Kluskens Stef

Adapting a Game Boy emulator to play Dr. Mario

Supervisor: Tesch Tom

Coach: Defoort Stephanie

Graduation Work 2023-2024

Digital Arts and Entertainment

Howest.be

CONTENTS

ABSTRACT & KEY WORDS

For this thesis and grad work, I decided to do something with a Game Boy. More specifically, I took an existing Game Boy emulator[1] that only plays Tetris and adapt it so that it can play Dr. Mario as well. I discuss the relevant information about what I changed to make this work. I also go over the Game Boy's opcodes, more specifically the ones that were missing. These were left out by the person who originally made the emulator, with good reasons, because they weren't needed to play Tetris.

Voor deze thesis wou ik iets doen met de Game Boy. Ik had besloten om een bestaande Game Boy emulator[1] die enkel Tetris speelt, die gemaakt is door een vorige student, aan te passen zodat deze ook het spel Dr. Mario kan spelen. Ik bespreek de relevante informatie over wat ik heb veranderd om dit te laten werken. Ik bespreek ook de opcodes van de Game Boy, meer specifiek de opcodes die ontbraken. Deze waren weggelaten door de persoon die de emulator oorspronkelijk maakte, met goede redenen, omdat ze niet nodig waren om Tetris te spelen.

PREFACE

I chose this topic because I've been interested in emulating a retro console for a while. I chose the Game Boy for nostalgic reasons. The Game Boy is the console I used the most when I was a kid. It holds a special place in my heart and that's why I wanted to delve deeper into its inner workings by working on an emulator.

At first, I wanted to create a full emulator with memory bank controllers included, but that wouldn't have been possible in the given time frame. That's when I decided to work from an existing emulator[1], made by a previous student, and add memory bank controller functionality to that. This would allow the emulator to play a wider range of games.

During work on the MBC implementation, I noticed how much work it was trying to create the MBC and fixing the emulator to play more than just Tetris. So, in discussion with my supervisor, I changed my topic. That's how I got to adapting the existing emulator to play Dr. Mario as well as Tetris.

LIST OF FIGURES

INTRODUCTION

For this thesis, I always had it in my mind to do something with the Game Boy. This is something that I first thought of doing back when I had classes about the Game Boy and has stuck with me until now.

I first wanted to do a full emulator complete with memory bank controllers. This proved to be a bit optimistic given the time frame that I had. After talking with my supervisor, I landed on emulating just the memory bank controller and add that to an existing emulator[1]. The reason why I wanted to do the memory bank controllers, is because the games that I wanted to emulate all required the same MBC, the MBC1. But after working on trying to get this to work, I felt like I wasn't going to get this to work. So, in discussion with my supervisor, I decided to change the topic slightly and adapt the given emulator to have it play Dr. Mario, as well as Tetris.

That is how I landed on my research question: *What needs to change in order to get a Game Boy emulator, that only plays Tetris, to also play Dr. Mario?*

This is my main question for this thesis. I wanted to figure out what the problems were with the original emulator[1] that made it only play Tetris. I wanted to do this, because both games are pretty similar, Dr. Mario is a slightly adapted game from Tetris. But, more importantly, technically they are very similar. They have the same ROM size, both have 2 ROM banks and they don't have a memory bank controller.

With the research question set, I started working on the hypotheses. I landed on these hypotheses:

The missing opcodes will have to be added in order to have all of the instructions necessary to play Dr. Mario.

The existing functionality that handles the updating of the graphics will need to be changed to handle graphics drawing for Dr. Mario.

To come up with these hypotheses, I was thinking about what the possible issues could be with the original emulator[1]. Reading though Brecht's thesis[1], I found that he didn't implement all of the opcodes. I wasn't sure if Dr. Mario needed those instructions, but it was a possibility.
Next, I looked at the visuals of the game when played through the emulator. It was immediately clear that the background was messed up. The sprites were all over the place, creating a mixed-up frame. So, my second hypothesis was easy to form. The graphics code definitely needed to be fixed.

For the emulator, I was given the grad work and thesis of Brecht Uytterschaut[1], who made a Game Boy emulator which runs Tetris. Once I familiarized myself with the codebase I was given, I started on debugging and looking for where potential issues were hiding.

LITERATURE STUDY / THEORETICAL FRAMEWORK

GAME BOY EMULATION IN C++

For my literature review, I have to start with the thesis on Game Boy Emulation in C++ by Brecht Uytterschaut. This is the backbone of my work as I am continuing on his work by adapting his emulator to allow it to play the Dr. Mario game.
In his thesis, he explains how the Game Boy's CPU works and how he implemented it. He talks about the opcodes, the instructions, that the CPU has. He mentions that he didn't implement all of the instructions, only those that are needed to run Tetris, one of the few games that can run without an MBC.
He then talks about the PPU, the Picture Processing Unit, which handles the graphics. He explains how the different layers work. These are the background, the window or UI, and the sprites.
He finishes by shortly explaining how he implemented the input and how a memory bank controller works.

CPU INSTRUCTION SET

The chapter on the CPU instruction set from gbdev.io gives an overview of all of the instructions the CPU can do. This overview made it easier to find what instructions were missing from the emulator I got. This page also gives a short explanation and how many clock cycles the instruction takes to complete.

