specifically the first **256 bytes** used during the boot sequence. What we are doing instead is to only copy the data starting from address **0x100**, while to temporarily store the first bytes inside a buffer. Once the boot ROM has finished executing, we then copy those initial bytes into memory to complete the emulation of the cartridge.

13.4 How the Debugging Process Changed

When transitioning to running Tetris on the emulator, I realized that disassembling the entire ROM manually would have been far too time-consuming. While I still aimed to understand the program flow and verify how the emulator behaved, I decided to rely on this existing disassembled version of Tetris.
This shift allowed me to stay focused on debugging and improving the emulator itself rather than investing excessive time deciphering game code. The disassembly served as a valuable reference, helping me interpret instructions and memory interactions more efficiently while still enabling a hands-on learning experience.

14 First Tetris Execution

At this point, I wanted to try executing the game for the first time, even though I knew there were still many features left to implement. As expected, after finally seeing our window with the whole Nintendo logo, running Tetris at this stage results in a completely white screen.

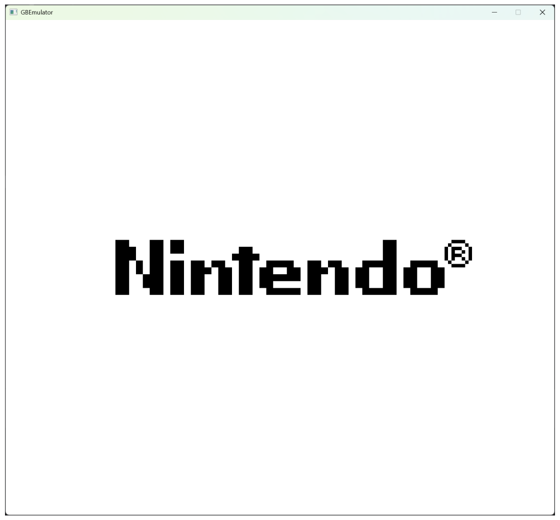


Figure 7: Final logo

It was finally time to analyze the disassembly. That is when I noticed that Tetris checks if all the arrow keys are down at the same time and, if it is the case, a reset occurs. Thus, the next thing to add is the joypad emulation.

```
; Read buttons & return values
l02c4:
; Function that returns a value in FF80
    call    l29a6           ; 02c4
    ...
; If all arrow keys are down
; (if lower 4 bits of value at FF80 are all 1s)
; at the same time, then jump to 21b
    ldh     a,(80h)         ; 02cd
    and     0fh             ; 02cf
    cp      0fh             ; 02d1
    jp      z,l021b         ; 02d3
```

What initially confused me was why the game assumed buttons were being pressed, even though I had not implemented input handling yet.

15 Joypad Emulation

The Game Boy features a simple input scheme: four directional buttons and four action buttons — A, B, Select, and Start.

15.1 Joypad Register

Games can access inputs through a register located at memory address **0xFF00**. Buttons were interestingly arranged as a **2x4 matrix**.

7 6	5	4	3	2	1	0
	Select buttons	Select d-pad	Start / Down	Select / Up	B / Left	A / Right

Table 10: Joypad flags

1. Select buttons: When this bit is 0, the state of the Start, Select, B, and A buttons can be read from the lower 4 bits.
2. Select d-pad: When this bit is 0, the state of the directional (D-pad) keys is available in the lower 4 bits.
3. The lower 4 bits are read-only. **Notably**, unlike most of the Game Boy's conventions, **a pressed button is represented by a 0**, not a 1.

Personally, I do not see why the controls could not have been arranged as a simple 1x8 matrix. I assume there must have been a specific reason why the designers opted for this layout instead.

15.2 Implementation

Simulating a joypad is as easy as getting the input from the keyboard using SDL and update the register flags accordingly.

