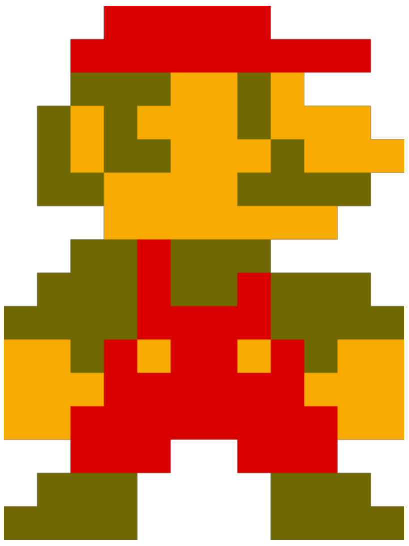
UNIVERSITY OF GOTHENBURG



YλNE

Yet Another NES Emulator

Kandidatarbete inom data- och informationsteknik

TOBIAS OLAUSSON
TOBIAS VEHKAJÄRVI
DAVID HADZIC

Göteborgs Universitet
Institutionen för data- och informationsteknik
Göteborg, Sverige 2009
Kandidatarbete/rapport nr 2009:63

Preface

Many people have grown up toying and playing with some form of console and/or personal computer system, especially with the big boom for home entertainment consoles such as Nintendo's NES in the 1980s. For us three growing up in these exciting times we had the opportunity to use these consoles and play countless games until our thumbs were sore. Remembering these fun times in our youth has been a major motivating factor in making this emulator, to mix our passion for functional programming with being able to play our beloved games was a win-win situation.

The reader should be familiar with Haskell and have a basic understanding of monads and digital computer systems.

This project was supervised by Koen Lindström Claessen.

Sammanfattning

Vi beskriver hur det funktionella språket Haskell kan användas för modellering av hårdvarukretsar på ett enkelt och smidigt sätt. Vidare ger Haskell en hög abstraktionsnivå för beskrivning av den hårdvara man emulerar, vilket ger både läsbar och modulär kod.

Det systemet vi har valt att emulera är den populära 8-bitars spelkonsollen *Nintendo Entertainment System* (NES) som skördade stora framgångar på 80-talet. Vi har lagt fokus på att skriva kod som är lättläst, modulär, och lätt att utöka för vidare projekt.

Resultaten visar tydligt hur Haskell lämpar sig för emulering med hjälp av ADTs, monader samt lathet. Latheten har varit både på godo och på ondo, då bristen på optimeringar i koden har gjort att latheten i Haskell medför långsammare exekvering. Detta har berott till största delen på tidsbrist.

Abstract

We show that the low-level programming and hardware emulation in general coupled with a functional paradigm of a high-level language like Haskell is indeed possible to achieve.

Haskell being a powerful, pure and type-safe language provides high abstraction and enables easy modeling of the hardware in question where different components are modular and decoupled from each other for simplicity and high readability.

Our chosen hardware to-be emulated is the highly popular 8-bit *Nintendo Entertainment System* (NES) console from the 1983 with a lot of effort put in to make the code simple, modular, extendable and if possible, efficient, meaning matching original hardware in speed.

The results in this project show clearly how emulation can be made easy using ADTs, Monads, and lazy lists. Haskell being lazy however, has made it hard to make the code run fast. That has been mostly due to time constraint.

Contents

1 Introduction **6**
1.1 Motivation . . . 7
1.2 Purpose & Goals . . . 7
1.3 Development Cycle . . . 8

2 Technical Background **9**
2.1 CPU Details . . . 9
2.1.1 Instructions . . . 10
2.1.2 Status Register . . . 10
2.2 Addressing Modes . . . 11
2.2.1 Memory Map . . . 13
2.2.2 DMA . . . 14
2.2.3 Memory Mapped I/O . . . 15
2.2.4 Interrupts . . . 15
2.3 PPU Details . . . 17
2.3.1 Memory Map . . . 18
2.3.2 OAM . . . 19
2.3.3 PPU Registers . . . 20
2.3.4 Screen Mirroring . . . 21
2.3.5 Scrolling . . . 21

3 Implementation **22**
3.1 CPU . . . 22
3.1.1 Modelling the Processor . . . 23
3.2 PPU . . . 25
3.3 Synchronization & Communication . . . 27
3.4 Enterprise Pulling . . . 28
3.5 I/O . . . 28
3.5.1 SDL & Graphics . . . 29
3.5.2 Loading & ROM layout . . . 29

4 Discussion **31**
4.1 Results 31
4.1.1 Correctness 32
4.2 Delimitations & Choices 32
4.2.1 Memory Mappers 33
4.2.2 Functional versus Monadic 33
4.2.3 CPU 33
4.2.4 PPU 34
4.2.5 Lazy versus Strict 34
4.2.6 Communication 35
4.3 GHC 6.10 36
4.4 Related Work 36
4.4.1 BeaNES 36
4.4.2 Nintendulator 36
4.4.3 OmegaGB 36
4.4.4 Coroutines 36
4.5 Future Work 37
4.5.1 Sound 37
4.5.2 Mappers 37
4.5.3 State saving 37
4.5.4 Netplay 37
4.5.5 Parellellism 37
4.5.6 PPU Improvements 37

5 Conclusion **39**

A Hardware Specs **43**
A.0.7 NES 43
A.0.8 Test Computers 43

B Dictionary **44**

C Instruction List **46**

D Module Hierarchy **50**

E Contribution Report **51**

Chapter 1

Introduction

The *Nintendo Entertainment System* is an 8-bit cartridge-based video game console first launched in 1983 to instant acclaim. NES quickly became the best selling game console and helped revitalize the rapidly declining home gaming industry.[25] It was a third generation console consisting of an 8-bit MOS Technology 6502 microprocessor produced by Ricoh which incorporated custom sound processing hardware on-die in addition to a *direct memory access controller* (DMA). The *Central Processing Unit* (CPU) ran at either 1.79MHz or 1.66MHz depending if the designated region was using either NTSC (US) or PAL (EU) systems respectivly. Graphics were taken care of by the *Picture Processing Unit* (PPU) also manufactured by Ricoh. The picture processor could display a maximum of 53 colors and had a resolution of 256x240 pixels. The built-in *Audio Processing Unit* (APU) had support for five sound channels with a volume control of 16 levels.

In addition the system contains a mediocre 2KB of *random access memory* (RAM) - connected to a 16bit adress and 8bit data bus respectively - which can be expanded by hardware contained in the game cartridges which of course also holds the necessary media in *read only memory* (ROM).[3]

Using Haskell was the first thing we agreed upon in this project and this made the modelling easy, since the language has support for ADTs. Haskell is statically typed and has type inference, because of that a lot of small mistakes and bugs are caught at compile-time instead of crashing the program at runtime.

Furthermore Haskell uses lazy evaluation which means that no values are computed unless they are actually needed. It follows from the laziness that one may write programs without actually knowing some values; for example in Haskell one can use infinite lists, a feature which is heavily used in our implementation. Since Haskell is a pure language, neither side effects nor variables in the classical programming sense are permitted. This is problematic due to the highly stateful characteristics of a CPU. To handle that, the need for monads arise. Monads provide a way to run actions in a sequential manner, encapsulating possible side effects such as a state.

Emulation is the art of imitating a system on another system be it either hardware or software. The interesting part of emulation is being able to reproduce the exact behaviour of the actual system; in our case the NES. Emulation is also a way of preserving an obsolete system since it might have been out of production for many years but convenient in the sense that a working emulator can run any program written for the NES.[26]

1.1 Motivation

Since the NES emerged, emulation has been on the rise. Most emulators however, are written in ASM/C and are highly optimized for speed. The fact is that many were written a long time ago when computers were not as powerful as they are these days. There are several consequences from that, most consequential being that the code looks awful and is not readable at all. The advantage when using Haskell is that the high abstraction level in the language makes it easy to model complex structures in a readable manner. Another motivating factor is that there are no NES emulators written in Haskell, at least to our knowledge.

1.2 Purpose & Goals

The goal of this project is to show that Haskell can be used in practical applications by implementing an emulator for the *Nintendo Entertainment System*, that is able to run a small subset of games in a clean and readable way by using Haskell's powerful type system and abstraction for modelling the NES hardware.

Since writing a complete emulator is very hard and a tedious process, we have decided on some sub-goals due to the time constraints for this project. We have worked on the project in the following steps:

1. Emulation of the CPU is considered complete when it follows the specification of the 2A03.[5] It should be able to run all supported instructions and return a "correct" state.

2. Emulation of the PPU is considered done when the emulated PPU behaves as the one in the real NES hardware would, that is when the graphic outputted to the screen is the expected one.

3. Synchronization between CPU and PPU is considered to work when a PPU write to a register is reflected in the corresponding CPU memory, and vice versa.

