UNIVERSITY OF COPENHAGEN

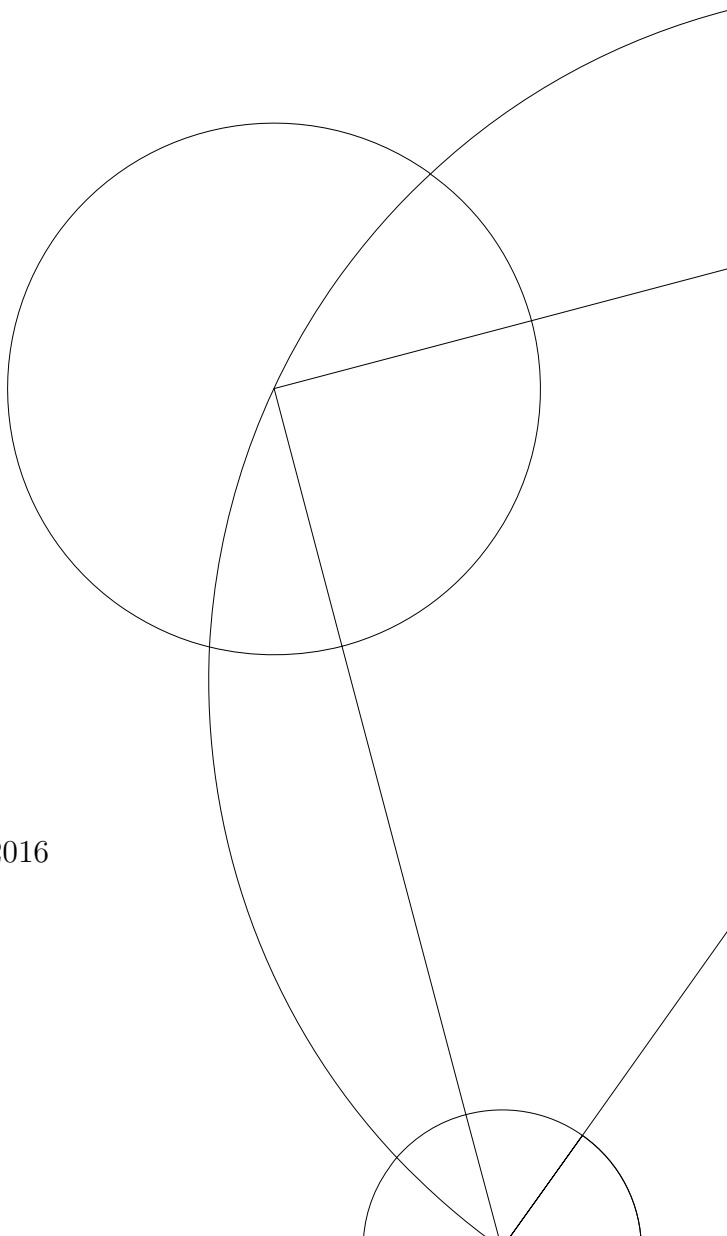
Emulation of Nintendo Game Boy (DMG-01)

PyBoy

Troels Ynddal
QWV828

Mads Ynddal
SJT402

Asger Lund Hansen
CMG881



January 18, 2016

Abstract

This project is covering an emulation of the Nintendo Game Boy (DMG-01) from 1989. The Game Boy has been emulated many times before, but this project will emulate it in the programming language Python 2.7. The implementation is not based on any existing emulator, but is made from scratch. The emulation has proven to be fast enough, to run software from cartridge dumps, with the same speed as the Game Boy. Most essential components of the Game Boy, are part of the emulation, but sound and serial port are not included in this project. The implementation runs in almost pure Python, but with dependencies for drawing graphics and getting user interactions through SDL2 and NumPy.

Forewords

The main purpose of this project is to learn how computer systems can be emulated and educate ourself in levels close to the hardware. By getting to work with the lowest levels of a system, we get an in-depth knowledge of the hardware.

The secondary purpose is to help others, who pursue the same project. During the project, it has become clear, that some of the specifications can be ambiguous, directly misleading, or undocumented, which will be described in this report too.

Aside from a general understanding of computers and object oriented programming, we expect the reader to have a level of understanding of hardware, comparable to introductory courses in machine architecture and operating systems. There will be comparisons to the MIPS architecture, but it is not a prerequisite to reading this report.

Contents

1 Introduction **6**
- 1.1 Motivation . . . 6
- 1.2 What is a Game Boy . . . 6
- 1.3 Terminology . . . 6

2 Emulation **6**
- 2.1 Architechture . . . 7
  - 2.1.1 Structure . . . 7
  - 2.1.2 Interconnection . . . 7
  - 2.1.3 Implementation . . . 7
- 2.2 Partial Conclusion . . . 8

3 Central Processing Unit **8**
- 3.1 Sharp LR35902 . . . 8
- 3.2 Opcodes . . . 9
- 3.3 Registers . . . 9
  - 3.3.1 General purpose registers . . . 9
  - 3.3.2 Special registers . . . 10
- 3.4 Operations . . . 10
  - 3.4.1 Arithmetic Logic Unit 10
  - 3.4.2 Load data . . . 10
  - 3.4.3 Jumps . . . 11
- 3.5 Interrupts . . . 11
  - 3.5.1 `VBLANK` . . . 11
  - 3.5.2 `LCDC` . . . 12
  - 3.5.3 `SERIAL` . . . 12
  - 3.5.4 `TIMER` . . . 12
  - 3.5.5 `HiToLo` . . . 12
  - 3.5.6 `HALT` . . . 12
- 3.6 Emulation . . . 12
- 3.7 Partial conclusion . . . 13

4 Boot-ROM **14**
- 4.1 Extraction . . . 14
  - 4.1.1 Physical . . . 14
  - 4.1.2 Software . . . 14
- 4.2 Disassembly . . . 15
- 4.3 Emulation . . . 15
- 4.4 Partial Conclusion . . . 15

5 Cartridge **15**
- 5.1 Memory banking . . . 16
- 5.2 Adding Functionality . . . 16
- 5.3 Memory Bank Controllers . . 16
  - 5.3.1 Cartridge types . . . 16
- 5.4 Emulation . . . 17
- 5.5 Partial Conclusion . . . 17

6 Random Access Memory **18**
- 6.1 Overview . . . 18
- 6.2 Banking . . . 18
- 6.3 Video RAM . . . 18
- 6.4 Special addresses . . . 18
- 6.5 Emulation . . . 19
- 6.6 Partial conclusion . . . 19

7 Display **19**
- 7.1 Tiles . . . 19
- 7.2 Tile Data . . . 20
- 7.3 Tile Views . . . 20
- 7.4 Viewport . . . 21
- 7.5 Sprites . . . 22
  - 7.5.1 DMA to OAM . . . 22
- 7.6 Registers . . . 22
  - 7.6.1 LCD Display . . . 22
  - 7.6.2 LCD Status . . . 23
  - 7.6.3 LCD Position and Scrolling . . . 23
- 7.7 LCD . . . 23
- 7.8 Emulation . . . 24
- 7.9 Partial conclusion . . . 24

8 Interaction **25**
- 8.1 Emulation . . . 25
- 8.2 Partial conclusion . . . 25

9 Verifying solution **26**
- 9.1 Debugging . . . 26
- 9.2 Unit test . . . 26
  - 9.2.1 CPU . . . 26
  - 9.2.2 Display . . . 27
  - 9.2.3 RAM . . . 27
  - 9.2.4 Cartridge . . . 27
- 9.3 Test-ROMs . . . 28
- 9.4 Partial Conclusion . . . 28

10 Performance **28**
- 10.1 Finding optimal datastruture 28
  - 10.1.1 Test purpose . . . 28
  - 10.1.2 Test setup . . . 29
  - 10.1.3 Datatypes . . . 29
  - 10.1.4 Test results . . . 29
- 10.2 Interpreter vs. JIT . . . 30
  - 10.2.1 Test purpose . . . 30
  - 10.2.2 Test setup . . . 30
  - 10.2.3 Results . . . 30
  - 10.2.4 Test enviorment . . . 31
- 10.3 Partial conclusion . . . 31

11 Conclusion **32**

Appendices **35**

A Class Map **35**

B Page 14 of Zilog Z80 Reference Manual **36**

C Boot-ROM picture **37**

D Performance test results **37**

E CPU dissection **38**

F Data basis for optimization efforts **38**

G Performance comparison between Pypy and Python **39**

H Results of Test ROMs **40**

Category
Hardware emulation

Keywords
Copenhagen, university, computer, science, emulation, game boy, nintendo, DMG-01, LR35902

1 Introduction

This project describes an emulation of the first Nintendo Game Boy (internally dubbed DMG-01). The emulation will be programmed in Python 2.7 and will have almost the same features as the real Game Boy. The sound driver and serial port will not be part of the emulation. We have chosen to call the emulator "PyBoy".

For every emulated component, there will be a description of how it works and how the emulation of it has been made. After the examination of the components, there will be a test section and an optimization section, which go through the results of the unit testing and show how the performance has been optimized.

1.1 Motivation

As written in the abstract, the main purpose of this project is to educate ourselves. But what can the emulator be used for?

Digitalbevaring.dk is a website by The Danish National Archives, State and University Library and The Royal Library. One of their common goals are to preserve current technology to save our cultural legacy for the future[25]. With an emulator, you have the ability to display old technology, that might become rare in the future. By making an emulator and documenting how it works, it will be possible for our ancestors, to understand how the technology worked.

It's also a matter of convenience. Emulating a piece of technology can be used for ease of use or for new hardware to be backwards-compatible[8].

1.2 What is a Game Boy

The first Game Boy was introduced on April 21, 1989, in Japan. The original Game Boy line sold 119 million worldwide as of April 2009[9].The first Game Boy was internally dubbed "DMG-01" (Dot-Matrix-Game), and these references are still visible on the printed circuit boards. The Game Boy is equipped with a 8-bit CPU clocked at 4 MHz and a monochrome, 4-tone LCD display[18]. Instead of installing games on the device, like you would on a personal computer, the games come in external, removable cartridges. All game data, including saved progress, is stored in the cartridge. The cartridge consists of a volatile and non-volatile memory. The game data is stored on the non-volatile memory as read-only, while the saved games is stored in battery powered volatile memory. This is due to the fact, that flash memory is too expensive at the time.[3] This choice of memory also mean, that the cartridge effectively expands the RAM/ROM bank of the Game Boy.

1.3 Terminology

Since some computer architectures and paradigms use different terms for bit switching and integer notation, these tables will show the expressions used in this paper.

For bit switching, we adopted the terms used in the Zilog Z80 reference manual:

Set	Switch to logical true
Reset	Switch to logical false
Test	Reading the state of a bit.

For integer notation, we use typical mathematical notation for equations and Python notation for code.

Abbreviation	Python	Mathematical
Hex	0xB	B_{16}
Dec	11	11_{10}
Bin	0b1011	1011_2

2 Emulation

An emulation is an imitation of software or hardware, which gives you the ability to

use a product, even if you don't have the original. An emulation has to imitate the real product as much as possible and even replicate known errors. An emulator distinguishes from a simulator by the ability to replace the original product completely. The focus of a simulator is to replicate a condition or situation[25].

An example of a simulator is the Apple iOS Simulator, which simulates the operating system of the ARM-based iPhone. The Apple iOS Simulator can simulate the conditions on the Apple iPhone, by building the application for the modern x86-architecture, which can be run on a desktop computer. This distinguishes it from a emulator, which would have reused the binary ARM-code and tried to emulate the hardware specification of the iPhone[1]. But since the desktop CPU is much faster, than the mobile ARM CPU, this simulator can't be used for performance tests, because the simulator doesn't imitate the CPU restrictions of the ARM CPU.

Our goal is to emulate the Game Boy to makes it possible to run the software from real game cartridges, like you could on the original Game Boy.

2.1 Architechture

2.1.1 Structure

Nintendo filed a patent in 1990, describing in clear details, the architecture of the Game Boy. Especially Fig. 4 of the patent (see Figure 1) shows the integration and connection between CPU, RAM, cartridge and display.

With inspiration from this, we will make classes in Python, for each of these components. This will set up the foundation for a "guest system", on our 'host system", running Python. This guest system will be the virtual Game Boy hardware, theoretically capable of running every existing piece of software, written for the Game Boy.

2.1.2 Interconnection

The real Game Boy runs all of the hardware in parallel, and synchronizes through interrupting the CPU. The interrupts work as

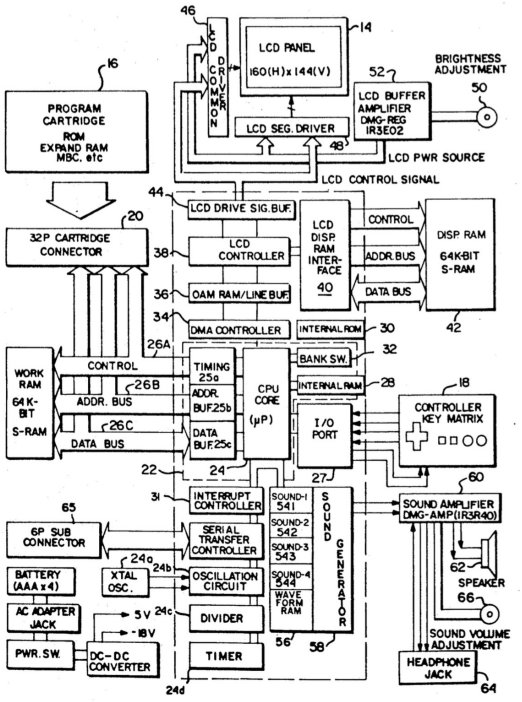


Figure 1: Architecture of the Game Boy from patent by Nintendo[28, Fig. 4]

a unidirectional message system, between the hardware parts. The LCD-controller might trigger the VBLANK-interrupt, to make the CPU do an updating routine, but the CPU can enable and disable interrupts, if it's an inconvenience to the programmer.

In our software, we have a `MotherBoard`-class, as the first step inside the guest system. This class will instantiate every other component of the Game Boy, like the `CPU`, `RAM`, `Cartridge`, and `Display` classes. These will all be covered in detail in the following sections.

The structure has a strict hierarchy, where parents call functions and accesses elements of children. By disallowing children to make calls to their parents, we keep the code less entangled and, thereby, easier to read.