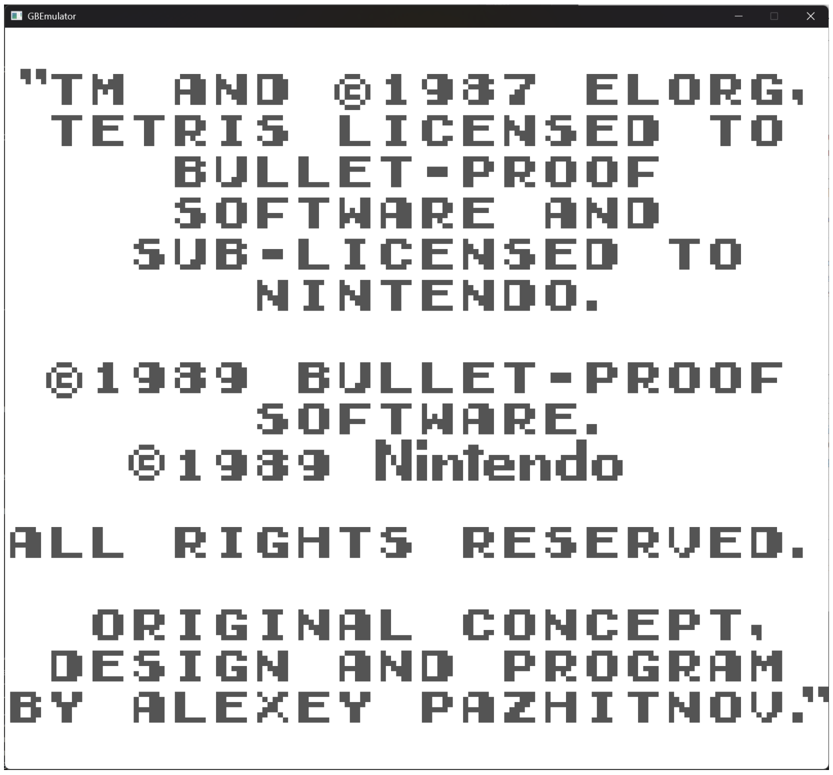
```cpp
class Joypad {
    static constexpr Word JOYPAD_REG = 0xFF00;
    static constexpr Word IF_REG = 0xFF0F;

    bool up = false;
    bool down = false;
    bool left = false;
    bool right = false;

    bool start = false;
    bool select = false;

    bool A = false;
    bool B = false;

    Byte old_values = 0xFF;
public:
    bool keyboard_event(const SDL_Event&);
    static bool high_to_low_transition(Byte first, Byte second, u32
     bit);
    static bool is_interrupt(Byte current, Byte old);

    void update(Memory& mem);
};
```

I chose to map the keyboard arrows to the directional inputs, keep "A" and "B" as-is, and use "T" and "Y" for Start and Select, respectively.

```cpp
bool Joypad::keyboard_event(const SDL_Event &event) {
    if (event.type != SDL_EVENT_KEY_DOWN && event.type !=
    SDL_EVENT_KEY_UP) return false;
    const bool new_value = event.key.down;

    switch (event.key.key) {
        case SDLK_A: {
            A = new_value;
            break;
        }
        case SDLK_B: {
            B = new_value;
            break;
        }

        case SDLK_T: {
            start = new_value;
            break;
        }
        case SDLK_Y: {
            select = new_value;
            break;
        }

        case SDLK_UP: {
            up = new_value;
            break;
        }
        case SDLK_DOWN: {
            down = new_value;
            break;
        }
        case SDLK_LEFT: {
            left = new_value;
            break;
        }
        case SDLK_RIGHT: {
            right = new_value;
            break;
        }
        default: {
            return false;
        }
    }
    return true;
}

void Joypad::update(Memory& mem) {
    Byte Joypad = mem[JOYPAD_REG];

    Byte values = 0x0;
    bool select_buttons = (Joypad & 0b100000) == 0;
    bool select_dpad    = (Joypad & 0b10000) == 0;

    if (select_buttons) {
        values |= !start << 3;
        values |= !select << 2;
        values |= !B << 1;
        values |= !A;
    }

    if (select_dpad) {
        values |= !down << 3;
        values |= !up << 2;
        values |= !left << 1;
        values |= !right;
    }

    if (!select_buttons && !select_dpad)
        values = 0xF;

    Joypad |= values;
    mem[JOYPAD_REG] = Joypad;

    if (is_interrupt(Joypad, old_values)) {
        mem[IF_REG] |= 1 << 4; // Calling interrupt
    }

    old_values = Joypad;
}
```

