Group Kappa

CESE4000: Software Fundamentals

NES Emulator

Friso Smit 5104904 Hashim Karim 4811518
Mihaly Fey 5476747 Rik Bieling 4955854

TU Delft

Contents

1: Introduction

The Nintendo Entertainment System (NES) is one of the most iconic gaming consoles of the 20th century, celebrated for introducing groundbreaking games like Super Mario Bros. (1985), The Legend of Zelda (1986), and Mega Man (1987). This project aims to recreate the NES's core architecture by emulating the functionality of its Central Processing Unit (CPU), the 6502, and its interactions with memory, cartridges, and the picture Processing Unit (PPU).

The 6502 CPU, widely used in early computers such as the Apple I, Commodore 64, and the NES, operates with a Complex Instruction Set Computing (CISC) architecture, where a single instruction can perform multiple low-level operations. For instance, a single instruction might load data from memory, manipulate it, and store it back—all in one operation.

Throughout this project, rigorous testing ensures accurate emulation of the NES architecture's core interactions and behaviors. Beyond replicating the NES processor, this project also enhances our understanding of collaborative software development, as we work together to bring this complex system to life.

2: Software components

The emulator splits the physically separate parts of the NES into different crates in the program as well. The CPU, the PPU, the APU, the cartidge and the controller. Of these the PPU and the internal parts of the PPU were implemented by the embedded systems teaching team[1][2] The rest is up to project group. The implementation details are described in this chapter.

2.1. Overall architecture

The groups implementation contains a top-level struct which implements the `TestableCPU` and the `CPU` traits. This struct contains the memory struct, which maps memory addresses to communication with the other parts of the NES system. The CPU handles the classical CPU functions, the program execution and the logging of its and the memory's operations. The PPU handles the graphics of the screen, by communicating with the CPU and the cartridge.
The overall system is in Figure 2.1.

2.2. CPU emulation

The main goal of a working CPU is to write the right data into the memory and the PPU registers at the right time. What data gets written and read is decided by the instructions, which are read from the memory as well. Each instruction is uniquely identified by the opcode, which determines what the instruction does, what it reads, where it writes.

Addressing modes The first thing in each instruction is the addressing mode, gets read first and determines two things, the addressing mode and the instruction type. The addressing mode determines how many bytes should be read from the memory and where the operand should be read from, then does the memory jump and other operations, and reads the memory value at the specified location, and passes it to the execution. There are two exceptions to this, the `implied` addressing mode does not require reading from the memory, and if the instruction type is write-only and does not need an input value from the memory, memory read is not performed, as unintended reads can cause unintended behavior when reading from registers such as PPUDATA. The addressing mode is otherwise common for all operations and was implemented such, in and `match` statement and returns options for both the operand value and the address it was read from.

Execution What the operations execute is independent of the addressing mode so the execution is just a matches the current instruction type to an execution process. This process handles the CPU registers, further read/writes to the memory and setting the processor status flags.