4. DMA is considered to work when it takes 512 cycles to move 256 bytes from CPU memory to PPU memory.

1.3 Development Cycle

We have followed a simple scheme for developing the application. It is always a good idea to start with research on the subject. To test and debug is also central to any programming project.[27]

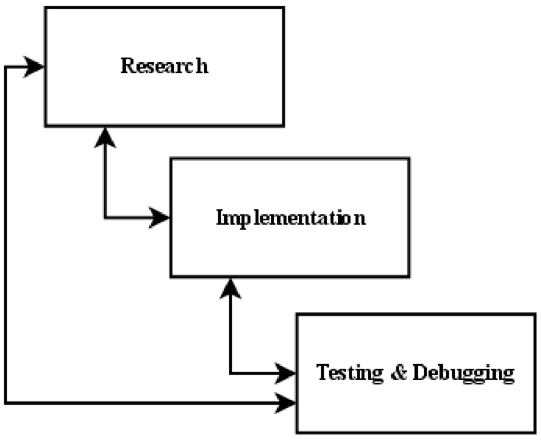


Figure 1.1: Modified waterfall model

Chapter 2

Technical Background

The NES architechture components are interconnected using a 8-bit control bus, 8-bit data bus, and a 16-bit address bus. There are two main computing components; the CPU (main data processing), and the PPU (graphics processing and rendering). On top of the CPU is an APU (audio processing unit) for producing sound.

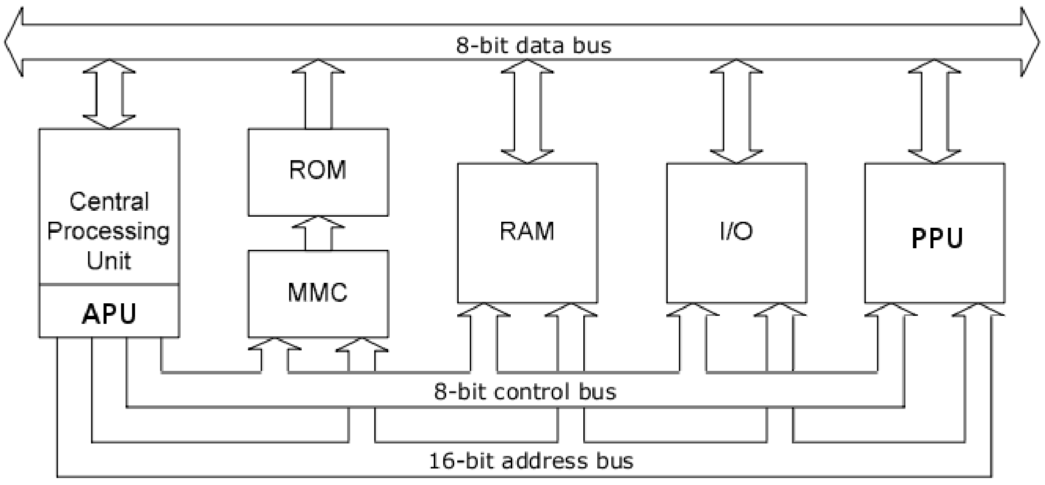


Figure 2.1: NES Architecture

2.1 CPU Details

The 2A03 or 6502 CPU as it is more commonly known as has 56 instructions. Together with the numerous ways a memory location can be addressed it can execute a total of 151 operations.[7] Furthermore it has three 8-bit general purpose registers: *accumulator* and two *index registers*. The remaining three registers are: *stack pointer*, *status register* and the 16-bit *program counter*.[1] Registers are very fast, small storage areas in the CPU, that are used for various computations and temporary storage.

2.1.1 Instructions

An instruction is indexed by its opcode, a hexadecimal value between 0x00 and 0xFF. When executing instructions, the cpu fetches and decodes the opcode, and runs a specific instruction. Each instruction takes some fixed number of cycles to execute, which is due to the internals of the CPU with its bus and memory. For example, there can only be one value on a bus at the same time.

The 56 instructions[9] can be categorized into different groups as follows:

- load/store - LDA, STA, TXS ...
- arithmetic/logic - ADC, ROLA, EOR ...
- branches/jumps - BEQ, BCC, JMP ...
- miscellaneous - SEC, BRK, CLI ...

2.1.2 Status Register

Status register or flag register holds the necessary state of the processor in between execution of instructions. This register holds 8 different flags where each flag is represented by one bit and the flags being set or cleared depending on the instruction and its operands being executed.

Carry Flag (C) - The *carry* flag is used to detect overflow in unsigned arithmetic operations, for example adding 255 with 1 would result in 0 since we are dealing with an 8-bit architecture, and the carry being set. So by using only one bit to represent overflow one can have support for calculations on numbers longer than 8-bits.

Zero Flag (Z) - The *zero* flag is set when the result of the last performed instruction was zero. It is often used when looping and branching while decrementing the accumulator, for example.

Interrupt Mask (I) - When set the system will be prevented from executing interrupts, although non maskable interrupts can not be disabled.

Decimal Mode (D) - Used to switch the 6502 to *binary coded decimal* (BCD) mode, however this is is not supported by the 2A03 so this flag is ignored.

Break Mode (B) - The break flag is set when the *break* (BRK) instruction has been executed causing a software interrupt to be generated.

Overflow Flag (V) - The signed equivalent of the carry flag, meaning that a positive result was generated when a negative one was excepted and vice versa. For example adding 127 with 1 results in -128 and the overflow flag being set.

Negative Flag (N) - Also known as the sign flag, is used to represent the sign of the last calculation. This flag only mirrors the seventh bit of the result since that last bit tells if the number is positive (0) or negative (1).

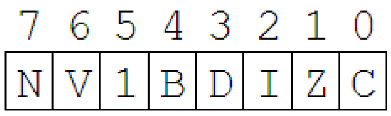


Figure 2.2: The status register[1]

2.2 Addressing Modes

There are several different ways to access the data in memory that one wants to operate on, these are called addressing modes. Many instructions are able to operate using several of these modes, and the following modes exists:[7]

Implicit - The data operated on is implied directly by the instruction. Used by instructions operating on the status register such as *Set Carry Flag* (SEC) for example.

Accumulator - Used by instructions operating directly on the accumulator, e.g. *Rotate Right Acc* (RORA). Very much like *implicit* mode, and internally treated the same.

Immediate - Allows the programmer to specify an 8-bit literal directly following the instruction, such as *Load Accumulator* (LDA #255)

Zero Page - A fast direct access mode that can only operate on the first page of memory from address 0 to 255 hence the name *Zero Page.* Since only one byte, the least significant byte, is needed to access this area, one extra cycle to fetch the most significant byte of the address is saved making the instructions that operate here faster. This also implies that the memory usage is minimized. Since the 2A03 has only a handful of usable registers, code should utilize Zero Page addressing as much as possible. It is also possible to use indexed versions with X or Y register to offset the address, see *Indexed Indirect* and *Indirect Indexed*

Absolute - The programmer supplies a 16-bit literal which is the absolute address operated on. Indexed versions of this mode can be used to offset the address using the X or Y register.

Relative - This mode can only be used together with branches. The programmer supplies an 8-bit signed value, which is added to the *program counter.* Thus, it is only possible to branch 128 bytes in memory in either direction.

Indirect - Like the absolute mode, the programmer supplies a 16-bit address. However, when using indirect addressing, the contents of that address, and the address+1 is interpreted as an address, which is then used. The 2A03 has a wrapping bug concerning this addressing mode; when the address is xxFF, the first byte read will be at 0x23FF (for example), and the second byte will be read at 0x2300. Normally one would expect the byte to be read at 0x2400 (0x23FF+1), but this is not the case here.

Indexed Indirect & Indirect Indexed - Both these modes operate on the zero page. The difference between them being what register is added as offset, and when it is added. The programmer supplies a one byte address pointing to the zero page which means that the address will be between 0 and 255. When using the indexed indirect mode, the contents of the X register is added to the address before it is read. When using indirect indexed, the contents of the Y register is added to the address after it is read.

2.2.1 Memory Map

The 2A03 is able to address up to 64KB memory due to the 16-bit addressing bus. However, the physical memory in the NES is no more than 2KB (addresses 0x0 to 0x800). Since the 2A03 uses memory mapped I/O, some addresses in logical memory are reserved for sound, PPU communication and DMA. Large parts of the logical memory are mirrored several times as well.[3]

The *first page* in memory (*0x0-0xFF*) is called the zero page. The usage of the zero page is special, as many instructions support some of the zero page addressing modes to save memory and gain speed.

The *second page* in memory (*0x100-0x1FF*) is reserved for the stack. The stack can never grow bigger than this, but instead wraps around. Thus, if the stack pointer is at 0x1FF and a new value was pushed onto it, the stack pointer would be updated to 0x100.

The pages between *0x200* and *0x800* are used as regular RAM for storing variables like game data. All memory locations between 0x0 and 0x800 are mirrored three times up to 0x2000. Thus, accessing 0x805 gives the same result as accessing 0x05.

Addresses *0x2000-0x2007* are memory mapped I/O registers used for communication between the CPU and PPU. These registers are mirrored every 8 bytes up to 0x4000.