2.1.3 Implementation

As the Game Boy has an 8-bit processor, and a modern computer has a 32-bit or 64-bit processor, we had to figure out a way to support the different architectures. Python doesn't support storing a single byte in a variable, like the C-type `char`. Our first

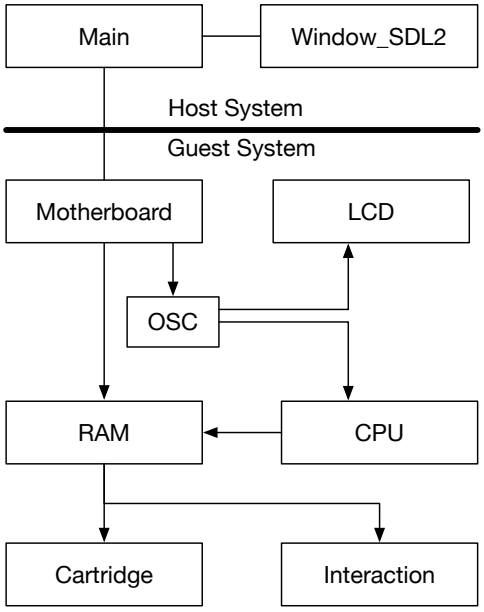


Figure 2: Simplified architecture of the PyBoy implementation (see Appendix A for the complete version)

approach had an "8-bit" class with a constructor and a set of functions, to make sure we didn't accidentally use more than 8 bits. This proved to be incredibly slow, and had to be optimized, which will be described in greater detail in the "Performance" section. Our second approach, which proved to keep the time boundaries, was to use a normal integers, instead. This means, that on a 64-bit system, every byte would take up 8 times as much memory and after every operation, that could possibly overflow into the 9^{th} bit, we would have to mask out excessive bits. This choice would not be optimal, if we were expecting to run the emulator on constrained hardware. We chose to prioritize speed and correctness of the CPU, and let it use the extra memory.

2.2 Partial Conclusion

The initial structure is to organize the program with classes, where each class represents its real counter part. This will yield a more natural flow in the code, and keep a logic seperation of the classes. It would have been possible to make a single-class program, but it would lose readability and become more difficult to maintain and develop.

Having a 32-bit or 64-bit processor will cause a memory penalty on the host system, when emulating an 8-bit processor.

To form an overview of the emulation, see appendix A where there is a class map, covering the architecture; including: `Display`, `Interaction`, `RAM`, `Cartridge`, and `CPU`. These emulations will be explained in the following sections.

[25]

3 Central Processing Unit

The CPU of the Game Boy is single cycled, meaning it has no pipeline[13]. Unlike modern consoles and personal computers, the Game Boy has only one processor. The Game Boy has a primitive sprite engine, which off-loads the CPU. The CPU still has to move and load background elements from the memory to the designated video memory.

3.1 Sharp LR35902

The CPU of the Game Boy was derived from the Zilog Z80, but was modified, to only include some parts of the Z80. The CPU of the Game Boy was officially named LR35902 and is produced by Sharp. The specification of the LR35902 includes a reduced instruction set and fewer registers than the Zilog Z80. The LR35902 has the same registers as the Intel 8080 (Introduced in April 1974)[12], but had some of the added functionality of the Zilog Z80 (introduced in July 1976)[24].

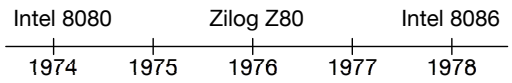


Figure 3: Release timeline of Intel 8080[12], Zilog Z80[24], and Intel 8086[14]

The LR35902 and Z80 both uses the same instruction set from year 1974. The Z80 has an 8-bit opcode length, allowing an instruction set of 256 opcodes. But by executing a special prefix instruction, called *opcode prefix*, the CPU will use a different lookup table, for the next opcode. When the prefixed opcode has been executed, it

will revert back to the main opcode table. The Game Boy's LR35902 is a hybrid of the Intel 8080 and the Zilog Z80 in the sense, that it includes all the features from Intel 8080, but also has some of the features introduced by Zilog. The Intel 8080 also carries a resemblance to the modern x86 (first introduced in 1978), since 8080's successor, Intel's 8086, defined the initial instruction set of x86. The opcodes of Z80 and LR35902 are variable in length and are determined by the byte-length of the opcode parameters. This means, that unlike the MIPS architecture, you can't do random lookups in the program code, without a known entry point.

3.2 Opcodes

The opcodes are constructed as seen in figure 4. The dashed lines are mandatory, depending on the opcode. The prefix is used to switch to other lookup tables. The LR35902 only supports one prefix, which is called the CB-prefix, derived from the hexadecimal value of the code.

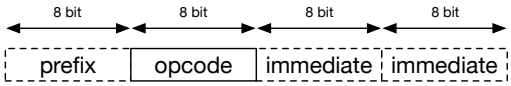


Figure 4: Z80/LR35902 opcode assembly

The CB-instruction set of the LR35902 includes operations for rotate, swap, and bit-wise test, set, and reset of registers. The Z80 has 4 additional instruction sets for negations and extra load functions[23].

The main instruction set includes: Arithmetic functions, jumps, calls, stack push/pop, and control instructions.

The opcode can have an immediate appended after it. The length of the immediate is predetermined by the opcode and can be either 8- or 16-bit. In general, the CPU only supports unsigned integers, with the only exception being relative jumps and two Stack Pointer operations. The signed data is formatted in two's-complement and have to be treated differently by the CPU, since every other integer is unsigned. The 16-bit immediates are in big endian, which must be taken into consideration when building the emulator.

3.3 Registers

Some registers of the Z80 has been omitted in the LR35902. The Z80 has an *Alternate Register Set*, in addition to the *Main Register Set*, which is 8 general purpose registers. The *Alternate Register Set* is no different from the main registers, but makes it possible for fast context switch, by swapping the register sets. Besides the 8 general purpose registers, 4 *Special Purpose Registers* has been omitted. These 4 registers were used for address indexing, memory refresh, and indirect calls by interrupt (see Zilog Z80 data sheet in appendix A).

The registers in the LR35902 are arranged as in figure 5.

Bit 15 ... 8	Bit 7 ... 0	
A (acc/arg)	F (flags)	General Purpose Registers
B	C	
D	E	
H (addr.)	L (addr.)	
SP (Stack Pointer)		Special Purpose Registers
PC (Program Counter)		

Figure 5: CPU registers arranged in 6 rows of 16-bit

3.3.1 General purpose registers

All the general purpose registers are 8 bits each, but a single row can be combined into a 16-bit register. This is typically used when addressing data from the memory, since the memory addresses are 16-bit. The registers are general purpose, but each register still serve a specific purpose. The A register is commonly used for accumulating numbers and as argument for calls. This is due to the fact, that most 8-bit arithmetic and logical instructions saves the result in register A (80 of 104 ALU instructions, in the main instruction set, use register A). The F register is a bit special, which we will get back to in a moment. Register B, C, D and E can be used for anything and

doesn't carry any special properties. Register H and L are a bit special, too, since they are often combined as HL, which is needed for indirect memory access. Indirect memory access is slower and takes 8-12 cycles, where the same instruction take 4 cycles on a register.

3.3.2 Special registers

Stack Pointer and Program Counter are self explanatory. The Stack Pointer points to the next unused space, of the stack, and the program counter points to the next instruction.

Back to the F register. The F register can't be used as operand for ALU(Arithmetic Logic Unit) and load operations, because the register is reserved for ALU flags.

The ALU can set 4 flags in the register

Z	Math operation resulted in zero
N	Math operation used subtraction
H	Math operation raised half carry
C	Math operation raised carry

Register F							
7	6	5	4	3	2	1	0
Z	N	H	C	0	0	0	0

Figure 6: ALU flags of register F

The flags span from the 8^{th} bit to the 5^{th}. The lower nibble is always zero. These flags can be used to check a condition before jumping or handling carry, caused by arithmetic operations.

3.4 Operations

3.4.1 Arithmetic Logic Unit

The LR35902 can't handle overflows, but it has a carry instead. The carry flag can be set, when an integer overflows, among other things.

Consider the case, where register $A = 200_{10}$ and $B = 250_{10}$ with the carry reset.

First, calculating the addition

$$200_{10} + 250_{10} = 450_{10}$$

By taking modulo 256 of the result, we will get the result capped to 8 bits.

$$450_{10} \bmod 256_{10} = 194_{10}$$

Dividing the previous result by 256 and flooring the result, gives us the new carry status.

$$\lfloor 450_{10}/256_{10} \rfloor = 1_{10}$$

The bit that overflowed will be moved to the carry and may be used in other operations.

The carry can be used to check a jump condition. If you were to repeat a loop 256 times, you could initialize a register to 0 and increment it with each repetition. When reaching 256 and incrementing once more, the register would overflow to 0 and the carry flag would be set. This would make it skip the jump and break the loop.

This case showed the carry used as borrow from an overflowed operation, but the carry is more than that. The carry flag is also used by rotate functions, where it is used as an extra bit or as a duplicate of the rotated bit. One case is the Rotate Right Circular (`RRC`) operation. With this operation, the 1^{st} bit is copied to the carry and to the 8^{th} bit, meanwhile, the other 7 bits are shifted to the right.

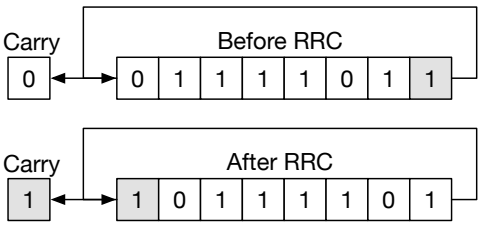


Figure 7: RRC operation

3.4.2 Load data

The LR35902 supports typical load operations, that moves data between registers and between memory and registers. But in an effort to optimize performance, when iterating over an array, the `LD`-operation can do a "Load from `HL` pointer and increment `HL`" or "Load from `HL` pointer and decrement `HL`". This can be helpful for iterations, because you don't have to increment the register, after each load, and thus it saves

4 of 12 clock cycles, otherwise used by `LD` followed by `INC`.

Even though, the Sharp LR35902 is an 8-bit CPU, it supports a variety of 16-bit operations. The CPU supports some unusual instruction like `LD (a16), SP`, where `a16` is a 16-bit unsigned integer and the parentheses means its pointing to an address. The operation *says* it will load the 16-bit Stack Pointer to an 8-bit cell of the memory (pointed by an 16-bit address pointer). But, the Stack Pointer is actually broken into 2×8 bits and is saved to the address of `a16` and `a16`+1.

3.4.3 Jumps

A major difference from MIPS to the LR35902, is how jumps and calls are handled. While the MIPS architecture has 32 registers, the LR35902 is limited to 8 registers[29, page 152]. In MIPS the Program Counter is saved to a reserved register, "Return Address", when jumping, but LR35902 can't afford to reserve a single register for jumps. For LR35902 return addresses are pushed to the stack, using `CALL`, and popped by using `RET` (Return from call)[26, page 278].

In addition, the LR35902 has an advantage compared to MIPS, since it can make jump across its entire 16-bit memory space, while 32-bit MIPS can only jump with 26 bits immediates, which doesn't cover its 32-bit memory space. MIPS does support 32-bit jumps, but will require the use of a register to hold the 32-bit address.

3.5 Interrupts

On hardware level, the CPU of the Game Boy has just one processing core. This means, that it has to serve a lot of different purposes. If one were to make a game without interrupts, it would possibly use a lot of time in spinlocks and consume more power (see `HALT` 3.5.6).

Timings would be close to impossible to control and changes in the code, would cause a cascade of changes. If one were to have a timer, that counted seconds, while moving around sprites and calculating game dynamics, some counters had to be used, and the time of each instruction should be accounted for.

Interrupts mitigates a lot of these issues and leaves the programmer with more readable and efficient code.

None of the interrupts are mandatory to serve. It's up to the programmer to enable the interrupts, which he finds useful.

When an interrupt is triggered, the CPU automatically disables further interrupts, to avoid nesting multiple interrupts. The Program Counter (PC) is automatically pushed to the stack and the appropriate interrupt vector is run. It is up to the programmer to reenable interrupts through the `EI` or `RETI` operations. This is often done, at the end of serving an interrupt.

If multiple interrupts are triggered at once, the first on the list below is served. It is undocumented, what happens with the ignored interrupts. Either, the CPU could interrupt again once the interrupts are enabled or it could be up to the programmer to poll the interrupt register.

The Game boy has 5 main interrupt vectors:

Name	Cause	Call
`VBLANK`	LCD has drawn a frame	40_{16}
`LCDC`	LCD controller changed	48_{16}
`SERIAL`	Serial transfer completed	50_{16}
`TIMER`	Serial transfer completed	58_{16}
`HiToLo`	User pressed a button	60_{16}

All main interrupts are enabled or disabled through the register at address $FFFF_{16}$. The cause of the interrupts, can be controlled individually.

3.5.1 `VBLANK`

The interrupt occurs, when the LCD has drawn a full frame. The programmer is not allowed to change the video-RAM (VRAM), during the rendering of a frame. This interrupt can be used by the programmer, to change the content of the VRAM, between frames.

Apart from triggering the game to update the frame, in Pokemon Blue, this interrupt has been observed to change to the

color palette, to make a white-to-black fading effect. It is also used in Pokemon Blue, to change the pattern of water, to make the waves move.

3.5.2 LCDC

This interrupt can be triggered by a range of causes. The main interrupt gets enabled through $FFFF_{16}$, but the cause are modifiable, from the `STAT` register (see section 7.6.2).

The main reasons are, when the LCD controller changes to a specified mode. It can also be used to wait for the screen to draw a specific line, through the `LYC` register (see section 7.6.3).

3.5.3 SERIAL

This interrupt is triggered, when using the serial port of the Game Boy, to communicate with another Game Boy. This is not covered in this project.

3.5.4 TIMER

The timer register can be setup to trigger at a specific time interval. The timer works by incrementing a special register, until it overflows. The overflow triggers the interrupt and the timer resets, to a predefined value, and start counting again[19]. The registers are:

Address	Name	Function
$FF04_{16}$	`DIV`	Divider Register
$FF05_{16}$	`TIMA`	Timer Counter
$FF06_{16}$	`TMA`	Timer Modulo
$FF07_{16}$	`TAC`	Timer Control

The divider register increments automatically, with a frequency of 16,384 Hz and is resets, if anything is written to it. The purpose of this register is not documented, but might be used for generating pseudo random numbers.

