UNIVERSITY OF COPENHAGEN

Investigating the Game Boy Cartridge

PyBoyCartridge

Mads Ynddal
SJT402

August 22, 2016

Abstract

Following the PyBoy project[7], which covered emulation of the Nintendo Game Boy, this project continues the development from another perspective.

Emulation is the process of mimicking an existing system. To achieve a good or perfect emulation, one has to probe the existing system to find details, which might not have been documented. This could be an odd behavior, which has become part of the system, but not from the intention of the original design.

This project takes on the aspect of physically interacting with a Game Boy, to advance the development of the emulator. The project will first focus dumping of game cartridges, by connecting it to a microcontroller and sequentially ask for each byte it stores. Afterwards, the process gets turned around to connect a microcontroller to the Game Boy as if it was a cartridge. This will open the possibility of running test code on the Game Boy to pinpoint the undocumented behavior.

Forewords

This is the further development of the PyBoy project, which I (Mads Ynddal), Troels Ynddal and Asger Lund Hansen wrote in the first semester of the school year 2015/2016. This report will have several references to the PyBoy report. It is therefore recommended, that the reader has already read the report from the PyBoy project, or has it at hand while reading this report.

Contents

1 Introduction **5**
- 1.1 Point of entry 5
- 1.2 Part one: Dumping cartridges 5
- 1.3 Part two: Emulating a physical cartridge 6
- 1.4 Future work 6

2 The Hardware **6**
- 2.1 Game Boys for testing 6
- 2.2 Microcontroller or FPGA . . 6

3 Game Boy Cartridges **7**
- 3.1 Pin layout 7
- 3.2 ROM and RAM banks 7
- 3.3 Memory Bank Controller . . 8
  - 3.3.1 Types 8

4 Dumping the cartridge **8**
- 4.1 Wiring to the Arduino 9
- 4.2 Cartridge headers 10
- 4.3 Dumping the first bank . . . 10
- 4.4 Changing banks 10
- 4.5 Dumping RAM 11
- 4.6 Verifying 11

5 Emulating the cartridge **11**
- 5.1 Dummy cartridge 12
- 5.2 Boot-ROM 12
- 5.3 Checking the data bus 13
- 5.4 Detecting the clock 13
- 5.5 Changing data on clock cycles 14
- 5.6 Reading the address 14
- 5.7 Deliver the Nintendo logo . . 15
- 5.8 Improving the timing 16
  - 5.8.1 Measuring the timing 17
  - 5.8.2 Interrupt controlled . 17
  - 5.8.3 Optimizing compiled code 18
  - 5.8.4 Moving port to optimize 18
  - 5.8.5 Testing the new routine 19
- 5.9 Future work 20

6 Conclusion **20**

Appendices **22**

A Arduino Mega **22**

Category
Hardware emulation

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

1 Introduction

After the PyBoy[7] project, I wanted to look further into the Game Boy hardware. The development of the emulator was mainly based on existing documentation and testing on existing emulators. To be able to source the information myself, I will need access to the actual hardware. The main goal being to execute my own code on the Game Boy hardware and return a result in some way or another.

1.1 Point of entry

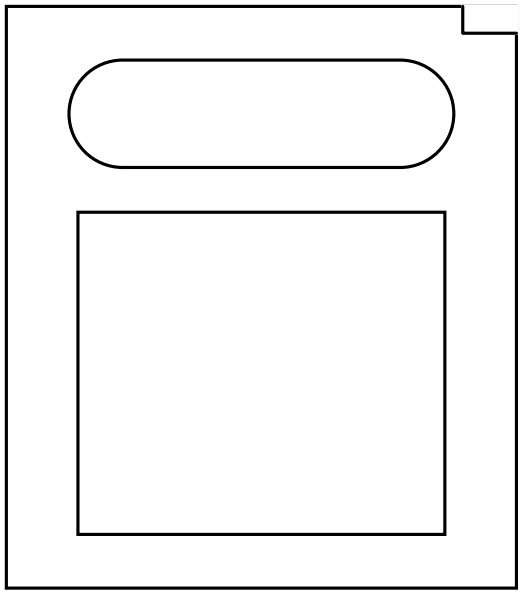

Figure 1: Game cartridge is inserted on the back before starting the Game Boy

So, how can the Game Boy be accessed for code execution? The Game Boy has two ways of communication. The primary is the game cartridge (see figure 1). Except for the boot-ROM; the cartridge holds all the software, that the Game Boy executes. If one were to create a custom cartridge, it would be possible to execute any code, as the software on the cartridge has full hardware access – as the Game Boy has no operating system.