Addresses *0x4000-0x4020* are used for sound, except for *0x4014* which is used for DMA, and *0x4016-0x4017* which are used for controller input. The addresses between *0x4020* and *0x6000* are used for expansion ROM; the area where memory mappers can swap in memory from cartridges.

Addresses *0x6000-0x8000* are used to access save locations on cartridges. The only way for games to save states is by having a small RAM on the cartridge, powered by a battery.

All addresses from *0x8000* and above contain the program ROM which is the game code divided into two banks. Some programs do not utilize more than one bank, and since *0x8000-0xFFFF* contains two banks (16KB each) the other one starting at offset *0xC000*, that program would be inserted into both banks by the system.[1,3]

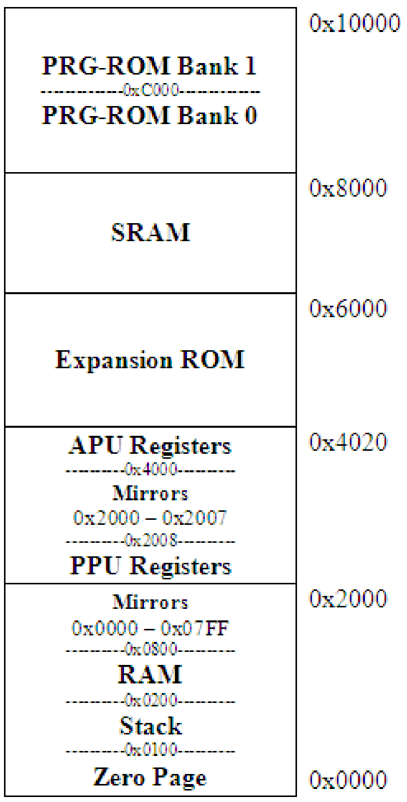


Figure 2.3: The CPU memory map

2.2.2 DMA

Direct Memory Access is a technique using dedicated controller hardware, that enables transferring large pieces of data between the main memory and the PPU memory in an efficient manner bypassing the CPU altogether. In the NES this is used to transfer a large chunk of sprites from CPU RAM to *Sprite RAM* (SPR-RAM) also known as *OAM* in the PPU.[10]

Activating DMA is done by writing one byte to *0x4014*. This byte is multiplied by *0x100* and is used as the base address where data is read from. The NES reads 256 bytes of data, and transfers them to the SPR-RAM of the PPU. The existing data in SPR-RAM is completely overwritten. The whole process takes 512 cycles. Doing everything manually would require the programmer to write an SPR-RAM address to the PPU OAM address register, and then copy the desired byte from CPU RAM into SPR-RAM register. That would take 4 cycles for each byte. However, when DMA is occuring the data bus is busy, so the CPU can not execute any other instructions at that time.[5]

2.2.3 Memory Mapped I/O

The CPU and PPU communicate via memory mapped I/O, on registers *0x2000-0x2007*. For the CPU, this is a regular memory location, but internally it is mapped to a register in the PPU. The same technique is used when the CPU outputs sound to the APU, via registers *0x4000-0x4020*. The addresses *0x4016/0x4017* are the locations where information about input from the controllers is stored.[1,2]

The internal structure of the controller mount point is a shift register. This means that data is shifted into the first bit once every read. Since one has the possibility of having both controller one and three connected to the same input, both are read from the same address. The NES controller has eight buttons which are read in the following order: *A*, *B*, *Select*, *Start*, *Up*, *Down*, *Left*, *Right*. To poll all information from a register it takes 24 reads, divided as follows (for 0x4016): 8 reads for P1, 8 reads for P3, 0, 1, B and after that 4 reads that does nothing. For the second controller, reads 17-19 is 1, 0, B instead.[1] For each read, at most one bit is set (bit 0) for the controller input. However, when using other controllers such as the infamous NES zapper (light gun), other bits in the register indicate if the gun hit a sprite or not. Writing to the controller registers has no effect.[1]

2.2.4 Interrupts

As the name implies interrupts are used to interrupt or prevent the normal sequential execution of code and make the processor jump to a predefined routine to execute some interrupt handling code. They are basically used to get the attention of the CPU for a short period of time and after that the regular execution is continued were the CPU was interrupted. For this to be transparent and the continuation of the execution possible a state needs to be saved before the execution of the interrupt and restored afterwards. For the 2A03, the program counter and the status register is saved on the stack before executing the interrupt handler.

So called interrupt vectors are used to point to the location of the interrupt handling routines, starting at memory location 0xFFFA and stretching to the end of the memory map at 0xFFFF.[2,11]
Interrupts can either be generated in hardware or software and NES has support for both.

The 2A03 has three different interrupts, shown in descending priority:

Reset - Signaled when NES is powered on or the reset button has been pressed. It has its reset vector at *0xFFFC* for the high byte of the address and *0xFFFD* for the low byte.

NMI - Non maskable interrupt is triggered by the PPU hardware after it has finished drawing a frame and is in a state called V-Blank, waiting for the TV hardware to retrace the cathod ray back to the top. This small time interval is the only time frame when the CPU can send data to the PPU memory without causing graphics corruption. It is heavily used in NES games to update SPR-RAM. The interrupt vector is located at *0xFFFA* and *0xFFFB.*

IRQ - Can be generated by memory mappers, or by the BRK instruction. An interrupt request is signaled, and interrupt vector is loaded from *0xFFFE* and *0xFFFF.* Interrupts generated by the BRK instruction are called *software interrupts* and are mainly used for debugging purposes on the NES.[1]

2.3 PPU Details

Graphics and video output are taken care of by the *Picture Processing Unit* (PPU) which is running in parallel to the CPU and was also developed by Ricoh. The PPU features 256 bytes of on-die *object attribute memory* (OAM) and a main working memory, these both are internal to the PPU and separate from the main CPU memory and bus.[6]

The PPU has a color palette of 48 colors and 5 grays but only 25 colors can be used at the same time on one scanline, where a scanline produces a row of pixels outputted to the *cathod ray tube* (CRT) television set. Sprites can be either 8x8 or 8x16 pixels in size and a total of 64 sprites can be displayed on screen any given time without reloading, with a limitation of 8 sprites per scanline. The maximum display resolution for the system is 256x240 pixels due to hardware limitations.[14]

The PPU also known as *2C02* is running approximately three times faster than the CPU depending on the region which equals to 5.37MHz on a NTSC system. The static procedure of displaying frames is executed 60 times per second where each frame consists of 240 visible scanlines which in turn have 256 pixels each. Like the CPU the 2C02 can also address 64KB of memory but it only has 16KB of *video ram* (VRAM) used for storing graphics related content such as palettes, patterns and name tables.

All the communication between the CPU and PPU takes place over the I/O-mapped registers and this is the only way the programmer can receive status information and control the PPU's behaviour. As previously mentioned these registers reside at locations *0x2000-0x2007*.

Since the NES was designed to work with the old CRT-based TVs the designers had to take the way a TV works into consideration due to the electron gun. It takes time to retrace from the bottom of the screen to the top to begin drawing a new frame, and this time is called the *vertical blank* (v-blank) period. It is only during this time that the CPU can transfer data to the PPU memory without risk of graphics corruption since the addresses used by the PPU for drawing can get altered when the CPU is writing to the various shared registers.

The basic idea of the PPU is to for each scanline go through the correct name table entries indexed by the current VRAM-address.[13] Then continuing with the object data from the OAM to find the first eight sprites located on this scanline to be drawn on top of the already generated background. This process is repeated 240 times before the PPU goes into v-blank state.

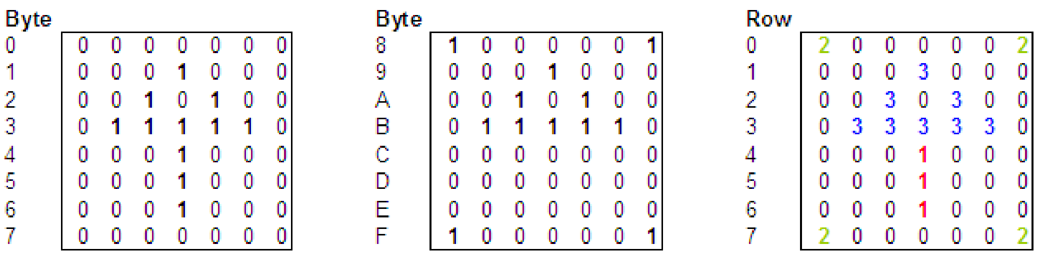
A typical frame takes 263 PPU scanlines, which equals 341*263 cycles, the different phases when rendering scanlines are as follows:[2]

- 0-19: Vertical blank period, this is the time it takes to move the electron gun to the top of screen.
- 20: A dummy scanline used to initialize internal variables and latches, does not produce any pixels.
- 21-261: Visible scanlines, renders pixels, a total of 240 scanlines.
- 262: Does nothing.
- 263: Clean-up and if enabled produces NMI.

2.3.1 Memory Map

The PPU can also address 64KB of memory but the actual size is only 16KB divided into four logical sections as seen in figure 2.5. Starting at address *0x0000* there are two pattern tables also known as tile tables each *0x1000* in size containing background and sprite tile data. These tables are used to store two least significant bits of the color data and together with the remaining most significant bits from the attribute table define a color used to offset with in the palette table.

