Abstract

The video gaming industry is growing and evolving faster than any other form of entertainment. And sometimes it is just nice to go back and see those revolutionary milestones which made the gaming what it is today.

Just like the Nintendo Entertainment System was. The goal of this project is to create an emulator for the NES system for either nostalgic reasons or maybe out of curiosity so we can live through the greatest gaming titles of the past on our PCs without the need to have the original hardware.

The result of the project at the end was a working NES emulator which can run a few of the original launch titles of the console.

Acknowledgments

I would like to thank Dr Robert Atkey for supervising the project during the year. His support was truly valuable and helpful during the project's lifetime.

I would also like to thank the Nesdev community to keeping alive this wonderful console with their support towards both game and emulator developers.

Contents

Chapter 1

Introduction

The Nintendo Entertainment System, frequently called NES, is a home gaming console developed by the Japanese company called Nintendo. Through its lifetime the from the early '80s up until the end of the '90s a lot of good quality titles were released to the platform and a wide range of audience was reached. Due to this quality of games and popularity which the NES gained at it's peak time people even nowadays like to pick up some of the most famous, or personal favourite titles and play through them. Sadly, due to the rapid evolution computing the current hardware are not compatible with old games developed for the NES platform also the old and still working consoles are really hard to get a hands on. Yet the gaming community still keeps alive these games and the way it is mostly done by using emulators.

To solve the compatibility issue with the modern hardware, without actually buying or refurbishing one, the community started to creating so-called emulators. Which are a special kind of software, it's purpose is to provide a virtual emulator environment of a given old or very specific hardware, such as the NES for instance, on top of a given system, like PC or any other gaming console. The goal of this project is to produce an emulator like this which, by emulating the NES hardware behaviour, provides a way to run the games without any need of the modification of the old game's code. Therefore, unlike porting the actual game, the user has access to an authentic experience. On the other hand, emulation is more resource dependent than a ported game, but less time consuming than porting every single game.

1.1 History

This console was the second console of which was created by the Nintendo Co. Ltd. and the first which was planned to be sold worldwide (Wikipedia 2019a). The gaming console was first released in Japan in 1983. The Japanese version was called Famicon and had a bright red toy-like design, unlike the

version which was released worldwide. The homeland release was followed by the USA and parts of Europe in 1986 and later in 1987 Australia and the rest of Europe.

However, these releases were concerning for Nintendo as '83 was also the year of the great video game crash happened in North America (Oxford 2011). But it turned out the perfect opportunity for the console. As the hugely saturated gaming console market shrunk down, most of the competitors get bankrupt, Nintendo rebranded their console and released it as Nintendo Entertainment System worldwide.

It was a huge success not just because the competitors fall out but also due to two important principle which Nintendo following since then. One of them is unlike other consoles before only certified games could be released to the platform, which meant degradation in quantity and increase in the quality of the games. The other principle was that the actual hardware built from not bleeding edge components, therefore, making it cheap, more easily accessible.

(a) Famicon (b) NES

Figure 1.1: The console design

1.2 Background

The NES hardware itself can be divided up to five big different part.

The console main chip was manufactured by Richo, which contains the CPU (Central Processing Unit) and the APU (Audio Processing Unit) (NesDev 2019). The processor itself is an 8-bit MOS Technology 6502 with a little difference that the decimal mode is not presented.

The PPU (Pixel Processing Unit), which was also shipped by Richo, is technically a primitive graphics card which is used by the system to colour and render the graphics pixel by pixel to the Television screens.

The Cartrdige which meant to provide the necessary binary code of the games and also the graphics data for the system. Also gave an opportunity to the developers to implement their own cartridge builds and use it to extend the console's capabilities one great example for that is the first The Legend of Zelda game. The game's cartridge also contains a battery powered RAM extension for players to save their game state (Gerstmann 2006).

Two 8 button, these buttons are up, down, left, right, select, start, A, B, controller provided the interface for user input to the system. The NES controller was the first controller which introduced the single button plus symbol shaped DPAD. Each Nintendo system was brought some revolutionary design idea to the world of gaming console controllers.

The RAM (Random Access Memory) is the central piece of the hardware which not only holds data but through memory mapping, it also connects all the other pieces to the CPU. Therefore the developers can control the full hardware behaviours trough with specific read and write operations to certain memory slots.

In this project, these core parts of the hardware will be emulated on x86-64 machines. As a result, provide an application which is able to run those games which were developed for the original NES system.

Figure 1.2: NES controller

Chapter 2

Project Goal

The main target of the project is to build an application which is able to run games developed for the original Nintendo Entertainment System. The main target platform for the emulator to support is 64bit Linux, but the project design allows easy implementation for other platforms too. This goal is achieved by emulating all the necessary hardware of the original NES system programmatically thus creating a medium for the game binaries to run without any modification at all to it.
However, as the sounds system (APU) is not necessary to be able to run the games this part of the system is not implemented.
The project goal can be divided into multiple stages which are the core and other advanced stages. The main goal is to achieve the Minimal Core stage by the project deadline. However, if the time allows it advanced features will be implemented.

Core - Minimal The emulator can emulate the perfectly the RAM and it's mirroring mapping behaviour properly.
The CPU capable to execute all the official operation codes, as the MOS 6502 has unofficial operation codes as well, also able to properly handle interrupts.
Graphics displayed on the screen with the usage of the PPU without any restriction about how it is doing it.
The user is capable to interact with the system with a keyboard.
Digitalized cartridge format (iNes) red and mapped with the basic NROM mapper (Mapper 0).

Core - Full) The graphics CPU and the whole system itself running with 100% cycle accuracy compared to the original gaming console. Some games are heavily rely on cycle accuracy, especially to provide some nice graphics implementations and also to provide more authentic experience for end users.

Advanced - Mappers Implementing additional Cartridge mappers other than the basic NROM mapperwiki 2019. This would allow the emulator to run a larger pool of games.

Advanced - Multiple Platform Due to the nature of the project design, see below, support for other systems could be more easily implemented due to the modular approach which was used to develop the emulator. Therefore, for instance, Windows and Android could be easily supported by the emulator which would further boost the user experience.

Advanced - State Save Allowing the user to make a snapshot of the current CPU, PPU and RAM state and be able to load it back at a later point in case if the user would like to continue the game right from the place where they saved it.

Advanced - Online Multiplayer The NES system supports two-player local multiplayer, as the system provides hardware to connect two controllers, by default which could be extended over a network, therefore, players could play two player games with the comfort of their homes.

Chapter 3

Project Execution

3.1 Design

Due to the fact that the NES gaming console's hardware could easily be divided up to 6 distinct, loosely coupled module, it was clear that the best way is to design the implementation around these modules:

1. Ram
2. CPU
3. PPU
4. Cartridge / Mapper
5. APU

This is because the system gives control over all of these pieces by wiring them into given memory slots and using CPU instructions to perform read and write operations on them. With this, trigger given actions on the corresponding hardware, all of this without letting the actual CPU Hardware know that it is accessing anything else. Therefore when the system could be implemented module by module in careful order. Due to this modular separations, it was easy and straightforward to build each module easily by separating them up to smaller task and gradually build them up.

Besides the separation of the emulator code to modules, the GUI, Graphical User Interface, is also separated from the actual emulator logic. This behaviour is implemented by compiling the NES emulator logic into a shared object, dynamic library for Windows users, and linked it with the GUI implementation. This decoupling has numerous advantages over shipping the whole application as a single executable.
Due to this separation, the emulator logic became easier to test on its own.
When a new patch is ready for the applications only part of the project needs

to be recompiled and updated. Which makes easier to implement a versioning and updating system.
The GUI implementation can be changed without affecting any of the core solutions, therefore, it is easier to port it to different platforms and devices.

3.2 Tools

The project is written in the latest standard of C++ 17 (cppreference.com 2019). The reason behind the choice of C++ was that the language itself really stable and mature and proven to be a great tool to develop clean, stable, high performant software solutions and also providing low-level solutions for given sets of problems such as pointers. Compared to C, a system level programming language, C++ provides a more readable code on exchange for minimal performance loss thanks to its rich standard library and object-oriented programming capabilities.

On top of C++ a small number, namely two, of external libraries were used to develop the project. Catch2 (Hořeňovský 2019) was used to carry out the automated tests which were written for the project in a BDD (Holmes 2017) style. The other one is SDL (SDL2 2019)

For source code compiling the GCC (GCC-Team 2019) version, 8.2.1 toolchain is used through the MAKE (GNU 2016) build system. And these are configured with CMake (CMake 2019) to which provides a cross-platform build tool configuring interface. Also, it provides an opportunity for developers to use their own preferred IDE, Integrated Development Environment, therefore boost their productivity.

The project documentation consists of two parts the report itself (this document) and the API doc (Sebestyén 2019a).
The report was created with LaTeX (LaTeX3-Team 2019a). On the other hand, the API documentation was generated with Doxygen (Heesch 2019) from the annotated C++ Header files of the source code.