MEMORY MAP

The chapter on the memory map from gbdev.io talks about how the Game Boy's 16-bit address bus stores its data. It gives the address ranges for the cartridge header, external memory, echo RAM and I/O ranges. This was useful for the address rage for the timer variables.

THE CARTRIDGE HEADER

The cartridge header chapter from gbdev.io gives a great overview on what all the memory addresses in the cartridge header hold. This is where I could find where in memory the ROM and RAM data was stored. This page also tells you where to find all game specific data for setup, like the name, manufacturer, licensee code, and more.

THE GAME BOY OPCODE TABLE

The opcodes table was one of the most useful pieces of reference I used. The website from izik1 doesn't just have a static table, but every opcode is selectable. This opens a window that shows information about the opcode. It tells you what instruction is used, with the parameters and how each of the flags are set. It also gives the amount of clock cycles it takes for the operation to complete.

THE NOOBBOY

The NoobBoy is an open-source Game Boy emulator. This was an excellent reference to compare against, as this emulator is complete. It has all of the functionality an original Game Boy had.
I used this emulator towards the end of the work on the project to see how somebody else did things.

THE ULTIMATE GAME BOY TALK

The Ultimate Game Boy Talk is a must see for anybody interested in the Game Boy. When I started on this project, that was the first piece of reference I saved. I've watched this video multiple times as it gives a thorough explanation in the inner working of the Game Boy.

CASE STUDY

The test object of my grad work experiment will be a Game Boy emulator which I modified to play Dr. Mario, as well as Tetris. For the testing, I will use a test rom. These will run some test code in order to see how accurate the emulator is compared to the original Game Boy. I will test the emulator before and after adapting the emulator. This will give me 2 sets of data to compare.

1. INTRODUCTION

Starting on this project was pretty daunting. An emulator is a pretty complex piece of software where a lot of parts need to work together correctly in order to emulate a game.

After familiarizing myself with the code base and looking at the differences in how Tetris and Dr. Mario worked through another emulator, I had a pretty good idea on where to start.

Because the problem was mostly graphical, I instantly knew if what I had changed in the code made any difference to running the game. With the image below as a starting point, it couldn't get much worse.

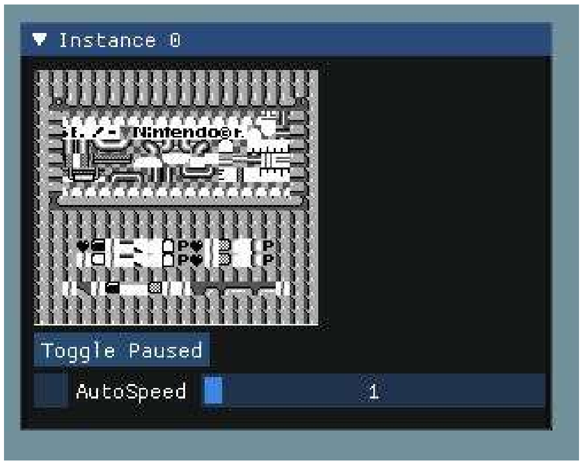


Figure 1: Dr. Mario's start screen in the original emulator.

2. OPCODES

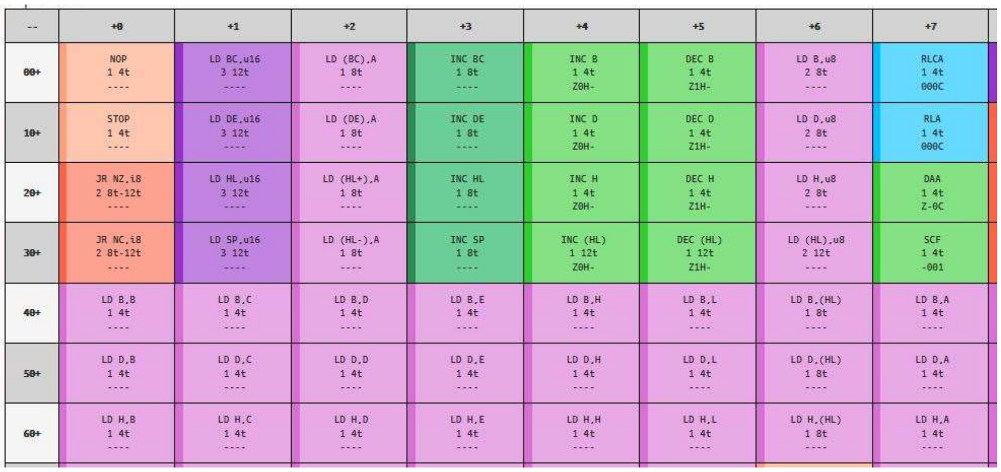
The opcodes of a Game Boy are the instruction that the CPU can perform. It has 2 sets of instructions, the main instruction set and the extended instruction set.

The main instruction set has 245 instructions that the CPU can execute. The extended instruction set has another 256 instructions. This brings the total amount of instructions to 501.

All of the opcodes are fully explained in the CPU manual, and numerous websites have opcode tables for easy referencing. I used an interactive table, hosted on github[5]. This table allow you to see what address an opcode has and what the instruction behind it does.