The Timer Counter increments, at the rate specified by `TAC`. When it overflows, it resets to the value of `TMA` and triggers an interrupt.

Timer Modulo contains the timer offset. Since the timer is only triggered on overflow, the only way to adjust the length of the timer, is to restart the timer with an offset.

The Timer Control sets the speed of the timer with a divider and tells the timer component to start or stop.

Bit	Function
0-1	00: 4096 Hz 01: 262144 Hz 10: 65536 Hz 11: 16384 Hz
2	0 = Stop 1 = Start

If the Timer Interrupt has been enabled, the CPU will jump to address 0050_{16}, when `TIMA` overflows. At this address, the programmer has prepared a routine that decides what to do.

3.5.5 HiToLo

When the user pushes a button, this interrupt is triggered, to let the game immediately react, to the user input. The user won't see the changes until the next frame, but it can make the game feel more responsive, to not lock the interaction to specific time of a frames.

3.5.6 HALT

Games are recommended to use the `HALT` operation, to save power [17, Using the `HALT` Instruction]. The `HALT` operation stalls the CPU, until an interrupt is triggered. The interrupt breaks out of the `HALT` and the CPU is transferred to the interrupt vector. When returned from the interrupt, it continues with the code, right after the `HALT`.

3.6 Emulation

The CPU is implemented as a class, that the `MotherBoard`-class controls. The CPU initializes the Program Counter (PC) to 0 and waits for a call to its Tick-function. The Tick-function will make one tick on

the emulated CPU and return to the `MotherBoard`-class.

The CPU imitates the registers using a list of integers. Each register is then referenced by `A`, `F`, `B`, `C`, `D`, `E`, `H`, `L`, `SP`, and `PC`, each containing their respective index in the list. Our emulator runs on 32-bit and 64-bit Python, which means, that the integer operations has to be restricted, due to the 8-bit architecture. During any arithmetical function, the function has to control overflow, underflow, and flags. These are set using the respective functions of the CPU class, that then sets the flags.

In order to run an opcode, each opcode has its own implementation in Python. The opcodes are arranged in a list with exactly 512 elements, illustrated in figure 8. The first 256 elements form the main instruction set and the last 256 elements form the CB-prefixed instructions. Since the opcode ranges from 0 to 255, we can use the opcode's literal value as an index in the list. If the opcode is CB_{16}, it reads the next byte as the new opcode and offsets the instruction by 256, to do a look-up in the extended instruction set. As mentioned before, each Python-function controls overflow, underflow, and flags, which means, the execution of Game Boy instructions is done by executing a Python-function. Executing the Python-function will return a new PC and the Tick-function is done.

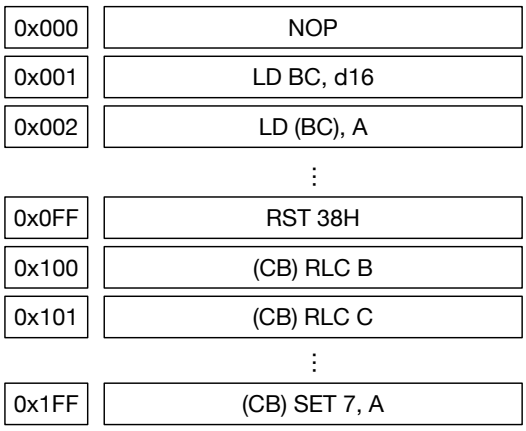


Figure 8: List of opcodes

The state of the CPU is preserved until the next tick. The CPU works by giving it a new tick every time, the `MotherBoard` class is ready for a new execution. By having this abstraction (that the CPU doesn't trigger a new tick itself), we can make the emulator single-threaded and control timing independently from the execution. This will give the motherboard time to execute the display emulator and keep the timing. Although, the Game Boy is essentially "multithreaded" in the sense, that components (Display Driver, CPU, Interrupts, etc.) operate independently, having a single thread of execution will reduce the complexity of the emulator. We could have implemented a multithreaded solution to benefit from the multicore host system, but having to synchronize threads and managing async queues would require more resources, than the actual instruction-execution. Since the multithreaded execution is not required to achieve the desired speed, the additional overhead of multithreading doesn't make up for the resource usage.

3.7 Partial conclusion

Despite the difference in architecture between MIPS and LR35902, they still carry a close resemblance. While there are minor differences from MIPS to LR35902, the overall layout of the opcodes remain the same. A major difference between the two, is the register layout. While MIPS has 32 registers, LR35902 only has 8 registers. This doesn't mean MIPS is objectively better, but rather, that it follows another paradigm. MIPS doesn't have a flags register in the same way as LR35902. On MIPS, zero and overflow flags is handled directly by the ALU and is not saved in a register. For jumps and branches, the result of a comparison is taken into action instantly, while LR35902 saves the result of a comparison in the `F` register.

When utilizing interrupts, the CPU gains lots of time for other tasks, instead of polling each component for updates. It also gives a greater accuracy to time critical operations, since they can halt the main program and do the necessary updates, as soon as the timer runs out.

As a programmer, it also gives a more reliable program with greater flexibility. Measuring time without a timer interrupt

can be nearly impossible.

4 Boot-ROM

In emulation, the boot-ROM can be crucial for a perfect emulation. The boot-ROM is the first code, that a system executes before a potential operating system, or in this case, a game. How the boot-ROM sets up the hardware, can in some cases be reverse-engineered by observing a startup using electronic probes. Although a definite answer can only be found by dumping the boot-ROM.

4.1 Extraction

The Game Boy has its boot-ROM on the CPU's die, which means, it is not accessible without breaking the integrated circuit packaging. The boot-ROM can't be dumped, just by using a modified cartridge either, because the boot-ROM on the Game Boy disables itself, at the last instruction[28, Fig. 9]. The boot-ROM has been successfully read by enthusiasts. The two following sections will describe two methods for dumping the boot-ROM.

4.1.1 Physical

On July 17, 2003, the user "neviksti" of cherryroms.com's forum posted a message describing how he extracted the boot-ROM [15]. He successfully etched the IC out of the CPU package and took pictures of it, through a microscope (see Appendix E). From the overview, he identified one of the blocks to be the boot-ROM. We stitched his sequential pictures into one, for readability (see Appendix C for the full picture).

From these images, he read out each bit, one at a time, with multiple passes [15]. The data is stored in 16 blocks of 16x8 bits. The blocks are read left to right. The data within each block is read as 16 bit words from south to north with 8 words from left to right (see figure 10). This is in total read as 2048 bits – equal to 256 bytes.

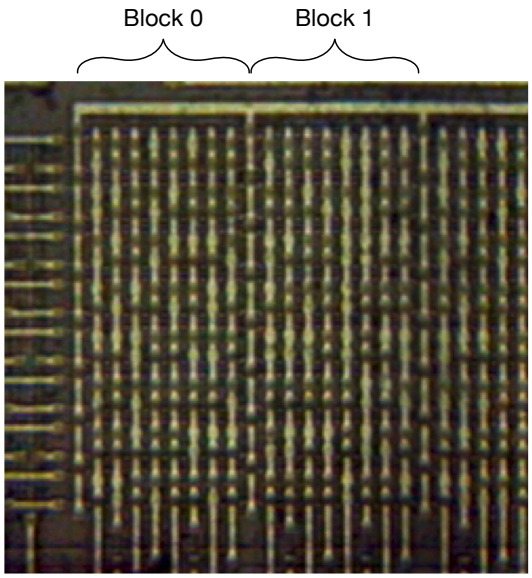


Figure 9: Shows the first two blocks from the full picture.

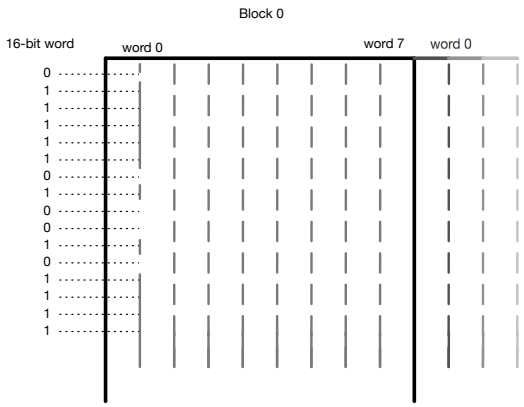


Figure 10: Illustration of how to read the data from the microscope

4.1.2 Software

On September 28, 2009, (6 years after neviksti), Costis Sideris posted a message on his blog, "FPGABoy" (a reimplementation of the Game Boy Color on an FPGA)[31]. He successfully dumped the boot-ROM by desoldering the oscillator crystal and the CPU VDD off the circuit board and controlled it with an FPGA. The trick itself was to let the boot-ROM initialize the system, but halt the oscillation of the CPU right before the disabling of the boot-ROM. He then lowered the CPU VDD to make the CPU unstable, causing the internal registers to be scrambled, with random data. After several tries, he got

the program counter to point to a random place in a modified cartridge and used a NOP-slide to align it with a dumping routine. The routine simply read the boot-ROM byte-for-byte and dumped it to the FPGA[6].

4.2 Disassembly

To understand what the boot-ROM does, we looked at the disassembly, that neviksti made. The disassembly is crudely commented and is mostly just one-to-one translations from opcode to assembly.

```
    LD SP,$fffe     ; $0000  Setup Stack

    XOR A           ; $0003  Zero the VRAM
    LD HL,$9fff     ; $0004
Addr_0007:
    LD (HL-),A      ; $0007
    BIT 7,H         ; $0008
    JR NZ, Addr_0007; $000a

    ...
```

We read through the boot-ROM and commented on sections that were important to our emulation. We skipped some sections of the boot-ROM, because we did not plan to support sound.

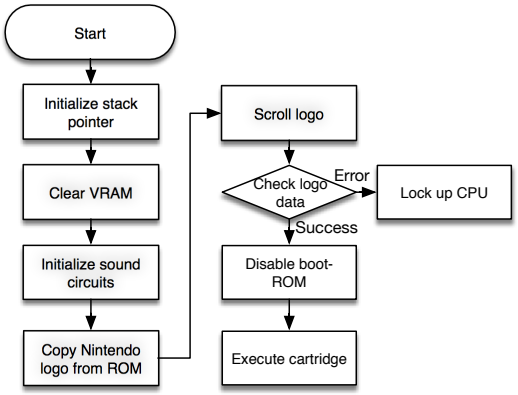


Figure 11: Flow of the boot-ROM

We used the boot-ROM as a guideline, for our emulation. As the boot-ROM did not use interrupts, nor other more advanced features of the CPU, it was a good starting point.

As from seen in figure 11, the boot-ROM initializes the hardware and shows the Nintendo logo on the screen. The hidden agenda of this, is to check for counterfeit games. The boot-ROM contains the Nintendo, and so does any legal cartridge. By having the copyrighted Nintendo logo on the cartridge, they effectively prevented third parties from manufacturing cartridges[28, claim 1].

When the boot-ROM has loaded, the Game Boy shuts the boot-ROM off and it is not accessible anymore. This is unusual, compared to traditional computers, that are assisted by an operating system. This is not the case with the Game Boy and all interfacing and memory transfers must be controlled by the game developer.

4.3 Emulation

Besides setting up audio, the boot-ROM doesn't bring anything to the Gamy Boy emulator. The first part of the boot-ROM initializes the RAM, but since Python already takes care of that, it is completely redundant. Next, it sets up the audio, but since our implementation doesn't support audio, it doesn't matter, if it gets initialized. Then, the Game Boy will start showing the Nintendo logo and play the start-up sound, which is just aesthetic. The final step is to verify the authenticity of the Game Boy ROM, but since this is an emulator, the authenticity has already been compromised.

4.4 Partial Conclusion

Due to previous work by "neviksti", the Game Boy boot-ROM is easy to obtain. Although, the boot-ROM doesn't seem to be a necessity to the emulator. What is mostly done, is initializing hardware, which is either way initialized by the emulator, show the Nintendo logo and check for counterfeit games.

5 Cartridge

The only software the Game Boy has internally is the boot-ROM. All software run on the Game Boy, has to be written to a cartridge (see figure 12) and inserted on the back.

Figure 12: Super Mario Bros. Deluxe and Star Wars cartiridges

5.1 Memory banking

The Game Boy has 16-bit addressing, meaning only 64 KB of storage can be accessed. As games may need more memory than this, the Game Boy utilizes memory banking[28, claim 6]. The banking works by switching small blocks of memory from a large storage, into designated areas of a limited memory space. Looking at the way the RAM is partitioned, there are two dedicated banking areas (see figure 14). The 16 KB allocated from address 4000_{16} is dedicated to ROM banking and can be switched using the memory bank controller (MBC) on the cartridge. A similar 8 KB area is allocated from $A000_{16}$ and is dedicated for RAM banking and is also controlled by the MBC.

5.2 Adding Functionality

The cartridge of the Game Boy is primarily used for storing a game, but also has the possibility of extending the RAM of the Game Boy, by using banking. Some cartridges adds other hardware, for example a vibrator, a camera, or sonar for locating fish[5].

5.3 Memory Bank Controllers

As the Game Boy often utilizes banking, but has no special opcodes for controlling it, a special chip has to be on the circuit board of the cartridge (see figure 13). This chip is called the memory bank controller (MBC). The MBC is connected directly to the external connector of the cartridge and redirects reads and writes to the separate RAM and ROM chips[16]. There are many different MBCs and they each have their own way off doing the banking (see figure 13).

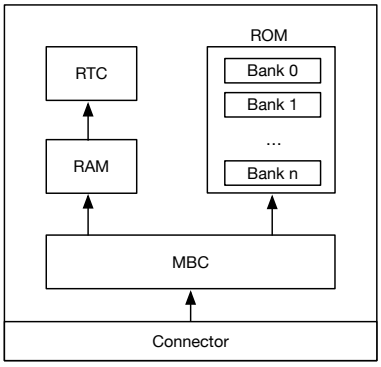


Figure 13: Internals of Game Boy cartridge

The MBCs only have a data channel; therefore, the commands are send to the MBC by issuing a write command to the ROM banks – which is read-only. The write address is used in the MBC as an opcode for the banking and the data as argument[16].

Choosing an MBC and knowing how to interface with it, is all up to the developer of the software, since the Game Boy doesn't have an operating system.

5.3.1 Cartridge types

As the Game Boy is oblivious to the MBC, every developer could make a custom chip, if they needed something specific. Luckily, most cartridges, we have observed, used only one of these four different types:

ROM Only
: No MBC is used. Only 32 KB ROM is addressable.