The source code along with the build system configuration and the documentation is version controlled with GIT and stored in the main Github repository (Sebestyén n.d.). As the project meant to be published as an open source project it is publicly available. Although, during the course of the project it was already publicly available and only modifiable by the project owner (Sebestyén Bence) and the project supervisor (Robert Atkey).
The project repository is also linked to a CI, Continuous Integration, the system called Travis CI (LaTeX3-Team 2019b). Which mostly used to provide security against merging any proposed changes which are not passing the build state or failing any of the automated tests which were supplied with the source code.

The CI system build and testing are triggered every time when a pull request for a change against the main repository is set up and blocking any merge until the errors within the proposed changes are not fixed.

3.3 Management

The development cycle managed through GitHub project page. For every task regarding the project execution, there was an issue opened. These issues were labelled up based on which development cycle, see below, it was accomplished, and added to the given project milestone based on which module it was belonged to.
This provided a great oversight over the project's evolution and also a faster more stable development cycle due to the small subtasks.

The development life cycle contains multiple stages which are built upon each other. These life cycle stages are the following:

1. Planing
2. Prototyping
3. Implementing
4. Testing
5. Documenting

Every small task went through a planning phase which mostly consisted idea how to actually code down the given task and how it is going to be fitting into the whole project current state meanwhile thinking about to keep as much space as possible for later new features and improvements.

Prototyping itself was more flexible, compared to the planning, it was different from task to task, some sort of prototype was also developed outside the project source to prove concepts of the planning, but the difference was that it was not always done to every single subtask. Sometimes it involved a whole module, sometimes it just has done for two tasks together.
The main goal here was to test the result of the planning in an actual working context and see if the result is clean, stable and extendable enough to be implemented in the actual project.

Through the implementation, the actual result of the planning and prototyping were added to the main code base.

Testing was carried out in two phases. Depending on the given issue, manual testing was carried out on task and a subtask level if it was needed and also automated BDT, Behaviour Driven Testing, where implemented for every module to test their provided functionality. And these automated tests were executed with the CI as well.

The API documentation was created after the given task was finished and updated with Doxygen.
Meanwhile, the report itself was uprated in larger chunks because this way there was always a bigger portion of the system was available therefore it was easier to position the actual chunk to the whole project.

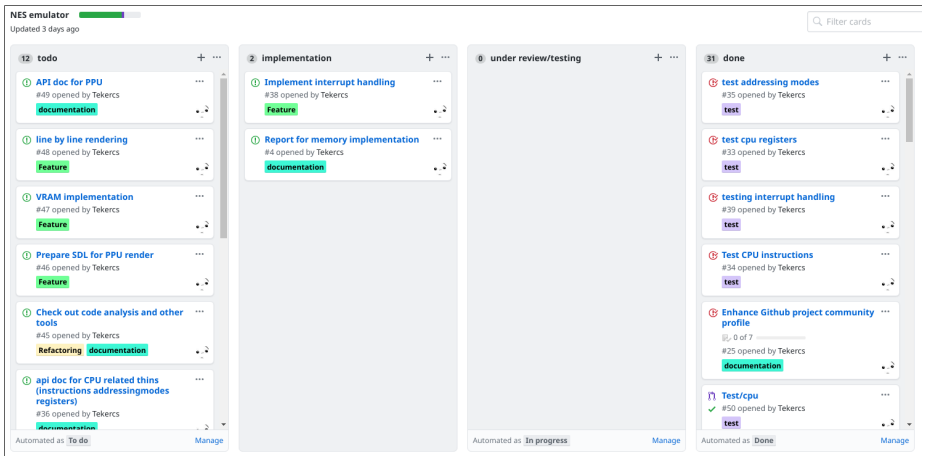

Figure 3.1: Github project page

Chapter 4

Specification and implementation

4.1 CPU

4.1.1 Specification

The NES CPU was the RP2A03/RP2A07 chip, a modified version of the MOS 6502 (Naberezny 2018) microprocessor, manufactured by Ricoh (Wikipedia 2019b). This chip was an 8-bit little-endian microprocessor running on 1,79/1,66 MHz, the European / Australian version (PAL) was running on lower clock speed.
The CPU contains 6 registers from that 3 are more general purpose and 3 are for special purposes (NESDEV-Wiki 2015a).

1. Accumulator
2. X index
3. Y index
4. Status flags
5. Stack pointer
6. Program counter

All of these registers are 8-bit long registers except the Program counter which is 16-bit long due to the memory address bus size, see below (Jacobs 2003a). Most of the work was done on the Accumulator, X and Y index registers as most of the arithmetic and load/store instructions were implemented for these. Also, the result of these operations was visible on the Status flags as well.
The Status flags was an 8-bit register where each bit, except the 5th bit, are representing a different state of the CPU.

Negative Flag, 7th bit This flag is changed whenever the last CPU instruction's result is negative or not. The reason why this flag is important is the CPU's instructions on its own cannot determine if the given 8-bit value which it is working with at the moment is negative or positive. Therefore this flag is used to tell the programmer about the result and give the control to their hand to handle the result in a way as it is good for them. By ignoring the flag the result can be treated as an 8-bit long unsigned value or, with the value of the flag kept in mind, as a signed 8-bit value.

Overflow flag, 6th bit The value of this flag determined by the sign flag of the result of the previously executed instruction is invalid or not.

Break flag, 4th bit Whenever the programmer decides that they want to force out an interrupt from the CPU by executing the BRK, Break, command this flag is set. Therefore the CPU know that after finishing the instruction an IRQ interrupt should be handled.

Decimal Mode, 3rd bit Decimal mode flag can be toggled by two CPU instruction but has no effect on the system due to the fact that the custom modified chip which is used by the emulator is not implementing this binary-coded decimal mode.

Interrupt Disable, 2nd bit This flag also connected to the chips interrupt handling. As if this flag is set, then the microprocessor will ignore all the IRQs.

Zero Flag, 1st bit Whenever an arithmetic operation's result is zero this flag is set. It is important not just for arithmetic but control-flow instructions using it as well for their checks whenever the processor should branch or not.

Carry flag, 0th bit The flag is set by either its designed instruction or any other arithmetic, shifting or rotating instruction which result did not fin in the 8-bit size. This is important to provide support for operations on larger values than 8-bit. One of the reasons is the fact that the memory's address bus is 16-bit.

Along the register, the other main capability defining aspect of the chip is its instruction set. Due to its 8-bit nature of the microprocessor, the number of available instructions is 256. Although not every opcode is implemented officially (Jacobs 2008). The 151 instruction which actually implemented is technically 56 instructions with different addressing modes (Jacobs 2003b). Each instruction has its execution time measured in CPU cycles, which depends on the addressing mode and the length of the instruction itself. Usually, the instruction is 1 - 3 bytes long, the 1st byte specifies the instruction and it's addressing mode and the optimal 2nd and 3rd are used to specify the operands of the instruction.

For example:

TXA instruction is 1 byte long.
STY $01 is 2 bytes long
STY $20AA is 3 bytes long

Important to notice that the Indirect jump instruction, JMP (opcode: $6C), contains a bug. Whenever the indirect instruction parameter points to $xxFF, where x is any hexadecimal number, the instruction fetches the low-byte from $xxFF then the high-byte from $xx00 instead of $(xx+1)00. This is an important bug because there are games which heavily rely on the behaviour of this instruction.

As is seen above on the STY example the same instruction could behave differently this is due to the different addressing mode. Each instruction has its own set of addressing modes, these are all the possible ones:

Implicit Instructions with Implicit addressing mode does not require any additional operand as the instruction itself contains every necessary information. E. g.: BRK

Accumulator The Accumulator addressing mode is used to tell the CPU instruction to carry out its task on the accumulator. E.g.: LSR A

Immediate Immediate addressing mode used when an instruction invoked with an 8-bit constant operand. E.g.: LDA #10.

Zero Page Similarly to the Immediate addressing mode Zero Page using an 8-bit constant operand as well, but in this case, it refers to a memory location starting with 0x00##. E.g.: STA $11.

Zero Page, X Works the same way as the Zero Page addressing but before the operand is used the content of the X index register will be added to it. Important to notice if the address after the addition cannot leave the zero Page address range, instead it wraps over. E.g.: STA $11,X.

Zero Page, Y Identical with Zero Page, X, but instead of X it uses the Y index register. E.g.: STX $11,Y.

Relative Relative addressing mode mostly used for branching instructions. The signed 8-bit constant operand is added together with the program counter. E.g.: BNE *+4.

Absolute Instructions with absolute addressing mode will receive a 16-bit long constant operand which used to identify the target memory location. E.g.: STA $2001.

Absolute, X Similar behaviour with Absolute addressing mode, but in this case, the X register's content added to the 16-bit constant address. E.g.: STA $2001, X.

Absolute, Y Similar behaviour with Absolute addressing mode, but in this case, the Y register's content added to the 16-bit constant address. E.g.: STA $2001, Y.

Indirect When an instruction executed with Indirect addressing mode it will receive a 16-bit constant operand which is a memory address which points to the least significant byte of the real target address. E.g.: JMP $1100.

Indexed Indirect It behaves just like the Indirect addressing mode but the operand of the instruction is a constant 8-bit zero page address which is added together with the X index register before use. E.g: LDA ($40, X).