Figure 2: Link Cable is used to interconnect two Game Boys

The second way of communications, is the serial port called a Link Cable. This port is used for games that have a multiplayer element. Even though it is not meant for tranferring code, it might be possible to find a bug in an existing game, that enables execution of arbitrary code. I've chosen to not use this method, as emulating a cartridge will allow us to even change interrupt-vectors, which are normally hardcoded on the cartridge ROM.

I have chosen the first option, as this will not depend on an existing cartridge, and we will also have more flexibility towards emulation, as the Link Cable method will be limited to code that can completely reside in RAM. This would rule out changing the interrupt routines, as they reside on the first ROM bank of the cartridge.

1.2 Part one: Dumping cartridges

The main goal is to emulate a physical cartridge on the Game Boy. But as an introduction to the problems I might face, I'll start off by copying the data of a cartridge onto my computer. The information I get from this, will be useful for the next part.

The games that gets dumped will also be useful for the PyBoy project, as this is the way to load games into the emulator. The dump will create a perfect one-to-one copy of the cartridge, and together with the PyBoy emulator, will form an almost complete replacement for a Game Boy.

1.3 Part two: Emulating a physical cartridge

Based on the experience and data from dumping the cartridges, I will proceed to emulate a cartridge for the physical Game Boy and thereby be able to run arbitrary code on the actual Game Boy hardware. This will enable me to create tests to execute on the hardware and determine ways to emulate its behavior correctly.

1.4 Future work

This project will not be improving the PyBoy directly. The project will only make it a possibility to access the hardware for the development of the emulator.

An example of what might be tested, is the DAA (Decimal Adjust after Addition), which proved to be problematic to implement because of different source's varying descriptions[7, Appendix H]. If we executed a test on the Game Boy, we could once and for all determine how the hardware would act.

Apart from running explicit tests, it would be a possibility to run a game on the hardware and emulator at the same time and inject code to dump the state of both machines. In case the two machines changes state in two different directions, I will be able to pin-point the operation, which caused it, and fix the emulation accordingly – as even faulty hardware has to be implemented in an emulation[7, Emulation].

2 The Hardware

The Nintendo Game Boy is a handheld gaming console. It was first released in 1989, but has during the '90 and early '00 seen several successors. The early successors all had complete backwards-compatibility with the games for the original DMG-01 Game Boy from '89.

2.1 Game Boys for testing

For this project, I will use a Game Boy Advance and a Game Boy Color. Neither of which are the DMG-01 Game Boy, which the PyBoy is trying to emulate, but this shouldn't become an issue.

The Game Boy Color is mostly a second revision of the original DMG Game Boy. It's processor has exactly the same instruction-set and features, it just has the possiblity of running at the double clock-speed[5, CGB Registers]. It also has a color screen and an extra bank of VRAM, which the programmer can utilize if they *want* to. This means it is effectively a "superset" of the original DMG-01 and is 100% backwards-compatible with the original DMG-01. The only things to watch out for, is an updated boot-ROM and some glitches of the old hardware, which won't be present on the new version – for example a memory scramble bug[5, Sprite RAM Bug].

The Game Boy Advance is effectively a completely different platform compared to the two older Game Boy models, which had alot in common on hardware level. But what is interresting, is the Game Boy Advance still is backwards-compatible with previous generations. Although the Game Boy Advance has an ARM CPU, it also has a complete Game Boy Color on the same CPU die[4]! The Game Boy Color mode is engaged by a small button inside the cartridge tray. The Game Boy Advance cartridges don't push this button, because they has an equivilant indentation. For the cartridge emulation part of the project, I will therefore use a first-generation cartridge to engage this button.

2.2 Microcontroller or FPGA

To interface with the Game Boy, I started to look at using an FPGA. On the prolist, it can achieve high clock speeds and maybe store a complete game in-memory.