This implies that each tile is built by reading 16 consecutive bytes where the bits from the first eight bytes are used as low bits and together with the bits from the last eight bytes form two bit color data, as shown in figure 2.4.[1,6]



Byte							
0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0
2	0	0	1	0	1	0	0
3	0	1	1	1	1	1	0
4	0	0	0	1	0	0	0
5	0	0	0	1	0	0	0
6	0	0	0	1	0	0	0
7	0	0	0	0	0	0	0

Byte							
8	1	0	0	0	0	0	1
9	0	0	0	1	0	0	0
A	0	0	1	0	1	0	0
B	0	1	1	1	1	1	0
C	0	0	0	0	0	0	0
D	0	0	0	0	0	0	0
E	0	0	0	0	0	0	0
F	1	0	0	0	0	0	1

Row							
0	2	0	0	0	0	0	2
1	0	0	0	3	0	0	0
2	0	0	3	0	3	0	0
3	0	3	3	3	3	3	0
4	0	0	0	1	0	0	0
5	0	0	0	1	0	0	0
6	0	0	0	1	0	0	0
7	2	0	0	0	0	0	2

Figure 2.4: Tile composition

Name tables are used to create the landscape by indexing into the pattern tables where a byte points to a specific tile. They hold 32x30 tiles which equals to a full frame of 256x240 pixels. There are two palette tables, one for the sprites and the other one used for background tiles. The first color in each is a transparent one used for the background.

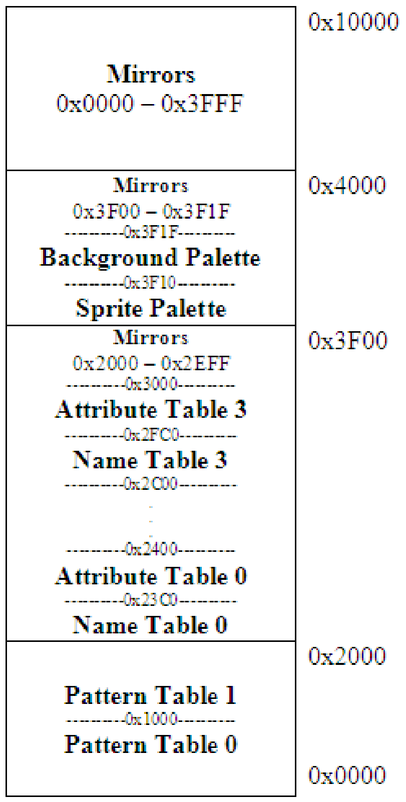


Figure 2.5: PPU memory map

2.3.2 OAM

Object attribute memory is a separate memory used to store attributes for a maximum of 64 sprites which results in a total size of 256 bytes. The first sprite is known as Sprite 0 and is commonly used as a technique for scrolling because the PPU will set the hit detection bit in the status register when this sprite is drawn on a non-transparent background. By placing sprite 0 in the status area together with some clever code one avoids this part of the screen being scrolled thus constantly displaying the status information the player needs such as remaining health and so on.[4]

Each sprite object entry in the OAM uses 4 bytes as follows:

1. Y-coordinate
2. Tile index in the pattern table
3. Attributes
   - Two first bits are the msb of color offset
   - Bit 5 is priority with respect to the background
   - Bit 6 is used for horizontal flipping
   - Bit 7 is used for vertical flipping
4. X-coordinate

2.3.3 PPU Registers

The I/O-mapped registers used for communication between the CPU and the PPU are as follows:[6]

- 0x2000 - Control register 1, write only.
  0. This combined with the next bit determines what name table should be used.
  1. Base table selection, together with bit 0
  2. Increase VRAM address by 1 or 32 when CPU reads *0x2007*
  3. Pattern table selection for background tiles.
  4. Pattern table selection for sprites.
  5. Size of sprites, 8x8 or 8x16
  6. Not used.
  7. Enables NMI interrupts at end of frame.
- 0x2001 - Control register 2, write only.
  0. Turns graphics black and white.
  1. Enable background clipping.
  2. Enable sprite clipping.
  3. Enable background rendering.
  4. Enable sprite rendering.

    5. Intensify red.
    6. Intensify green.
    7. Intensify blue.

- 0x2002 - Status, read only.
  Note that when reading, bit 7 is cleared.

    5. Sprite0 hit detected.
    6. More than 8 sprites in a scanline.
    7. Signals that a vblank has occured.

- 0x2003 - Sprite Address, write only

- 0x2004 - Sprite Data, read and write access.

- 0x2005 - PPU Scroll, affects VRAM Address, requires two writes since the addresses are 16-bits

- 0x2006 - PPU Address, affects VRAM Address, requires two writes since the addresses are 16-bits

- 0x2007 - PPU Data, read and write data to or from the VRAM on the VRAM-address.

2.3.4 Screen Mirroring

The NES has support for four name tables but due to the memory limitation only two can be used at once without using mappers to extend hardware functionality. For games without a need for scrolling effects only one name table is used. Otherwise horizontal or vertical mirroring is used to produce the scrolling effect. To achieve a horizontal scroll one needs to mirror vertically so that the first and second name tables are aligned in order and remaining two are mirrors of these. Vertical scroll is analogous to this.[4]

2.3.5 Scrolling

There are two different ways to scroll the background, horizontal and vertical. Producing the scrolling effect is done by the programmer by writing to the PPU registers *0x2005* and *0x2006*. Since the name tables are mirrored only two can be used at a time at most. Each table contains one frame worth of pixels and they can either be used separately or in a combination thus achieving a scrolling effect by filling in tile data in advance.[4]

Chapter 3

Implementation

Let us define some basic concepts that will be used later on in this document. Since we are working with references, a *reader* monad transformer[16] is used to keep track of these in a read-only fashion. Monad transformers are used to add several layers of monads to combine their functionality as a new monad. Here we encapsulated the ST monad within the reader transformer to store and manipulate a state where the memory is represented by mutable arrays and internal registers by value references. ST stands for State Thread, and works as follows: the ST monad holds an internal state and each operation performed in the monad, updates this internal state. So far, the ST monad works very much like the IO monad but without I/O operations. The main difference is that actions performed in the ST monad are indexed by a type variable *s*, which is quantified so that no two ST threads may interact with each other, achieving a clean separation of actions in a controlled manner. The only way to get actual values out from the ST monad is to invoke runST which has the following type:[15]

```
    runST :: (forall s. ST s a) -> a
```

3.1 CPU

We have focused on writing readable code that is easy to understand and maintain. CPU instructions have been mapped to Haskell functions, and as a CPU by nature holds a state, the CPU will be represented as a reader monad transformer over a state thread (ST) monad. The reader environment contains the necessary registers simulated as variable equivalents, and functions for changing the state built as a small library on top.

```
type CPU s a = ReaderT (CPUEnv s) (ST s) a
```

Since mapping low-level hardware instructions to high-level functions, the possibility of having a Domain Specific Embedded Language (DSEL) arises but we forwent this idea early on in the design process due to the fact that it would add more complexity to the code thus making the simulation slower. The idea was to map the instructions to as basic as possible Haskell functions without sacrificing simplicity and readability.

```
type Memory s = STArray s Address Operand

-- | CPU State that keeps track of registers and status flags.
data CPUEnv s = CPUState
   { aReg      :: STRef s Operand
   , xReg      :: STRef s Operand
   , yReg      :: STRef s Operand
   , sp        :: STRef s Word8
   , pc        :: STRef s Address
   , status    :: STRef s Operand
   , lowMem    :: Memory s     -- | Address range: 0    - 7FF
   , ppuMem    :: Memory s     -- | Address range: 2000 - 2007
   , uppMem    :: Memory s     -- | Address range: 4000 - FFFF
   , action    :: STRef s Action
   }
```

3.1.1 Modelling the Processor

All Processors work at the same basic level; they fetch, decode and execute instructions. We try to encompass this inherent functionality but on a higher level using higher order functions, function composition and monads to increase readability and code simplicity.

```
-- | Main CPU loop, fetches and executes instructions in order
loopCPU :: CPU s ()
loopCPU = do
    handleIRQ
    fetch >>= execute
    loopCPU

-- | Fetches the address PC currently points to, and updates PC
fetch :: CPU s OPCode
fetch = do
    op <- getPC >>= readMemory
    alterPC (+1)
    return op
```

Fetch works by extracting the current *program counter* (PC) from the current state, reading the value contained at that memory location which is the current opcode to execute, altering the PC to point to the next instruction and finally returning the fetched opcode to be decoded and run by the execute function.