Indirect Indexed Instructions with this addressing mode will receive a constant 8-bit zero page address which points to the least significant byte of a 16-bit memory address. After it is loaded in the content of the Y register will be added to it. E.g: LDA ($40), Y.

4.1.2 Implementation

When it comes to implementing the processor emulation it was carried out in multiple stages.

Registers First of all the register emulation had to be implemented first, as all of the CPU instructions accessing at least one of the registers. Each of the registers is emulated by a ***uint8_t*** typed variable, or ***uint16_t*** for the program counter, wrapped in a class called ***Registers*** (Sebestyén 2019c) with the necessary accessors functions implemented within. This separation from the actual CPU logic is the result of the break up of the big classes for testing and readability purposes.

Instructions and Addressing modes The first thing to start with instruction implementation is to separate the logic of the addressing mode and the core instruction itself.
For instance, the ADC instruction has 8 different addressing mode. Each of them specifies the way how the instruction obtains the 8-bit value which then will be then just added to the accumulator along with the carry. With this separation, there were only need for 51 core instruction implementation and

separately 13 addressing modes instead of the total 151 instruction.
Each of the instruction and addressing mode implementation received it's own function (Sebestyén 2019b) to make the code as easy to read/understand and debug as it is possible.

Memory accessing As some of the instructions are accessing the main memory of the system for either reading or writing. Therefore the ***Cpu*** class also receives a smart pointer to a ***Memory*** object, for implementation details see below.

CPU Cycle The CPU Cycle emulation was implemented with a different approach. As there is no straight correlation between the cycle and anything else, other than the fact that the PPU itself roughly working on 3X as much cycle rate, the possibility is open to implement it to check for interrupts and execute a single instruction in each step instead of just a single cycle (an instruction usually executed through several CPU cycles 2 - 7 cycle/instruction).
Due to the computational power difference between the NES and today's average PC this approach does not prevent emulator to work with 100% cycle accuracy. As even the most complex instruction can be easily executed within the necessary time frame even with the fact that the emulator running on the top of the normal system. This, of course, could cause issues within an emulator if the emulator's target platform would be stronger such as an Xbox 360, PS 4 or Nintendo Wii.
The instruction execution implementation is consist of a mapping of the 8-bit opcode of the instruction to a paired function call for the core implementation and the addressing mode. This mapping is implemented with a switch statement which readability wise is not the best option due to it has 151 case but that is nearly the best way performance wise.
Although, if it is needed there is room for further performance increase. By monitoring the emulator through different games it is possible to find the most used instructions. With this information in hand, the most used instructions could be lifted out from the switch statement and implemented with an if-elseif structure, in order the most used ones checked first and the least used one checked last from the selected set of instructions. By doing that, the compiler could actually provide further optimization on this part of the implementation.

4.1.3 Testing

The testing of the CPU was carried out by using a test cartridge developed by Kevin Horton (Horton 2004). As his test ROM is an all-inclusive test when it comes for CPU and its instruction's behaviour.
The test carried out by implementing a helper logging function for the CPU implementation. A state log was generated based on the upcoming instruction to be executed and the current state of the registers. Then this log was compared in every processor cycle with a log which was provided along with the test ROM.

As the invalid opcodes were no implemented in the project those tests were skipped.
Testing this type of software happens to be challenging. Even the fact that a pre-written testing ROM was used to identify if any bug left in the system, finding the actual source of the problem was proved to be challenging. Mostly to follow through instruction by instruction the executed operations to see which instruction's resulted in faulty execution. Besides, it is time-consuming and requires the tester to maintain the track of the execution in real time with the software as the problem source from time to time can even happening in a completely different execution step compared to where the bug was actually found.
The most effective bug findings were the result of using a debugger tool and next to it a pen and paper approach.

Also, each of them after implementation were tested manually as well. Due to the fact that the testing ROM mentioned above was only usable after the whole CPU emulation completed.

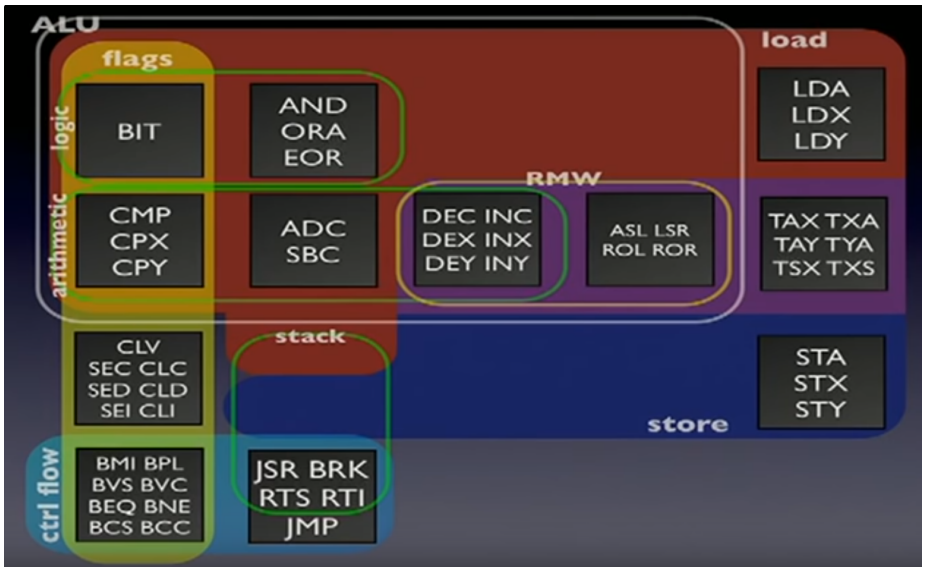

Figure 4.1: MOS 6502 instruction map (Steil 2010)

4.2 RAM

4.2.1 Specification

The NES RAM (NESDEV-Wiki 2016a), unlike RAMs nowadays, fulfils multiple purposes. One of them is providing storage for the game program code, and the other one is to provide a connection for the CPU to the other parts of the console.

Storage The console manufactured with a fix 64KB of RAM. This amount of storage was accessible through a 16-bit address bus and an 8-bit data bus. The fact that the memory is addressed by 16-bit addresses the CPU required extra cycles to read and write from the RAM as it can only handle 8-bit values. This is where the different addressing modes for the instructions became effective useful to provide faster access to the RAM, therefore, increase performance.
Interestingly from the 64KB of RAM, only 32KB were actually available for the programmers to store their games' binaries. Also from this 32KB, another 6 Byte comes down which are the necessary reset vectors reserved for the CPU to specify where to search for the first instruction after power-on/reset or where to jump to handle an interrupt. Other than that the developer has full access over this 32KB so-called PRG ROM or Program Rom. It is separated up to two 16KB chunks as it is stored this way on the ROM Cartridges. The way how it is filled up is defined by the cartridge and its mapper. There are tricks on how to utilize more storage then 32KB but this needs a special Cartridge format, see below at section 4.3.

Zero page On the other hand, the memory has other parts which are still available for the programmer with a few restrictions. Such as the Zero page. This is a special 256 Byte-sized area at the very beginning of the RAM. As this area's address space is 0x0000 - 0x00FF to access this we only need an 8-bit long address, therefore, the accessing speed of this part of the ram is doubled compared to the rest of it. One of the main purposes of this Zero Page area is to provide additional fast storage for the CPU to carry out complex calculations as it is already mentioned above the NES' microprocessor only contains 3 general purpose registers from which 2, the X index and Y index, has only limited capabilities for arithmetic operations.

Stack This area located right after the zero page at the address from 0x0100 up to 0x2000 reserved for the stack operations which are carried out either by a subroutine/interrupt call or forced by the programmer with the necessary CPU instructions. Interestingly, unlike on the modern systems, stack overflow never happens. Instead, the stack wraps around and starts overwriting itself. Of course, this behaviour contains the pros and cons. For example, this type of stack provides more flexibility for the developers as they can exploit this behaviour, on the other hand, it could results loss if information and crash the program as the return from interrupt or subroutine will put the microprocessor

to a wrong address, therefore, causing memory corruption.
Important to notice that as the stack resides in the RAM the developers have read and even write access to these parts of the memory. Therefore the data stored here can be manipulated without any stack specific instruction and just treating it as a normal RAM. It is dangerous but widely used at more complex games which are heavily pushing the limits set up by the hardware.

RAM The actual RAM, which is functionally identical to the modern RAMS, are 1,5KB starting from address 0x0200. This is the amount of storage space given to the developers to use it freely for their games. Technically, there is no difference between this section and the Zero Page other than the fact that the Zero page accessing time can be two times faster, therefore, it is better to think about it as some sort of CPU registers for performance reasons.

The Zero page, Stack and RAM section together mirrored three times in the upcoming 6KB to fill the addressing space up to 0x2000. Which means whenever a memory accessing carried out on the address range of 0x0800 - 0x1FFF is equivalent to do it on 0x0000 - 0x07FF.
The reason behind this type of memory mirroring is that when the hardware decoding the given address line a lot fewer pins on the chip are required to do that by limiting down the address space.