Although, after looking into the use of it, it seemed too demanding for the size of the project, as it would require a lot of time to learn to use the FPGA and I likely wouldn't have time for more than the first part of the project.

I chose to use an Arduino Mega. It has enough I/O pins to not require shift registers or other I/O expanders. The clock-speed is quite limited, as it can "only" acheive 16Mhz. The Game Boy runs at 4Mhz with 4 clock cycles per instruction. This means, that I have a maximum of 16 clock cycles on the Arduino for each intruction on the Game Boy. The Arduino also has a very limited amount of the memory. With 8KB of RAM and 256KB of Flash memory, only some games will be possible to be load onto the Arduino. To overcome this, I might look into a way of stalling the Game Boy while paging in data from a PC over a serial connection.

3 Game Boy Cartridges

Before we begin on the two parts of this project, I'll go through what these cartridges actually are.

The Game Boy cartridge is the only way to load software onto the Game Boy. The cartridge is a small plastic box of rougly $5,7 \times 6,5 \times 0,8$ cm. The box has a 32-pin connector on the bottom, which is the bottom part of the circuit board of the electronics inside.

The internals of the cartridge can vary greatly. Excluding cartridges with hardware extension; most cartridges has a Memory Bank Controller (MBC), a ROM chip, and optionally a RAM chip.

When the Game Boy is connected to the cartridge, the Game Boy will access these pins and communicate with the hardware inside the cartridge directly. As the Game Boy has no operating system, it has no awareness of how the cartridge works – that's up to the developers of the cartridge to decide. The only requirement is for the cartridge to have code at the entry point on address 100_{16}, as this is where the Game Boy starts to execute after leaving the boot-ROM.

3.1 Pin layout

As said, the cartridge uses a 32-pin connector for the Game Boy to interface with. The connector itself is a proprietary standard, and is used across all the Game Boy generations. Although the purpose of each pin changes between the first and later Game Boy Advance generation (see appendix A).

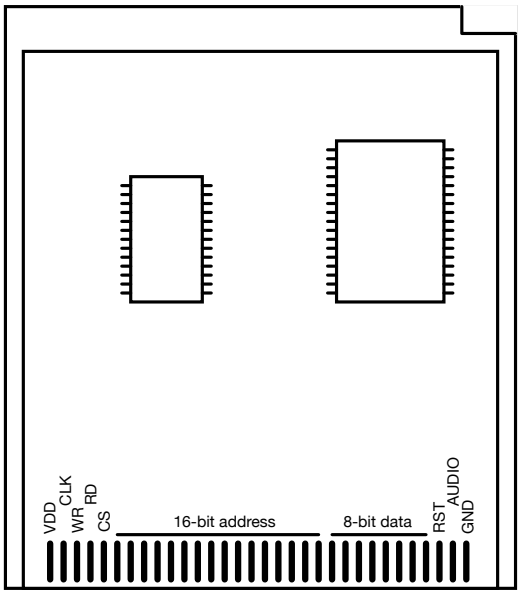

Figure 3: Illustration of pin layout of game cartridge

From the illustration above, the main interest is the 16-bit address bus and the 8-bit data bus. The `WR`, `RD` and `CS` pins are used respectively to control read, write and whether to direct the read/write operation to the RAM or ROM.

3.2 ROM and RAM banks

Eventhough the address bus is a mere 16-bit, some games are multiple mega-bytes in size. This is done through a technique called *banking*. The system has similarities with paging in a modern computer. The internal ROM of the cartridge stores the complete game, but the banking makes only a small section of 16KB addressable through a predefined address range.

The layout of the memory of the Game Boy is as pictured in figure 4. In this

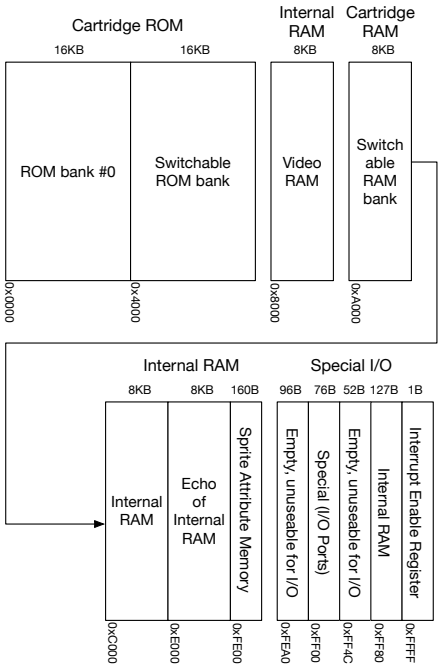

Figure 4: Address space of Game Boy from PyBoy report

project I focus on the first two sections of 16KB and the 8KB section marked as cartridge RAM. The first section will always address the first bank in the ROM. The second section can be swapped by the MBC, when the game requests it.

