Software fundamentals NES emulator – Group 17

Marnix Crouzen (5655633, mcrouzen@tudelft.nl), Christian Larmann (5861101, clarmann@tudelft.nl),
Jiacong Li (5656354, jiacongli@tudelft.nl), Gefei Zhu (5651727, gefeizhu@tudelft.nl))

November 4, 2022

1 Introduction

This report is made for the Software Fundamentals NES emulator assignment. Described here are the inner working of the software, the features added, and the reflection on the teamwork.

2 Overview of the architecture

Figure 1 and 2 in the appendix show how different parts are connected in an NES system using different mappers. The connections between the components are basically identical in both figures. The main differences are the data distribution in program ROM, and the ways of reading and writing data from and to program ROM. For NROM, whether data is mirrored or not depends on the size of the program data in the cartridge, and the data location is fixed in the ROM. For MMC1, different data banks are dynamically loaded in program ROM according to different MMC1 `PRG_BANK` modes. This means the data in ROM is flexible. And also, MMC1 needs to switch banks when the PPU tries to access the character ROM from cartridge.

In the design, we created different structures for the CPU, cartridge, mapper and bus. For the structure `CPU`, an array with size of 0x10000 is used to represent the CPU memory map. Furthermore, the structure holds several CPU registers. The structure `Cartridge` consists of header, program ROM data and character ROM data. Finally, every part is connected with `Bus`. All functions needed for reading and writing data, and communication between different parts are implemented for the structure `Bus`.

Figure 3 in the appendix generally shows how those functions work with each other. The program starts from `main()`[1] which contains the function `get_data()`, of getting data from an iNES file (which represents a cartridge), and `run_cpu()` for running the CPU. After that, the function `tick()`[2] is run in loops. The actual execution of instructions happens in `tick()`, where every `tick()` represents one CPU cycle. Depending on the instruction, `data_write()` or `data_write()` will be called. The program counter (PC) changes when executing instructions. Except that, PPU can also generate `NMI_IRQ` which changes program counter. More details about that will be discussed in the next chapter.

[1] see main.rs
[2] see cartridge.rs

3 Features added

In this chapter, all the features added will be presented and explained.

3.1 Instructions handling and addressing modes

The function `tick()` continuously calls `do_instruction()`[3], which handles all instruction requests. It will read the function opcode and execute the instruction accordingly. Each instruction is a structure containing the instruction type, addressing mode, and number of cycles it would take an original CPU to execute it. One interface, the `get_data_address()`[4] function, fetches an address for each of the different addressing modes. This function is called before all instructions that need an address to work on. It also sets the program counter according to the bytes it has read. The interfaces `data_read()`[5] and `data_write()`[6] handles all read and writing operations (more on this in chapter 3.2). The instruction logic and setting the flags (in the CPU structure) is implemented here too, according to the specifications of the 6502 instructions. The function increments the program counter to do the next instruction. This function is able to execute all instructions, official as well as unofficial.

3.2 Memory map

The 6502 memory is divided into sections related to different functionality. The sections are: program ROM, program RAM, PPU registers, etc. To handle this, an interface for writing and reading data from the 6502 memory was created called `data_read()` and `data_write()`. This checks the address and performs the appropriate function. The PPU memory (0x2000-0x4000) calls the PPU read/write functions, location 0x8000 and above calls the mapper function (more on that later in chapter 3.3), etc. Mirroring was handled with PPU, RAM, and the mappers.

3.3 Mappers

Two mappers were implemented for this project: NROM and MMC1. An element of the `Bus` structure (named mapper) is used to access the mapper functions. For this, a mapper type (named `MapperType`[7]) was created to be able to distinguish them. This

[3] see instructions.rs
[4] see instructions.rs
[5] see bus.rs
[6] see bus.rs
[7] see mapper.rs

type is a enumeration (`enum`) of structures, so each mapper could store the relevant information. The `MapperType` implements `get_mapper_address()`[8] to fetch addresses (here mainly used to handle mirroring) and `write_mapper()`[9] to parse options to mappers.

The two mappers are briefly explained in the following sections.