I/O Registers The following memory addresses starting from 0x2000 up until 0x0401F are specially mapped to other parts of the NES console. This way the system provides an opportunity to the CPU, along with that to the developers, to control the other hardware parts by writing or reading values from these special locations.
0x2000 - 0x2007 address range are mapped to specific PPU, Pixel Processing Unit, Registers. Which can be used to query the current status of the PPU, control it's behaviour or even to copy data to its VRAM. See Section 4.4 for more details. This range of PPU memory mappings is also mirrored through the following address place from 0x2008 up to 0x3FFF.
In the 0x4000 - 0x401f address range the APU and Controller mappings reside. As this project not implementing the Audio Processing Unit the only important registers are the 0x4016 and 0x4017. These memory mappings provide the possibility to query the two controller's input or change it's reading behaviours.

4.2.2 Implementation

The RAM implementations itself was done by using an array of 8-bit long unsigned integer pointers. Unsigned variables have used the fact, as it was mentioned above, that the CPU along with the whole system handles every value as an unsigned integer. The reason behind using an array of pointers instead of an array of values is the easily mimic the memory's mirroring behaviour. During the initialization of the RAM, the same set of pointers were copied to the given

parts of the array. With this whenever a given address was written every other address, the given element of the array were updated as they all pointed to the same value.
The above-mentioned connections between given memory addresses and the other hardware parts were done via a listener system. Which consist of an EventSource and a Listener. In this scenario, the Memory is the event source and notifies all the subscribed components whenever a read or a write executed on the memory. For instance, the PPU can handle the reads and writes executed on it's mapped registers.

4.2.3 Testing

As the most part of the RAM is just reading and writing the given indexed value of the array the carried out tests were focusing on to make sure that the memory mirroring implemented properly.
This was done by executing a certain set of reads and writes to evaluate that for each write and read all the necessary memory slots were properly updated.

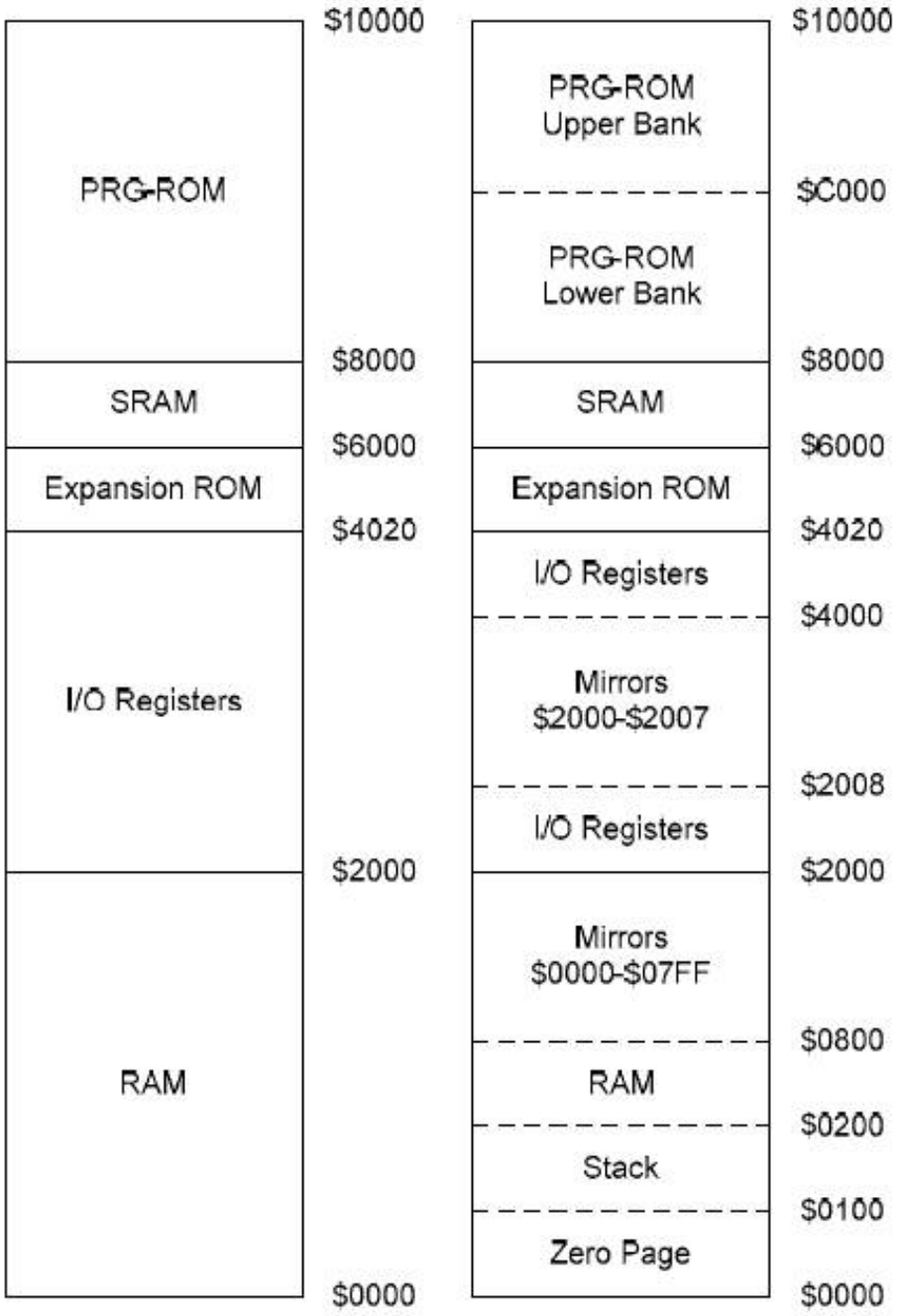

Figure 4.2: Memory map (Diskin 2004)

4.3 Cartridge (ROM)

4.3.1 Specification

The cartridge is a very special part of the NES console ecosystem. As the hardware does not contain any storage capacity on its own, it was solved by distributing the games and the necessary programs on Cartridges. These Cartridges usually contained the program data and the sprite pattern data, for more details about the sprite data see section 4.4. Whenever one of these connected to the console then the data is loaded up from the Cartridge to the RAM and the VRAM.

The console itself does not know where to put the data which was held on the cartridge because they, unlike modern ROMS like DVDs and Blu-rays, came in different layouts. To handle these differences the cartridges contains some extra, special circuits called mappers (wiki 2019). A mapper provides a great potential to the developers to not just deliver the necessary data of the game but to even extend the console capabilities. By design, the console itself provides space for 8KB of sprite data and 32KB for program data. Just like the very first cartridges to the console, see Figure 4.3, came with an 8KB CHR ROM (sprite data) and either 16KB or 32KB PRG ROM (program codes). When the cartridge only contained 16KB of PRG ROM than the memory was filled up by copying the data twice after each other. These were the so-called NROM (NESDEV-Wiki 2016b) mapped cartridges.

Of course, there were cartridges which designed to be more complex such us the MMC5 (NESDEV-Wiki 2019b)). One of the most famous games shipped with this type of Cartridge was the Castlevania III: Dracula's Curse. The MMC5 itself was capable to ship 1MB of PRG ROM and 1MB CHR ROM which is truly extended the possibilities what a developer could achieve. But that is not everything, the cartridge also provided, along with the huge storage capacity, 128KB of extra RAM for the system, extra sound channels for audio, extra interrupts on each scan-line instead of only at the VBlank.
This greatly extendable possibility through different Cartridges was one of the reasons why this system was so successful and could live up until the late '90s.

As these cartridges were designed to especially for the NES console a new file format was designed to hold a digitalized version of these cartridges. This format called Ines (Fayzullin 2005a) or the later extended version the NES2.0 (Horrton 2006). These formats considered by the de facto standard format for storing and distributing these old games. And also to use for emulator developers. The usual structure of such file is the following:

Header section The first 16 byte, which from the first 3 bytes is "NES<EOF>" string, is the header section. These bytes are either flag which provides some

information about how the console different parts meant to be configured up. Or providing details about the actual ROM file by defining what type of binary data represented within the file and what size of the given parts are.

Training Area Right after the header section, if presented, coming up the training area. This is a special, fix 512-byte long sequence which used for providing extra code for compatibility reasons. This either for some special hardware inside the Cartridge or to provide helpers to run a game which was originally developed for a different gaming console but it was ported to the NES system.

PRG-ROM The upcoming part after the training area is the PRG-ROM. This section contains essentially the game logic itself. It is coming on 16KB batches and even if it is not filling up the last batch it has to be padded out to be the right size. This is required mainly for compatibility reasons, to mimic the actual PRG-ROM chip behaviour form the physical cartridges.

CHR-ROM The CHR-ROM is very similar to the PRG-ROM in behaviour within the iNes format. The difference comes from the batch sizes, as it is only 8KB rather than being 16KB. The format itself does not put a restriction on the developer what they can put in here, but the mapper usually loads the data found here to the VRAM's Pattern table.

Miscellaneous ROM The remaining bytes in the file belongs to the Misc ROM. This is rarely used by special types of the console and mapper to provide some extra executable code or audio data.