The reason the first bank is always addressed, is not documented, but I expect it to be due to speed and simplicity. The interrupt vectors are in this address space. If the vectors were placed on another bank, it would require time to swap around the banks by the Game Boy. This would also require the Game Boy to be aware of the MBC, which would complicate the hardware – I will get into details in the next section.

3.3 Memory Bank Controller

As said before, the cartridge only has a 16-bit address bus. That means, that only 65536 bytes can be addressed at once. Because of this limitation, most cartridges include a Memory Bank Controller (MBC). The game on the cartridge can make the Game Boy send commands to this MBC to change which ROM-bank the cartridge should make accessible at the address space 4000_{16} to $7FFF_{16}$ as shown in figure 4.

There is no dedicated communication line between the Game Boy and the cartridge. The MBC exploits the fact, that ROM is read-only. When "writing" to a special address range in the ROM, the MBC will intercept it and take it as a command to change a property in the MBC. In practice, this means, the Game Boy itself has no idea, that the banking is happening.

3.3.1 Types

The PyBoy report has alot of details about the addresses and values to write to change banks in section 5.3.1 "Cartridge types". To summarize it quickly, there is a range of common MBCs, but in theory the MBC could be completely unique for every game. We can only dump games from cartridges, whose MBC has been reverse-engineered.

The common practice is to use no MBC at all for games smaller than 32KB and to use the types named 1, 2 and 3 for larger games. In general, each type has a different balance between ROM and RAM size.

They all have in common, that writing to the address space of 2000_{16} to $3FFF_{16}$ will change the ROM bank. Type 1 of the MBC will only change the 5 least significant bits when written to this address. The last bits are written to the address space of 4000_{16} to $5FFF_{16}$. There is no rational behind this from the game's or the programmer's point of view. It is a matter of technicalities in the way the MBC was designed. When we dump the cartridge or if we get as far as to emulate the cartridge, we will have to watch out for these characteristica.

4 Dumping the cartridge

Dumping the memory of the Game Boy cartridges will give me a one-to-one copy of the exact contents of the cartridge. The dump can be used as backup, for analysis or to load into an emulator.

In this project, we want to copy it, so we can use it for the second part of this project, and secondarily to have games to load into the PyBoy emulator.

In the section above, we went through how a cartridge works internally. Because the first ROM bank is always accessible from the lower 16KB of the address space, I will start by dumping that part. Afterwards, I will advance into copying the other banks by sending commands to the MBC. This will of course require me to determine what kind of MBC the cartridge utilizes.

4.1 Wiring to the Arduino

The Arduino Mega has several 8-bit ports, which we can manipulate directly as variables in the code. I will assign two ports for the lower and higher part of the 16-bit address bus. I will also assign an 8-bit port for the data bus and 6 seperate pins to control the `CLK`, `WR`, `CS`, `RD`, `RST` and `AUDIO` pins on the cartridge.

To see the layout of the Arduino Mega, see appendix A.

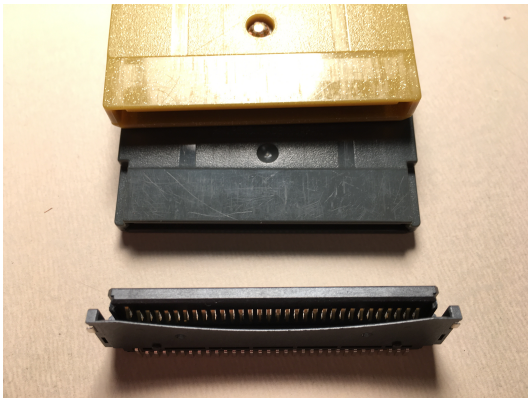

Figure 5: Picture showing a black Game Boy Advance game and a golden Game Boy Color game