When reading through the thesis of Brecht[1], I found that he didn't implement all of the instructions. Not sure which instructions were missing, I took a table of opcodes and started going over all of them and fill in the missing instructions.

Eventually I found that there were 3 instructions missing. The RL, RRC and SRA.



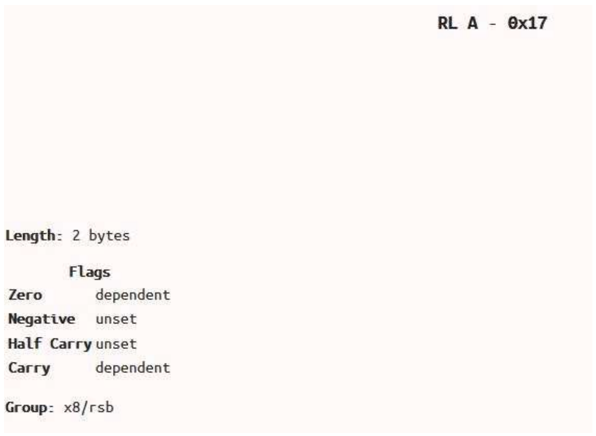
--	+0	+1	+2	+3	+4	+5	+6	+7
00+	NOP 1 4t ----	LD BC,u16 3 12t ----	LD (BC),A 1 8t ----	INC BC 1 8t ----	INC B 1 4t Z0H-	DEC B 1 4t Z1H-	LD B,u8 2 8t ----	RLCA 1 4t 000C
10+	STOP 1 4t ----	LD DE,u16 3 12t ----	LD (DE),A 1 8t ----	INC DE 1 8t ----	INC D 1 4t Z0H-	DEC D 1 4t Z1H-	LD D,u8 2 8t ----	RLA 1 4t 000C
20+	JR NZ,i8 2 8t-12t ----	LD HL,u16 3 12t ----	LD (HL+),A 1 8t ----	INC HL 1 8t ----	INC H 1 4t Z0H-	DEC H 1 4t Z1H-	LD H,u8 2 8t ----	DAA 1 4t Z-0C
30+	JR NC,i8 2 8t-12t ----	LD SP,u16 3 12t ----	LD (HL-),A 1 8t ----	INC SP 1 8t ----	INC (HL) 1 12t Z0H-	DEC (HL) 1 12t Z1H-	LD (HL),u8 2 12t ----	SCF 1 4t -001
40+	LD B,B 1 4t ----	LD B,C 1 4t ----	LD B,D 1 4t ----	LD B,E 1 4t ----	LD B,H 1 4t ----	LD B,L 1 4t ----	LD B,(HL) 1 8t ----	LD B,A 1 4t ----
50+	LD D,B 1 4t ----	LD D,C 1 4t ----	LD D,D 1 4t ----	LD D,E 1 4t ----	LD D,H 1 4t ----	LD D,L 1 4t ----	LD D,(HL) 1 8t ----	LD D,A 1 4t ----
60+	LD H,B 1 4t ----	LD H,C 1 4t ----	LD H,D 1 4t ----	LD H,E 1 4t ----	LD H,H 1 4t ----	LD H,L 1 4t ----	LD H,(HL) 1 8t ----	LD H,A 1 4t ----

Figure 2: Section of the opcode table.

2.1 RL

RL, or rotate left through carry, is an operation that shifts the bits of a given register one position to the left. As it says in the name, the carry flag is involved. Because the bits are all shifted to the left, there is 1 bit that gets shifted out. This bit gets put into the carry flag. The contents of the carry flag before the shift are put into the least significant bit, LSB, of the target register.

During this operation, the carry flag gets a new value. The other flags are also set when the RL instruction is called. The zero flag gets set if the result of the shifting is 0. The subtract flag is reset, because there is no subtracting during this operation. The half-carry flag is also reset, this is mainly to emphasize that this flag isn't modified during this operation.
This instruction takes either 8 or 16 clock cycles. This depends on the register that needs to shift. If it's a single register, like register A, the it takes 8 cycles. If it's a larger register, like register HL, then it takes 16 cycles[2].



RL A - 0x17

Length: 2 bytes

	Flags
Zero	dependent
Negative	unset
Half Carry	unset
Carry	dependent

Group: x8/rsb

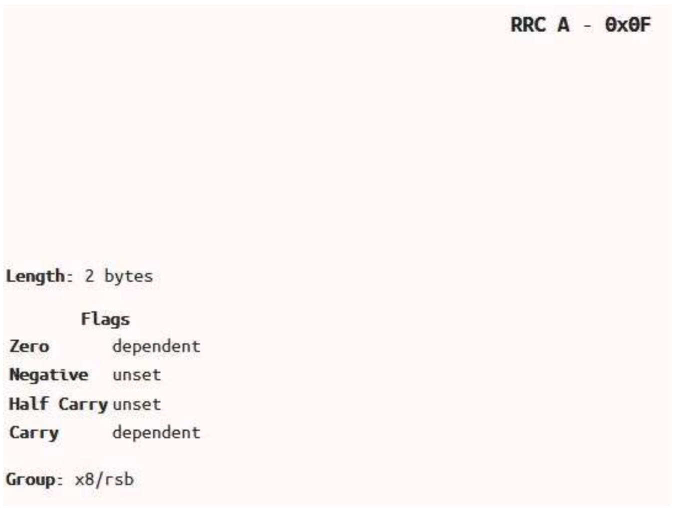
Figure 3: The RL explanation from the opcode table from [5].