3.3.1 NROM NROM is the easiest mapper to implement for an NES. It is not possible to write to physical NROM, so the emulator ignores any writes to the addresses. The read only mirroring has to be accounted for. When the size of the program ROM is 16kB, then all addresses should be mirrored on both addresses. This is implemented by simply subtracting 16 kB from the address if it is above the mirrored starting point. If the program ROM is 32 kB, no mirroring is required.

3.3.2 MMC1 MMC1 can switch out 16 kB banks into the 32 kB ROM of the CPU. To do this, the mapper can be written to (simply by trying to do write instructions to the 6502 ROM memory). These have many options, but the important ones for this implementation are:

- **Program bank mode:** Specifies which part of the bank on the CPU should be fixed (to first or last part of program ROM on cartridge) with one switchable bank or if the entire bank should be switchable.

- **Program bank number:** Specifies which bank of the cartridge program ROM should be in the switchable memory.

Writing to MMC1 addressing is done bit-by-bit with a shift register. So a counter is kept to store the data. After five shift are done, the data is loaded on the register corresponding to the final address that was used for shifting. The settings are set accordingly and if the banks of the program bank has changed, then an additional operation is performed to handle this.

As mentioned in chapter 2, the memory is stored in an array, so a bank switch has to copy the cartridge ROM into the 6502 memory space. Using the bank number, the correct address of the cartridge ROM vector can be calculated.

3.4 Standard Controller

The controller is another peripheral in the bus structure. This of itself is a structure with a strobe bit and an index. Writing to address 0x4016 allows to set/reset the strobe, which continuously reloads a shift register when set. Whenever a byte from location 0x4016 is read, one button is read on the LSB. When the strobe bit is set a shift register makes sure that another button can be read on the same memory location. The button input is handled by the PPU. A second controller was not implemented.

[8] see mapper.rs
[9] see mapper.rs

4 Reflection on teamwork

At the start of the project some rules were set up for git usage, keeping the git history clean for the most part. The CI/CD also helped with this. For the rest everything got along.

The code works well enough and can play Super Mario and other games. Some features were not implemented though, such as the second controller, proper timing, etc. Games using MMC1 seem to have issues with the graphics, which is possibly related to bank switching. The `ALL_INSTRUCTS`, `NROM`, and `NESTEST` tests all pass. No warnings are present for `cargo fmt` and `cargo clippy` and `cargo doc` can be used to build the documentation.

Some problems arose when implementing the structure (such as generic function calls to read/write data), requiring different parts to be rewritten and merge conflicts to happen. At the start of the project some basic well thought-out interface with all inputs and outputs should have been agreed upon.