16 The Credits Screen

Finally, we can see the credits scene, confirming that the essential game systems—such as input handling, memory mapping, and rendering—are now working together as intended. This scene displays the developer and publisher credits, and is one of the earliest indicators that the emulator is beginning to execute the game logic correctly.
This moment is significant: it demonstrates that the CPU is able to fetch and decode instructions, graphics are being drawn through the PPU correctly.
However, this scene is static by design. The game does not yet allow any interaction, and the issue now is that the game does not progress at all–it remains stuck on the credits screen indefinitely.



"TM AND ©1987 ELORG,
TETRIS LICENSED TO
BULLET-PROOF
SOFTWARE AND
SUB-LICENSED TO
NINTENDO.

©1989 BULLET-PROOF
SOFTWARE.
©1989 Nintendo

ALL RIGHTS RESERVED.

ORIGINAL CONCEPT,
DESIGN AND PROGRAM
BY ALEXEY PAZHITNOV."

Figure 8: Tetris Credits

17 The Title Screen

After further debugging, I figure out that I had forgotten to trigger the V-Blank interrupt in the PPU update method. This interrupt is crucial because, after completing its initialization, the game waits for a V-Blank signal to proceed. Without it, execution stalls, leaving the game effectively paused on this screen.

```
case 0: { // Simulating H-Blank
    total_cycles += 2;
    if (total_cycles >= 456) {
        sprites_buffer.clear();
        total_cycles = 0;
        mem[LY_REG]++;
        if (mem[LY_REG] >= 144) {
            mode = 1; // Entering VBlank
            mem[IF_REG] |= 1; // <-- Calling interrupt
        } else {
            mode = 2;
        }
    }
    break;
}
```

This little distraction of mine, which actually took me a few days to find, was the final obstacle that was preventing the **title screen** to show.

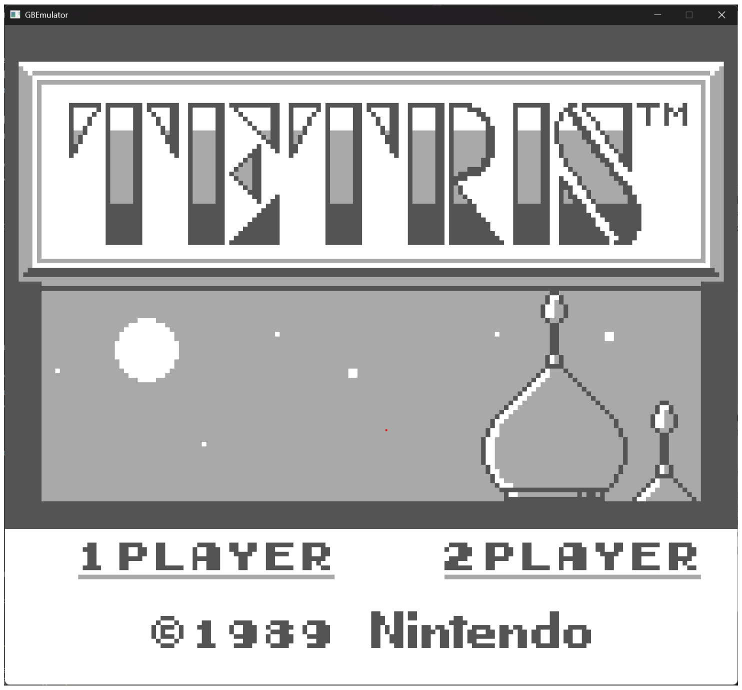


Figure 9: Main Screen

However, one element is still missing from this screen: the arrow that indicates the player's selection. This brings us to the next major step in developing our emulator–implementing **sprites**. The arrow, as well as the **Tetriminos** and other elements, are moving objects, which are still to be implemented in our **rendering pipeline**. To support them, we will need to implement DMA transfers and extend the PPU to handle sprites rendering.