2.2 RRC

RRC, or rotate right through carry, is an operation that shifts the bits of a given register one position to the right. Like with the RL instruction, the RRC also uses the carry flag. The bit that gets shifted out on the right side is placed both into the carry flag and into the lest significant bit, the LSB, of the target register.

During this operation, the carry flag gets a new value. The other flags are also set when the RL instruction is called. The zero flag gets set if the result of the shifting is 0. The subtract flag is reset, because there is no subtracting during this operation. The half-carry flag is also reset, this is mainly to emphasize that this flag isn't modified during this operation.

This instruction takes either 8 or 16 clock cycles. This depends on the register that needs to shift. If it's a single register, like register A, the it takes 8 cycles. If it's a larger register, like register HL, then it takes 16 cycles[2].


RRC A - 0x0F

Length: 2 bytes

Flags	
Zero	dependent
Negative	unset
Half Carry	unset
Carry	dependent

Group: x8/rsb

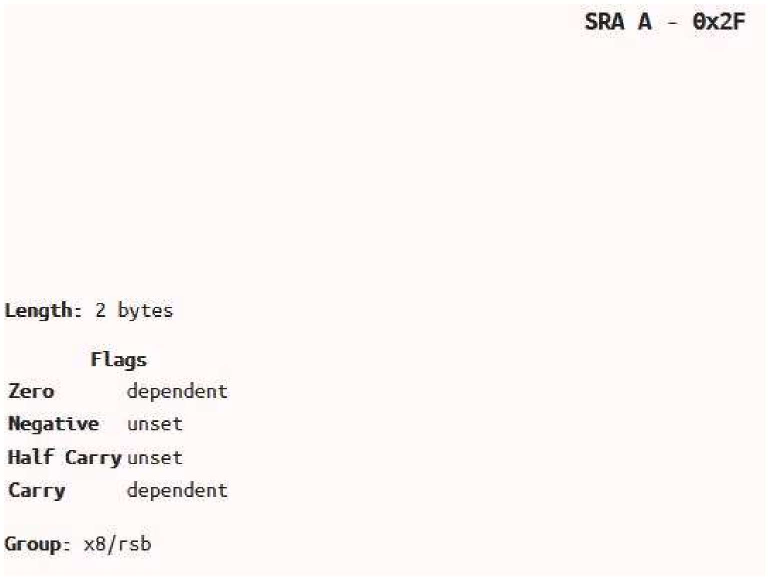
Figure 4: The RRC explanation from the opcode table from [5].

2.3 SRA

SRA, or shift right into carry, is an operation that shifts the bits of a given register one position to the right. Because this is an arithmetic shift, the most significant bit, the MSB, is duplicated into the vacated bit spot. This means that the MSB doesn't change when execution this operation. The MSB is the sign bit in signed numbers, so this operation is useful for signed number manipulation, as it preserves the sign bit. The bit that got shifted out, is put into the carry flag.

The other flags get the following values during an SRA operation. The zero flag gets set if the result of the shifting is 0. The subtract flag is reset, because there is no subtracting during this operation. The half-carry flag is also reset, this is mainly to emphasize that this flag isn't modified during this operation.

This instruction takes either 8 or 16 clock cycles. This depends on the register that needs to shift. If it's a single register, like register A, the it takes 8 cycles. If it's a larger register, like register HL, then it takes 16 cycles[2].



SRA A - 0x2F

Length: 2 bytes

Flags	
Zero	dependent
Negative	unset
Half Carry	unset
Carry	dependent

Group: x8/rsb

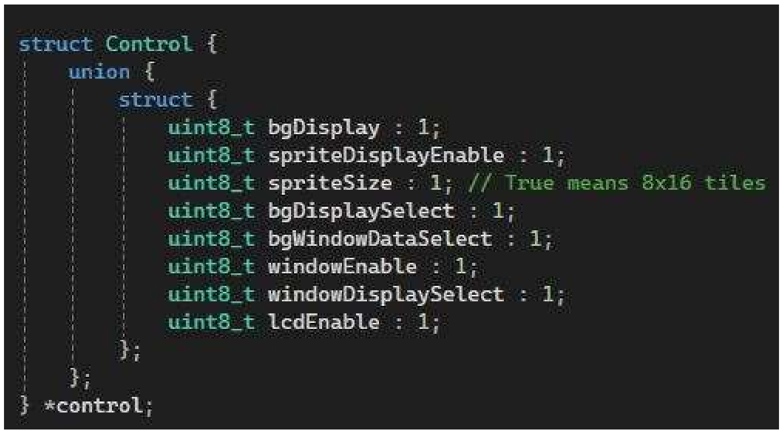
Figure 5: The SRA explanation from the opcode table from [5].

3. GRAPHICS

Adapting the graphics code was a big task. This code had some of the biggest issues in order to run Dr. Mario. At every point where a register needed to be read from memory, there was a call to the Read function. I changed these to be a pointer variable, so it's clear what is being read from memory. The same goes from writing.