4.3.2 Implementation

The implementation of this piece of hardware was separated into two parts. First the iNes format parser and secondly the mapper implementation itself.

Parser Once the binary file was opened for reading purposes the first thing which has to be done is to read the first 16 bytes. These bytes are the so-called header of the file. The header includes important information about the cartridge. Such as how big are the PRG and CHR ROMs or is there any training area presented or which mapper meant to be used with this ROM. Once these details are carefully extracted from the header, all the necessary information available for the parser to identify what each byte represents. Knowing that the separate portions of the file can be parsed and presented as a well-structured object which could be used by the mapper.

Mapper As there is a large pool of mappers (wiki 2019) which are used in the NES system, due to time limitation and project scope only the NROM mapper was implemented. But the actual implementation was carried out in mind of the code has to be easily extendable with other mapper implementation. That

led to the current implementation which consists of an abstract Mapper class and a helper method to create a concrete mapper object based on the parsed Cartridge.

4.3.3 Testing

The testing of the parser was implemented in a rather simple way which consisted of multiple steps. First of all, a NES ROM was selected as the target of the test, due to the nature of the test it could be any valid iNes binary file. Then the selected file was parsed by our emulator.
Based on the result of the parsing the source file was replicated then checked off the source binary file is exactly the same as the replica.

This theoretically supposed to be enough but as new logic introduce, namely the file replication from the parsed format, another step was required to verify it to be 100% right. In the next step, the file which was the result of the replication was also parsed and checked if it is equal with the previously parsed object from the original test file.

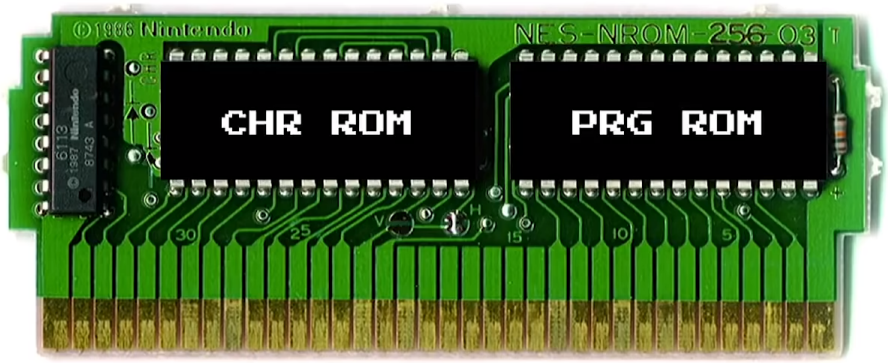

Figure 4.3: NROM cartrdige (Joudrey 2017)

4.4 PPU

4.4.1 Specification

The PPU (NESDEV-Wiki 2018a), Pixel Processing Unit, is one of the most complex part of the NES system, form an emulator development perspective. Compared to it's time it was a really advanced chip. Just like the CPU it was also manufactured by Ricoh and had two different versions the RP2C02 and the RP2C07 for PAL regions. The chip itself contained 16KB dedicated VRAM which was used to store the screen and sprite graphics. In terms of speed the PPU runs 5,36MHz, this results that meanwhile, a CPU does 1 cycle the PPU executes 3 of them. And as the name show these cycles is used to draw the screen pixel by pixel with a 60FPS, Frames Per Second, based on the settings of the chip and the content of it's dedicated ram. Both of these are accessible and modifiable through memory mapped register for the CPU, therefore to the developers as well.

The PPU contains 9 registers which are mapped to a given address of the main memory of the system. Each of these registers can be read, write or both to control the behaviour of the PPU (Fayzullin 2005b). Whenever any of these accessing happens it goes through a special latch. This latch is capable to hold data temporarily whenever a read or a write happens. It is most useful when larger 16-bit numbers are transferred between the CPU and the PPU as both chip handles only 8-bit values.

PPUCTRL This register mapped to the main memory at 0x2000. It can be used by writing an 8-bit value to the address where each bit has its own effect on the PPU.
Bit 0 and 1 can be used to select which name table out of the four meant to be the main one.
Bit 2 sets development perspective. Compared to it's time it was a really advanced chip. Just like the CPU it was also manufactured by Ricoh and had two different versions the RP2C02 and the RP2C07 for PAL regions. The chip itself contained 16KB dedicated VRAM which was used to store the screen and sprite graphics. In terms of speed the PPU runs 5,36MHz, this results that meanwhile, a CPU does 1 cycle the PPU executes 3 of them. And as the name shows these cycles is used to draw the screen pixel by pixel based on the settings of the chip and the content of it's dedicated ram. Both of these are accessible and modifiable through memory mapped register for the CPU, therefore to the developers as well.

how the memory address pointer incremented after each data writing. This is mostly used for name tables, as it is provided with an option to fill them line by line or column by column.
Bit 3 provided an option to select which pattern table used to supply the graphics details for sprite rendering.

Bit 4's behaviour is the same as bit 3's, the difference is that this bit set the source palette table for the background evaluation.
Bit 5 was the flag which used to toggle between 8x8 and 8x16 sprite size.
Bit 6 is meant to be used to change the behaviour of the EXT pin on the PPU chip but due to its formation when it set it can damage the board, therefore, it is highly discouraged to set this flag.
Bit 7 is the NMI flag. When this flag is set and the PPU reaches the VBlank state it generates an NMI interrupt in the CPU.

PPUMASK The register which is mapped to the 0x2001 memory address is responsible for masking options. It is write-only. With these writes the way how the PPU colours the pixels can be changed.
Bit 0 is used to toggle greyscale colour rendering.
Bit 1 can be used to toggle the rendering of the background for the first column of the screen (8 pixels on the left side of the screen).
Bit 2 identical with bit 1 except it used for the sprites.
Bit 3 is the flag which toggles whenever the background should be rendered or not.
Bit 4 is again identical with bit 3 but it toggles the sprite rendering.
Bit 5, 6 and 7 used to make the Red, Green and Blue colour displayed stronger.

PPUSTATUS This is a read-only register, at memory location 0x2002, used to report back events which generated meanwhile the screen is rendered. Also reading this register has a side effect on the above-mentioned latch which is used for communication between the CPU and the PPU. Namely, after every read Bit, 7 became 0 and the address latch is set back to the default name table address which is set by the control flag.
Bit 5 is set when in a single scan line more than eight sprites meant to be rendered.
Bit 6 is set when any of the non-transparent parts of a rendered sprite overlaps with a non-transparent part of the background.
Bit 7 is set when the PPU reached the VBlank state.

OAMADDR This write-only register used to specify the OAM, Object Attribute Memory, address. This address fulfils multiple purposes. One is to provide an address which specifies the data which should be read or overwritten by the OAMDATA registers. The other is to set which byte is the first byte used in the OAM ram when the sprites are rendered on the screen. This register is mapped to 0x2003.

OAMDATA This register used to either retrieve data from or set it in the OAM memory. Both of these functions can be achieved with the corresponding read or write to the 0x2004 memory location. To carry out both functionality the OAMADDR register's value used as an address. Important to notice as this

register modifies the sprite data it is highly discouraged to write to it meanwhile the screen is under rendering to prevent rendering errors.

OAMDMA OAMDMA register is used to trigger a DMA, Direct Memory Access, by writing an 8-bit value to the 0x4016 memory address. This 8-bit value will be used then as the high byte for accessing the memory and filling up the OAM memory by copying the values from 0x##00 - 0x##FF memory location where the ## is the 8-bit value above mentioned. Meanwhile, DMA is carried out by the PPU, both the data and address bus of the main memory is occupied therefore the CPU is unable to fetch the next instruction and carry on until the DMA is not finished. Due to this and the fact that the PPU works 3 times faster than the CPU, this is the preferred way to populate the OAM memory.

PPUSCROLL PPUSCROLL register used to modify the background rendering starting point, which is set by the PPUCONTROL register, by adding a certain offset to it. This offset can be specified by two writing to this register at memory location 0x2005. At the first write the X offset will be set and on the second write, the Y offset will be set. It is highly recommended to execute a PPUSTATUS read before this operation to make sure the base address is pointing to the right place and no garbage value left behind by any previous operations.

PPUADDR To set the value of this register two writes necessary to the 0x2006 address, as the VRAM, just like the main memory, working with 16-bit addresses. Important to notice that the first the upper 8-bit, and with the second writing the lower 8-bit can be set.

PPUDATA Right after the PPUADDR at 0x2007 this register responsible for writing data or return the data from the VRAM address specified by the previously mentioned register. After each writes/read the PPUADDR register will be incremented based on the settings specified in the PPUCONTROL. Along with the address increment, there is another hidden behaviour. Every read for this register is buffered, which means to read the right value after changing the address register two reads has to be performed and with the second read, the right value will be returned.

The PPU's own dedicated VRAM. Even though the 16-bit addressing let the system address up to 64KB, in reality, it only uses 16KB due to cost efficiency in my when the hardware was assembled. Therefore each access which targets address 0x4000 or above will be mapped down to an appropriate address.
The VRAM is separated up to multiple sections. Each of these combined together producing the picture which is displayed on the television screen.

