Emulation

Takanen Edoardo

April 1, 2025

Abstract

MOVE TO CARTRIDGE SECTION
Another interesting fact I found out when making this emulator is what is inside a cartridge. It is fascinating to know that some cartridges would not only include the ROM banks with the game code, but they could also supply their own additional SRAM (Static Random-Access Memory), as well as a battery to preserve the game saves. Due to limiting memory sizes, games could also have a Memory Bank Controller (MBC) to change what ROM should be pointed for memory region $4000 - 7FFF$.
Notice that all these additional components are **not** supported on my emulator, since the original Tetris DMG cartridge just had a 32 Kb ROM.
(still to be placed)
images from:

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

Contents

1 Introduction

As a kid, I used to play with some Nintendo (copyright symbol) consoles like the Wii or the DS and I've 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's 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 exacly 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

```cpp
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's 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's 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**.

I think 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's registers

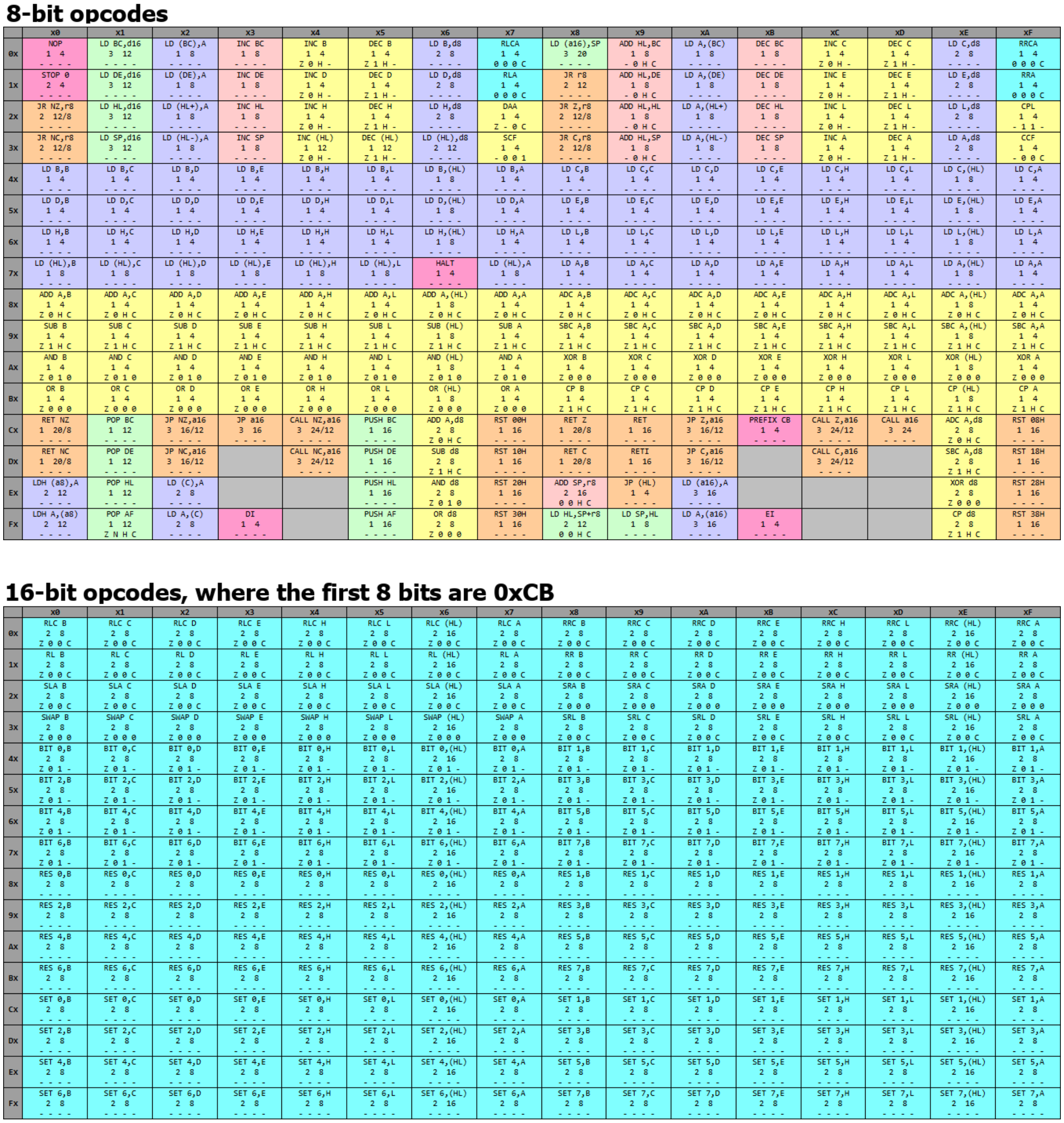
Registers are the fastest memory to access because it is already inside the processor. However their size is very limited, so we cannot have everything in them. That is why the CPU has a set of instructions for loading data to and from larger memory.

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's 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's 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's 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. (TODO)

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's processor behavior.

```
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.

```
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.

```
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.

Some peculiar things 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.

```
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's 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's 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 VBlank

This interrupt is requested every time the Game Boy enters the VBlank 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	VBlank

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	VBlank

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 (VBlank) 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.

```
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