Appendix

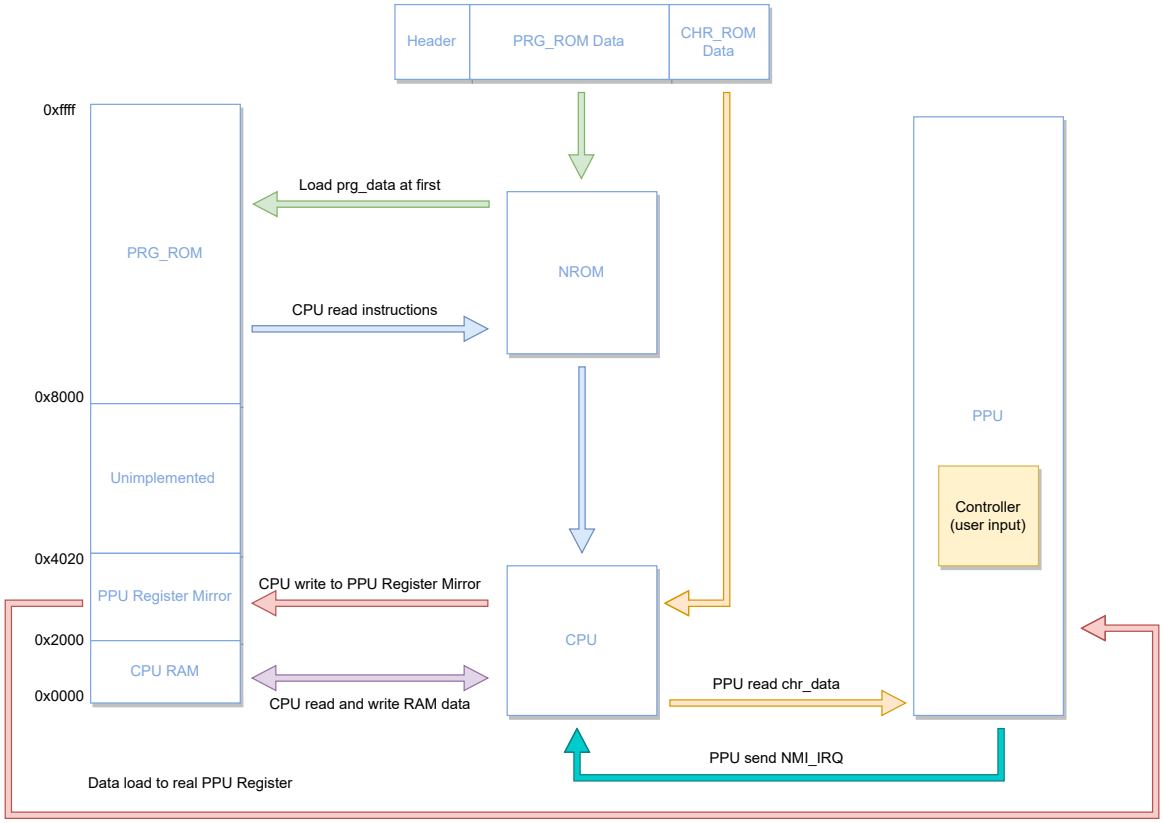

Figure 1: System with NROM

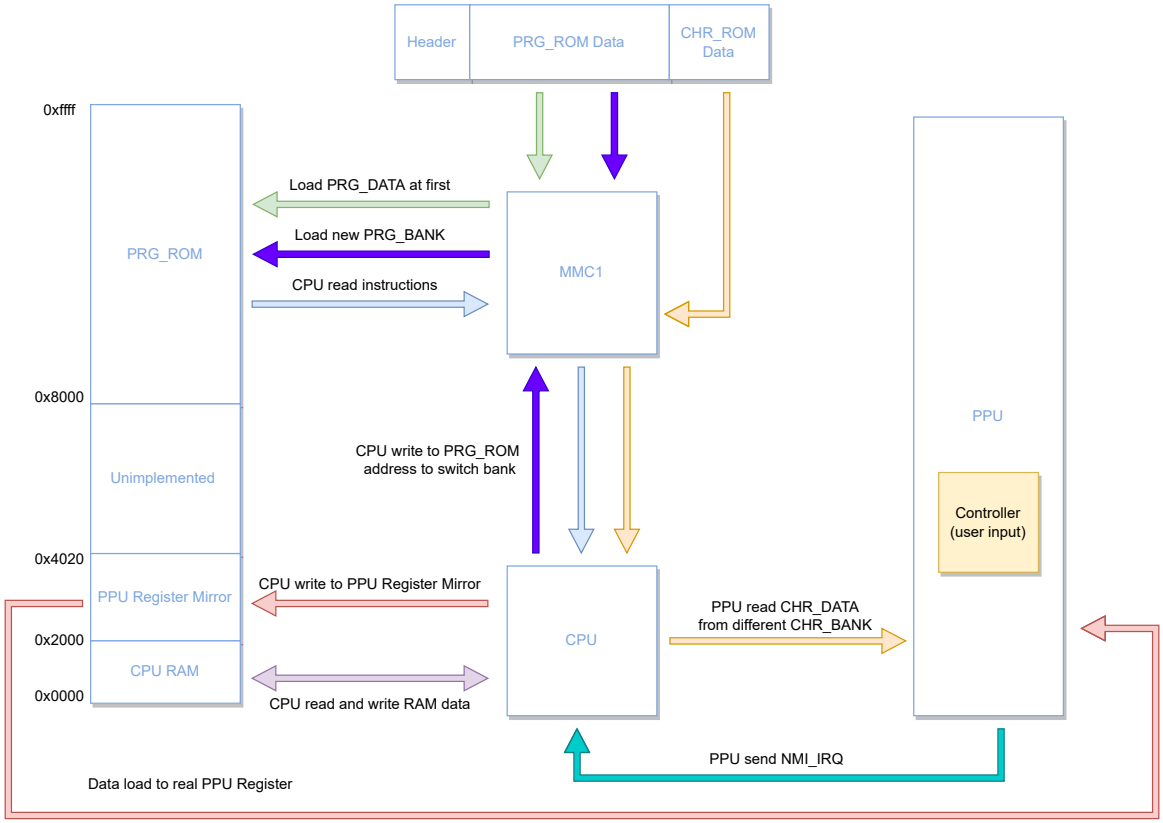