I'll need a connector to plug a cartridge into. These connectors are not manufactured anymore. Reading online, I found suggestions, that pointed to use a connector for one of the newer generations of the Game Boy. I purchased a replacement connector for a Nintendo DS Lite, which is compatible with cartridges for Game Boy Advance. The connector for the original cartridges and the Advance cartridges is the same 32-pin connector. But the pins are used differntly depending on the two platforms. The Nintendo DS connector has two indentations to make it physically incompatible with the original Game Boy cartridges. This is because the Nintendo DS does not include the extra co-processor, like the previous generations did.

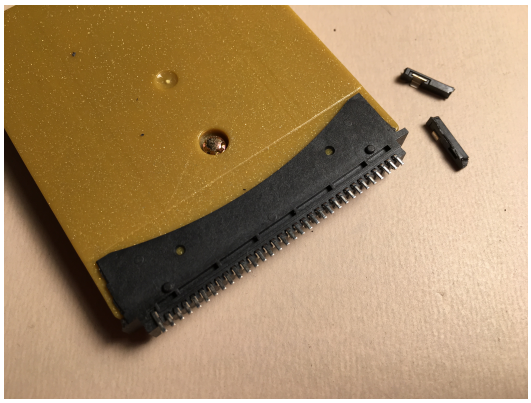

Figure 6: The connector modified to allow original cartridges

I'm only planning to dump games for the original Game Boy, therefore I take off the sides of the connector with a pair of pliers. This makes it possible to connect any cartridge. I soldered a ribbon cable the connector and connected it to the Arduino which dumps the cartridge to a PC.

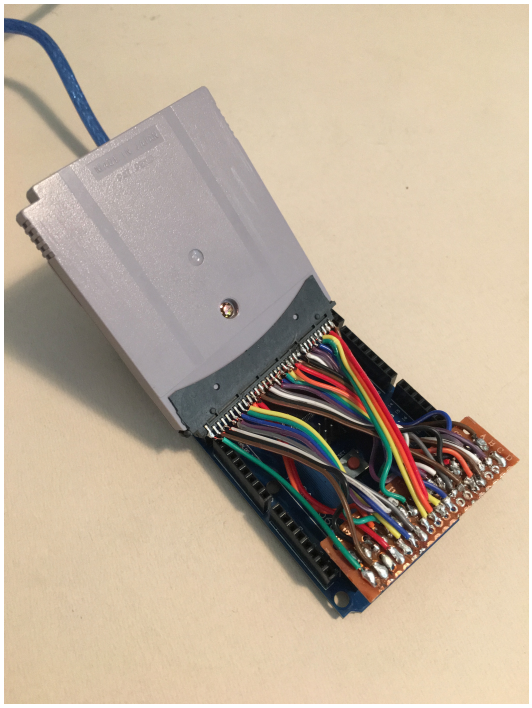

Figure 7: The setup used to dump cartridges

4.2 Cartridge headers

Every cartridge has a special set of values in the address space from 100_{16} to $14F_{16}$ of the first ROM bank. These values are meta data about the cartridge's hardware and the game. Most of the values are information for debugging and is not used by the game. To quickly summarize the impportnant fields, this is area contains: Entry point after exiting the boot-ROM, data for the Nintendo logo, title of the game, cartridge type, ROM size, RAM size, and two different checksums.

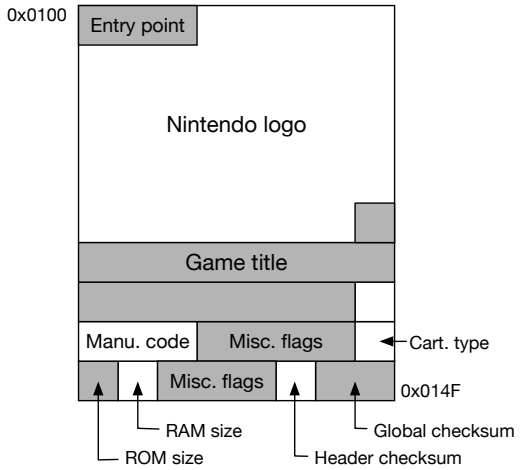

Figure 8: Illustration of the cartridge's header fields

4.3 Dumping the first bank

The Arduino Mega has just 8KB of RAMA, which is relatively small compared to the 16KB ROM banks. Therefore, the data I dump from the cartridge will have to be transferred to a PC as they get copied from the cartridge.