I also added some structs to keep data that is related to each other together. These struct I got from the reference[6] I found. These structs made everything clearer.

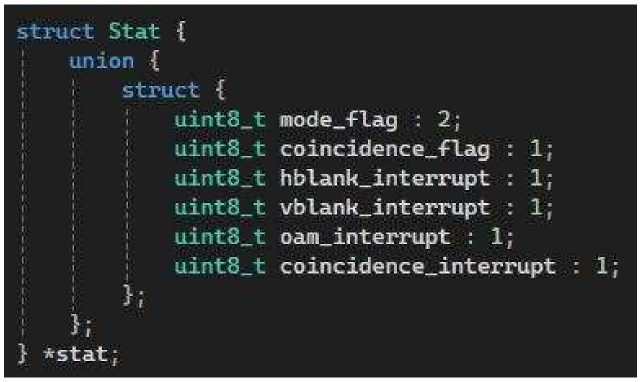
First there is the struct for the LCD control. The LCDC is used to configure how the graphics are displayed on the screen. It holds if the display is enabled, if the window or background tile map needs to be selected, if the window is enabled, if it needs to select the window or background tile data, and the sprite size.



```
struct Control {
    union {
        struct {
            uint8_t bgDisplay : 1;
            uint8_t spriteDisplayEnable : 1;
            uint8_t spriteSize : 1; // True means 8x16 tiles
            uint8_t bgDisplaySelect : 1;
            uint8_t bgWindowDataSelect : 1;
            uint8_t windowEnable : 1;
            uint8_t windowDisplaySelect : 1;
            uint8_t lcdEnable : 1;
        };
    };
} *control;
```

Figure 6: The LCD Control struct.

Next there is the struct for the LCD stat. The LCDS works together with the LY and LYC registers to synchronize the LCDC's timing. It also handles events related to the different display modes and scanlines. It holds the coincidence interrupt, the interrupts for the different display modes, the coincidence flag and the display mode flag.



```
struct Stat {
    union {
        struct {
            uint8_t mode_flag : 2;
            uint8_t coincidence_flag : 1;
            uint8_t hblank_interrupt : 1;
            uint8_t vblank_interrupt : 1;
            uint8_t oam_interrupt : 1;
            uint8_t coincidence_interrupt : 1;
        };
    };
} *stat;
```

Figure 7: the LCD Stat struct.

3.1 UPDATE GRAPHICS

The first thing I tested, was if the graphics function that is called in the update loop, HandleGraphics, was implemented correctly.

This function is should be switching the graphic modes. There are 4 modes available: H-Blank, V-Blank, OAM and VRAM.

H-Blank, or horizontal blank, is a process that occurs when the pixel drawing has reached the end of the scanline and needs to reposition to the start of the next one. During this period, the CPU executes some tasks including updating sprite information, modifying background and tile data, and performing Direct Memory Access, DMA, transfers.

V-Blank, or vertical blank, is similar to H-Blank, but instead of when the pixel drawing is at the end of a horizontal scanline, V-Blank occurs when the pixel drawing is at the end of the screen and needs to travel back up. During this small break, the CPU has the time to execute some tasks like updating game logic, handling input and modifying graphics data.

OAM, or object attribute memory, is a section of memory that stores information about sprites, such as the position, size and attributes. During the OAM mode, the graphics hardware, the PPU, retrieves the data from OAM to render the sprites on the screen. OAM DMA transfers are common to happen during the H-Blank phase.

VRAM, or video RAM, stores background and tile data. During VRAM mode, the PPU retrieves data from VRAM to render the background and tiles. During this mode, the CPU can update background and tile information without causing visual bugs.

In the original function, the LCD status was updated in a different way than what I found in reference. The function was also split up in 2, the main HandleGraphics function and a function to configure the LCD. I changed this to be 1 function dealing with the configuring, as well as calling the Draw function.

The Draw function itself, calls the 3 functions to draw the background, the window and the sprites.

3.2 BACKGROUND DRAWING

In the original emulator, the function starts with an assert to make sure that the LCD is enabled. If the LCD is not enabled, then there shouldn't be any background drawing. This is to make sure that this doesn't happen.

All of the required variables are retrieved from memory or calculated. These include, the tile map address, the horizontal and vertical scroll positions, the wrap-around positions.

With these values, the offset for the tile map and framebuffer can be calculated. These offsets are used during the rendering to determine where to read the date from the tile map and where to write the data in the framebuffer. This makes sure that the correct tiles are used to get a correct frame.

Next, the function configures the correct colours. The colour palette is read from memory and is used to fill the colour array, which are of type uint8_t.

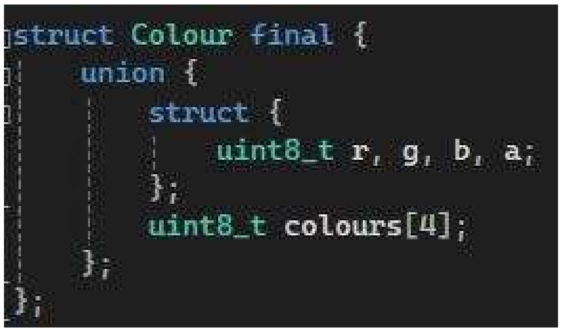
The framebuffer itself is retrieved as a reference from the main Game Boy class. The framebuffer is set up as a bitset of size (160 * 144 * 2). The values used to calculate the size are the screen width, screen height and the 2 is because every pixel in the framebuffer uses 2 bits.