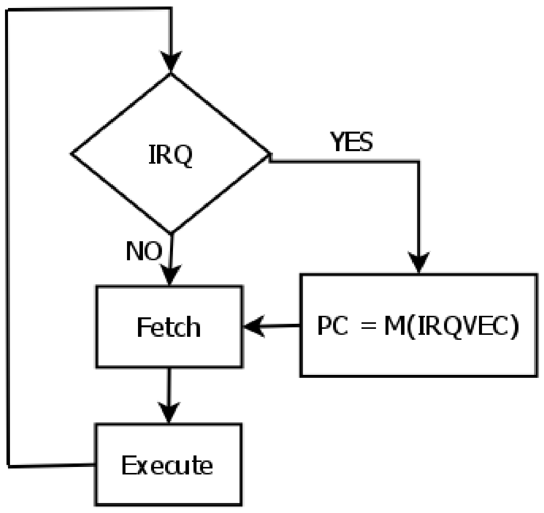


Figure 3.1: Fetch/Execute cycle

```
 - | Decodes and Executes instructions
execute :: OPCode -> CPU s Int
execute op = case op of
    -- ADC
    0x69 -> immediate >>= adc >> return 2
    0x65 -> zeropage  >>= readMemory >>= adc >> return 3
    0x75 -> zeropageX >>= readMemory >>= adc >> return 4
    0x6D -> absolute  >>= readMemory >>= adc >> return 4
    0x7D -> absoluteX >>= readMemory >>= adc >> return 4
    0x79 -> absoluteY >>= readMemory >>= adc >> return 4
    0x61 -> indirectX >>= readMemory >>= adc >> return 6
    0x71 -> indirectY >>= readMemory >>= adc >> return 5
    ...
```

Instruction decoding and execution is handled by a function conveniently named *execute*, which basically is a big lookup function that translates opcodes to Haskell functions and also decodes the correct addressing mode for each instruction and returns the number of cycles it would take the real 2A03 to execute.

The code snippet above shows mapping and decoding of the ADC instruction with all its addressing modes. *ADC* stands for *Add with Carry* and it adds the contents of the memory with the accumulator together with the carry bit from the status register.

Modeling 2A03 instructions to Haskell functions is a pretty straight forward process albeit slower since we are translating processor instructions to higher level code that will potentially be compiled to many instructions but we gain so much more in having a simple, readable and manageable code.

```
-- | Logical AND
-- | A,Z,N = A&M
and :: Operand -> CPU s ()
and op = alterA (.&. op) >>= setZN
```

Example of how a Haskell function describing the *AND* instruction is modelled. The contents of the A register is logically AND:ed with the decoded and fetched operand using the helper function *alterA* to updated the register with the new result and finally binding this result to the *setZN* function used to set the status register, in this case the *zero* (Z) and *negative* (N) flags respectively if the result was either zero or negative.

```
setZN :: Operand -> CPU s ()
setZN op = setFlagZ (op == 0) >> setFlagN (op < 0)
```

3.2 PPU

The PPU is modeled in a similar fashion to the CPU where we use a Reader monad transformer to hold the environment state containing references while letting ST be the innermost monad.

```
type PPU s a = ReaderT (PPUEnv s) (ST s) a
```

The environment is moldeled as a Haskell data record containing the necessary shared registers defined as `ppuState` and memory layout where the memory is represented as STArrays. Finally some internal helper variables are defined for the helper functions.

```
data PPUEnv s = PPUEnv
    { ppuState      :: STRef s PPUState -- | PPU Registers

    -- | Memory layout.
    , patternTables :: Memory s -- | Address range: 0x0000 - 0x2000
    , nameTables    :: Memory s -- | Address range: 0x2000 - 0x3000
    , palettes      :: Memory s -- | Address range: 0x3F00 - 0x3F20
    , ppuOAM        :: Memory s -- | OAM (also known as SPR-RAM)
    -- | Internal variables of the PPU.
    , scanline      :: STRef s Int       -- | Current scanline
    , scanlineCycle :: STRef s Int       -- | Current clock cycle
    , sprAddr       :: STRef s Address   -- | SRAM Address
    , vLoopy        :: STRef s Address   -- | VRAM Address
    , tLoopy        :: STRef s Address   -- | Temporary VRAM Address
```

```
  , xLoopy         :: STRef s Address  -- | X-scroll
  , pixels         :: STRef s [Pixel]  -- | Generated pixels
  , firstWrite     :: STRef s Bool     -- | First write to SCROLL/ADDRESS
  -- | Actions used for synchronization.
  , ppuActions     :: STRef s [Action] -- | Actions perfomed
 }
```

As mentioned previously, when the CPU reads or writes to and from certain registers it will affect different parts of the internal PPU registers. This is handled by the *action decoder* explained in more detail in the synchronization section later.

To simplify the implementation of the PPU and by doing this also have a faster graphics back-end we based our emulation on a per-scanline approach. It may not be as precise as a per-pixel based solution since it doesn't allow updates of registers mid-scanline but it is sufficient for a majority of games since only a handful need such precise timing.

For each clock cycle the CPU runs, the PPU runs three cycles. We wanted to control the number of cycles the PPU executed by using the number of cpu cycles it took to execute the last instruction multiplied with three as an argument to *runPPU*. In addition to the above mentioned instruction cycle counter we have to manage a *ppu cycle counter* (pcc) since it takes the PPU 341 cycles to generate a scanline worth of pixels. Generating the pixels in a scanline is handled by the *runScanline* function that basically maps the current scanline to the correct PPU phase. As explained in the PPU details section a scanline consists of 5 phases: initialization, rendering, cleanup, idle and v-blank.

```
runPPU :: Int -> Int -> PPU s [Pixel]
runPPU _ 0  = return []
runPPU pcc cycles
    | pcc == 341 = do
        xs  <- runScanline            -- xs  :: [Pixel]
        xss <- runPPU 0 (cycles - 1) -- xss :: [Pixel]
        return (xs ++ xss)
    | otherwise = do
        runPPU (pcc + 1) (cycles - 1)
```

When *runScanline* is in the phase of rendering pixels it will call the *genBackground* function that will generate a whole scanline worth of pixels from the current row of 32 tiles. To achieve this it will combine color data from the *pattern tables* with the *attribute tables* to produce a color offset.[13] This offset is then used to index into a table mapping NES colors to RGB values to be outputted to the screen.

3.3 Synchronization & Communication

Since we modelled the CPU and the PPU as two separate components as they also are by design in hardware, we needed a simple and efficient way to synchronize information between them. Our solution was to use a kind of message-passing where we utilized infinite lazy lists that pass around *Actions* between the two components.

```
data Action
    = Read Address
    | Write Address Operand
    | DMA [Operand]
    | NMIRQ
    | NOP
```

Actions are produced for every step in the execution of the CPU and PPU, default action being *no operation* (NOP) which means that nothing of interest has happened hence no need for synchronization. When a read or write is made on the shared registers (0x2000-0x2007), a *Read* or *Write* action is sent respectively by both CPU and the PPU.

When the CPU writes to 0x4014 a DMA action is encoded and sent together with 256 bytes to be inserted into OAM by the PPU. Last *Action* is the NMIRQ which is sent when the PPU is done generating a frame to signal v-blank.

```
writeMemory :: Address -> Operand -> CPU s ()
writeMemory addr op
    ...
    -- PPU shared registers (with mirroring)
    | addr <  0x4000 = do
      let addr' = (0x2000 + addr `mod` 8)
      mem <- asks ppuMem
      lift (writeArray mem addr' op)
      setAction (Write addr' op)
    -- DMA
    | addr == 0x4014 = do
      let addr' = op * 255
      ops <- forM [addr'..addr'+255] readMemory
      setAction (DMA ops)
    ...
```

As explained above, the code excerpt shows the encoding of actions performed when writing to memory in the CPU. The PPU works in a similar fashion.

For each step the components run, they synchronize by decoding the actions sent from the other component. For the CPU, this looks as follows (the CPU is only interested in writes and NMI).

```
cpuDecodeActions :: [Action] -> CPU s ()
cpuDecodeActions = mapM_ handle
    where
        handle a = case a of
            Write addr op -> do
                mem <- asks ppuMem
                lift (writeArray mem addr op)
            NMIIRQ        -> setVar irq (Just NMI)
            _             -> return ()
```

3.4 Enterprise Pulling

Enterprise pulling (EP) is a technique used for communicating between the CPU and the PPU. It is responsible for delivering messages between the two components of the program. EP relies on the fact that laziness in Haskell makes it possible to use infinite lists. Two separate infinite lazy lists are used; the list of CPU actions which is fed to the ppu function, and the list of PPU actions which is fed to the cpu function (figure below). Each step in the cpu and ppu functions respectively consumes one element in the input list, and produces one element in the output list. Thus, only one element in each list will be computed at each step in the program.

The enterprise pulling is initiated as follows:

```
enterprisePulling :: [Maybe Input] -> [Pixel]
enterprisePulling keys = pixels
    where (actions, pixels) = ppu (cpu keys actions)
```

Relying on Haskell being lazy, one can use the "actions" in the cpu function even though they have not yet been computed. Since the Enterprise pulling works by taking advantage of this inherent functionality, nothing is computed until it actually is needed. This makes it easy to control the emulation speed, since only the output that is visible to the user are the pixels returned by the PPU. By adding latency in the PPU, latency will automatically be achieved in the CPU as well.