Pattern Table The pattern table NESDEV-Wiki 2015b is a 4KB section inside the VRAM. It is occupying the 0x0000 - 0x0FFF and 0x1000 - 0x1FFF address space as there is two of them. These pattern tables are used to store sprite patterns. Each of these sprite patterns describes an 8x8 pixel blocks. A sprite pattern consists of 16 Bytes which used to describe which colour meant to be used from a certain palette to draw each of the pixels within this block. Each of these pallets contained 3 different colours and background colour, therefore 2 bits was required for each pixel to identify which colour needed to be used. That is why the 16 Byte was utilized in a certain way. The first 8 byte sets the low bit of the colour by storing each 8 wide line information as a byte from top to bottom. The second 8 byte works in a similar fashion but it provides a high bit of the colour.
The above-described structure also well represented in the actual addressing as well. Whenever a byte in the pattern table addressed the top 4 bit sets which pattern table are meant to be used it could be either 1 or 0. The next 8 bit used to identify which pattern is targeted. Meanwhile the least significant 4 bit of the address used to identify the number of the pixel rows, 0x0 - 0x7 low bit rows, 0x8 - 0xF high bit rows.

Name Table The name table (NESDEV-Wiki 2019c) is the part of the VRAM where the background is stored. One of the name tables holds enough information to cover the whole screen. To do this, a single name table is 960 bytes long which is enough to fill the while screen with 8x8 pixel blocks, where each block is represented by a single byte. The byte was needed to store the pattern index for colouring the block. Inside the memory, the name table bytes are aligned by row by row starting with the byte of the top leftmost 8x8 pixel block. Although to provide scrolling functionality more than one was needed.
The PPU designed to be able to handle four of them to provide proper vertical and horizontal scrolling functionality at the same time. To do this the PPU used the four name table as they are ordered in a 2x2 manner and a 1x1 name table sized portion was selected from this to be drawn on the screen. The four name table can be found in the memory each starting at the given addresses: 0x2000, 0x2400, 0x2800, 0x2C00.
But as the system was not shipped enough VRAM just for two name table the other two were just mirrors. The way how the mirroring executed is specified by the cartridges. There is two way how the PPU on its own can implement the mirroring. Either vertical, when the top 2x1 name table equals the bottom 2x1 name table or horizontal when the left 1x2 name table is mirrored to the right 1x2 name table section. In these cases, background scrolling only works in one direction.
On the other hand, games like the Super Mario Brothers 3 implements scrolling on both direction at the same time. This was achieved by shipping extra ram for the PPU inside the Cartridge. This extra ram then used to provide enough space to use 4 different name table at the same time.

Attribute Table Right after each name table in the memory, there is a 64-byte long attribute table (NESDEV-Wiki 2019d). This section of the memory holds data that provides information about which colour palette used for each 8x8 pixel block described in the name table. As the size of this section not enough to cover each name table entry one by one, a more compressed structure is used.
Each 8-bit entry, that contains 4 different 2-bit long pattern table index, refers to a 4x4 block of name table entries. The 4 indexes is divided up between the included pixel blocks by following the convention that the least significant 2 bit refers to the top-left 2x2 block, within the 4x4 target, the next 2 bit specifies the top-right section, the 4th and 5th bit belongs to the bottom-left one and the 2 most significant bit describes the bottom-right 2x2 name table entry block.
Although, this solution is storage efficient it comes with certain drawbacks. Mainly it reduces how many colours can be displayed on the background per frame as each colour palette is referenced by multiple adjacent nameable entries. Also due to this grouping, when screen-scrolling involved the side of the screen can suffer from a colour glitch, this can especially be noticed at the right side of the screen while playing Super Mario Brothers 3.

Palette Table The palette table (NESDEV-Wiki 2018c) holds the colour palette definitions which used for both background and sprite rendering. Each palette consists of 4 different values which can be used to colour the given segment of the picture. In the VRAM there is 32 Byte space available, starting from 0x3F00, to define these palettes. Each of them takes up 4 byte, where only 3 Byte represents a real colour index from the NES colour palette, meanwhile, the first one is always mirrored to the background colour.
As the size indicates 8 colour palettes can fit in at the same time. From which the first 4, 0x3F00 - 0x3F0F, used to indicate the colours used for the background rendering and the second 4. used for the sprite rendering.

OAM Along with the VRAM sections described above, there is a separate 256 Byte area called OAM, Object Attribute Memory, (NESDEV-Wiki 2018b) where the sprites attributes are stored. This provides enough storage for 64 different sprites as each sprite attribute is 4 byte.
The 1st byte and the 4th byte specifies the Y and the X coordinate of the sprite on the screen. The 2nd byte gives the index of which pattern meant to be used to render the given sprite. Closely related to that the 3rd byte provides some flags which can change the behaviour of the rendering for the specific sprite only.
With the bit 0 and 1, the colour palette for the sprite can be set. Bit 7 and 6 provides the capability to draw the sprite vertically and/or horizontally flipped, this is a very usefully capability of the PPU which could save up a huge amount of palette RAM. For example the big mario character instead of using 6 different patterns it can be made using only 4. by using a flipped version of the chest and leg part.

Also, with bit 5 we can set the rendering priority of the sprite. Which could result that the sprite will be rendered behind non-transparent background elements.

Along with storing the necessary graphics data, the PPU's other important responsibility is to draw the screen. The way how this process works is in close relationship with the way the CRT screens behaviour. After the PPU went through it's warm-up time which is roughly 88974 cycles long it starts rendering the screen scan-line by scan-line.

A scan-line is a horizontal row of pixels, and it is starting at the top left corner of the screen. Each of these lines contains 256 pixels and there are 240 scan-lines. A complete render cycle of the PPU contains 261 scan-line evaluation which roughly equals to 890001 PPU cycle or 29667 CPU cycle. In the first scan-line the PPU usually just repopulate render data to its register and sets/clears the necessary state flags to begin the actual screen rendering at the next scan-line. The actual screen rendering takes place between 1 and 240 scan-line. Once it is finished the remaining 20 scan-line is the VBlank state. This state is when the CRT screen not drawing anything on the screen as it reached the bottom right corner, therefore to start the drawing again it has to reposition to the top-left corner. At the beginning of the VBlank, an NMI interrupt fired by the PPU to notify the CPU that the screen rendering is finished and the VRAM is available for safe reading/writing. The CPU has 2160 cycles then to run any game logic which is needed to update the VRAM for the next screen rendering. If the CPU is not finished with this operation in time then it could corrupt the graphics which is being rendered as both hardware trying to read/write the same memory location.

Once the rendering is started, it is time to evaluate the content fo the VRAM to produce the pixels which need to be drawn as a background and sprite separately. First comes the background evaluation (NESDEV-Wiki 2019e). In this case, the name table byte, the corresponding attribute table byte and the two patterns bite loaded into the internal latches. Then the PPU shifts out the correct pixel values and looks up the necessary colour from the colour palette for them to display it. Then proceeds to the next pixel. These fetch/draw operations repeated for every 8 cycles in the scan-line. Also at the end of each scan-line two bytes are loaded in for the next one.
The colour of each pixel is evaluated by following the given formula: 0x3F00 + (two attribute table bit-shifted left by 2) + (the two bit from the pattern table). This will produce a 16-bit long address which then can be used to load in the 8-bit long colour index. This colour index is within the range of 0x00 - 0x3F. Then this is used to look up the colour definition, see Figure 4.6.

Along with the background evaluation the sprite drawing (NESDEV-Wiki 2019f) took place here as well. For each scan-line, the PPU goes through the

OAM and search for sprites which are located in the given scan-line. Then fills up the secondary OAM with the found sprites. The first eight will be drawn on the screen, if more then eight found then the Sprite overflow flag in the PPUSTATUS register will be set to indicate the user that not all of the sprites will be drawn on the screen.
Pixel evaluation for sprites is very similar to the background evaluation. The difference is the Y and X coordinate of the top-left pixel of the sprite is stored in the OAM along with the sprite attribute. Which not only provides the 2 palette bit, that can be used for the same colour evaluation but with 0x3F10 base, but also gives extra flags for fine positioning. Such as vertical/horizontal flip and order of the sprite as it should appear behind or in front of the background.

```
Bit Planes                Pixel Pattern
$0xx0=$41  01000001
$0xx1=$C2  11000010
$0xx2=$44  01000100
$0xx3=$48  01001000
$0xx4=$10  00010000
$0xx5=$20  00100000             .1.....3
$0xx6=$40  01000000             11....3.
$0xx7=$80  10000000  =====      .1...3..
                                .1..3...
$0xx8=$01  00000001  =====      ...3.22.
$0xx9=$02  00000010             ..3....2
$0xxA=$04  00000100             .3....2.
$0xxB=$08  00001000             3....222
$0xxC=$16  00010110
$0xxD=$21  00100001
$0xxE=$42  01000010
$0xxF=$87  10000111
```

Figure 4.4: Pattern table example (NESDEV-Wiki 2015b)

