NES Emulator

Sebestyén Bence

March 22, 2019

Contents

Chapter 1

Introduction

The Nintendo Entertainment System, frequently called NES, is an home gaming console developed by the Japanese company called Nintendo Company, Limited. Through it's lifetime the from the earl '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 were 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 the 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. This projects goal 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 games 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 Nitnedo 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 trough 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 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. Controller
5. Cartridge / Mapper
6. APU

This thanks to the fact that 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 instructions on them to trigger given actions on them, 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 each module.

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 it's rich standard library and object oriented programming capabilities.

On the 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 (Bence 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 a main Github repository (Bence n.d.). As the project meant to be published as an open source project it is publicly available. Although, under the time of the project during the course of the project it was also publicly available the repository was strictly available for editing for 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 were 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.

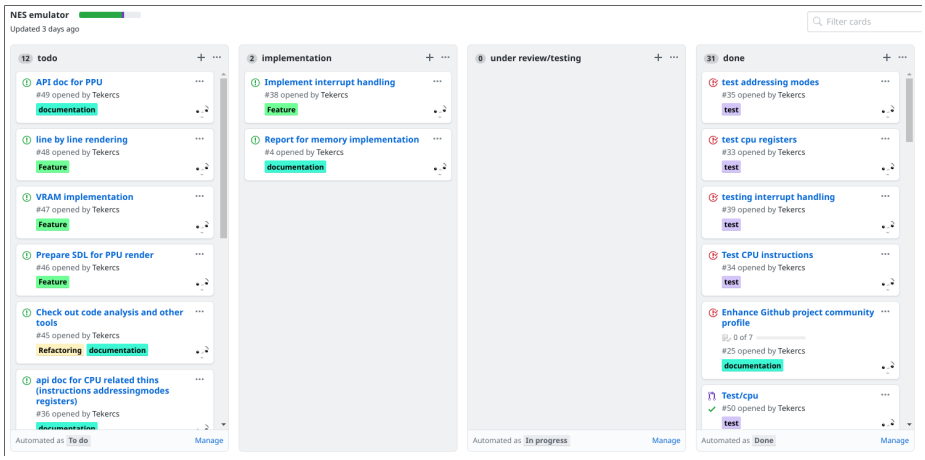

Figure 3.1: Github project page

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 a lover clock speed. The CPU contains 6 registers from that 3 are more general purpose and 3 are special purpose (NESDEV-Wiki 2015).

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 the most of the arithmetic and load/store instructions were implemented for these. Also the result of these operations were 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 can not 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 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 an 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 al the IRQs.

Zero Flag, 1st bit Whenever an arithmetic operation's result are 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 it's instruction set. Due to its 8-bit nature of the microprocessor the number of available instructions are 256. Although not every opcode is implemented officially (Jacobs 2008). The 151 instruction which actually implemented are technically 56 instructions with different addressing modes (Jacobs 2003b). Each instructions has its execution time measured in CPU cycles, which depends on the addressing mode and the length of the instruction itself. Usually the instruction are a 1 - 3 byte 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 byte long
STY $20AA is 3 byte long

Important to notice that the Indirect jump instruction, JMP (opcode: $6C), contains a bug. When ever 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 the behaviour of this instruction.

As it 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:

1. Implicit
2. Accumulator
3. Immediate
4. Zero Page
5. Zero Page,X
6. Zero Page,Y
7. Relative
8. Absolute
9. Absolute,X
10. Absolute,Y
11. Indirect
12. Indexed Indirect
13. Indirect Indexed

4.1.2 Implementation

When it came to implement 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*** (Bence 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 mode instead of the total 151 instruction.
Each of the instruction and addressing mode implementation received it's own function (Bence 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 the 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 was no implemented in the project those tests were skipped.
Testing these type of software happens to be challenging. Even the fact that a pre-written testing ROM were 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 trough instruction by instruction the executed operations to see which instruction's resulted 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 were 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 along the implementation phase when it came to implement the instructions and the addressing modes. 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.

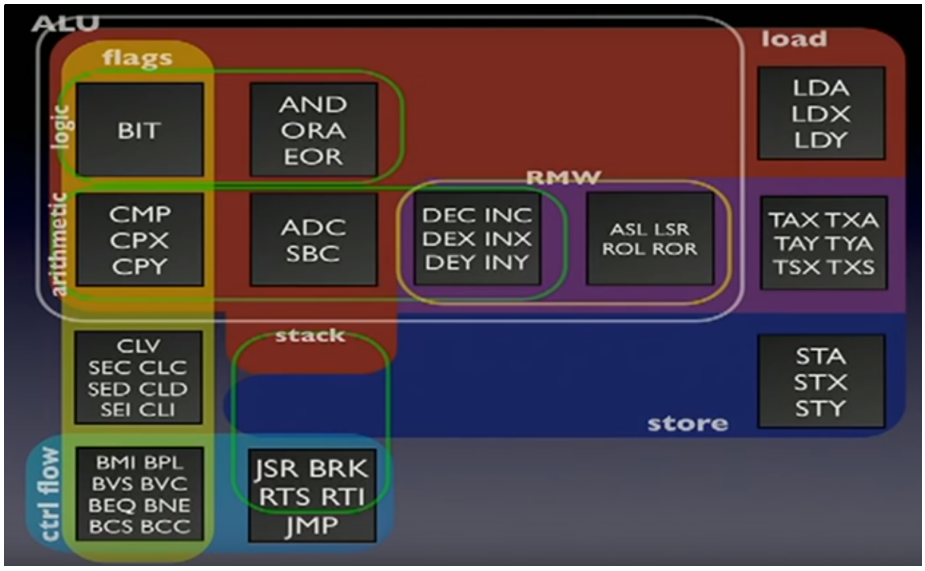

Figure 4.1: MOS 6502 instruction map (Steil 2010)

4.2 RAM

4.2.1 Specification

The NES RAM, unlike RAMs nowadays, fulfils multiple purpose. One of them is providing a storage for the game program code, and the other one is to provide 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 trough a 16-bit address bus and a 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 have 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 it's mapper. There are tricks 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 restriction. 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 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 purpose 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 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 form interrupt or subroutine will put the microprocessor to a wrong address therefore causing memory corruption.