3.5 I/O

The only output in the program are the pixels generated by the PPU, which are passed to the graphics library (SDL). The input to the program is a lazy list of key events. This list is checked in each step of the cpu loop, and if the first element in the list is *Nothing*, then nothing is performed, otherwise the value corresponding to the pressed key is written to the appropriate memory

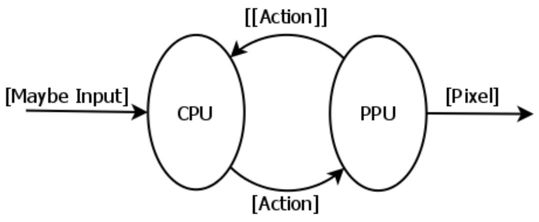


Figure 3.2: Enterprise Pulling

location (0x4016 for P1, 0x4017 for P2). The unfortunate thing about the input is the need for *unsafeInterleaveIO* since the IO monad is strict and we want the list to be lazy. The use of unsafeInterleaveIO makes the function return immediately, and nothing is computed. When the value of the unsafe call is needed, the computation will start.[20]

3.5.1 SDL & Graphics

We have chosen to use SDL as the front-end library for this project. This was a natural choice for us, since SDL supports both graphics, input and sound. Even though sound is not considered at this point, it is convenient not to have to change the library if the time comes. Since SDL is only responsible for drawing pixels to the screen and for polling keyboard events, the code required for this to work was very small.[19]

```
draw :: [(Pixel,Int,Int)] -> IO ()
draw pxs = do
    surface <- getVideoSurface
    forM_ pxs $ \(p,x,y) ->
        fillRect surface (Just $ Rect x y 1 1) p
    SDL.flip surface
```

3.5.2 Loading & ROM layout

Since most images, be it hardware dumps from real game cartridges or homebrew games use some kind of a header to inform the emulator of various things such as the region format etc, the data had to be parsed correctly so we had to implement a basic loader for these images. Our choice was the *iNes* header format since it is the one mostly used, a de facto standard in the NES community.

Size of the header is only 16 bytes and it holds important data needed by the emulators to load the games properly for example the number of program or code banks and character banks to load in to the emulator memory and the region format mentioned above. Character rom is loaded into the PPU memory as pattern data and the program rom is loaded into the main memory starting at location *0xC000*.

The header for the iNES format looks as follows:[34]

- Bytes 0-3: "NES".
- Byte 4: 0x1A.
- Byte 5: Number of 16K PRG ROM Banks.
- Byte 6: Number of 8K CHR ROM Banks.
- Byte 7: Control Byte (lsb)
  - Bit 0: Mirroring
    - 0 = Horizontal Mirroring
    - 1 = Vertical Mirroring
  - Bit 1: Backed battery present (yes/no)
  - Bit 2: Trainer data present (yes/no)
  - Bit 3: Four screen mirroring (yes/no)
  - Bits 4-7: Mapper id, four lower bits.
- Byte 8: Control Byte (msb)
  - Bit 0: VS Unisystem Arcade (yes/no)
  - Bit 1: Playchoice-10 Arcade (yes/no)
  - Bits 2-3: Not used
  - Bits 4-7: Mapper id: four higher bits.
- Bytes 9-15: Reserved, must be zeroes

Chapter 4

Discussion

4.1 Results

Emulating only the CPU, speeds of approximately 10MHz are achievable without any major optimizations being done. Testing the CPU speed was done using a small custom written program in assembly code. The program loaded all registers with *0xFF*, and contained three nested loops decrementing these registers, which made a total of 83 million cycles.

At that point, a program was considered finished when encountering the *no operation* (NOP) instruction which in this test case was placed after the loops for the purpose of terminating the execution. The 10MHz result was computed on an 800MHz dual-core laptop computer running GNU/Linux, compiled with -O2 optimizations. The compilers used were ghc 6.10.2 and 6.8.2. Since the program was written in a strict manner at that point, the 6.10-bug (see below) did not manifest itself.

Testing the PPU was difficult, as one had to have a fully working graphics core working in tandem with the CPU core and all this synchronized with the *Enterprise Pulling* component. And even then one had to visually compare the results with the expected ones. These tests proved to be good enough together with printouts and tracing of the program. Being able to print the current VRAM-address, together with seeing how colours and tiles appeared on the screen was a great aid in correcting PPU bugs.

Due to our solution with separate components and gluing them together with *Enterprise Pulling*, lazy evaluation was a must. This together with a lack of strictness annotations because of lack of time resulted in speed slow down of about a factor five compared to a strict CPU. We managed to minimize this factor by running only the communication lazy, and the CPU and PPU strict.[15]

The original NES has a framerate of 60 *frames per second* (FPS), whereas our implementation has around 10 FPS. This does not mean that lazy programs execute slower than the strict ones, but rather that the lazy evaluation requires more annotations and careful profiling to achieve the same speed. We have been profiling the code to some extent, but we have not had time to analyze the results to the extent that would have made it really useful, thus optimizations have not been a priority.[22]

4.1.1 Correctness

The CPU has been very carefully implemented to imitate the original behaviour of the 2A03. This has been straightforward as the 2A03 is well-documented. Not only has the expected behaviour been implemented but also various bugs that exist in hardware. For example, when executing the *push status to stack* (PHP) instruction, the break flag is set.[12] With the PPU on the other hand one can only do visual comparison of the graphics output, making it hard to test if the result was the expected one. We have used both the real hardware and other emulators for doing this comparison with satisfying results.

4.2 Delimitations & Choices

When working on a project on a very limited time frame, there are many things that need to be taken into account, and also many things that one might not have time to implement. We have had to decide both on what we wanted to include and exclude in the emulator, and most importantly how to do it.

Very early in the project we realized that the APU for the NES is not only poorly documented, but also a completely analogue device. This led to the decision that we would not include it at all, unless we were pretty much done with the more important parts and had time over. Another decision that was made early in the project was that we were to code for simplicity and readability rather than for speed.

4.2.1 Memory Mappers

We have chosen not to implement memory mappers. There are many different mappers, and implementing all of them would have been very time consuming ordeal which would have taken focus off the more important parts in the project. Adding mappers is discussed in the future work section.

4.2.2 Functional versus Monadic

Purely functional programs have the benefit of not having any side effects what so ever. A CPU however, is highly stateful, and passing a state as explicit argument to each function would be tiresome. Therefore, the program is written using monads that can hold an implicit state. To use explicit argument passing was discarded early in the planning phase. In addition to this one can layer monads to achieve more functionality.[28]

4.2.3 CPU

The CPU has been one of the more central parts of the project, which naturally led to many insights during the development. We had to change several things in the implementation of the CPU realizing that we needed a specific feature, for example faster mutable arrays.[29] The main changes in the CPU have been regarding data structures. We have gone through several "phases" in modelling of the CPU:

State Monad

State monad was initially used for the CPU, holding a record with memory and internal state such as various registers. At this point the memory was a purely functional array, *DiffUArray*, residing inside the state record. All instructions operated more or less directly on the state. After a while we realized it would be a lot better to have a shell of library functions operating on the CPU, so that the underlying implementation could be changed without having any effect on the instruction set.

ST Monad

Switching to the ST Monad allowed us to use *STUArray*, as *DiffUArray* proved to be all too slow for our needs. When using mutable arrays, we had to change the implementation of the CPU type from

```
State CPU a
```

to

```
ReaderT (STRef s CPUEnv) (ST s) a
```

Adding the *reader monad transformer* enabled us to simplify the code and increase readability by having an implicit state. This implementation became the final one with only minor changes.

Memory

The implementation of memory has also changed several times during the project. At first when we were working with the State monad, the memory was simply a pure array from 0x0000 to 0xFFFF. Since this array proved to be very memory consuming because of it being purely functional and all different versions of it residing in the memory, we switched to the mutable ST array library. At the same time, we also decided that it would be logical to divide the memory into several smaller arrays, partly because this would make mirroring a lot easier, but also due to the fact that the memory in the real NES is divided into logical sections (fig 2.3). In the working implementation there are three parts of the memory; *lowmem* (0x0-0x800), *ppumem* (0x2000-0x2007) and *uppmem* (0x4000-0xFFFF).

4.2.4 PPU

The PPU component did not go through as many changes as the CPU since we learned a lot how we would go about from the modelling of the CPU. From the get go we used the same memory layout with mirroring and implementing the whole PPU environment in a similar fashion as the CPU. Furthermore we based the graphics on per-scanline rendering instead of per-pixel due to the more simplified approach this entailed as explained in the PPU implementation section. Lastly we did not have time to finish the sprite rendering part and thus it is a future work.