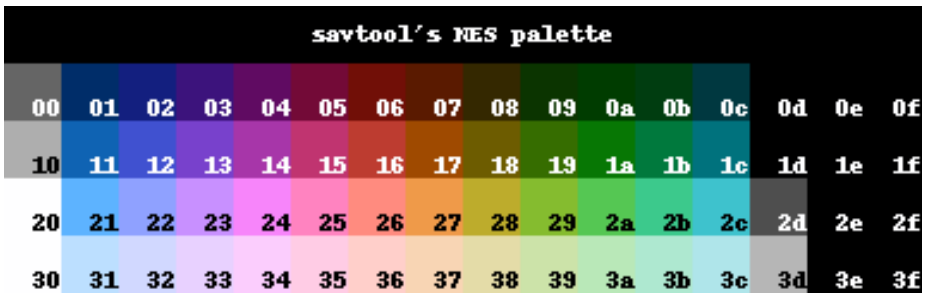

Figure 4.5: NES Colour palette (Tepples 2015)

4.4.2 Implementation

Unlike the real hardware in the current level of the implementation, the PPU and the CPU executes their cycles after each other instead of running parallel. As the microprocessor not working in a cycle based way but executing instructions after instruction to line up with this behaviour the CPU broadcasts how many cycles were required to execute the given instruction once it's done. Then the PPU listens for these broadcasts and executes just enough PPU cycles as many would have been executed along the CPU. As it is not a cycle accurate approach during the planning phase it was kept in mind that later on the implementation should be able to easily refactored to concurrent execution.
Of course, this serial execution sequence has its drawbacks on performance. But based on how fast even the low tier CPU-s nowadays it is hardly noticeable.
As it above mentioned in the specification, the implementations are also separated into two different part, the memory handling and the rendering.

Memory handling The base concept of the implementations was exactly the same what was used at the Main Memory. But instead of everything stored within a single array, in this case, two was used. The reason behind this was that the OAM has a separate address space from the rest of the VRAM. Therefore a separate set of accessors were defined for that. This design choice was also justified by the way how the PPU registers handlers the separately the OAM and the rest of the VRAM.
Along with the rest of the register of the PPU, the accessors for the VRAM was connected to Memory events. The PPU holds a reference both memory and listening to Memory read and write events to trigger the necessary accessor calls or register modifications.

Rendering For rendering the plan was to build it the easiest way as possible and providing just enough feature to be able to run the simplest games which were released to the console. And later on, once it is finished and working as it is intended then gradually build up toward the perfect cycle accurate rendering. The bare minimum implementation contained a basic background colour painting, background sprite evaluation, sprite drawing on top of the background with the possibility of horizontal and vertical flipping. And providing an interface for GUI implementation on top of that. Which in this case was implemented with SDL2 (SDL2 2019), but as long as the necessary Renderer functions are provided SDL could be swapped out for any other UI library.
To provide the necessary pixel information for the UI a simple whole-screen rendering was implemented. This greatly reduced the complexity compared to the scan-line based one. Simply after enough PPU cycle passed, 74640 as there are 240 scan-lines and each of them takes 311 cycles, simple the whole screen evaluated and rendered out by following the given order. First, the background painted on the while screen because every palette has the same background colour, therefore no need to handle it later. Then the background evaluated by going through the selected name table and rendered every 8x8 pixel block where

a given colour was set. Once the background rendering finished the next step is the sprite rendering, which in some ways similar to the background except sprite direction flips and better position handling was used. Then in the next cycle, when the VBlank started, the NMI interrupt is triggered therefore the CPU has an opportunity to refresh the VRAM with the new content.
In its current form rendering does not support scrolling behaviour, sprite hit check and any other more complex solutions as they are barely used in NROM games, which are the first goal to run.

4.4.3 Testing

Although amputated Behaviour Driven tests were developed for the PPU, they only covered minimal VRAM access cases. Most of the memory access and rendering tests were carried out manually by using test Roms (NESDEV-Wiki 2019a) developed by the community. Or using actual game titles and running them on the emulator and on a NES Classic Mini, and comparing the two outputs.
As this part of the is by far more complex and heavily coupled together, it was important that the developer who wrote the emulator and the test is different. The reason behind this is that if the developer understanding on the system/ background knowledge is not precise the test could easily written to be false positive. Therefore a third-party test, which is widely recognised within the NES developer community, is used then this could be easily avoidable. Mainly the following test Roms were used:

color_test.nes Created by rainwarrior (Smith 2015), this ROM provides a simple interface for testing colour palette rendering and the different masking behaviours of the PPU.

vram_access.nes This test was created by Blargg (Green 2005), This heavily tests VRam accessing through the CPU.

sprite_ram.nes This test was created by Blargg (Green 2005), the goal of this test is to verify if the system executes OAM read/write and DMA requests.

Chapter 5

Evaluation

Emulators by nature can be evaluated by looking at two important aspects of the software. First of all, how accurately the system was emulated by the software. Which includes from the very basic components up until the most complex and edge-case behaviours even the bugs as well.
Secondly, how well the emulator performs, as it is technically a "hardware" run on real hardware. It could put great strain on the target computer and provide bad experience for the users or even worse make it unable to run on most of the systems due to the high requirements.

5.1 Emulation Precision

The main memory of the system is precisely emulated unlike the rest of the parts.

The processor is nearly at the perfect state. Compared to the specification all the behaviours are well implemented along even with the known bugs of the operations codes. On the other hand, the so-called unofficial opcodes implementation are missing. These opcodes are never really meant to be used but some of them provide really interesting and useful behaviours which are used in some of the more complex games. Based on how rarely they are used it is not essential to have but it would greatly increase the user's experience by supporting more games. Unlike the unofficial opcodes, the cycle accurate CPU emulation would be a much-needed feature which this emulator lacks at the moment. The current solution, the instruction based stepping, is not bad and it would be sufficient up to the mid-high tier quality games. But above that, the games are pushing the limits of the hardware and heavily relying on every small detail to push out quality gaming experience and this solution would not be sufficient for that.

Cartridge handling has a strong and stable foundation. But, roughly more than 400 mappers exits for the NES system meanwhile this emulator only han-

dles one, the most basic NROM mapper. As this type of Cartridge mostly contained the launch titles of the console, they barely tapped into the potential of the hardware itself and not even seeking about the wast ability of the NES to expand its hardware with special Cartridges. Also, the most memorable games are all developed on a more complex mapper and there is a high chance that the users will find no support for their most precious game within this emulator.

Probably the biggest weak spot of this emulator is the PPU implementation. As it is just barely emulated with enough precision to run the most basic games. The main issue here is the rendering precision. As without scan-line based rendering the system would not be able to render games which are contains split screens or using any special effect which requires a careful interaction with the PPU meanwhile it is rendering the scene. Also, there is a missing specification implementation which is required by even the not so complex but more appealing titles the scrolling mechanism.
Therefore at its current immature state, the PPU severely holding back the list of the supported games.

5.2 Performance

As the emulators, generally speaking, emulating a set of hardware on top of another hardware through software the system requirements for these are quite high compared to a normal program. As there is a certain hardware specification which the emulator has to provide runtime even with great optimization it will consume at least as many resources as the target system has. Also, the behaviour of the software itself sets up a barrier for optimization. Therefore the goal is to try to get the software's resource consumption as close to the original hardware spec. as it is possible without hurting the accuracy of the emulation.

Due to the project timeline, there was no effort made yet to provide a more optimal solution regarding the system requirements of the emulator. Although the following measurements providing a great starting point for the start of the optimization phase.
The system specification of the development machine which was also used for these measurements are the following:

- 16GB DDR4 2400MHz RAM
- Intell i7-7700HQ, 6MB cache 28GHz (3,6 GHz on Turbo) 4 CORE 8 THREAD
- NVIDIA GeForce GTX 1050 TI 4GB GDDR5 vram
- 256GB SSD storage

The simulated hardware's specification:

- 64KB Ram
- Ricoh RP2A03 1,79MHz 1 CORE 1 THREAD
- Ricoh RP2C02 5,37MHz 16KB + 64Byte vram
- No data storage

The following results which are represented in Figure 5.1 are mixed. In the case of memory usage, the 50MB memory usage is a good level, of course, this could be optimized further but it is not a bad result.

On the other hand, the CPU usage, compared to how powerful the host systems CPU, is a terrible result. Using 100% of a single CPU thread to emulate a system which is by on it's own an at least 1000x weaker in clock speed than the host is terrible. This is an issue which must be addressed. Possible solutions for this is to introduce a proper multithread CPU support and also further investigate what which parts of the application are poorly written in terms of CPU utilization.

The application itself, without the SDL2 libraries, including the executable which contains the execution logic and the shared object file which contains the emulator logic together taking up 516KB space. As is shown in Figure 5.2.