Important to notice that as the stack is reside in the RAM the developers has 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 in functionality 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 less 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 opportunity ot the CPU, along 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 it's VRAM. See Section 4.4 for more details. This range of PPU memory mappings are also mirrored through the following address place from 0x2008 up to 0x3FFF.
In the 0x4000 - 0x401f address range the APU and Controller mappings resides. As this project not implementing the Audio Processing Unit the only important registers are the 0x4016 and 0x4017. These memory mappings provides 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 were 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 th 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 when ever 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 was done via a listener system. Which consist 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. Therefore the 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 buy executing a certain sets of reads and writes to evaluate that for each write and read all the necessary memory slots were properly updated.

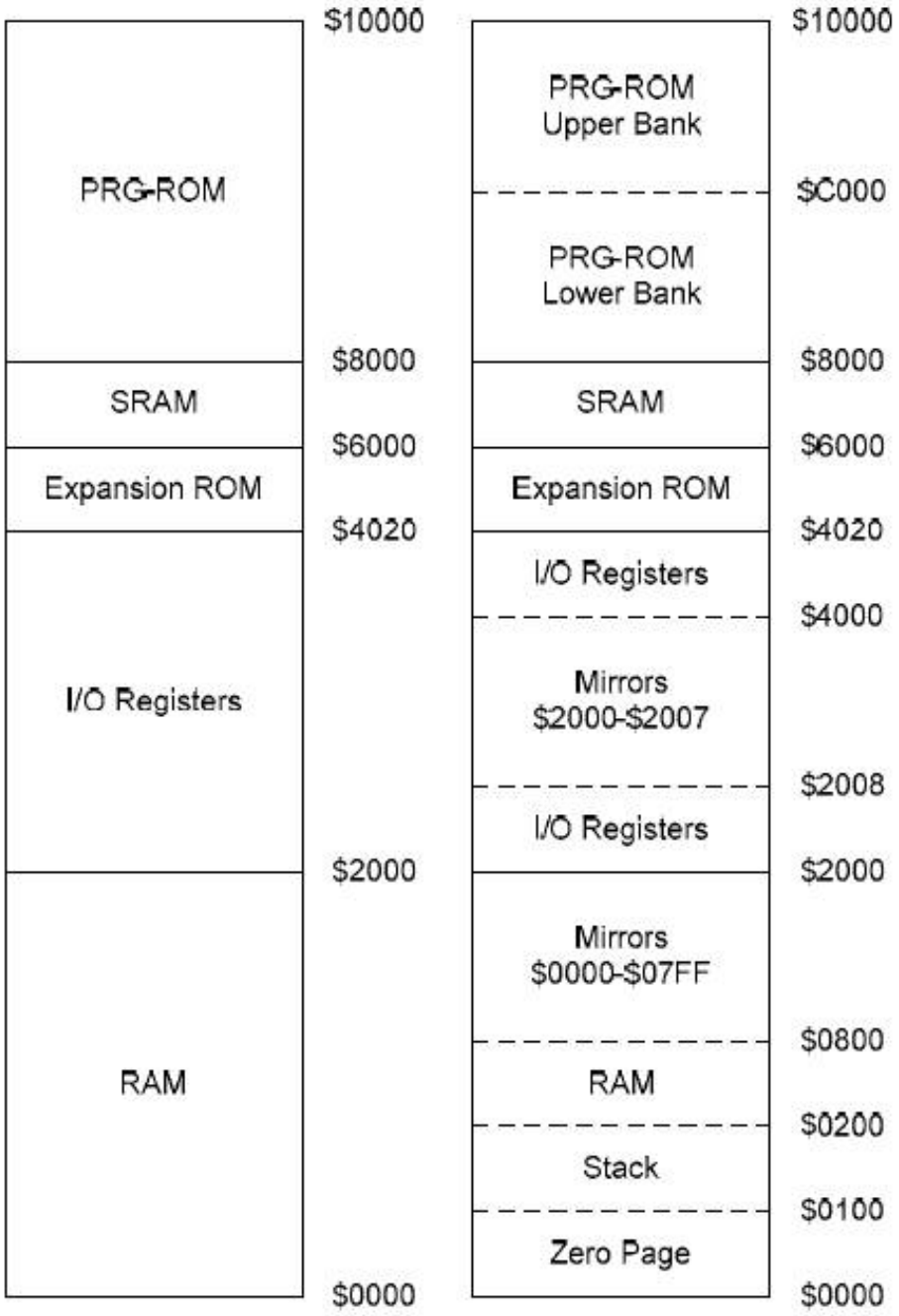

Figure 4.2: Memory map (Diskin 2004)

4.3 Cartridge (ROM)

4.3.1 Specification

4.3.2 Implementation

4.3.3 Testing

4.4 PPU

4.4.1 Specification

4.4.2 Implementation

4.4.3 Testing

Chapter 5

Evaluation

5.1 Performance and Precision

5.2 Game Performance

Chapter 6

Conclusions

Bibliography

Bence, Sebestyén (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.

— (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.

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.

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/.

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.

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.

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

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, 2015). *CPU registers - Nesdev wiki*. URL: https://wiki.nesdev.com/w/index.php/CPU_registers.

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.

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.

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.