4.2.5 Lazy versus Strict

Our implementation relies on Haskell being lazy because the two communicating components, the CPU and the PPU, are constantly consuming each others lists and at the same time producing more data. Writing that in a strict manner would not have been possible, since one would have to compute the whole list before using it, and computing an infinite list is impossible. From our testing running things strict was generally faster because computational thunks are minimized and lazy computations demand more profiling and optimizations to achieve the same speeds.[22] When we worked solely with the CPU, using a strict monad was not a problem, since there was no need for synchronization nor communication. However, when producing lists in a lazy manner such as:

```
cpu = do
    opCode <- fetch
    x      <- execute opCode
    ~xs    <- cpu
    return (x:xs)
```

there is no way to do that strict, since that version of the function will try to evaluate xs and it will never terminate.

4.2.6 Communication

We had several choices on how to handle the communication between the CPU and the PPU. There were three different techniques that were feasible besides running both components in the same loop; *STM*,[17] *MVars*[18] and *Enterprise Pulling*. Running both units together (unified looping) in the same loop was considered first, but we found it unsatisfactory as we wanted to avoid an inelegant monolithic approach.

STM is an interesting technique because it allows for changes as transactions and thus avoiding readers/writers problem and similar concurrency related problems. However a big problem using STM is that all actions have to be performed in the IO monad, which has been one of the things we wanted to avoid as much as possible.[23]

Another possible way to communicate would be to use MVars, which are part of the concurrency library in ghc. This was of course considered, but since this is the more low-level way of doing concurrency with locks and semaphores, this was directly disregarded in favor of STM.

The reason why we chose the enterprise pulling technique over all others, is that it is completely free from side effects, as it only relies on lazy lists being consumed and produced. The *unified looping* style would of course also be free from side effects, but where the enterprise pulling is elegant, the unified looping is awful.

4.3 GHC 6.10

When we ran the CPU code with strict ST monad, all worked well. However, when changing to lazy which was needed to make the communication with PPU possible, we encountered a bug in ghc 6.10. As it turns out, lazy ST is broken in that version of ghc, making recursive loops as needed by our main loop yield ≪loop≫ as output instead of the intended result. Normally, ≪loop≫ is outputted when the next thunk of data refers to the current, thus entering an infinite loop without a change. In our program however, every loop modifies the current state, so the current thunk can never refer to the previous one. When this was first encountered we thought that some part of our code was erroneous and a lot of time was spent on debugging. Finally we posted on the haskell-cafe mailing list, where the code was found to be correct, and a bug was filed.[24]

4.4 Related Work

4.4.1 BeaNES

NES emulator written in Java but does not seem to be under development any more, only an early alpha released. Straight forward implementation overall, code is easy to read.[30]

4.4.2 Nintendulator

NES emulator written in C. Precise emulation but not readable code because of high optimizations and a pixel based PPU.[31]

4.4.3 OmegaGB

GameBoy emulator done in haskell. Very messy structure, unmaintainable code. Uses state monad. Project is abandoned.[32]

4.4.4 Coroutines

Coroutines is a concept which can be used very much like Enterprise pulling, but is mostly used in an imperative setting. Coroutines are commonly used to implement lazy lists, producer/consumer patterns etc.[33]

4.5 Future Work

4.5.1 Sound

Adding sound would probably be the most prioritized feature to be implemented, since the sound brings another level of immersion to the gameplay. Implementing sound would be similar to the implementation of other components where the communication would still be managed by the *enterprise pulling* technique.

4.5.2 Mappers

One thing that was not considered at all in the project was the use of memory mappers. It would be very interesting to add this feature in the future, as most NES games use them. One approach for doing this would be by writing a typeclass for Mappers, having read and write functions. This would be similar to an interface in Java, which is how BeaNES does it.

4.5.3 State saving

State saving would likely be the easiest one to implement. One would only need to dump current CPU and PPU state. The hard thing when implementing this would probably be to have a good representation of data and code for loading external states.

4.5.4 Netplay

Netplay would be an interesting extension that would make it possible to play against other people online, thus simulating the feature of several controllers when using the real NES.

4.5.5 Parellellism

Since the CPU and PPU are two separate components, it would be interesting to parallellize the program. Haskell has support for parallellizing programs using the *par*[21] function.

4.5.6 PPU Improvements

Currently the rendering in the PPU is based on scanlines. This makes some trickery used by programmers to create cool effects impossible, since they rely on changing the values in PPU memory in the middle of a scanline. To achieve this one would have to be even more stateful in the PPU and keep track of x and y coordinates on the screen.[31] As mentioned previously, very few games rely on such precise timings, so this has been disregarded.

Another thing that we would like to work on in the future, is to alter the PPU into a more readable implementation, since the current algorithm is a very low-level one, obtained by reverse engineering.[13] One way to optimize the PPU could be done by simulation, instead of emulation. Then one would not need to store all low-level details, but instead focus on internal data structures and optimize these.

Chapter 5

Conclusion

We have shown that modelling hardware in Haskell can easily be done using abstract data types and monads. Representation of the CPU and its instructions felt very natural and was modelled intuitively when abstracting low-level instructions to Haskell functions. The PPU on the other hand, was not modelled in the same clean fashion, since it involves a lot more low-level operations.

Communication between the CPU and the PPU was solved in a clean and pure fashion using the *enterprise pulling* technique. The code became very modular, thus changing the internal representation of the CPU with the underlying data structures does not entail major changes to the core implementation.

Running only the CPU in a strict setting was fast and efficient. When adding the PPU and the communication between the two units, the need for laziness arised in our implementation. This made the program execute slower, which we suspect is due to the program thunking a lot of data which could be solved with various strictness annotations and optimizations. This has been partially solved by running everything but the communication strict. However, many more optimizations could have been done, but it was not prioritized, mostly due to lack of time.

Bibliography

[1] NESDoc.pdf http://nesdev.parodius.com/NESDoc.pdf

[2] NESDev wiki http://nesdevwiki.org/

[3] nintech.txt http://nesdev.parodius.com/nintech.txt

[4] nestech.txt http://nesdev.parodius.com/ndox200.zip

[5] 2A03 Technical Reference
http://nesdev.parodius.com/2A03 technical reference.txt

[6] 2C02 Technical Reference
http://nesdev.parodius.com/2C02 technical reference.TXT

[7] 6502 reference http://www.obelisk.demon.co.uk/6502/

[8] 6502 documentation http://nesdev.parodius.com/6502.txt

[9] 6502.org http://www.6502.org

[10] 6502 Hardware Manual
http://users.telenet.be/kim1-6502/6502/hwman.html

[11] 6502 Programming Manual
http://users.telenet.be/kim1-6502/6502/proman.html

[12] nesbugs.txt http://nesdev.parodius.com/nesbugs.txt

[13] loopyppu http://nesdev.parodius.com/loopyppu.zip

[14] Wikipedia on CRT http://en.wikipedia.org/wiki/Cathode_ray_tube

[15] ST monad documentation
http://hackage.haskell.org/packages/archive/base/4.0.0.0/doc/
html/Control-Monad-ST.html

[16] Reader monad documentation
http://hackage.haskell.org/packages/archive/mtl/1.1.0.2/doc/html/Control-Monad-Reader.html

[17] Software Transactional Memory
http://www.haskell.org/ simonmar/papers/stm.pdf

[18] Thread Synchronization
http://haskell.org/ghc/docs/latest/html/libraries/base/Control-Concurrent-MVar.html

[19] SDL Haskell bindings
http://hackage.haskell.org/packages/archive/SDL/0.5.5/doc/html/Graphics-UI-SDL.html

[20] unsafeInterleaveIO
http://haskell.org/ghc/docs/latest/html/libraries/base/System-IO-Unsafe.htm

[21] Control Parallel http://hackage.haskell.org/packages/archive/parallel/1.1.0.1/doc/html/Control-Parallel.html

[22] Real World Haskell - Chapter 25: Profiling
http://book.realworldhaskell.org/read/profiling-and-optimization.html

[23] Real World Haskell - Chapter 7: I/O
http://book.realworldhaskell.org/read/io.html

[24] GHC Bug Tracker http://hackage.haskell.org/trac/ghc/ticket/3207

[25] Wikipedia article on NES
http://en.wikipedia.org/wiki/Nintendo_Entertainment_System

[26] Svenska Akademiens Ordlista (SAOL) p. 180
13th edition 2006.
ISBN13: 9789172274198

[27] Managing the Development of Large Software Systems
Dr. Winston W. Royce
http://www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf

[28] All About Monads
http://www.haskell.org/all_about_monads/html/index.html

[29] Array Summary
http://haskell.org/haskellwiki/Arrays

[30] BeaNES http://sourceforge.net/projects/beanes/

[31] Nintendulator http://www.qmtpro.com/˜nes/nintendulator/

[32] OmegaGB http://www.mutantlemon.com/omegagb/

[33] Coroutines http://en.wikipedia.org/wiki/Coroutines

[34] Reference to the iNES header format
http://nesdev.parodius.com/neshdr20.txt

Appendix A

Hardware Specs

A.0.7 NES

Ricoh 2A03 CPU 1.78MHz
Ricoh 2C02 PPU 5.37MHz (NTSC)
2kB CPU RAM
16kB PPU RAM
64kB address space
Little endian architecture