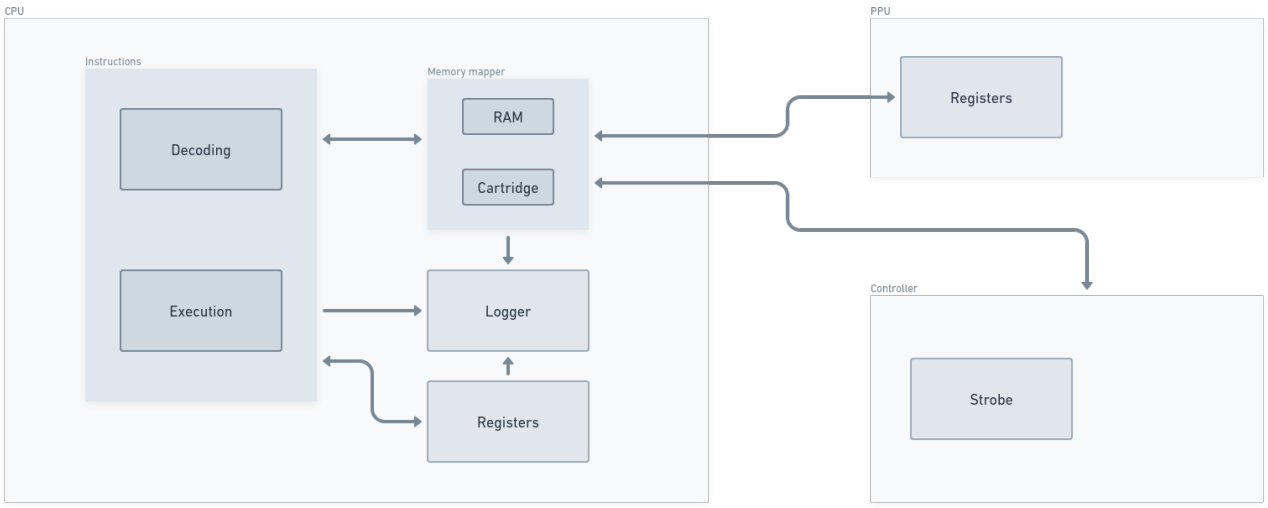

Figure 2.1: Overall system architecture

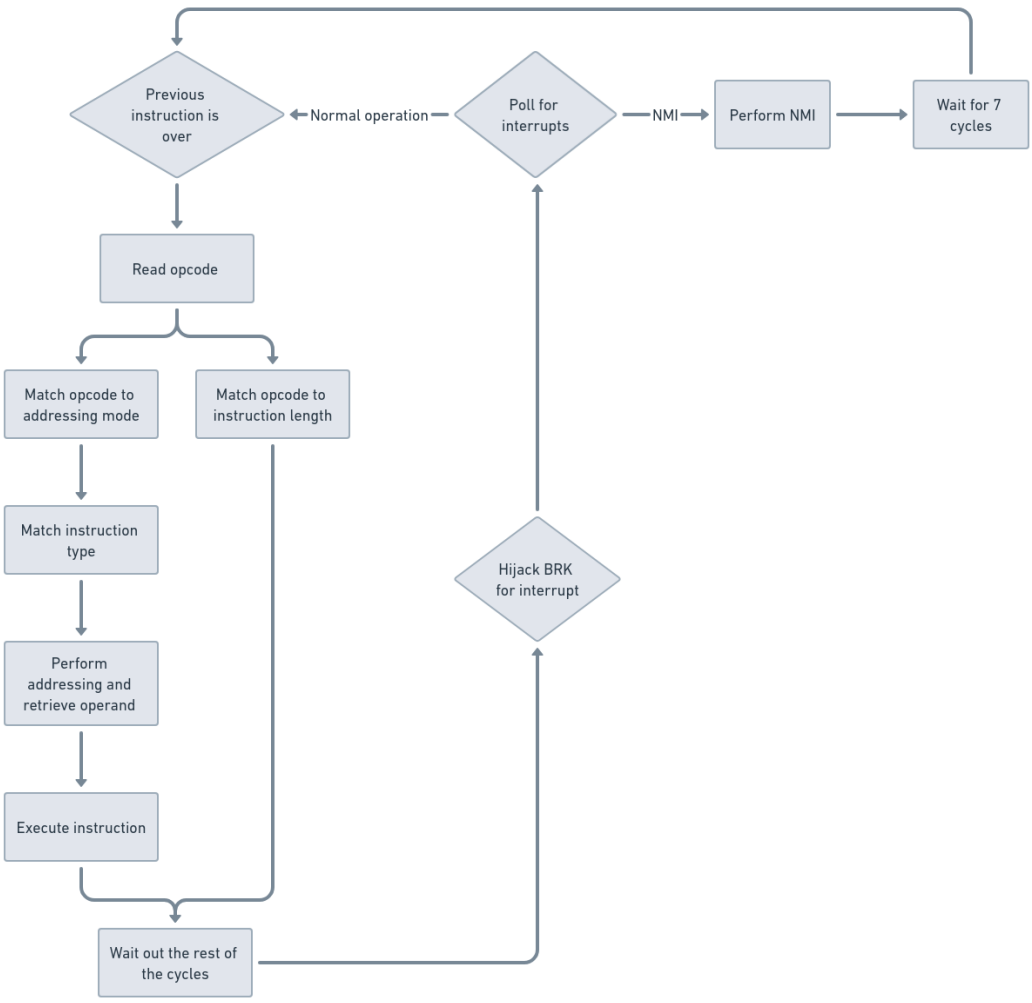

Figure 2.2: Instruction execution loop in the CPU

Instruction timing The opcode is matched to the cycle length of the instruction. All actions of the instructions are performed when the instruction is read, and the processor is idling in the rest of the instruction cycles. In case of page crossing and branch success one additional cycle is added to the total number of cycles.

Interrupts In the base implementation the only type interrupt that has to interrupt that has to be implemented is the Non-maskable interrupt (NMI), as no component can send a maskable interrupt and there is no reset button on the implemented controller. The CPU saves arriving interrupts and polls for them before the time of the current instruction cycles run out. Then it does the interrupt operation cycle instead of reading the next opcode. The interrupts are not polled at the end of interrupt cycles, they are left for the end of the next instruction. Interrupt hijacking is also implemented, when a BRK operation is performed the program counter is set to the address under FFFC-FFFD.
The overview of the CPU operation is in Figure 2.2.