18 DMA

Among other initialization instructions from **0x0199** to **0x02C2**, something called "DMA transfer" was being executed. This was not initially necessary for displaying the credits screen, as all the text was rendered on the **Background layer**.
Direct Memory Access (DMA) is a process by which bytes from a specified memory region are copied by hardware into another region, instead of relying on copying with software. On the Game Boy, the destination of this transfer is always the Object Attribute Memory (OAM). This approach is significantly faster than copying data through software, although it comes with certain hardware limitations, as we will see.

18.1 DMA Control Register

Writing to the DMA control register, located at **0xFF46**, initiates a DMA transfer. The value written to this register determines the **source address**, and the **destination** is always fixed:

```
Source*: 0xXX00 - 0xXX9F
Destination: 0xFE00 - 0xFE9F
*(XX) is the value inside the register
```

After writing to the register, a delay of **4 clock cycles** occurs. Then, one byte is transferred every 4 clock cycles, for a total duration of **640 clock cycles**, plus the initial 4-cycle delay, totaling **644 clock cycles**.

18.2 DMA Bus Conflicts

Although DMA is highly efficient, it introduces a key limitation. While DMA is transferring bytes, the **data buses** used to access RAM and ROM become unavailable, effectively **stalling the CPU**. Fortunately, **High RAM** is not affected by this restriction. Hence, games usually would copy small routines into this region that simply wait for the DMA transfer to complete.

18.3 DMA Transfers in Tetris

That is exactly what Tetris is doing.
First, during game initialization, a routine of **12 bytes** at region **0x2A7F - 0x2A8A** is copied to region **0xFFB6 - 0xFFC1**.
This routine will write to 0xFF46 (DMA control register) to copy bytes from **0xC000 - 0xC09F** to **0xFE00 - 0xFE9F** and wait for **DMA to complete**.

```
; Copy DMA transfer routine to ffb6
    ld      c,0b6h          ; 0293
    ld      b,0ch           ; 0295
    ld      hl,l2a7f        ; 0297
l029a:
    ldi     a,(hl)          ; 029a
    ldh     (c),a           ; 029b
    inc     c               ; 029c
    dec     b               ; 029d
    jr      nz,l029a        ; 029e

...

; Initiate DMA transfer from C000 to FE00
l2a7f:  ld      a,0c0h          ; 2a7f
        ldh     (46h),a         ; 2a81
        ld      a,28h           ; 2a83
l2a85:  dec     a               ; 2a85
        jr      nz,l2a85        ; 2a86
        ret                     ; 2a88
```

Then this routine will be called **once** at first an then once every time a V-Blank interrupt is called.

```
; VBlank Interrupt Routine
l017e:
    push    af              ; 017e
    ...
l0199:
    call    l21e0           ; 0199
    ...
; Initiate DMA transfer
    call    0ffb6h          ; 01d5
    ...
    reti                    ; 020b
```