Type 1
: This MBC has two modes. A 2 MB ROM/8 KB RAM and 512 KB ROM/32 KB RAM. Meaning, that this MBC is used for two different cartridges, not that this is expected to be switched during use[16]. The MBC defaults to ROM mode. Writing 0 or 1 into the $6000_{16} - 7FFF_{16}$ area, will set

the mode to 2 MB/8 KB or 512 KB/32 KB respectively.

Selecting a ROM bank is done by writing a 5-bit value to the $2000_{16}-3FFF_{16}$ area. This select the appropriate ROM bank at $4000_{16} - 7FFF_{16}$. Values of 0 and 1 are both pointing to ROM bank 1, as ROM bank 0 is always addressable from 0000_{16} (see figure 14).

When having 2 MB of ROM, the 5-bit address is not sufficient to locate the individual banks. The 6^{th} and 7^{th} bit of the bank number, is selected by writing a 2-bit value to the $4000_{16} - 5FFF_{16}$ area. The MBC will automatically combine the 5-bit and 2-bit numbers.

When having 32 KB of RAM, selecting a RAM bank is done by writing a 2-bit value to the $4000_{16} - 5FFF_{16}$ area. This select the appropriate RAM bank at $A000_{16} - C000_{16}$.

Type 2

This MBC has a 256 KB ROM and 256 bytes of RAM.

The RAM available when using this controller is quite special. The RAM doens't fill out the allocated address space on the Game Boy. Aside from this, the RAM only support storing 4-bit values in each byte, as the upper 4 bits are always zero when read.

Type 3

This MBC has a 2 MB ROM, 32 KB RAM and may have a Real Time Clock (RTC).

Selecting a ROM bank is done by writing a 7-bit value to the $2000_{16}-3FFF_{16}$ area. This selects the appropriate ROM bank at $4000_{16}-7FFF_{16}$. Values of 0 and 1 are both pointing to ROM bank 1.

Selecting a RAM bank is done the same way as type 1.

To read the Real Time Clock (RTC), the programmer can select a non-existing RAM bank between 8_{16} and C_{16}. After this, the current state of the RTC can be read from any address of the cartridge's RAM. The current time can be latched, by writing 00_{16} followed by 01_{16} to any address between 6000_{16} and $7FFF_{16}$.

5.4 Emulation

To use a real cartridge, special hardware is required. This hardware is out of the scope of this project. Therefore, we have used a file containing an exact copy of the ROM banks of a cartridge. These "ROM files" has been dumped from physical cartridges by emulating a physical Game Boy, which tells the MBC to read out every byte in sequence.

Our implementation loads a ROM file into a Python list in 16 KB blocks. These blocks are used as banks, to be swapped into the emulator.

All blocks are read into the computer's RAM; although, they are read-only to the Game Boy. This restriction is implemented in the `Cartridge` class by defining the special functions `__setitem__(self, address, value)` and `__getitem__(self, address)`.

We determine the MBC by reading address 0147_{16} of the first ROM bank. The value of this byte is used as index, in a lookup table, to find the corresponding MBC specifications.

The MBC specifications are implemented for MBC Type 1 and partially for MBC Type 3.

After the MBC is determined, we initialize the required RAM.

The cartridge is not directly accessible to the CPU. The CPU accesses the general `RAM` class. The `RAM` class redirects the calls to the cartridge, if the address points into the memory intervals of the cartridge.

5.5 Partial Conclusion

All software on the Game Boy has to come from the cartridge. Some cartridges give additional functionalities to the Game Boy and can have different MBCs. A lot of cartridges uses the same MBCs and the emulation of the cartridges is working with the

MBC type 1.

Banking extends the possibilities for games for the Game boy considerably. Although, each bank of the cartridge is just 16 KB, it is still possible to store games with a total size of 2 MB.

6 Random Access Memory

The CPU chosen for the Game Boy has a different memory composition, than a modern computer. The LR35902 has a 16-bit address space, which was a substantial constraint even at production time. Most of the design of the RAM and cartridge, is about solving these constraints.

6.1 Overview

The address space of the LR35902 is quite limited. With 16-bit address space, it is only possible to allocate 64 KB[1]. This is a significant constraint, as some popular games are in the range of 512 KB to 2 MB[6].

The engineers at Nintendo chose to split the RAM into groups, which are logically related to each other. First, memory allocated from the cartridge. Second, internal RAM of the Game Boy. Third, special addresses for input, output, sound, communication, interrupts etc (see figure 14).

6.2 Banking

Banking is used in the Game Boy to better utilize the limited address space. A specific interval of the address space is allocated as access to a single memory bank (see "Switchable ROM Bank" on figure 14). The bank can be switched between one of multiple banks available on the cartridge. Only one bank can be selected at once, therefore, the programmer has to manage and control the memory banks manually.

The memory banking will be discussed in further detail in the "Cartridge" section.

[1] $2^{16} = 65.536$ bytes

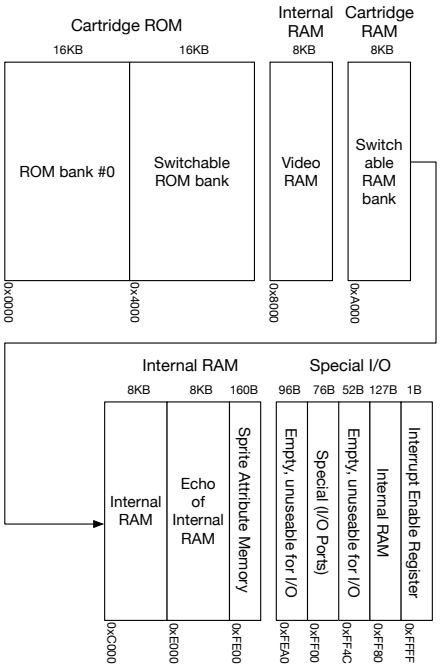


Figure 14: The RAM layout of the Game Boy

6.3 Video RAM

When a game wants to display a tile[2], it has to be loaded from the cartridge, into the area for tile data (8000_{16} to 9800_{16}). Depending on the placement, the tile will be available for the first, second, or both Tile Views (see figure 16). This is done to increase the utilization of memory, as elements might occur in both views at once.

Read more details about the video RAM in section 7.

6.4 Special addresses

Almost every setting of the Game Boy is set in the last addresses of the RAM. These are reserved for setting up interrupts, timers, sound, and parameters related to drawing the display (see figure 15).

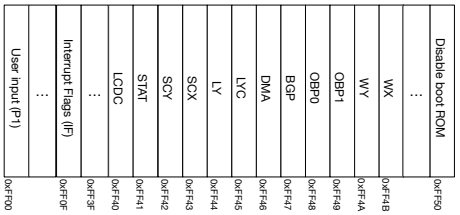


Figure 15: Addresses of special registers

[2] See section 7 about the display, for a defintion of "tile"

The list is very long and will not be described, in detail, in this section.

6.5 Emulation

When the emulator starts up, the RAM initializes the internal RAM as seen in figure 14. The RAM areas are initialized with zero written to all bytes. This is not part of our implementation, but a part of how Python manages memory.

On the real Game Boy, the RAM contains random values. This is caused by the transistors being in an unstable state before being turned on completely. When they are turned on, and ready for use, they remain to show the value they randomly had before being stable [17, Power Up Sequence].

Access to the `RAM` class is implemented by defining the special functions `__setitem__(self, address, value)` and `__getitem__(self, address)`. In case the address is not pointing to the RAM, but to either the `Cartridge` or a special register, the call is redirected to the correct class.

This could easily be emulated, by generating random numbers for all the cells in the RAM. We have chosen not to do this, to have a better overview when debugging. Not doing this, also allows for more deterministic errors, in cases where the wrong memory is accessed.

6.6 Partial conclusion

Although, the RAM is a central part of the operation of the Game Boy and this emulator, it has a fairly simple implementation.

The address space of the Game Boy is quite limited, but the extensive use of banking allows the Game Boy to do much more complex tasks.

For increased flexibility, the RAM handles all of the memory accesses for the CPU, but redirects some tasks to the cartridge and other components. This moves the complexity into specialized classes.

Although, the Game Boy has memory, which is initialized with random data, this is not a necessity to emulate. That could be implemented to increase correctness, but will also make debugging more complex.

7 Display

The Game Boy draws graphics onto the built-in LCD display. The graphics are 2-dimensional and constructed through "tiles". Tiles can be placed side-by-side through 2 views: Background and window. Tiles can be selected to work as sprites with transparency, using a color key. Sprites are generally used for objects moving freely on the screen.

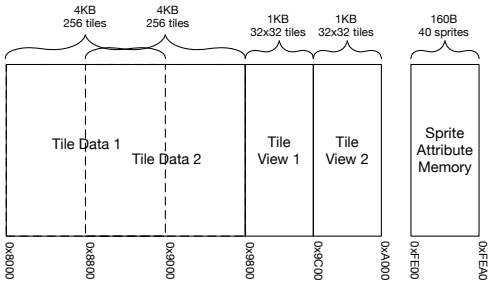


Figure 16: Allocation of video RAM

7.1 Tiles

Tiles are used in 2D-graphics and can greatly improve performance. The concept of tiles is rather simple. Instead of keeping the complete scene in a bitmap, you have a map of references to sub-images. The idea is to reuse the sub-images to minimize memory consumption. These sub-images could be parts to construct the background or terrain (grass, trees, water etc.). An example of use of tiles can be seen in figure 17.

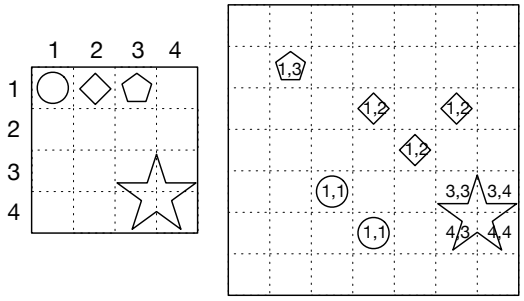


Figure 17: Illustration of wrapping and loading ahead

An alternative would be to have a frame buffer. This would require 5,760 bytes[3] for

[3] 160 pixels $*$ 144 pixels $*$ 2 bit colors/8 bits $*$

the Game Boy and compared to the 10 KB allocated to tiles and tile views, this would be an expansion of a little more than a percent. On the Game Boy, the tiles are of 16 bytes each. This would mean, that any change of a tile on the display, would require to move 16 bytes, rather than a single byte referencing a tile. Although, having a screen buffer will allow the programmer to make single-pixel drawing, and therefore having greater flexibility. It would also allow for much larger tile maps, as they do not have to be stored in the VRAM.

By having tiles instead, you still have to render the frame, but it is not necessary to store more than one horizontal line at a time, since they are continuously send to the LCD controller.

This also gives a great advantage, when rendering games, that scroll from side to side, since you only have to extend the tiles in a given direction, but more on that in section 7.7.

7.2 Tile Data

To conserve memory, the Game Boy is designed to reuse as much memory as possible. At the time the Game Boy was designed, it wasn't feasible to store a complete screen buffer for each of the background, overlay and moving characters. Therefore, to show any graphics on the screen, it has to be loaded from the ROM of the cartridge into a limited RAM area as a tile.

The tiles are of 8×8 pixels and have a 2-bit color palette. The tiles have to be stored in the area $8000_{16} - 9000_{16}$ or $8800_{16} - 9800_{16}$, as seen in figure 16[28, Fig. 13]. The area is chosen from the LCDC register, as described in 7.6.

The RAM areas allow for a total of 384 tiles (see figure 18). 128 of the tiles are overlapping between the two tile areas. This makes it possible to use common tiles in the "background", "window" and as sprites. This overlap is made to better utilize the limited RAM.

2 views = 5,760 bytes

Figure 18: Showing grid of tiles from Super Mario Land

7.3 Tile Views

The Game Boy has two static views, each spanning 32×32 tiles. These views are used as a screen buffer and spans each a total of 256×256 pixels.

The background tile view can be scrolled behind the window and sprites. The scrolling can be used to make a smooth transition in the game, for example when the player is moving towards the edge of the screen. The background tile view will always be behind the window and sprites.

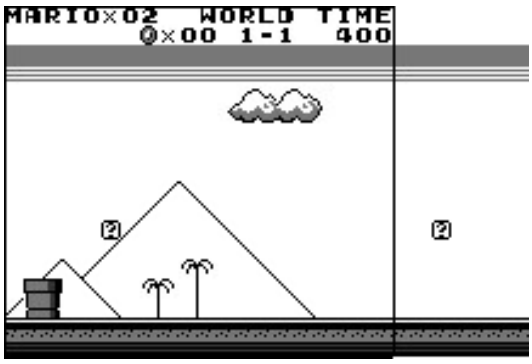


Figure 19: Background tile view from Super Mario Land. The black frame is showing the view port.

The position of window view can be set using `WX` and `WY` with single-pixel precision (see details in 7.6). The window is shown in front of the background, but can both be in front or behind the sprites. The tiles on the window are not transparent, and will hide the background underneath[17].

Depending on the settings in the LCDC register, the tile views can show data from one of two tile data areas (see 7.6). The first data area is index from 0 to 255 and the second data area is index from -128 to 127. The choice of first signed, then unsigned indexing, is a clever idea from the designers.

When a programmer is using the first

area, the indexes of tiles are simply an 8-bit numbering. Then, let us imagine, that the second area was indexed from 0 to 255 as well. When the programmer changes to the second tile area, all of the indexes for the common tiles have now shifted by 128 tiles. This means, all of the common indexes has to be subtracted by 128.

Instead of this, the second tile area uses signed 8-bit integers. This has the side-effect, that common tiles keep their binary value, but are interpreted as negative numbers (see figure 20).

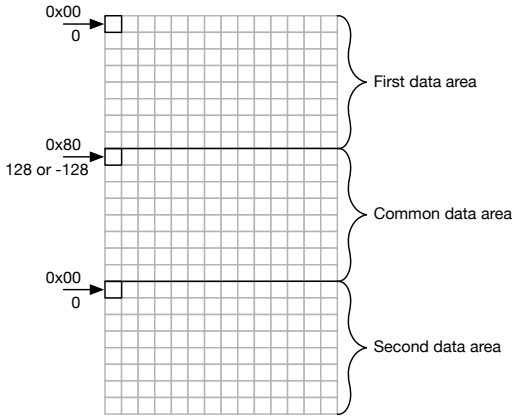


Figure 20: Illustration of how signed and unsigned numbers are used when indexing the tile data

7.4 Viewport

The Game Boy has a built-in LCD screen of 160×144 pixels. The two tile views are each larger than the screen area; therefore, the programmer has to choose a section to show on the screen. This is defined by the two registers `SCY` and `SCX`, each respectively located at $FF42_{16}$ and $FF43_{16}$ [28, Fig. 14].