A.0.8 Test Computers

- Dual Core 800MHz CPU, 1024MB RAM, GNU/Linux, GHC 6.10.2
- Dual Core 1800MHz CPU, 2048MB RAM, GNU/Linux, GHC 6.8.2

Appendix B

Dictionary

- ADT: Abstract Data Type.
- APU: Audio Processing Unit.
- ASM: Assembly Language.
- CHR-ROM: Character ROM.
- CPU: Central Processing Unit.
- DMA: Direct Memory Access.
- GHC: The Glasgow Haskell Compiler.
- GNU: GNU is Not UNIX.
- IO: Input/Output.
- IRQ: Interrupt ReQuest.
- LSB: Least Significant Byte.
- MHz: MegaHertz
- MMC: Memory Mapper Controller
- MOS: Metal Oxide Semiconductor.
- MSB: Most Significant Byte.
- NES: Nintendo Entertainment System.
- NTSC: National Television System Committee

- NMI: Non-Maskable Interrupt.
- OAM: Object Attribute Memory.
- PAL: Phase Alternating Line.
- PC: Program Counter.
- Pixel: Colour information for one spot.
- PPU: Picture Processing Unit.
- RAM: Random Access Memory.
- ROM: Read-Only Memory.
- SDL: Simple DirectMedia Layer.
- SPR-RAM: Sprite RAM, also known as SRAM or OAM.
- Sprite: Tile that is used for moving objects.
- ST: State Thread.
- STM: Software Transactional Memory.
- Tile: 8x8 pixels.
- VBlank: Vertical Blank.
- VRAM: PPU RAM.

Appendix C

Instruction List

- **ADC:** Add with carry. Adds the contents of memory to the accumulator, and if carry is set, it adds 1 to that result, otherwise 0. Flags affected: Z,C,N,V.
- **AND:** Logical and between memory and accumulator. Result is stored in accumulator. Flags affected: Z,N.
- **ASL:** Arithmetic shift left on memory. The contents of bit 7 in the memory operand is set to carry. Bit 0 is cleared. Flags affected: Z,C,N
- **ASLA:** Arithmetic shift left on accumulator. Like ASL but on accumulator.
- **BCC:** Branch on carry clear. Branches if C flag is 0.
- **BCS:** Branch on carry set. Branches if C flag is 1.
- **BEQ:** Branch if equal. Branches if Z flag is 1.
- **BIT:** Test bit, does and between accumulator and memory, V is set to bit 6 in memory, N is set to bit 7 in memory. Z is set if the result of the and was 0.
- **BMI:** Branch on minus. Branches if N flag is 1.
- **BNE:** Branch if not equal. Branches if Z flag is 0.
- **BPL:** Branch on plus. Branches if N flag is 0.
- **BRK:** Software Interrupt. PC+1 is pushed onto the stack. B flag is set to 1, status register is pushed onto stack, I flag is set to 1, interrupt vector is loaded to PC.

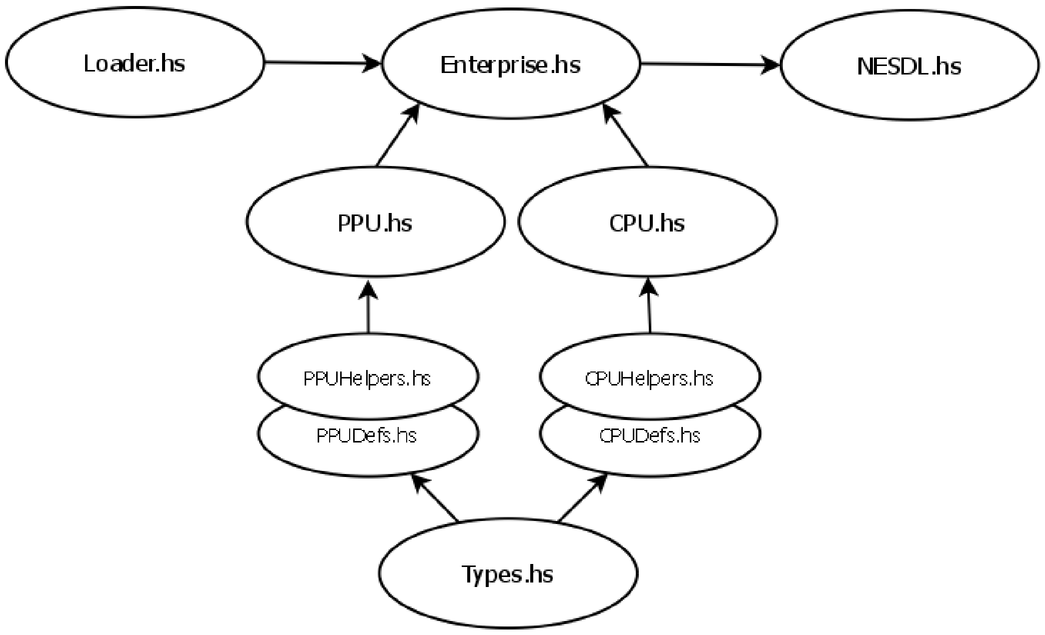
- **BVC:** Branch if overflow clear. Branches if V flag is 0.
- **BVS:** Branch if overflow. Branches if V flag is 1.
- **CLC:** Clear carry flag, set C to 0.
- **CLI:** Clear interrupt disable flag, set I to 0.
- **CLV:** Clear overflow flag, set V to 0.
- **CMP:** Compare accumulator with memory. If accumulator-memory is 0, Z flag is set. If the result is negative, N flag is set. If the accumulator value is bigger or equal that memory, C flag is set.
- **CPX:** Compare X with memory. Like CMP but on X register.
- **CPY:** Compare Y with memory. Like CMP but on Y register.
- **DEC:** Decrement memory. Affected flags: Z,N.
- **DEX:** Decrement X. Affected flags: Z,N.
- **DEY:** Decrement Y. Affected flags: Z,N.
- **EOR:** XOR between memory and accumulator. Affected flags: Z,N.
- **INC:** Increment memory. Affected flags: Z,N.
- **INX:** Increment X. Affected flags: Z,N.
- **INY:** Increment Y. Affected flags: Z,N.
- **JMP:** Jump to address. No affected flags.
- **JSR:** Jump to subroutine. No affected flags.
- **LDA:** Load accumulator from memory. Affected Flags: Z,N.
- **LDX:** Load X reguster from memory. Affected flags: Z,N.
- **LDY:** Load Y register from memory. Affected flags: Z,N.
- **LSR:** Logical shift right. C is set to bit 0 of memory. Memory bit 7 is cleared, and the memory is shifted one step to the right.
- **LSRA:** Like LSR but on accumulator.
- **NOP:** Does nothing.

- **ORA:** Logical or between memory and accumulator. Affected flags: Z,N.
- **PHA:** Push accumulator to stack. No affected flags.
- **PHP:** Push status register to stack. No affected flags.
- **PLA:** Pull accumulator from stack. Affected flags: Z,N.
- **PLP:** Pull status register from stack. Affected flags: Z,N.
- **ROLA:** Rotate accumulator left with carry. Carry flag is set to bit 7 of accumulator, and bit 0 in accumulator is set to old carry flag.
- **ROL:** Like ROLA but with memory.
- **RORA:** Rotate accumulator right with carry. Carry flag is set to bit 0 of accumulator, and bit 7 in accumulator is set to old carry flag.
- **ROR:** Like RORA but with memory.
- **RTI:** Return from interrupt. Pulls status register and PC register from stack.
- **RTS:** Return from subroutine. Pull PC from stack.
- **SBC:** Subtract with carry. Accumulator is subtracted with memory and (1-C). Affected flags: Z,C,N,V.
- **SEC:** Set carry flag, C is set to 1
- **SED:** Set decimal mode flag (has no effect on computations). D is set to 1.
- **SEI:** Set interrupt disable flag. I is set to 1.
- **STA:** Store accumulator to memory. No affected flags.
- **STX:** Store X register to memory. No affected flags.
- **STY:** Store Y register to memory. No affected flags:.
- **TAX:** Transfer A to X. Affected flags: Z,N.
- **TAY:** Transfer A to Y. Affected flags: Z,N.
- **TSX:** Transfer stack pointer to X. Affected flags: Z,N.

- **TXA:** Transfer X to accumulator. Affected flags: Z,N.
- **TXS:** Transfer X to stack pointer. Affected flags: Z,N.
- **TYA:** Transfer Y to accumulator. Affected flags: Z,N.

Appendix D

Module Hierarchy



Appendix E

Contribution Report

Being only three people in the group has made it hard to divide the work, so all of us have been involved in every part of the project. However, we have had different responsibilities as stated below:

David has been the project leader and has been coordinating the work. He has also been responsible for the CPU part in this report, and has together with Tobias V done most work on the PPU.

Tobias V has been web admin, and has been responsible for the wiki. Furthermore he has together with David been responsible for the PPU. He has also been the one making the most research, and was responsible for the technical background in this report.

Tobias O has been the secretary of the project. He has also been responsible for the CPU and SDL code, and for the discussion and appendicies of this report.