18.4 Implementation

The implementation of the DMA class is relatively straightforward.

```cpp
class DMA {
    bool transferring = false;
    u32 transfer_pos{};

    static constexpr Word DMA_REG = 0xFF46;
public:
    void transfer(u32 cycles, Memory& mem);

    /**
     * @param cycles M-Cycles
     * @param mem Memory reference
     * */
    void update(u32 cycles, Memory& mem);

    bool is_transferring() const { return transferring; }
};
```

```cpp
void DMA::transfer(u32 cycles, Memory& mem) {
    for (u32 i = 0; i < cycles; ++i) {
        Byte DMA = mem[DMA_REG];

        Word addr = (DMA << 8) + transfer_pos;
        Word dest = 0xFE00 + transfer_pos;

        mem[dest] = mem[addr];
        transfer_pos++;

        if (transfer_pos > 0x9F) {
            transferring = false;
            mem[DMA_REG] = 0x00;
            return;
        }
    }
}

void DMA::update(u32 cycles, Memory& mem) {
    for (u32 i = 0; i < cycles; ++i) {
        Byte DMA = mem[DMA_REG];
        if (DMA != 0x00 && DMA <= 0xDF && !transferring) {
            transferring = true;
            transfer_pos = 0;
            continue;
        }

        if (transferring)
            transfer(1, mem);
    }
}
```

To keep things simple, though, I decided to ignore the bus conflict issue. I relied on the assumption that Tetris had it handled and was correctly waiting for DMA to finish, following the good practice explained above.

19 PPU – Sprites

As mentioned in 11, the PPU architecture includes **two FIFOs**. Similarly to the steps that we described in 11.2, the **Sprites Fetcher** scans the OAM local buffer to identify which sprites should be rendered.

19.1 Timing Issues

Although this stage might seem straightforward at first glance, sprite fetching introduces subtle and nontrivial timing complexities. When a sprite fetch is triggered, the Background Fetcher is **immediately reset** to step 1 and **paused**. It remains inactive until the sprite's data has been fully fetched and loaded into the Sprite FIFO. Once the sprite fetch completes, the PPU begins pushing pixels to the LCD **immediately**. At the same time, the Background Fetcher is restarted from step 1 to resume preparing background pixels. However, fetching the next group of 8 background pixels always takes **6 clock cycles**. During this time, the PPU continues shifting pixels out of the FIFO at the usual rate—**1 pixel per clock cycle**. If the Background FIFO had fewer than 6 pixels left at the moment the sprite fetch ended, the PPU will stall and wait for the FIFO to refill. This results in a significant rendering delay, creating an important timing dependency and leaving little tollerance between background and sprite rendering. It should go without saying that implementing this stage introduced a fair amount of complications.

19.2 Implementation

Given all these considerations, I modified the previously existing methods to support sprite rendering.

```
void PPU::draw(u32 cycles, Memory& mem) {
    for (u32 i = 0; i < cycles / 2; ++i) {
        background_draw(mem);
        sprites_draw(mem);
    }
}

void PPU::pop_and_mix_to_lcd(u32 cycles, LCD& lcd) {
    if (sprites_drawing_step > 1) return;

    for (u32 i = 0; i < cycles; ++i) {
        if (background_fifo.is_empty()) return;

        Pixel pixel = background_fifo.pop();
        if (!oam_fifo.is_empty()) {
            Pixel oam_pixel = oam_fifo.pop();
            pixel = oam_pixel;
        }
        lcd.push(pixel);
    }
}
```

This is followed by the **sprites_draw** method.

```cpp
void PPU::sprites_draw(Memory& mem) {
    switch (sprites_drawing_step) {
        case 1: {
            sprites_fetch_tile_number(mem);
            break;
        }
        case 2: {
            sprites_fetch_tile_data_low(mem);
            break;
        }
        case 3: {
            sprites_fetch_tile_data_high(mem);
            break;
        }
        case 4: {
            sprites_push_tile_data(mem);
            break;
        }
    }
}
```

And now the main screen is done!