The function then loops over each pixel in the width of the screen, from 0 to 160. In this loop, the function retrieves the correct tile data and uses it to calculate the palette information. Using the tile data with the palette information, the framebuffer got filled in with pixel data to create the background.

This function worked correctly when running Tetris, but when I tried to run Dr. Mario, the background was completely messed up. So, I took some reference and adapted this function to try and get a correct background drawn on screen.

In my adapted function, I made use of the Control and Stat structs that I made. The first difference in the function, is how I get the tile map address. I now use a flag in the Control struct, this flag determines from what range in memory the tile map display needs to be read from.

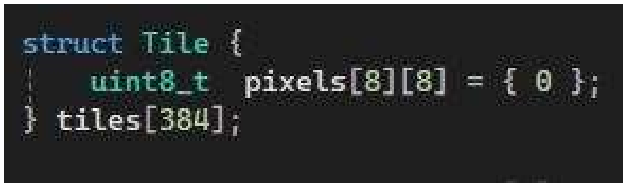
I also made some other helper structs, to keep relative data together. These were also done according to my reference[6], after seeing that it would help the emulator. I first made one for the colour data. This is the simplest struct I made, as it just holds the 4 values for a colour and a simple array to hold the 4 values.



```
struct Colour final {
    union {
        struct {
            uint8_t r, g, b, a;
        };
        uint8_t colours[4];
    };
};
```

Figure 8: The Colour struct.

Next, I added a struct for the tiles. Since the background is made up of separate tiles, it makes sense that there is a tile variable.



```
struct Tile {
    uint8_t  pixels[8][8] = { 0 };
} tiles[384];
```

Figure 9: The Tile struct.

With the background code fixed, I had a very big jump forward in order to play Dr. Mario.

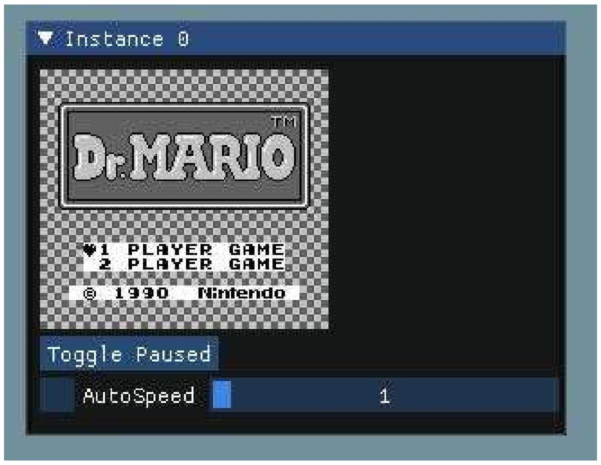


Figure 10: Dr. Mario's start screen after fixing the graphics code.

3.3 WINDOW DRAWING

The window drawing in the original emulator did not work at all. I only found this towards the end, because both Tetris and Dr. Mario don't use a window for their UI. For these games, the UI is part of the background. I only found out about this when, out of interest, I tried emulating a different game.

The original window drawing code was almost exactly the same as the original background drawing code. Since the original background code had issues, I could make a reasonable guess as to where the issues could be with the window drawing.

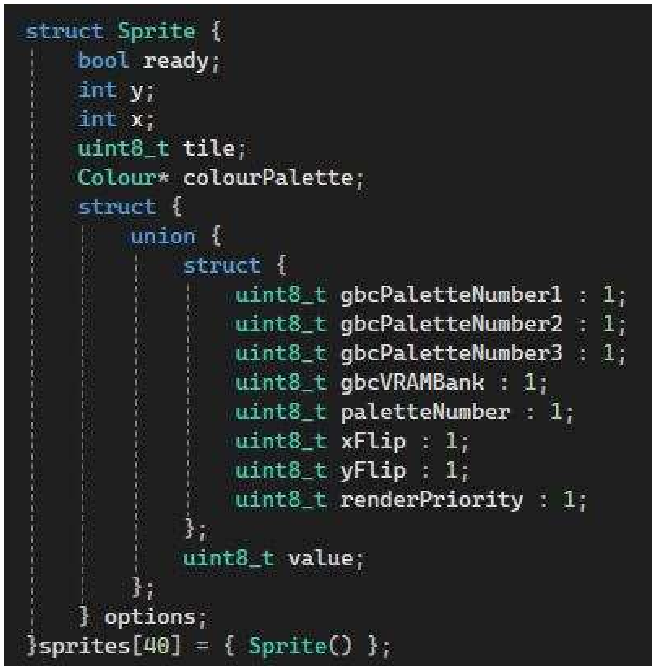
The fact that the 2 functions were almost the same is normal, because the background and window should be drawn in almost the same way. So, my new window drawing code looks similar to the new background drawing code.