I setup a serial connection to a PC for the Arduino to off-load the data. I will run a small Python script on the PC to initialize the connection and create a file to store the dump. The script will determine the filename from the title field of the header as well as verifing the header-checksum and also keep count of the amount of ROM banks to store.

To dump the cartridge I can simply iterate through the addresses from 0000_{16} to $3FFF_{16}$ by assiging the high and low byte to the ports I have wired to the cartridge connector.

```
for (j = from_high; j<=to_high; j++){
    AddrHigh = j;
    for (i = from_low; i<=to_low; i++){
        AddrLow = i;
        if (i==0){
            delayMicroseconds(1);
        }
        data = DataBus;
        Serial.write((short)data);
    }
}
```

After assigning the address to the ports, the byte for that address will become available on the port I have assigned for the data bus. Because I only transfer data to the PC with a rate of $1,000,000$ baud, there does not seem to be any issue with timings, as the cartridge is rated to deliver data at atleast 1 MB/sec – reading a byte at every intruction from the Game Boy.

The only place I was forced to add a delay was after changing the higher byte of the address. Without this delay, the higher part of the address did not get registered by the cartridge before I had already read out the next byte. This resulted in the first byte of every 100_{16} bytes to be read as a wrong value.

The cartridges which support the Game Boy Color are likely to support the double transfer speed because of the Game Boy Color's feature of running the CPU at the double clock speed. This does not become relevant, as the Arduino does not allow me to choose the serial port at any higher baud rate.

4.4 Changing banks

To change bank, we will first have to determine the MBC type. Luckily enough, the cartridge header contains the information for this exact reason. Although it might have been included by Nintendo to simplify debugging.

After reading through the Pan Docs of the Game Boy[5, Memory Bank Controllers], I created a simple look-up table that links together the types of MBCs to the way we will communicate with them.

```
void switch_bank(short MBC, short bank) {
    switch (MBC) {
```

```
        case 0:
            break;
        case 1:
            write_cartridge(0x2000,
                bank & 0b00011111);
            write_cartridge(0x4000,
                bank & 0b01100000);
            break;
        case 2:
            write_cartridge(0x2000,
                bank & 0b00001111);
            break;
        case 3:
            write_cartridge(0x2000,
                bank & 0b01111111);
            break;
    }
    delayMicroseconds(5);
}
```

After creating this function, it becomes quite simple to dump all the ROM banks, as the number of banks is given by the cartridge header. It just ends up being a triple-for-loop of: For each bank, for each higher address, for each lower address, read the byte and send it through the serial port.

4.5 Dumping RAM

It is also possible to read and write the RAM banks of the cartridge. I chose not to include this, as it will not have any benefit for the second part of this project.

In case it becomes interesting, it should be somewhat easy to just change the address of which we are reading off data and change the value of the `CS` pin.

From the perspective of emulation, it might be interesting to read out existing data from the RAM, as this is where saved games are stored. This will enable the user to pick up on the emulator where they left off on the Game Boy. It could even write the saved game back onto the cartridge to keep the Game Boy and the emulator syncronized.

4.6 Verifying

To verify that the process works, I loaded Pokemon Blue onto my PC and added a checksum routine to the Python script which dumps the data. The cartridge header specifies a checksum value, which can be used to verify, that the ROM has not been corrupted. The algorithm for the checksum is defined as:

```
x = 0
for m in range(0x34, 0x4D):
    x = x - ord(header[m]) - 1
    x &= 0xff
```

Meaning that we take the sum of all the header fields, excluding the checksum itself. The calculation has to be done as 8-bit arithmetic and the single byte we get as output has to be exactly the same as the one in address $014D_{16}$.

As can be seen in the screenshot in figure 9, the calculation checks out and I am able to load the dumped cartridge into the PyBoy emulator.