2.3. Memory emulation

The memory system has an overview of the full address space of the NES. It is responsible for giving the right value when an address is being read, and writing to the right place when an address is being written. The main components are as follows:

- Memory
- Cartridge
- Controller

2.3.1. Memory struct

All reads and writes first go through the `Memory` struct. It where reads and writes need to be redirected, based on the address. Some reads and writes will be redirected to the internal memory or the PPU. Others are redirected to the cartridge or the controller.

2.3.2. Cartridge struct

The `Cartridge` struct is owned by the memory and is responsible for reading rom files, interpreting the header and performing mapping operations. It implements the NROM and MMC1 mappers. Within the cartridge struct the following fields are defined:

- Header
- prg data
- chr data
- prg bank
- chr bank 0
- chr bank 1
- shift register
- prg bank mode
- chr bank mode
- prg ram
- chr ram
- init code

Header

The header contains all the data from the .nes file header.

prg data

The prg data part of the cartridge contains the entire program Read-Only Memory (ROM) as a vector of bytes.

chr data

The chr data part of the cartridge contains the entire cartridge ROM as a vector of bytes (if chr ROM is present).

prg bank

The prg bank holds the current selected bank for the cartridge if a program ROM with bank switching is present.

chr bank 0

Chr bank 0 holds the selected 4 KB bank for the cartridge if a character ROM with bank switching is present.

chr bank 1

Chr bank 1 holds the selected 4 KB bank for the cartridge if a character ROM with bank switching is present.

shift register

The shift register contains a u8 that is modified based on the writes sent to the mmc1 mapper to control the editing of the mapper control and bank registers.

prg bank mode

The prg bank mode determines what type of bank switching is active for the program ROM and is used to determine memory reading addresses during program ROM reading.

chr bank mode
The chr bank mode determines what type of bank switching is active for the character ROM and is used to determine the memory reading address for the character ROM reads.

prg ram
The prg Random Access Memory (RAM) is an 8KB array that functions as the program RAM of the cartridge, this array always exists even if the prg ram is disabled.

chr ram
The chr RAM is an 8KB array that functions as the character RAM of the cartridge, this array always exists but only gets used when chr ram is enabled by the header specifying a chr rom length of 0.

init code
The init code contains a copy of the code found at the end of the program ROM. This is saved because not all .nes files save a copy of the relevant jump registers at the end of every 16KB.

2.3.3. Controller struct
The `Controller` struct is handling reads and writes to controllers. The memory owns the controller (in a `Refcell<>`). It emulates a standard NES controller.

2.4. Debug printing
The emulator implements CPU instruction logging using the same format as some other emulators. Using this logging format, the emulator can be compared to other emulators instruction for instruction to find possible errors in the implementation. The emulator also implements debugging using the Log crate. This integrates with the logging of the PPU and allows you to set the log level with a single change in *main.rs*. One line of the logging looks like in the following; with relevant parts of the CPU printed for every instruction.

```
C009  AD 02 20  LDA A:00 X:FF Y:00 P:26 SP:FF CYC:201
```

2.5. Continuous Integration
The group has enabled Gitlab's Continuous Integration. This integration tests if the code compiles, runs the tests successfully, uses correct formatting and that it passes Clippy's linting tests.

3: Conclusion and Reflections

At the start of the project hours were set up to be 8 hours per week, and the group kept to it and everyone showed up throughout the whole project. Other aspects of the teamwork were also great, the discussions were frequent and productive. The gitlab workflow was also very nice, issues and pull request have been used exclusively for development and planning, apart from a couple of accidents. In the end a couple of hard to find bugs delayed the finishing and not everything was done that was planned for the whole course of the project, planning could have better accounted for that. It could have been anticipated that bugfixing takes a lot of time.

Glossary

Notation	Description
CISC	Complex Instruction Set Computing
CPU	Central Processing Unit
NES	Nintendo Entertainment System
NMI	Non-maskable Interrupt
PPU	Picture Processing Unit
RAM	Random Access Memory
ROM	Read-Only Memory

References

[1] Jonathan Dönszelmann. *Requirements and Grading - Software Fundamentals*. 2024. URL: `https://cese.ewi.tudelft.nl/software-fundamentals/part-2/requirements.html`.

[2] Jonathan Dönszelmann. *Computer and Embedded Systems Engineering / Software Fundamentals / NES emulator graphics library*. 2024. URL: `https://gitlab.ewi.tudelft.nl/cese/software-fundamentals/tudelft-nes-ppu`.