Since this part of the graphics wasn't that important for my research, I didn't fully go into testing it. I wanted to spend more time on more important tasks to make Dr. Mario play correctly.

3.4 SPRITE DRAWING

The sprite drawing changed in a similar fashion to the background and window drawing code. I used my reference[6] to compare what was different and made changes where I felt they would help in fix the emulator.

Like with the background changes, I also added a struct to help with the sprite drawing. In line with my reference[6], I created a Sprite struct that holds all of the information of a sprite.



```
struct Sprite {
    bool ready;
    int y;
    int x;
    uint8_t tile;
    Colour* colourPalette;
    struct {
        union {
            struct {
                uint8_t gbcPaletteNumber1 : 1;
                uint8_t gbcPaletteNumber2 : 1;
                uint8_t gbcPaletteNumber3 : 1;
                uint8_t gbcVRAMBank : 1;
                uint8_t paletteNumber : 1;
                uint8_t xFlip : 1;
                uint8_t yFlip : 1;
                uint8_t renderPriority : 1;
            };
            uint8_t value;
        };
    } options;
}sprites[40] = { Sprite() };
```

Figure 11: The Sprite struct.

With the information in this struct, it was easier to adapt the sprite rendering code. Not much actually changed in this function, since most of it already worked. The biggest difference is the use of this struct to hold the data, instead of multiple read and write calls to memory.

4. TIMER

While debugging the emulator, in order to play games other than Tetris, I came across the timer function. When comparing this timer function to the reference[6] I found, I noticed some big differences.

4.1 DIV

The first difference I noticed was how the DIV cycles and DIV timer were updated.

The DIV cycles represent the passage of time within the system. They are used for certain tasks like creating time delays and generating random numbers. They are also used to implement time-dependent events.

In the original emulator, it was updated by using a cycles budget. This budget was than divided by 16.384, which is the number of times the DIV register increments per second. This is to create a cycles budget per DIV. This value was then used to compare the current DIV cycles too and increase the DIV timer if needed.
I changed this to not use the cycles budget. Instead, I immediately increase the DIV cycles with the current step cycles and compare that to a fixed value. If that comparison triggers, I update the DIV cycles and timer.

4.2 TAC & TIMA

Next, were the TAC timer, the TIMA cycles and the TIMA timer.

In the original emulator, the TIMA cycles got incremented by the step cycles. After that, a threshold was calculated based on the cycle budget. That threshold was then used to compare against the TIMA cycles. As long as the TIMA cycles were larger than the threshold, the TIMA cycles and timer were updated.

I changed this according to my reference[6]. I still add the step cycles to the TIMA cycles, before calculating a threshold. I don't use the cycles budget for this, I set the threshold to a value based on the case in the same switch statement as the original emulator.

I then do the same while loop functionality except for the setting of the TIMA timer. I read a certain memory address[3] for that, instead of setting it to the value of the TMA timer.

DISCUSSION

In updating the graphics, especially the updating of the LCD registers was the main part of fixing the issues this emulator had with running Dr. Mario correctly. Setting the correct bits made is so that the correct elements are displayed on the screen in the right place at the right time.
Mostly, it were the bits related to the background and window that gave issues. The important bits in question were bit 3 – 6. These buts control the window enabling and tile map area, and the background tile data and map area.

For the timer, the use of the cycle budget made it so that the threshold was incorrect, leading to an incorrect setting of the TIMA timer. This then messed up the flow of Dr. Mario, but apparently not for Tetris. This is something that’s probably a fluke.

CONCLUSION

HYPOTHESES

Hypothesis 1: *The missing opcodes will have to be added in order to have all of the instructions necessary to play Dr. Mario.*

I implemented the 3 missing opcodes, and calling them from the ExecuteOpcode function. While testing the Dr. Mario game, I set a breakpoint on those instruction calls. None of those breakpoints got hit, meaning that the instructions were never called.
Looking back, this doesn't surprise me. Because the issue was mainly a graphical one, implementing missing opcodes was unlikely to fix the issues with the emulator. This is because the PPU, the pixel processing unit, in the Game Boy works by writing to different registers than the CPU. The fact that this didn't help in fixing the issues was predictable.
It was, however, not a waste of time. It could have been that there were opcodes missing that Dr. Mario uses, but Tetris doesn't. It also increases the accuracy in which the emulator emulates the original Game Boy.

Hypothesis 2: *The existing functionality that handles the updating of the graphics will need to be changed to handle graphics drawing for Dr. Mario.*

This was the main chunk of the issue the emulator had when trying to run Dr. Mario. The image that I got for the main menu was completely wrong. The sprites that were drawn were in the wrong place and the Nintendo logo showed through. This is something that shouldn't have happened, because the emulator skips the boot rom in order to avoid the security the Game Boy games had built in.
Both the LCD control and stat register were being updated incorrectly. This made the graphics update incorrectly, creating a wrong frame.
I also noticed that the window drawing didn't work at all. This didn't impact the emulation for Dr. Mario at all, but out of interest, I tested another game and there I noticed that the window did not draw at all. Fixing this was a nice extra when I was working on the graphics anyway.