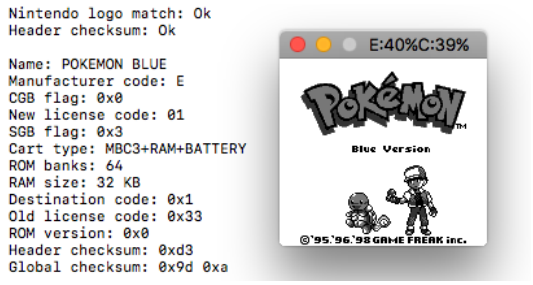

Figure 9: PyBoy emulator running the dumped cartridge of Pokemon Blue

5 Emulating the cartridge

This is what all the preparation has been for. I will now try to emulate a physical cartridge to make the Game Boy execute code, which I have the ability to control.

To deliver code to the Game Boy, we will first have to overcome the Digital Rights Management (DRM) routine inside the boot-ROM. I can overcome this by using the data which I retrieved in the previous sections. But to do this, I will have to be able to read the addresses the Game Boy is requesting and respond with the corresponding byte. This will require the Arduino to be able to intercept when the address changes, find the data and output it to the data bus within the time the Game Boy expects.

From my research of the cartridge, I know the Game Boy has an output-pin on the cartridge connector, which gives us a clock frequency to follow. I will monitor this clock and time when to read the ad-

dress and when to set the data bus to output.

5.1 Dummy cartridge

Wiring up the cartridge was quite simple. I opened up a cartridge and wired off the chip on the inside.

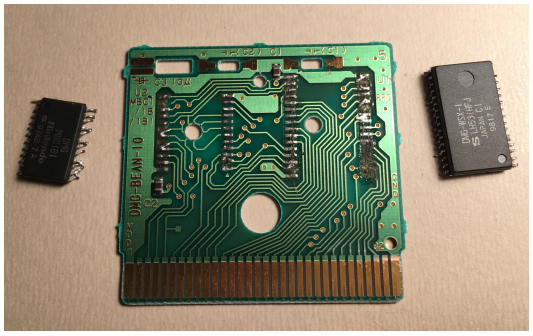

Figure 10: Emptying the cartridge

Then I soldered a ribbon cable to each of the pins to make it extend out of the cartridge.

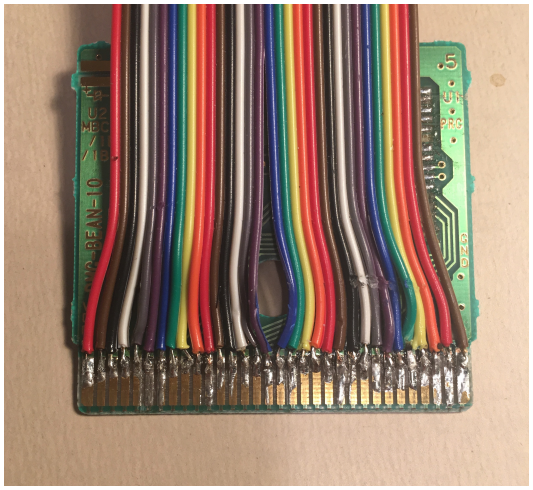

Figure 11: Wired ribbon cable to pins

The other end is connected to the Arduino and forms the dummy cartridge I will plug into the Game Boy.

5.2 Boot-ROM

As we found out during the PyBoy project, Nintendo implemented a DRM routine in their boot-ROM, which will copy the Nintendo logo from the cartridge and compare it to a copy, which it stores within the boot-ROM. The data for the nintendo logo is

Figure 12: The setup used to dump cartridges

available from multiple places on the Internet. One of them being the boot-ROM dump, which we covered in the PyBoy project[7, Boot-ROM]. Because I dumped several cartridges in the previous section, I can also retrieve the data myself.

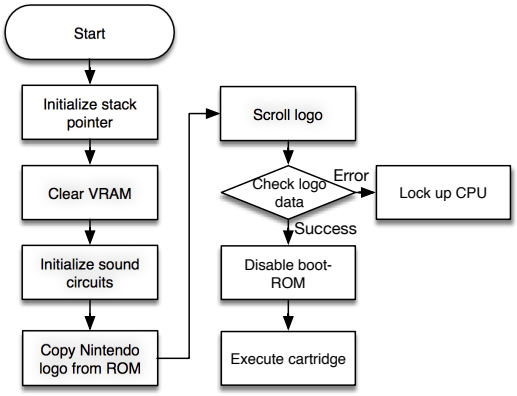

Figure 13: Flowchart of the bootROM from PyBoy report

Apart from copying the Nintendo logo for verifying authenticity, it also gets used as a self-test. To be precise, the boot-ROM copies the boot-ROM twice. First, it gets copied to the video-RAM and displayed on the screen. The second time, it gets used for the DRM routine.