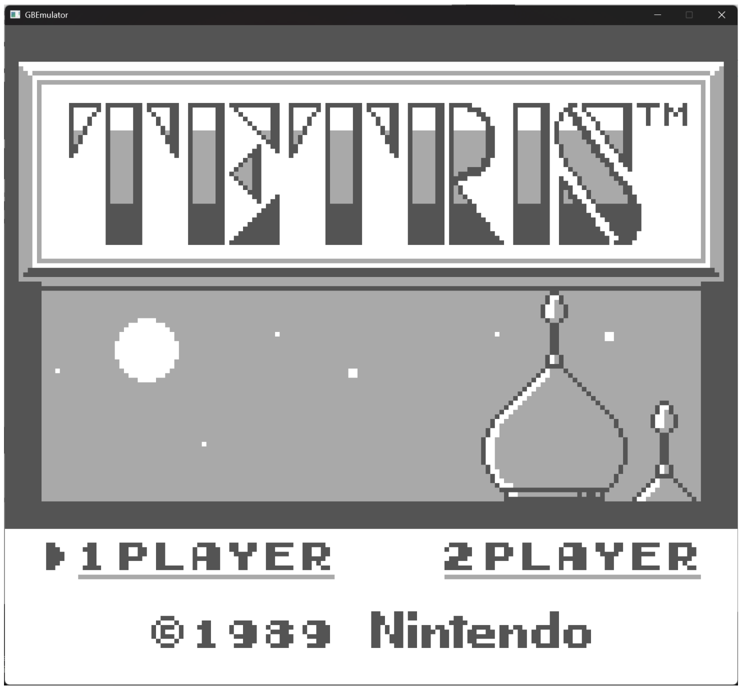


Figure 10: Final Tetris Main Screen

20 A Surprising Bug: Writing to ROM

After all this work, it was finally time to try playing the game. Unfortunately, my excitement was short-lived-when attempting to start an actual game, the software unexpectedly reset. Actually, when I first analyzed the code, I noticed a peculiar segment of code I had ignored for a long time.

```
; Set Rom bank to zero
; (Not needed since the original has no MBC.)
        ld      a,1             ; 0252
        ld      (2000h),a       ; 0254
```

This snippet attempted to write a byte to **address 0x2000**. Obviously, on a real Game Boy, such a write would silently fail, thus, this issue was not relevant. However, due to my own decision I stated at the beginning of this paper (4.2), my emulator **allowed** this memory to be overwritten.

```
l1ffe:
    ldi     (hl),a          ; 1ffe
    dec     b               ; 1fff
    jr      nz,l1ffe        ; *2000*
    add     hl,de           ; 2002
    dec     c               ; 2003
    jr      nz,l1ffc        ; 2004
    ret                     ; 2006
```

This caused the original conditional jump to be modified into:
`ld BC, 0xFC20`
After fixing this issue, the game finally became playable.

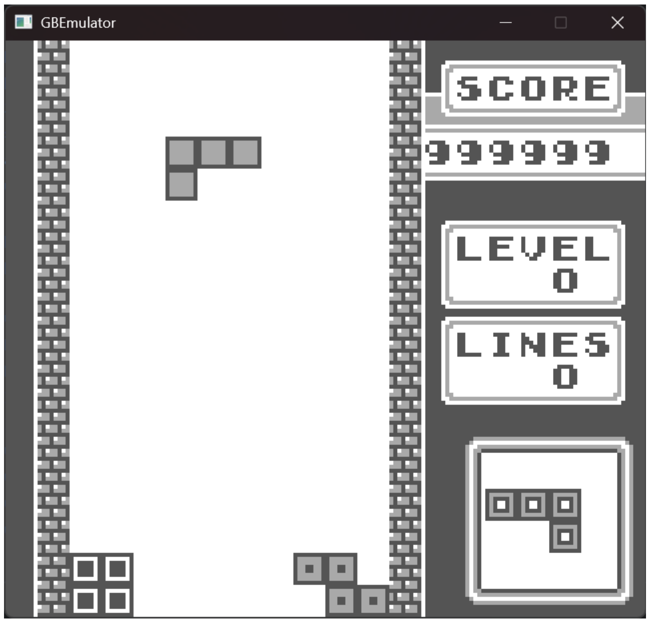


Figure 11: The Gameplay

21 Final Troubleshooting

Before considering the project complete, there were still two remaining bugs to address—one relatively minor and unrelated to the main game, and another that was more critical.