A typical Game Boy game doesn't show the complete game area at once. Only the area immediately adjacent to the player is shown. The limited viewport allows the programmer to load ahead of the player.

If the viewport is located closer, than 160 pixels to the right side or closer than 144 pixels to the bottom, the view port will wrap around the edge of the tile view to the opposite side (see figure 22). Meaning, the programmer can avoid moving tiles, that are already on the screen, and instead only copy new tiles in.

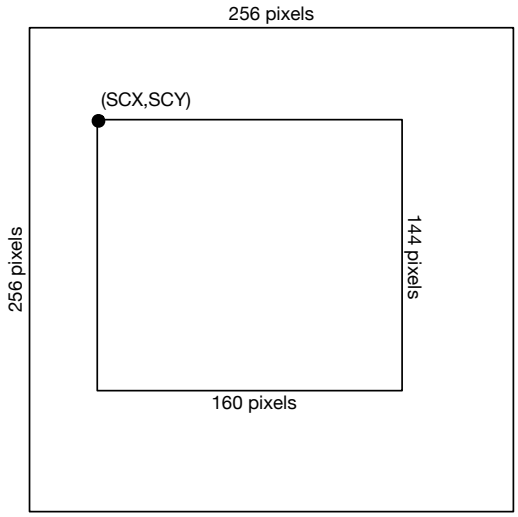


Figure 21: Viewport of the screen. `SCX` and `SCY` defines the location on the tile view

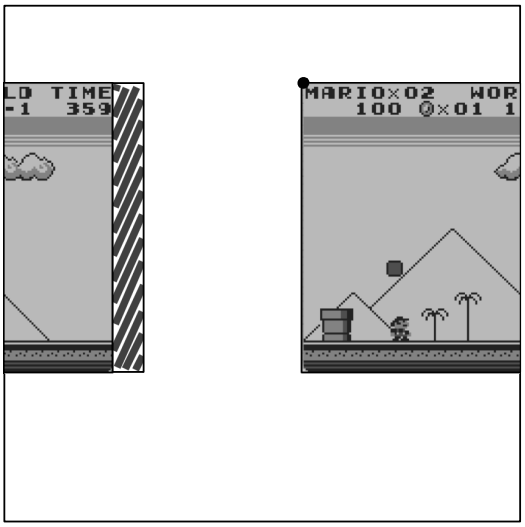


Figure 22: Illustration of wrapping and loading ahead

We observed, that Pokémon Blue will load the tiles, just outside the viewport, when the player moves in that direction. This dynamic loading, is most likely because of the very large game area.

Another scenario, would be a game that only has a 256×256 pixel game area. This could all be loaded at start-up and save time on loading during game play.

Generally speaking, it is recommended to use the background for the "game area", window for user-interface and sprites for moving objects.

In Pokémon, this wasn't possible with the graphical elements they wanted. In Pokémon they wanted an user interface, that partially overlapped the background (see figure 23). Since the window has no alpha channel or color key, they had to do something else. They did this by copying

Figure 23: Screenshot of Pokémon Blue with overlapping window

the current background view to the window view and moving the window in front of the background. This makes a static screenshot, that can be manipulated with. They have then drawn the menu on top of the screenshot. When the player exits the menu, the window view would simply be moved out of the view.

7.5 Sprites

All tiles are restricted to be drawn side-by-side in a grid. If this was the only way to draw graphics on the screen, it would become a big challenge to show any form of moving players or animations.

Luckily, the Game Boy also had a hardware sprite engine. A tile can be referenced in the Object Attribute Memory (OAM) (see figure 16) and become a sprite.

Sprites sacrifices one of the 4 colors in its palette in exchange for transparency. This means, the sprite can be drawn on top of the tile views in non-rectangular shapes. The sprites also have the advantage of being moved around with 1-pixel accuracy (tiles being 8-pixel accuracy).

The sprites can be referenced in either 8×8 pixels or 8×16 pixels. The hardware supports up to 40 sprites in either 8×8 pixels or 8×16 pixels. The sprites are kept in Object Attribute Memory (OAM). The sprites are not actually stored in the OAM, but rather referenced using pointers to the tile data[20].

Although, only 10 sprites can be shown on any horizontal line at once. If more than 10 sprites are to be show on a single horizontal line, only the 10 with the lowest memory addresses will be shown.

7.5.1 DMA to OAM

Programmers may want to switch between sprite tables in different scenes, but due to the limited memory, it can be necessary to do this quite often. In order to speed up this otherwise time consuming process, the Game Boy has special hardware to accelerate this task. The process is called a Direct Memory Address (DMA) transfer. It works by writing to address FF46_{16} with a prefix for the start address. The "source" is an interval from $\text{XX00}_{16} - \text{XX9F}_{16}$, where XX is the byte written to FF46_{16}. This will initialize the transfer process and copies all bytes from source to destination. The destination is fixed to $\text{FE00}_{16} - \text{FE9F}_{16}$. The transfer takes 160 microseconds and after that, the Game Boy can use the new sprites for drawing. While transferring, the Game Boy can only access memory from $\text{FF80}_{16} - \text{FFFE}_{16}$. Reading or writing to other addresses, in the mean time, will cause undefined behaviour.

7.6 Registers

For controlling the display, the Game Boy has an array of special registers. These registers, are not like the registers in the CPU, but regular memory in the main memory bank, accessible by the display driver.

7.6.1 LCD Display

The LCD Display Register, on address FF40_{16}, controls, if the display and tile views are individually enabled or disabled. It also controls where each view loads tiles

from[28, Fig. 15A].

Bit	Purpose	Reset	Set
0	Background Display	Off	On
1	Sprite Display	Off	On
2	Sprite size	8×8	8×16
3	Background tile view	View 1	View 2
4	Background tile data	Area 2	Area 1
5	Window display	Off	On
6	Window tile view	View 1	View 2
7	LCD Enable	Off	On

7.6.2 LCD Status

The LCD Status (`STAT`) register, on address $FF41_{16}$, are like the LCD Display register, but "Display Mode" spans over 2 bits and can be switched between 4 modes. Besides Display Mode, the LCD Status register are single-bit-switches to turn off and on[28, Fig. 15B].

Bit	Purpose	Value	Meaning

1-0	Display mode	00
		01
		10
		11
			`Mode 0`
			`Mode 1`
			`Mode 2`
			`Mode 3`

2	Coincidence flag	0 1	`LYC` $\neq$ `LY` `LYC` $=$ `LY`

		Reset	Set
3	`Mode 0` interrupt	Off	On
4	`Mode 1` interrupt	Off	On
5	`Mode 2` interrupt	Off	On
6	`LYC` = `LY` interrupt	Off	On
7	Unused		

7.6.3 LCD Position and Scrolling

These registers are located from $FF42_{16}$ to $FF4B_{16}$ and are illustrated in the original Game Boy patent[28, Fig. 15C-K]. All registers are 8 bits and used by software to control the video driver.

Name	Description
`SCY`	Scroll Y-coordinate
`SCX`	Scroll X-coordinate
`LY`	LCD Y-coordinate
`LYC`	LY Compare
`BGP`	Background color palette
`OBP0`	1. Sprite color palette
`OBP1`	2. Sprite color palette
`WY`	Window X-coordinate
`WX`	Window Y-coordinate

`SCY` and `SCX` defines the X- and Y-coordinate of the background tile view. This can be used for special effects and doesn't affect sprites or window tile view.[17, LCD Position and Scrolling].

`LY` indicates the line, that is about to be drawn, and setting `LYC` can enable an interrupt, when `LY` and `LYC` are equal. If the condition is met, a flag in the `STAT` register is set.

`BGP` is the color palette for the background and window tile view. The color palette defines up to 4 different colors and defines the colors of the tiles. Each pixel of the tiles reference a color in the palette. As a result of this, you can change the color dynamically to create effects. An example of this is to increment the darkness of each color in the palette to achieve a fade-out effect.

The sprites are split between two color palettes, `OBP0` and `OBP1`. They work the same way as `BGP`, but the lower two bits are not available, because they are reserved as transparent.

`WY` and `WX` defines the X- and Y-coordinate of the window tile view[17, LCD Position and Scrolling].

7.7 LCD

In modern computer graphics, a bitmap is rendered in memory and is stored in a memory buffer (often even a double buffer) until the monitor is ready for a refresh. The refresh is then done by transferring the final image to the screen from the buffer.

On the Game Boy, this single buffer, would require 57.6% of the current video memory. Therefore, the rendering of the display, is performed one line at a time. This is called "scanline rendering"[27]. This type of rendering means, that the display only needs a buffer equal to the amount of either the horizontal or vertical pixels.

The graphics are rendered from top to bottom in a loop like this:

- Load row `LY` of the background tile view to the line buffer.

- Overwrite the line buffer with row `LY` from the window tile view.
- Sprite engine generates a 1-pixel section of the sprites, where they intersect `LY` and overwrites the line buffer with this.

The background, window, and sprites are merged together and we end up with a single line for background, window and sprites. Generating 144 of these lines, while incrementing `LY`, will result in one frame on the LCD.

7.8 Emulation

The Game Boy has a color palette of only 4 shades of grey. A modern computer has 24-bit color palette arranged in 1 byte for each of the colors: Red, Green and Blue (RGB). This poses an incompatibility of using the Game Boy graphics. Therefore, we have chosen 4 shades of grey in RGB, that we will use for conversion.

Apart from the color palette, the way the data is arranged, is also different.

A traditional RGB color is arranged as an array of 3 bytes, where each byte represent the intensity of each sub-pixel.

The Game Boy has the pixel's color code arranged across two bytes; meaning, if you were to load 8 pixels on the screen, you would load in 2 bytes – giving 2 bits per pixel. The leftmost of the 8 pixels would get its color from the 8^{th} bit of the first byte and the 8^{th} bit of the second byte. The bit from the first byte is used as the most significant bit of the color code. These 2 bits represents the grey shade of 1 pixel (see figure 24).

Because of this, we will have to convert the colors from the Game Boy, before we can draw it on the screen. To do this, we have made 3 buffers as NumPy arrays of integers: Tile Map, Tile View 1 and Tile View 2. For every frame, we run through all the bytes in the Tile Map of the Game Boy and transfer them into a NumPy buffer. For the tile views, we copy blocks of 8×8 pixels from the Tile Map's NumPy buffer into the referenced places in the tile views.

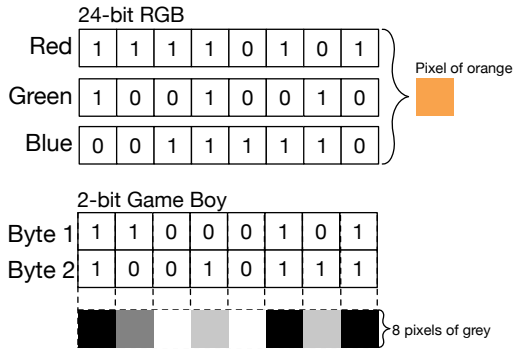


Figure 24: 24-bit Red, Green, Blue (RGB) compared to 2-bit grey-scale for the Game Boy

There is no form of caching or check to only blit the areas that have changed. We chose to extend the emulator instead of optimizing this, therefore, it would be a good place to optimize. A crude test shows a $4\times$ performance boost, when all SDL conversions are disabled. The conversions has to take place, of course, but it still shows how much performance is lost proportional to the rest of the emulator.

The Sprite buffer has, at the time of writing, yet to be implemented.

When the buffers for tile views and sprites are created in SDL, they can be copied into the final image to show on the display of the emulator (see figure 25).

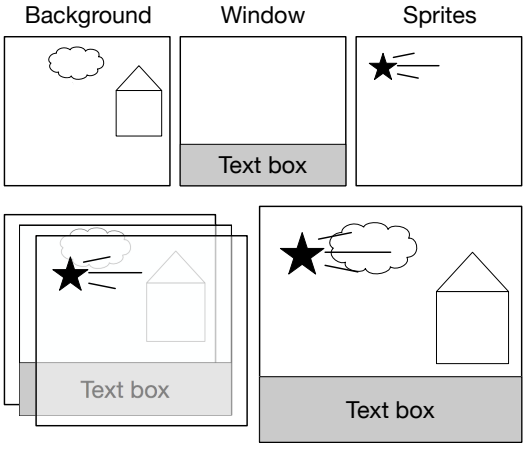


Figure 25: The three graphic layers

7.9 Partial conclusion

The Game Boy's use of tiles is an efficient way of getting simple 2-dimensional graphics on a relatively simple processor. The use

of tile views combined with sprites makes a fast and flexible graphics processor. The fast and precise sprites compensates for the static and limited tile grids.

Because of the interrupt-system of the Game Boy, it is possible to refresh and make quick changes to sprites while rendering the data on screen.

The sprite engine and tile views are not CPU-demanding and free up alot of resources in the Game Boy. Because of the difference in architecture from the host system, all these features has to be emulated in software and takes up alot of resources. The drawing routines are a place where optimizations will possibly have a high performance gain.

8 Interaction

The joypad of the Game Boy has 4 standard buttons (start, select, A, and B) and 4 directional buttons (up, down, left, and right). They are represented in the memory address $FF00_{16}$ in an 8-bit system as figure 26 illustrates.

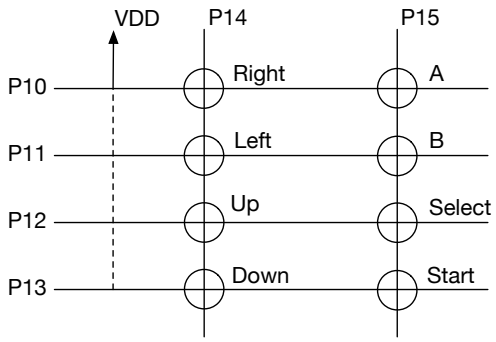


Figure 26: Interaction illustration. Replica of Nintendo's patent from 1990.[28, Fig. 8]

The 8-bit system is as follows:

Bit	Name	Description
7		Always set
6		Always set
5	`P15`	standard buttons
4	`P14`	directional buttons
3	`P13`	down **or** start
2	`P12`	up **or** select
1	`P11`	left **or** B
0	`P10`	right **or** A

The bits from 0-3 are directional buttons or standard buttons. When bit 4 is selected (signaled by reset), the directional buttons are probed and when bit 5 is selected, the standard buttons are probed. It is up to the programmer, to ask the Game Boy for either directional buttons or standard buttons.