UNEXPECTED ISSUE

An unexpected issue I found was how the timer got updated.
I already fixed the background issue, so I had the correct image rendered to the screen. But, while the input was responding, it didn't respond correctly at certain points. For example, on the start screen, I could move the selection up and down to select the number of players. But when I tried to move on and start the actual game, it would glitch and not move on.

I first debugged the input, but that was all responding correctly, so I was quite sure it wasn't that. I eventually found that it was the timer being updated incorrectly. Fixing the timer handling code after fixing the graphics code, lead to the emulator being able to run Dr. Mario without any issues.

RESEARCH QUESTION

What needs to change in order to get a Game Boy emulator, that only plays Tetris, to also play Dr. Mario?

My main question for this thesis. What did I need to change to the original emulator to make it play Dr. Mario?

As I expected when I made my hypotheses, the graphics code needed to change. This was clear when I tried to run the game through the emulator and the background was scrambled up. It wasn't just the way the emulator updated the graphics, the separate functions that render the actual sprites needed changing as well. I think the most important change in the graphics code was how the LCD registers were updated. Having these now update correctly will have improved the emulator's accuracy a lot.

It turned out that the timer was also an issue. This surprised me because it could run Tetris. This means that the original timer handling code was made in a way that it only correctly updated Tetris. Because the original emulator was only made to run Tetris and not other games, it probably wasn't tested with other games. So, mistakes like this weren't noticed.

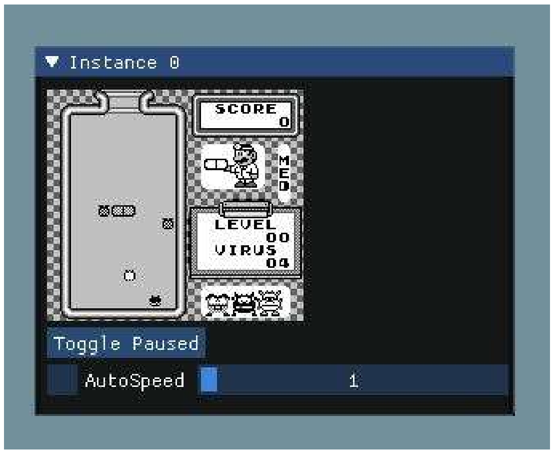


Figure 12: Gameplay from Dr. Mario after fixing the graphics and timer code

FUTURE WORK

For future work on this emulator, there are a couple of things to improve the project.

MBCs:
The emulator doesn’t have working MBCs. MBCs allow the Game Boy to play larger games. This would allow the emulator to play a larger range of games, such as Batman or Super Mario Land.

Sound:
I did not have the time to implement sound on top of the MBC1. Adding sound would be interesting as it would be nice to hear those nice 8-bit sounds being played again.

Link connection:
The original Game Boy had the option of connecting 2 Game Boy’s with a link cable. This allowed for multiplayer between 2 consoles. It would be an interesting challenge to try and emulate that connection with 2 instances of the emulator.

CRITICAL REFLECTION

Working on this thesis and project made me appreciate even more all of the effort that goes into the emulation of retro consoles and games. The attention to detail that goes into even thinking about creating a piece of software like this is incredible and not easily achieved.

Personally, as someone who stresses easily, I learned a lot about pacing myself. When I started fully focussing on this project, I spend more than 12 hours a day trying to make it work. Naturally, it didn't take long before that burnt me out. This then led to more stress and a feeling of doom started to nest itself into my mind. That's when I stopped work for a day, to have a reset. After this, and some helpful insights from my supervisor, I was ready to get back to work, this time without overworking myself.

Now, at the end of the project, I am glad I took that day off to reset. It helped in restoring my mindset towards this project and, ultimately, helped me reach my goals for this project.

Looking back, I am happy that I chose to go down the road of Game Boy emulation. Even though I found it very difficult, I think it helped me, as an aspiring programmer looking to start his game development journey, that even though it's difficult, it's doable.

REFERENCES

1. Uytterschaut B. Gameboy Emulation In C++. Howest; 2019.

2. gbdev. CPU Instruction Set. Published May 28, 2021. https://gbdev.io/pandocs/CPU_Instruction_Set.html

3. gbdev. Memory Map. Published May 28, 2021. https://gbdev.io/pandocs/Memory_Map.html

4. gbdev. The Cartridge Header. Published May 28, 2021. https://gbdev.io/pandocs/The_Cartridge_Header.html

5. izik1. The Game Boy Opcodes Table. https://izik1.github.io/gbops/

6. Mika412. NoobBoy - GameBoy Emulator. Github. https://github.com/Mika412/NoobBoy/tree/master

7. The Ultimate Game Boy Talk.; 2016.
https://www.youtube.com/watch?v=HyzD8pNlpwI&ab_channel=media.ccc.de

ACKNOWLEDGEMENTS

Firstly, I would like to thank my supervisor, Tom Tesch, for guiding me through the process of this grad work. His insight into the working of the Game Boy has been a great source of help.

Secondly, I would like to thank my coach, Stephanie Defoort, for motivating and advising me during our meetings.

Last, I want to thank the people around me for putting up with me during this stressful time.

APPENDICES

The result of the adapted emulator can be found on my github page:

https://github.com/StefKluskens/GameBoyEmulator