21.1 Score Issue

As you may have noticed in figure 11, the score is stuck at **999999**. After investigating, I discovered that the root cause was a mistake in my implementation of the **DAA** (Decimal Adjust Accumulator) CPU instruction. Specifically, I actually made a typo at the beginning of the project in the calculation of the **carry (C) flag**. In my initial implementation, I mistakenly added adj + 0xFF instead of masking adj with 0xFF, as shown below:

`c = ((A & 0xFF) + (adj + 0xFF)) > 0xFF;` → `c = ((A & 0xFF) + (adj & 0xFF)) > 0xFF;`

This error caused the carry condition to **always evaluate as true**, forcing the **C flag** to be **set** regardless of the actual result. As a consequence, the score logic malfunctioned, constantly pushing the counter toward its maximum value of **999999**.

21.2 GPU Memory Leakage

A more critical issue emerged from the CUDA video scaling implementation. Initially, for each frame, I was allocating memory on the GPU for the pixel buffers. However, I was not freeing that memory afterward. Over time, this led to memory leaks, and eventually the program would freeze as the GPU ran out of allocatable memory. What I ended up doing, instead, is to pre-allocate these buffers only once at the start of the emulator's execution–since their size does not change throughtout runtime–and free them upon closing the window. This approach ensures stable memory usage and prevents the emulator from crashing due to memory exhaustion.

```
int main() {
    ...

    Byte* cuda_video;
    size_t pitch = 0;

    Uint32* cuda_scaled_video;
    size_t scaled_pitch = 0;

    prepare_pointers(
            &cuda_video,
            &pitch,
            &cuda_scaled_video,
            &scaled_pitch,
            DISPLAY_WIDTH, DISPLAY_HEIGHT,
            scale
        );

    ...

    while (running) {
        ...

        if (lcd.is_ready() && !rendered) {
            build_video(
                    cuda_video,
                    cuda_scaled_video,
                    lcd.get_display(),
                    pixels,
                    DISPLAY_WIDTH, DISPLAY_HEIGHT,
                    scale
                );
            rendered = true;

            SDL_UpdateWindowSurface(window);
        } else if (!lcd.is_ready()) {
            rendered = false;
        }
    }

    SDL_Quit();
    free_pointers(cuda_video, cuda_scaled_video);

    return 0;
}
```

With these final issues resolved, the game is now fully playable, behaving as expected from the original Tetris experience.

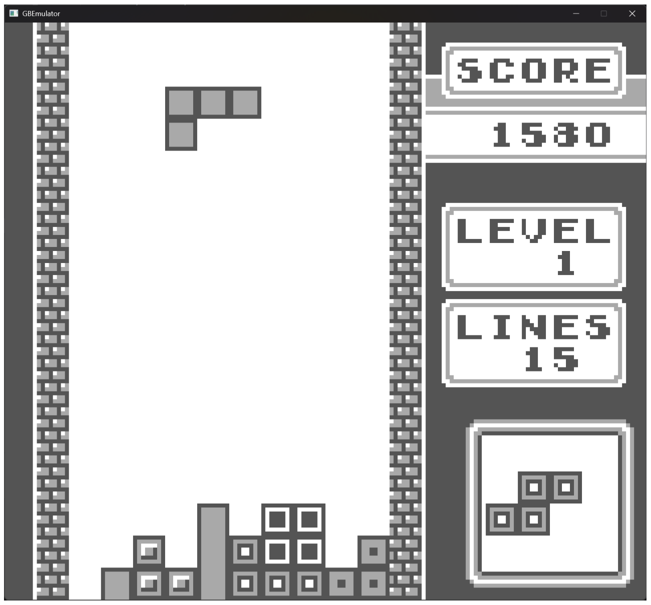


Figure 12: The Final Gameplay