A reset bit indicates that the player pressed a button. A case could be, if the player wants to go left in a game. The programmer would probe the controller and ask for the directional buttons. The controller would then return the byte shown below:

$$left = 11101101_2$$

Bit 4 indicated, that the programmer asked for a directional button and bit 1 indicates, that the left button was pressed. It is possible for the user to use multiple buttons at the same time. It will require two probes to get standard and directional buttons.

8.1 Emulation

The implementation of this has been done by making a class named `Interaction`. This class has three functions and a stack. One of the functions is for appending key presses to the stack. The stack is used to save all key presses between each frame. In the end of each frame, there is used a flush function, which flushes the stack and the unused key presses are deleted. The pull function analyzes all keys in the stack by the 8-bit system, as described, and returns an 8-bit number. The returned byte indicates all key presses from the stack either for the directional buttons or the standard buttons.

8.2 Partial conclusion

The interaction has been emulated with success, by using a stack. The interaction illustration from Nintendo's patent was a big help with the understanding and implementation of this section. From the 8-bit system it has been easy to analyzes key presses with the pull function.

9 Verifying solution

When designing software, it is hard to proof the correctness of all functions, just by observing it in action. Especially, in CPU emulation, we have seen, even the slightest variations from the real CPU, can produce largely different outcomes. There has been made some tools to ease the debugging with tile maps and tile views. Additionally, unit test has been made, which will help us to prove the correctness of the implementation. This would also ensure, that side effects caused by further development, would be caught instantly.

9.1 Debugging

Besides unittests, we implemented a simple routine to output the program counter and registers for each instruction. We implemented the same routine on a competitive emulator called Gearboy[30]. We would then search for differences in execution states and in that way locate bugs in our code. The Gearboy is not guaranteed to be correct either, but it would pin-point differences and give us a hint on where to look.

If the CPU encounters a situation with undefined behaviour (ideally a software bug in the cartridge), it would dump the entire memory, CPU state, key presses, and save all SDL buffers as BMP.

In order to debug the visual appearance, the PyBoy implementation features a "debug mode", where both tile views and tile maps are displayed separately from the game. This gives an overview of the loaded graphics while running the game.

Since the Game Boy can change tile offset and have different ways of reading the

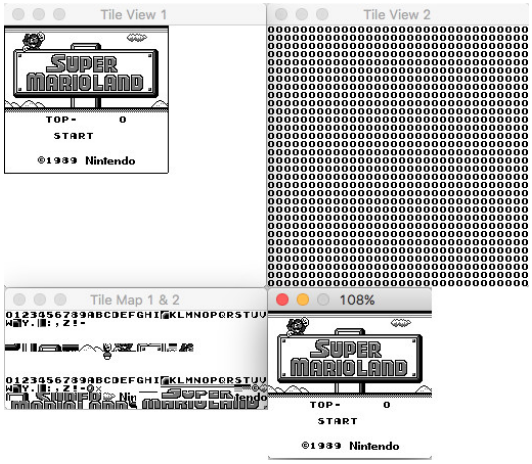


Figure 27: Screenshot of PyBoy running in debug mode

tile data, bugs can easily occur. In this example, the offset was not set correctly, and all sprites were shifted (28). Comparing

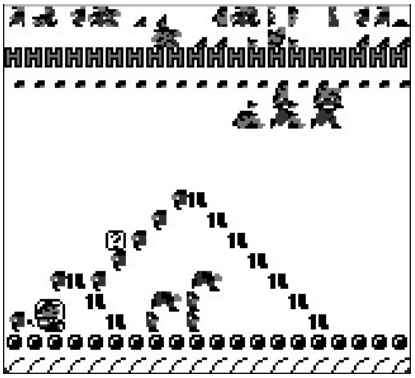


Figure 28: Screenshot of PyBoy with wrong tile offset

states with the Gearboy would not show this kind of errors, since the bug was an error in the SDL implementation.

9.2 Unit test

9.2.1 CPU

Testing the CPU has proven to be a necessity in making the emulator. It is realistically impossible to find existing software, that can proof the correctness of all operations.

Normally, when running software, it can be obvious where the errors are, from a stack trace. But at the hardware level of the Game Boy, there exist no errors, nor stack traces. Most operations can be im-

plemented completely wrong, but still let the CPU continue running.

An example could be an ALU operation, that didn't set the carry flag. This would cause no visible errors, as the software will most likely still run, but the sematics of the software has changed.

An issue, we observed, in this project, was the operation `LDD (HL), A`, which loaded the value of register `A` into the memory address of the register pair `HL` and, importantly, *decremented* the value of `HL`. In our early implementation, LDD wrongfully *incremented* `HL`, instead.

A loop in the boot-ROM runs `LDD` until the 8th bit of H turns to zero. This happens in both cases, but in our version due of an overflow, and not because of subtraction, in register H. This made it seem like the code ran fine, as it continued.

Another example, found in the boot-ROM, was a missing 8-bit mask. Most ALU-operations end with an 8-bit mask. This was overseen, and when the CP operation checked, if register A was 0, it actually compared 500_{16} to 0.

Testing
The tests of the CPU has been done by creating unit tests for every opcode. All tests are checking the mathematical correctness and flags. For operations like jump and load, the tests would check, if it jumped or loaded to the correct address or register. For operations, which deals with the stack, the tests verifies that push and pop are implemented correctly, in respect to the stored values and endianness.

Testings all possible states of the CPU, is practically impossible, but a thorough test should be able to ensure an accurate implementation, defined by the requirements specification. We can conclude, that our unittests doesn't suffice, since running a test-ROM reveals multiple errors. This is probably due to corner-cases or inadequately documented features.

9.2.2 Display

Testing the display is in common cases very simple. Compared to the CPU, a wrongfully implemented part of the display will not have consequences for the Game Boy.

If the CPU calculates something wrong, it will keep working with this error, and in worst case crash the Game Boy. If the display shows something it shouldn't, or something is missing, it would possibly look wrong, but in most cases not have permanent damage, as it will be overwritten at some point. The point here being, that the display only shows whatever the CPU has made it show, but the display will not alter the data.

The display has registers that show, how far it is in drawing on the screen and when it is safe to change the data for the display. If these are not set, it will possibly lock up a game, that waits for a register to become a specific value. This is often seen, when a game spinlocks for the `VBLANK` interrupt, by comparing memory address $FF44_{16}$ to be above or equal to 90_{16}.

Testing
The tests of the display has been done by comparing the visual result on the display with other emulators. The position of the tiles has been measured and by comparing to other emulators, the tiles has been determined to be correctly selected.

9.2.3 RAM

Most of the RAM can be statically tested, to see if a value can be stored and read. It's also important to check, that the parts of the address space, that isn't actually located in the RAM, is forwarded to the boot-ROM or cartridge.

Testing
Testing of the RAM has not been done formally, because we have not experienced any problems with the implementation yet. The RAM is tested indirectly by the unit tests of the CPU, as the opcodes uses the RAM.

9.2.4 Cartridge

The cartridge has mostly read-only memory, which has to be checked to not be

writable.

The MBCs has to be checked, to be sure, if they can change correctly and doesn't work in ways the Game Boy can't. An example of this could be to have ROM-bank 0 accessible at both $0000_{16} - 4000_{16}$ and $4000_{16} - 8000_{16}$. In this example, the Game Boy will allocate ROM-bank 1 to $4000_{16} - 8000_{16}$, even though the programmer asked for bank 0. This is an implementation detail of the Game Boy, which has to be accounted for.

Testing
Testing the cartridges has not been done because we have not experienced any problems with the implementation yet. A good test would be to write some software, which go though the implemented MBCs and check that the read-only memory is not writable. The data from the emulated cartridge can be verified using a checksum.

9.3 Test-ROMs

For validating our emulation, we have found Blargg's tests, which are designed to validate Game Boy emulators. These tests are detailed and even popular emulators such as Gearboy, no$gmb, and VGB does not pass all of them, even though, they work fine with many games. Blargg's tests goes through CPU, sound, memory timing, instruction timing, and OAM[2].

Our implementation did not pass any of these tests, when the boot ROM was running successfully. Only after debugging, some of the tests passed and the emulator can now pass the tests for 10-bit operations, jumps, and register loads.

Test results are in Appendix H.

9.4 Partial Conclusion

Comparing CPU states has proved to be very effective. The same result could have been achieved without it, but it speed up the debugging process substantially. A correctly implemented unit test would have given the same result, but, with Gearboy, we were able to locate overseen features, that were not addressed in the unittest.

The unittests has helped bringing the small and important details up, as these can take a long time to backtrace. This made sure, we could keep progressing in the development and get closer to running a game. Only a few of the tests from the test-ROMs succeeded, but this does not mean that the implementation is completely wrong and will still work in a subset of programs.

10 Performance

10.1 Finding optimal datastructure

This sections was found to be necessary when the former implementation was too slow compared to the Game Boy. The time for each frame was 253 milliseconds and made the emulation too slow compared to the Game Boy, which shows frames at an interval of 16.7 milliseconds (see Appendix F for test results). This led to changes, that improved code performance substantially.

10.1.1 Test purpose

Our initial implementation of the opcode table, was very time consuming and we found, that lookups took the majority of the execution time of each instruction. The initial implementation had an if-statement for each type of instruction and it would have to iterate through all if-statements until it found the right operation. Instead, we would implement a lookup table using a list, since we already knew the integer value of the opcode.

The old implementation would look up each opcode using an if-case like this:

```
if opcodes.NOP == inst.operation:
    ...
elif opcodes.EI == inst.operation:
    ...
elif opcodes.LD == inst.operation:
    ...
```

By having all opcodes in a list, we would potentially gain a penalty in memory usage, due to repetition. Since most of the opcodes look alike, with only minor changes,

we would have the same function written multiple times. We would compensate this by making selected functions generic and referencing them by a function pointer. The new implementation was drafted like this:

```
opcode = self.bootROM[pc]
operation = opcodes.opcodes[opcode]
```

Using the PC as an index in the boot-ROM and fetching the next instruction value. The next line would find the opcode in the op-table.

10.1.2 Test setup

The test would benchmark the candidates in the same scenarios, but with variable table sizes. We were interested in finding the fastest static lookup table, but also *when* to use different tables. Our hypothesis was, that every candidate would have certain advantages in memory consumption, look-up time or initialization time. Our main concern was look-up time, at this point, so initialization time was not in the scope of this test, nor was memory consumption.

The test would measure the lookup time of each list implementation with lengths of 2, 4, 8, 16, 32, 64, 128, 256, and 512 elements. For each test, we would generate a list of integers imitating opcodes and generate lists of dummy functions imitating the Python implementation of the CPU instructions.

The test would look up all elements from 0 to n, where n is the length of the list. This means, that all opcodes will be looked up once for each test. Each test would be repeated 30.000 times to minimize noise from the operating system and other applications on the test machine. Then, an average of all 30.000 tests would be calculated and used for the final result.

10.1.3 Datatypes

We wanted to test the following datatypes and techniques:

1. NumPy Array
2. Lists (Python built-in)
3. Tuples (Python built-in)
4. If-statements

NumPy Arrays are arranged like C-arrays, which means, that lookup is done by calculating the byte-offset from the beginning of the array. This gives an expected worst-case of O(1)[21]. Python lists are a combination of linked lists and C-arrays. It will dynamically allocated more space to the list when needed, but it still manages to deliver a O(1) look up time like in C-arrays. The reallocation isn't a major concern in this context, since this will only affect initialization[10].

Python tuples are meant to imitate C-structs by not being mutable. The performance difference in tuples and lists are not documented, which is also one of reasons for this test[22].

The if-statements are expected to be the slowest solution, since a lookup have a worst case of O(n), where the other candidates have O(1).

10.1.4 Test results

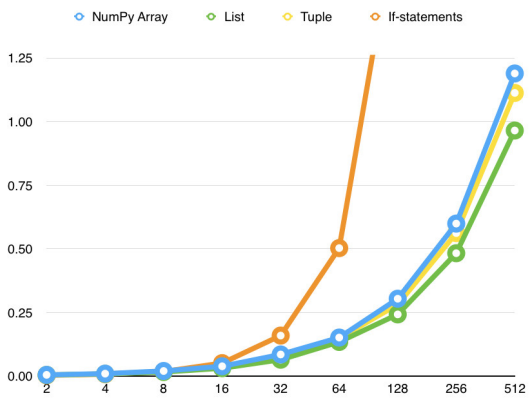


Figure 29: Comparison of average lookup time for NumPy Arrays, Python lists, Python tuples and If-statements (lower is better)

This test shows that if-statements are clearly outperformed in lookup time by the other candidates. With a peak of 27.29 seconds, it doesn't even come close to the others. It shows a close competition between the three list implementations. However, the built-in lists have a slightly lower lookup time, than the two others.

The test result may look like the lookup time increased exponentially, but that is due to the fact that the tested length increased exponentially and the tests became longer proportionally to the increased length.

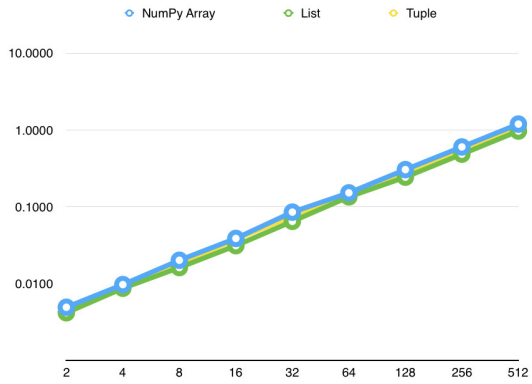


Figure 30: Comparison of average lookup time for NumPy Arrays, Python lists, and Python tuples (lower is better)

If we look at the data logarithmically, instead, we will see a linear growth, which supports the claim, that the tests have increasing length. This verifies the time complexity to be O(1).

10.2 Interpreter vs. JIT

10.2.1 Test purpose

With the initial setup, we used the Python interpreter developed by Python Software Foundation. We found, that this interpreter wasn't able to produce enough Frames Per Second (FPS) to run the emulator. We optimized the code as described previously in section 10.1, although, this didn't yield sufficient results. The FPS was still below the threshold.