Figure 2: System with MMC1

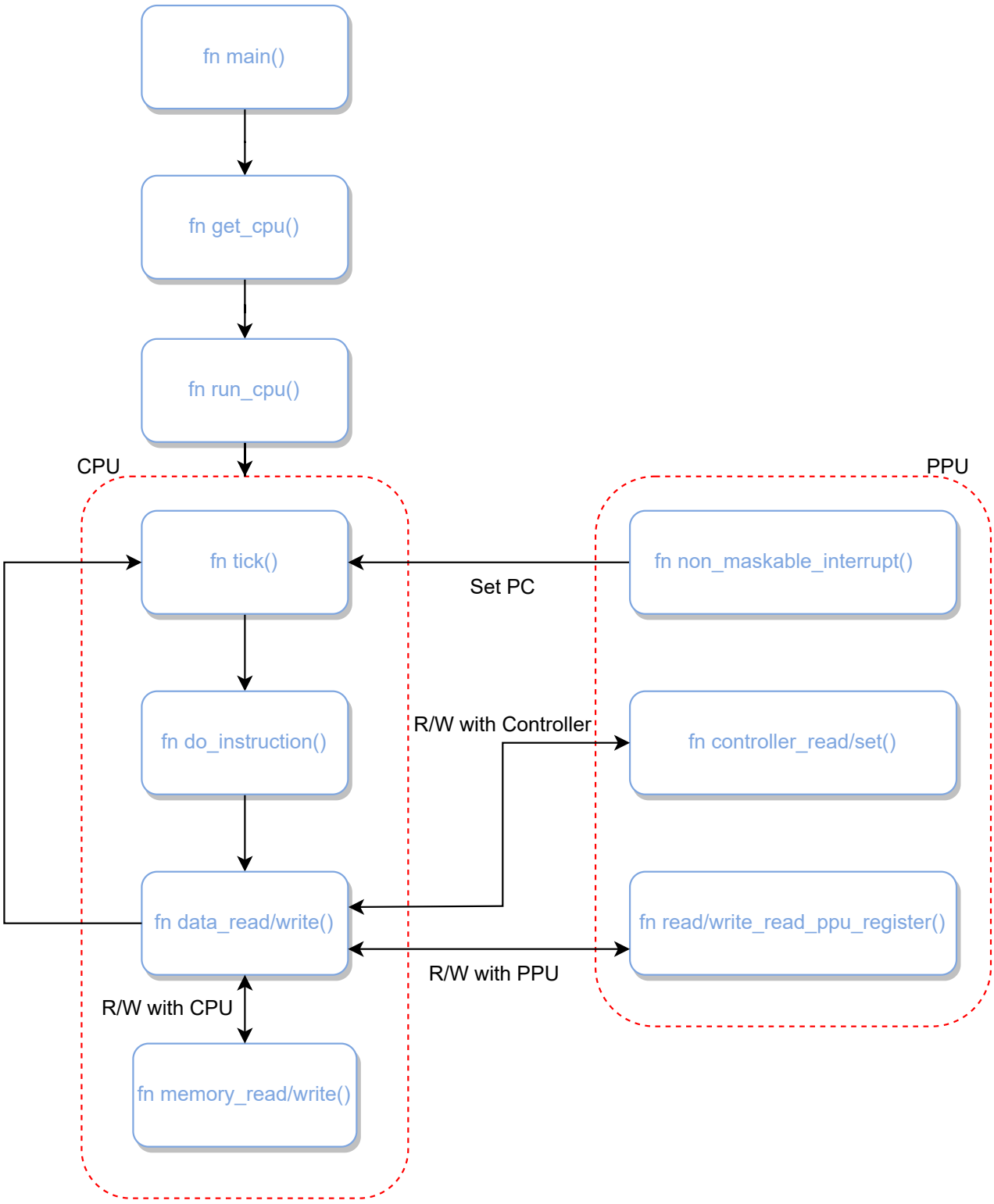

Figure 3: Flowchart of the program