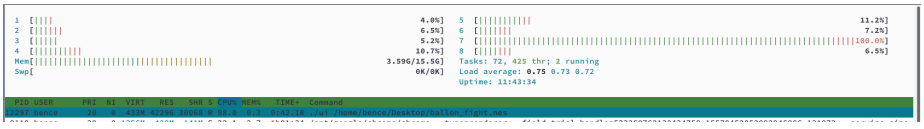

Figure 5.1: Resource usage

```
total 524K
drwxr-xr-x  2 bence users 4.0K Mar 25 05:36 .
drwxr-xr-x 14 bence users 4.0K Mar 25 05:35 ..
-rwxr-xr-x  1 bence users 328K Mar 25 04:21 libemulator.so
-rwxr-xr-x  1 bence users 188K Mar 25 04:21 ui
```

Figure 5.2: Resource usage

Chapter 6

Conclusions

The project at the end reached what it was proposed as a minimal goal. Unfortunately, this goal was not reached without bugs. Currently the there are issues with the rendering regarding colour definitions and some misplaced/missing sprites. But nonetheless the emulator, in its current state, is capable to run simple games without any modification to the original source code itself.

The journey through the learning and implementation of the NES system was truly beneficial and a great experience. It was astonishing to see how the original developers of the console solved the issues within the context of the available technology of the early '80s. And to see how the game developers exploited every small detail of the system's behaviour to produce a state of the art game titles. Also in the context of the academic studies, it was a great introduction for system level programming which is something completely different compared to what I have done up until this point.
But most importantly the time and pace management experience which acquired through the year thanks to this project. Although in the first half of the project the weekly progress pacing was not well thought compared to the project size. But it was a great lesson for further projects.

Of course, this project has a lot of untapped potential in it for further development. Such as the extra project goals mentioned above.

Bibliography

CMake (Mar. 21, 2019). *CMake*. URL: https://cmake.org/.

cppreference.com (Mar. 16, 2019). *C++ 17 - Feature set*. URL: https://en.cppreference.com/w/cpp/compiler_support.

Diskin, Patrick (Aug. 1, 2004). *Nintendo Entertainment System Documentation*. URL: http://www.nesdev.com/NESDoc.pdf.

Fayzullin, Marat (Jan. 24, 2005a). *Nintendo Entertainment System Architecture*. URL: http://fms.komkon.org/EMUL8/NES.html#LABM.

— (Jan. 24, 2005b). *Nintendo Entertainment System Architecture*. URL: http://fms.komkon.org/EMUL8/NES.html#LABM.

GCC-Team (Feb. 22, 2019). *GCC, the GNU Compiler Collection - GNU Project - Free Software Foundation (FSF)*. URL: https://gcc.gnu.org/.

Gerstmann, Jeff (Nov. 22, 2006). *The Legend of Zelda Review*. URL: https://www.gamespot.com/reviews/the-legend-of-zelda-review/1900-6162256/.

GNU (May 20, 2016). *Make - GNU Project - Free Software Foundation*. URL: https://gcc.gnu.org/.

Green, Shay (Sept. 15, 2005). *blargg_ppu_tests*. URL: http://blargg.8bitalley.com/misc/6502_asm_test.zip.

Heesch, Dimitri van (Mar. 19, 2019). *Doxygen: Main Page*. URL: http://www.doxygen.nl/index.html.

Holmes, Richard (Nov. 6, 2017). *Writing BDD Test Scenarios - Department of Product*. URL: https://www.departmentofproduct.com/blog/writing-bdd-test-scenarios/.

Hořeňovský, Martin (Mar. 21, 2019). *Travis CI - Test and Deploy Your Code with Confidence*. URL: https://github.com/catchorg/Catch2.

Horrton, Kevin (Sept. 18, 2006). *Nintendo Entertainment System Architecture*. URL: http://forums.nesdev.com/viewtopic.php?p=17727#p17727.

Horton, Kevin (June 9, 2004). *The ultimate NES CPU test ROM*. URL: http://www.qmtpro.com/~nes/misc/nestest.txt.

Jacobs, Andrew (Aug. 9, 2003a). *6502 Registers*. URL: http://obelisk.me.uk/6502/registers.html.

— (Dec. 28, 2003b). *CPU registers - Nesdev wiki*. URL: http://obelisk.me.uk/6502/addressing.html.

— (Feb. 17, 2008). *CPU registers - Nesdev wiki*. URL: http://obelisk.me.uk/6502/reference.html.

Joudrey, Christian (May 4, 2017). *!!Con 2017: Writing NES Games! with Assembly!!* URL: https://www.youtube.com/watch?v=IbS7uEsHV_A.
LaTeX3-Team (Mar. 21, 2019a). *LaTeX - A document preparation system*. URL: https://www.latex-project.org/.
— (Mar. 21, 2019b). *Travis CI - Test and Deploy Your Code with Confidence*. URL: https://travis-ci.org/.
Naberezny, Mike (Oct. 16, 2018). *6502.org: The 6502 Microprocessor Resource*. URL: http://6502.org/.
NesDev (Mar. 7, 2019). *CPU - Nesdev wiki*. URL: https://wiki.nesdev.com/w/index.php/CPU.
NESDEV-Wiki (Oct. 20, 2015a). *CPU registers - Nesdev wiki*. URL: https://wiki.nesdev.com/w/index.php/CPU_registers.
— (Oct. 17, 2015b). *PPU Pattern Tables - Nesdev wiki*. URL: https://wiki.nesdev.com/w/index.php/PPU_pattern_tables.
— (July 8, 2016a). *CPU memory map - Nesdev wiki*. URL: https://wiki.nesdev.com/w/index.php/CPU_memory_map.
— (Oct. 14, 2016b). *NROM - Nesdev wiki*. URL: https://wiki.nesdev.com/w/index.php/NROM.
— (Dec. 15, 2018a). *PPU - Nesdev wiki*. URL: https://wiki.nesdev.com/w/index.php/PPU.
— (Dec. 15, 2018b). *PPU OAM - Nesdev wiki*. URL: https://wiki.nesdev.com/w/index.php/PPU_OAM.
— (Dec. 15, 2018c). *PPU palettes - Nesdev wiki*. URL: https://wiki.nesdev.com/w/index.php/PPU_palettes.
— (Mar. 8, 2019a). *Emulator tests - Nesdev wiki*. URL: https://wiki.nesdev.com/w/index.php/Emulator_tests.
— (Feb. 19, 2019b). *MMC5 - Nesdev wiki*. URL: https://wiki.nesdev.com/w/index.php/MMC5.
— (Jan. 29, 2019c). *PPU nametables - Nesdev wiki*. URL: https://wiki.nesdev.com/w/index.php/PPU_nametables.
— (Dec. 15, 2019d). *PPU nametables - Nesdev wiki*. URL: https://wiki.nesdev.com/w/index.php/PPU_attribute_tables.
— (Jan. 22, 2019e). *PPU rendering - Nesdev wiki*. URL: https://wiki.nesdev.com/w/index.php/PPU_rendering.
— (Jan. 22, 2019f). *PPU sprite evaluation - Nesdev wiki*. URL: https://wiki.nesdev.com/w/index.php/PPU_sprite_evaluation.
Oxford, Nadia (Sept. 21, 2011). *TEN FACTS ABOUT THE GREAT VIDEO GAME CRASH OF '83*. URL: https://uk.ign.com/articles/2011/09/21/ten-facts-about-the-great-video-game-crash-of-83.
SDL2 (Mar. 21, 2019). *Simple DirectMedia Layer - Homepage*. URL: https://www.libsdl.org/index.php.
Sebestyén, Bence (Mar. 9, 2019a). *Nes Emulator API doc*. URL: https://tekercs.github.io/nes_emulator/doc/api/html/.
— (Mar. 25, 2019b). *NES emulator: Emulator::Cpu::Cpu Class Reference*. URL: https://tekercs.github.io/nes_emulator/doc/api/html/classEmulator_1_1Cpu_1_1Cpu.html.

Sebestyén, Bence (Mar. 25, 2019c). *NES emulator: Emulator::Cpu::Register Class Reference*. URL: https://tekercs.github.io/nes_emulator/doc/api/html/classEmulator_1_1Cpu_1_1Registers.html.
— (n.d.). *Project Repository*. URL: https://tekercs.github.io/nes_emulator.
Smith, Brad (Sept. 15, 2015). *Another palette test*. URL: http://forums.nesdev.com/viewtopic.php?t=13264.
Steil, Michael (Dec. 27, 2010). *Reverse Engineering the MOS 6502 CPU 3510 transistors in 60 minutes*. URL: http://mirror.fem-net.de/CCC/27C3/webm/27c3-4159-en-reverse_engineering_mos_6502.webm.
Tepples (Sept. 17, 2015). *Savtool swatches*. URL: https://wiki.nesdev.com/w/images/5/59/Savtool-swatches.png.
wiki, Mapper - Nesdev (Mar. 7, 2019). *TEN FACTS ABOUT THE GREAT VIDEO GAME CRASH OF '83*. URL: https://wiki.nesdev.com/w/index.php/Mapper.
Wikipedia (Mar. 7, 2019a). *Nintendo Entertainment System - Wikipedia*. URL: https://en.wikipedia.org/wiki/Nintendo_Entertainment_System.
— (Mar. 8, 2019b). *Ricoh - Wikipedia*. URL: https://en.wikipedia.org/wiki/Ricoh.