We searched for other Python implementation and found "PyPy", that advertised improved performance for repetitive operations[11]. Since our implementation will rerun the same Fetch-Decode-Execute cycle, once for every instruction in the ROM, our case would fit this description quite well.

Since PyPy is *just* another Python implementation, no code had to be rewritten and the Python implementations still remain interchangeable.

10.2.2 Test setup

We needed a test to compare the two Python implementations and decided the boot-ROM would represent a suitable test, since it has a clear start and stop point and runs several loops. We could have constructed a test-ROM, that would run each instruction or the most common ones, but the boot-ROM seemed like a better real-world example. The boot-ROM has preparation-loops, while showing an animation on-screen at 60 FPS (max speed). Also, the boot-ROM doesn't utilize interrupts – only spinlocks. This means, that the emulators never reach an idle state, by calling `HALT` or `STOP`.

10.2.3 Results

The results are shown in figure 31. The delta time for each frame rendered over a total of 355 frames. The dashed line shows the 60 FPS barrier, at which the Game Boy runs. The data is the average for three runs each.

The graph shows an initial spike in time, which normalizes within 5-10 frames. Python and PyPy both have the same spike to begin with, but PyPy will decrease its execution time dramatically. Our hypotesis is, that PyPy needs a few runs in order to start the optimization, but even with PyPy's relative speed to Python, there still is a quick fluctuation. When investigating the boot-ROM, we found, that the first step is to zero the VRAM, which spans 8192 bytes and is performed by a routine of 48 clock cycles.

$$8192 \text{ bytes} * 48 \text{ cycles} = 393216 \text{ cycles}$$

The CPU runs at an average of 4194304 Hz[4], which means, we can calculate the expected time frame for the zeroing routine.

$$\frac{393216 \text{ cycles}}{4194304 \text{ Hz}} = 0.0938 \text{ seconds}$$

$$\frac{0.0938 \text{ seconds}}{0.01667 \text{ seconds per frame}} = 5.625 \text{ frames}$$

From the execution, we estimated 5-10 frames, which fits well with the expected time of 5.625 frames. Notice, that the spike is above the 60 FPS line, this will make the 5.625 frames seem longer in reality, but the emulated game won't notice.

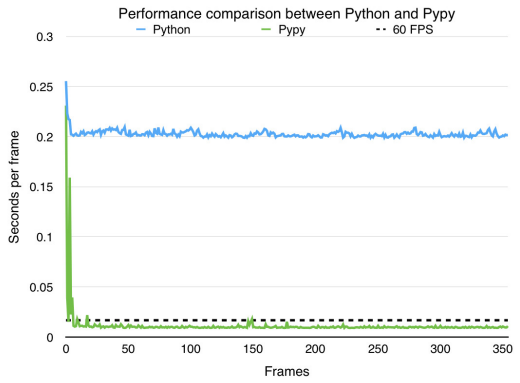


Figure 31: Render time for each frame in the boot-ROM. See Appendix G for greater detail

After the preparation of the VRAM, the Game Boy is made busy with spin locks, which are easily optimized by PyPy. As seen in the graph, the Python interpreter is running behind, and will have to run 12 times faster (0.200 seconds/$\frac{1}{60}$ = 12), just to keep the timing – leaving no head room.

10.2.4 Test enviorment

The tests were performed on a MacBook Pro (Late 2013) with a dual-core 2.4 GHz Intel Core i5 and 8 GB 1600 MHz DDR3 RAM. The test machine was rebooted before the test, and left idle for 300 seconds after reboot to make sure the OS had finished loading.

10.3 Partial conclusion

The implementation will use Python's built-in lists due to the better performance in lookup time, but we found, that the lists are only faster for tables above 8 elements. The full table is located in appendix D. It is noteworthy, that both lists and tuples were faster than NumPy Arrays in every test-case.

PyPy made it possible to achieve desired performance without optimizing the code extensively, which made it possible for us to spend more time on extending the emulators functionality. Even with heavily optimized code, we might not have been able to accomplish satisfying performance with the standard Python interpreter.

11 Conclusion

The Game Boy emulator has a structure, where each class represents its real counter part. This gives a more natural flow in the code, and keeps a logical separation of the classes. The implementation is working on a subset of programs. An example of this is the boot-ROM, which runs without any errors. The cartridge dump for Super Mario Land is working as well, but is only able to load the main menu, let the player press "start", and load the first scene. Although, sprites are not yet implemented; therefore, the sprite of Mario is not visible.

The `CPU`, `Cartridge`, `Motherboard`, `RAM`, `Display`, and `Interaction` has been implemented with success. Emulating the boot-ROM turned out not to be unessential as it only initializes hardware, shows the Nintendo logo, and checks for counterfeit games. These things are either way initialized by the emulator.

The cartridge emulation has been found to make use of memory bank controllers. These controllers make banking possible and extends the possibilities for games for the Game boy considerably. Only Type 1 of the controllers is implemented with success.

For the implementation of the RAM, it does not seem to have any bugs. For increased flexibility, the RAM handles all of the memory accesses for the CPU, but redirects some tasks to the cartridge and other components. This moves the complexity into specialized classes.

The display part of the Game Boy seems to work fully from what we observed, on screen. Tiles and positions are correct on the screen, when comparing to other emulators. The sprite engine and tile views allow for flexible 2D graphics on the Game Boy. Even though, the possibilities to draw graphics are limited, it still produces 60 frames per second, while not using excessive time on the CPU. Because of the difference in architecture from the host system, all these features has to be emulated in software and takes up a lot of resources. The drawing routines are a place, where optimizations will possibly have a high performance gain.

We experienced bad performance with the emulator during the project. We made some performance tests, which ended in a new and better implementation. The optimization utilized look-up tables for decoding of opcodes. The use of the PyPy Just-In-Time compiler, instead of the Python interpreter, made it possible to achieve the desired performance. This was possible without optimizing the code extensively, which gave us more time to spend on extending the emulator's functionality.

References

[1] About Simulator. https://developer.apple.com/library/ios/documentation/IDEs/Conceptual/iOS_Simulator_Guide/Introduction/Introduction.html. [Online; accessed on 4 Januar 2016].

[2] Blargg's tests. http://gbdev.gg8.se/wiki/articles/Test_ROMs. [Online; accessed on 12 Januar 2016].

[3] Charting the Rise of the Solid State Disk Market. http://www.storagesearch.com/chartingtheriseofssds.html. [Online; accessed on 21 December 2015].

[4] Emulating the Core, Part 2: Interrupts and Timing. https://realboyemulator.wordpress.com/2013/01/18/emulating-the-core-2/. [Online; accessed on 7 Januar 2016].

[5] Game Boy accessories. https://en.wikipedia.org/wiki/Game_Boy_accessories. [Online; accessed on 12 Januar 2016].

[6] Game Boy Bootstrap ROM. http://gbdev.gg8.se/wiki/index.php?title=Gameboy_Bootstrap_ROM&oldid=192. [Online; accessed on 22 December 2015].

[7] Game Boy chips - CPU. http://www.tinytransistors.net/index.php?option=com_content&view=article&id=11&Itemid=11&limitstart=1. [Online; accessed on 13 Januar 2016].

[8] GBA4iOS Features. http://www.gba4iosapp.com/features/. [Online; accessed on 12 Januar 2016].

[9] Happy 20th b-day, Game Boy: here are 6 reasons why you're #1. http://arstechnica.com/gaming/2009/04/game-boy-20th-anniversary/. [Online; accessed on 21 December 2015].

[10] How are lists implemented? https://docs.python.org/2/faq/design.html#how-are-lists-implemented. [Online; accessed on 21 December 2015].

[11] How fast is PyPy? http://speed.pypy.org. [Online; accessed on 7 Januar 2016].

[12] Intel 8080. https://en.wikipedia.org/wiki/Intel_8080. [Online; accessed on 21 December 2015].

[13] Intel 8080 Architecture. https://en.wikipedia.org/wiki/Intel_8080#/media/File:Intel_8080_arch.svg. [Online; accessed on 4 Januar 2016].

[14] Intel 8086. https://en.wikipedia.org/wiki/Intel_8086. [Online; accessed on 5 Januar 2016].

[15] "Manually" extracting a ROM (Thread of a web forum). http://web.archive.org/web/20060507053755/http://forums.cherryroms.com/viewtopic.php?t=3848&start=75. [Online; accessed on 21 December 2015].

[16] Memory Bank Controllers. http://gbdev.gg8.se/wiki/articles/Memory_Bank_Controllers. [Online; accessed on 12 Januar 2016].

[17] Pan Docs. http://problemkaputt.de/pandocs.htm. [Online; accessed on 12 January 2016].

[18] Pan Docs: Game Boy Technical Data. http://problemkaputt.de/pandocs.htm#gameboytechnicaldata. [Online; accessed on 18 January 2016].

[19] Pan Docs: Timer and Divider Registers. http://problemkaputt.de/pandocs.htm#timeranddividerregisters. [Online; accessed on 13 January 2016].

[20] Pan Docs: VRAM Sprite Attribute Table (OAM). http://problemkaputt.de/pandocs.htm#vramspriteattributetableoam. [Online; accessed on 13 January 2016].

[21] The N-dimensional array. http://docs.scipy.org/doc/numpy/reference/arrays.ndarray.html. [Online; accessed on 21 December 2015].

[22] Why are there separate tuple and list data types? https://docs.python.org/2/faq/design.html#why-are-there-separate-tuple-and-list-data-types. [Online; accessed on 21 December 2015].

[23] Z80 Undocumented Features (in software behaviour). http://www.z80.info/z80undoc3.txt. [Online; accessed on 21 December 2015].

[24] Zilog Z80. https://en.wikipedia.org/wiki/Zilog_Z80. [Online; accessed on 21 December 2015].

[25] Det Kongelige Bibliotek. Emuleringsstrategien. http://digitalbevaring.dk/emulering/. [Online; accessed on 22 December 2015].

[26] Zilog Inc. Z80 Microprocessors. www.zilog.com/manage_directlink.php?filepath=docs/z80/um0080, 2015. [Online; accessed on 29 December 2015].

[27] Marc Erich Latoschik. Realtime 3D Computer Graphics / Virtual Reality. http://www.techfak.uni-bielefeld.de/ags/wbski/lehre/digiSA/WS0607/3DVRCG/Vorlesung/13.RT3DCGVR-vertex-2-fragment.pdf. [Online; accessed on 12 Januar 2016].

[28] S. Okada. System for preventing the use of an unauthorized external memory, 1992. US Patent 5,134,391.

[29] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2013.

[30] Ignacio Sanchez. GitHub - GearBoy. https://github.com/drhelius/Gearboy. [Online; accessed on 14 Januar 2016].

[31] Costis Sideris. FPGABoy. http://www.fpgb.org. [Online; accessed on 12 Januar 2016].

Appendices

A Class Map

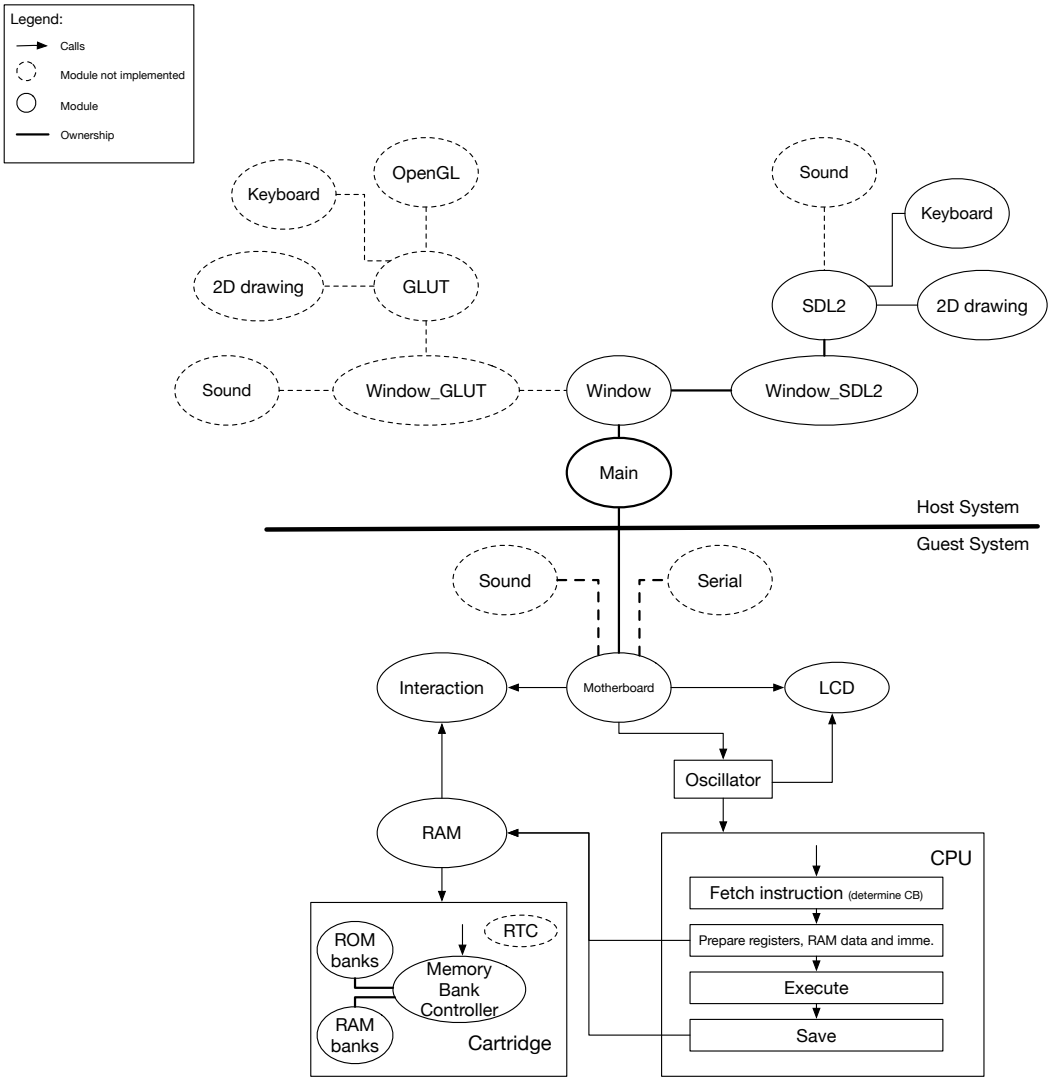


Figure 32: Map of all classes and their relations

B Page 14 of Zilog Z80 Reference Manual

CPU Register

The Z80 CPU contains 208 bits of read/write memory that are available to the programmer. Figure 2 shows how this memory is configured to eighteen 8-bit registers and four 16-bit registers. All Z80 CPU's registers are implemented using static RAM. The registers include two sets of six general-purpose registers that can be used individually as 8-bit registers or in pairs as 16-bit registers. There are also two sets of Accumulator and Flag registers and six special-purpose registers.

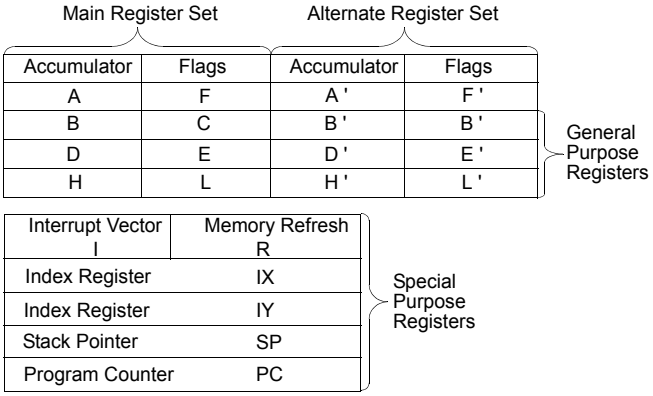


Figure 2. CPU Register Configuration

Special-Purpose Registers

Program Counter (PC). The program counter holds the 16-bit address of the current instruction being fetched from memory. The Program Counter is automatically incremented after its contents are transferred to the address lines. When a program jump occurs, the new value is automatically placed in the Program Counter, overriding the incrementer.

Stack Pointer (SP). The stack pointer holds the 16-bit address of the current top of a stack located anywhere in external system RAM memory. The external stack memory is organized as a last-in first-out (LIFO) file. Data can be pushed onto the stack from specific CPU registers or popped off of the stack to specific CPU registers through the execution of PUSH and POP instructions. The data popped from the stack is always the most recent data pushed onto it. The stack allows simple implementation of multiple level interrupts, unlimited subroutine nesting and simplification of many types of data manipulation.

C Boot-ROM picture

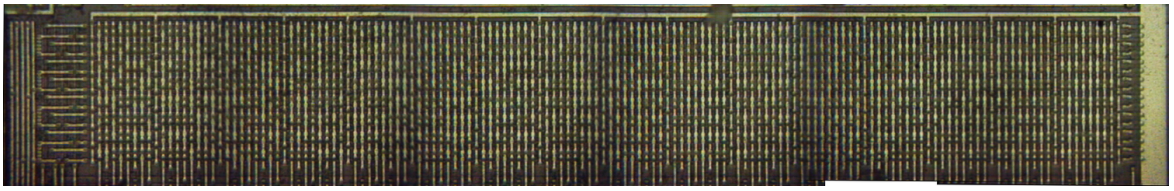


Figure 33: Complete picture of the boot-ROM

D Performance test results

Lists vs. Tuples vs. If-statements

	NumPy Array	List	Tuple	If-statements
2	0.0049	0.0042	0.0049	0.0032
4	0.0097	0.0087	0.0090	0.0075
8	0.0201	0.0162	0.0178	0.0181
16	0.0387	0.0311	0.0377	0.0510
32	0.0850	0.0645	0.0755	0.1593
64	0.1515	0.1343	0.1488	0.5029
128	0.3040	0.2425	0.2813	1.8293
256	0.5992	0.4828	0.5619	6.9974
512	1.1889	0.9648	1.1125	27.2913

Figure 34: Test results when comparing NumPy array, Python lists, Python tuples and if-statements. Values are in seconds with 30.000 repeatitions. Green indicates the fastest technique. Yellow is the second fastest technique.

E CPU dissection

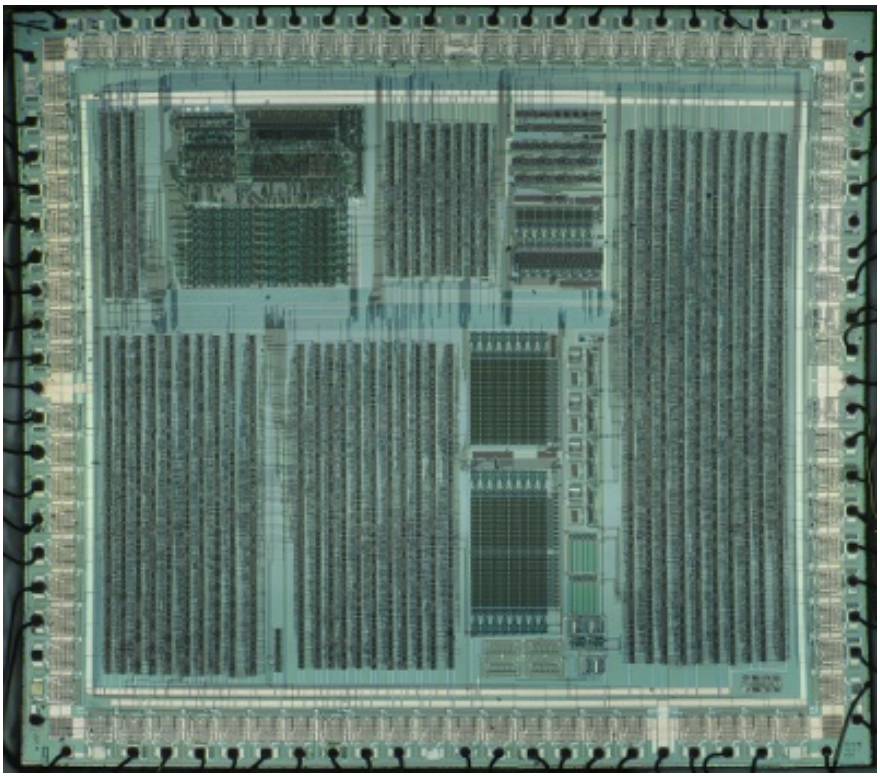


Figure 35: Overview picture of the Sharp LR35902 CPU[7]

F Data basis for optimization efforts

Init Cartridge

Commit Hash	Python	Pypy
4bba9c737572	2.5337	0.3299
117beb6c059a	2.1089	0.4079
d459b4e9df91	2.4332	0.3811
d09ab2fcf116	2.3255	0.3596
9a712a0fb67f	2.3872	0.3711
86c6076929f5	2.1833	0.3870
112d55a40832	0.2629	0.0501
098253632c6e	0.2727	0.0516

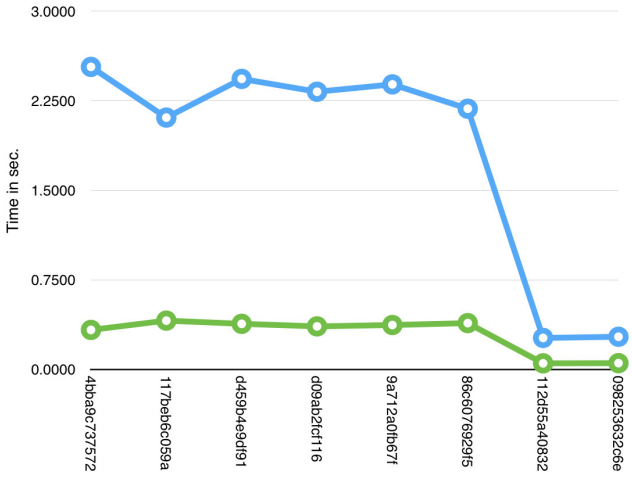


Figure 36: Render time for each frame in the boot-ROM with the original implementation

Keep in mind, that this test was conducted before all opcodes had been completely implemented. The two Python implementations would run the same code, but the code didn't behave as expected by a Game Boy.

G Performance comparison between Pypy and Python

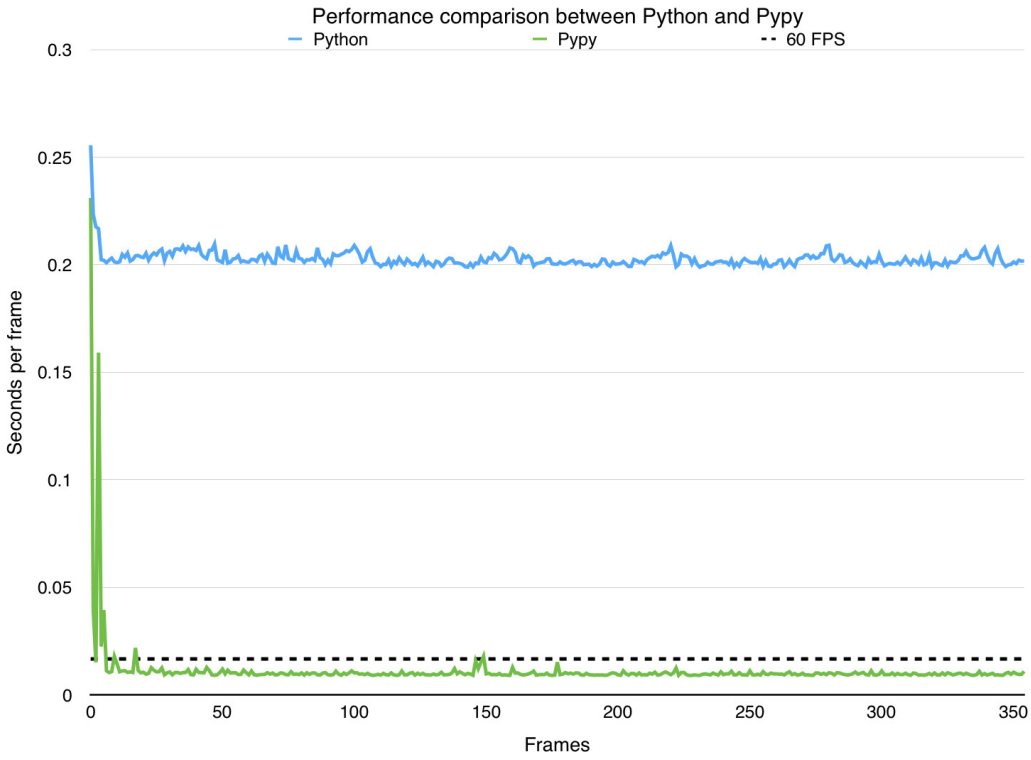


Figure 37: Render time for each frame in the boot-ROM

H Results of Test ROMs

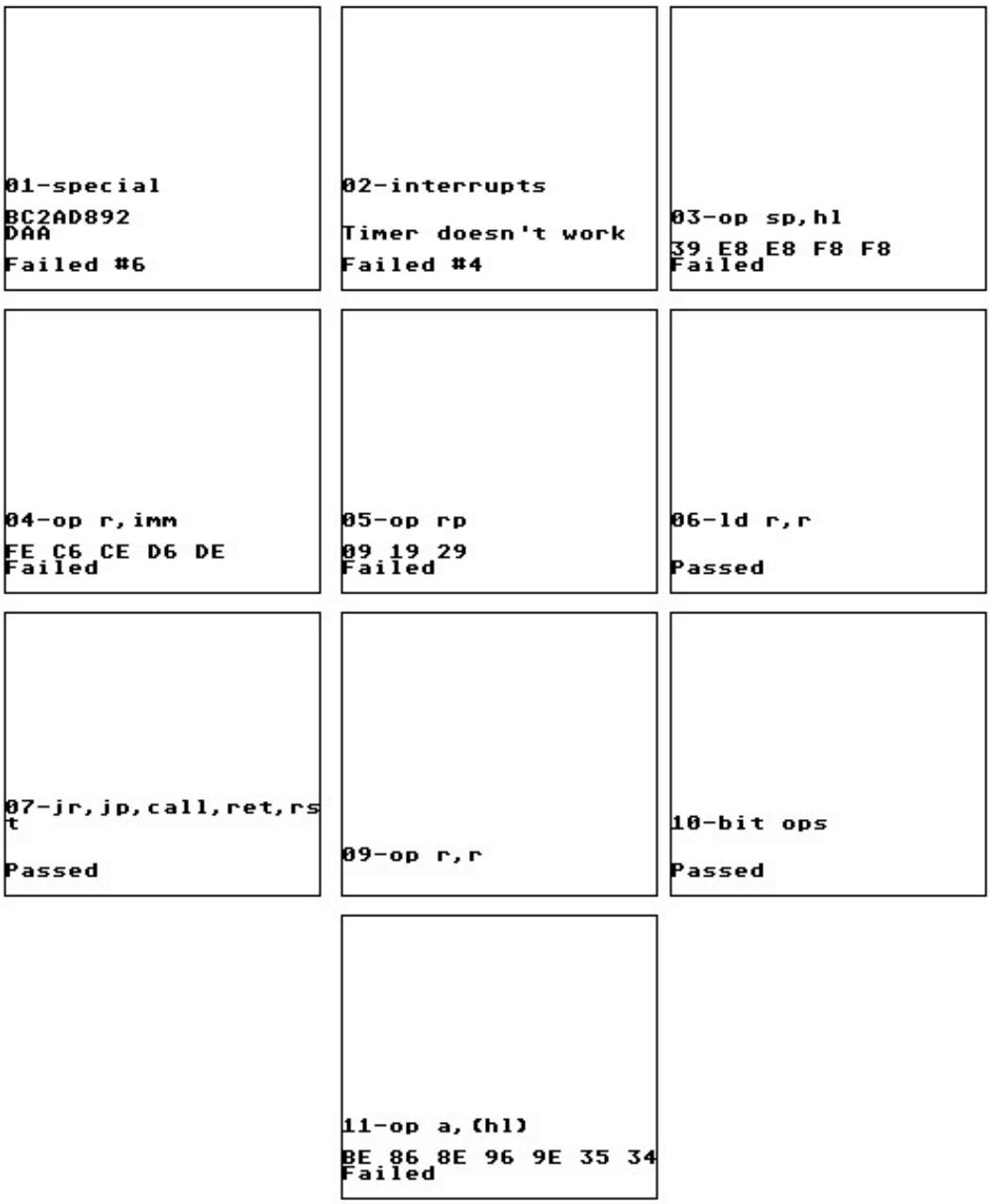


Figure 38: Results of Test ROMs