I have not found an official source that clarifies why the logo gets copied from the cartridge, but I expect it to be part a self-test that will show if there is a pin on the data bus or address that doesn't function

correctly. In case one of the pins are not connected, it will be shown as block dots on the logo and the DRM routine will make the Game Boy halt until it is reset.

I exploit this characteristica of the boot-ROM to test out, that my system is working. If the Game Boy did not have this, it would be practically impossible to verify if the data bus was working before we could pass the DRM routine.

5.3 Checking the data bus

First step is to prove that we can write just a single byte, which the Game Boy will display on the screen.

The boot-ROM doesn't write anything to the cartridge during the initial boot-sequence. Therefore, we can safely put our emulated cartridge in read mode and deliver a constant on the data bus.

I've chosen to send a byte of 00110011_2 with the expectation, that multiple vertical lines will replace the Nintendo logo on the screen. It will of course not fool the DRM routine and will lock up the Game-Boy. Which is just to our benefit, as we then have a chance to examine the data on the screen.

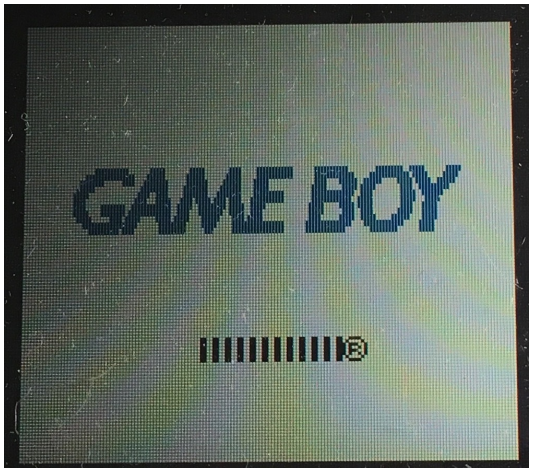

Figure 14: Showing our proof that we can send the Game Boy data

As can be seen in the picture above, the Game Boy did exactly as expected. I will run a test where each bit is enabled one at a time, to verify, that the connector is soldered properly.

7	6	5	4
3	2	1	0

Figure 15: The order the data pins relate to the pixels

From a quick test, I can see that the layout of the pixels on the screen is as shown in figure 15.

By counting the off-set of the pixels compared to the data, I found out I had swapped bit 6 and 7 by accident.

As the databus is now verified to be working, we can continue to deliver acutal data to the Game Boy.

5.4 Detecting the clock

Now that we have proven that we can make the GameBoy read our data, the next step will be to change the data for each clock. If we can't change the data at the correct pace, we will neither be able to read the address nor respond in time.

To try to detect the frequency, I create these few lines of code:

```
cli(); // disable all interrupts
int i = 0;
while ((PINB & 1) == 1){}
while ((PINB & 1) == 0){  // Await falling edge
  i+=1;
}
sei(); // enable all interrupts
Serial.print(i, DEC);
```

The code will disable interrupts to make sure we don't mess up the counting. Then I make sure to not start count in the middle of a cycle, by first waiting for the clock to go high and then count when it gets low. After the test, I enable interrupts and send the result to the connected PC. Then the test starts over.

The result of the test was not a success. The returned value is always 1, meaning that the code could only execute one cycle before the Game Boy's clock had changed. This is no where the efficiency that we need.

After examining the assembly code, I can't determine the reason the code will not atleast count a few times. I decide to move on, to see if I can point out the error.

5.5 Changing data on clock cycles

By "blindly" waiting for the clock to rise and fall, I can try to change the data on the bus. The data will be shown in place of the "Nintendo" logo the Game Boy shows when it boots.

The following code changes between 4 states on the data bus, which will will each show one unique pixel.

```
while(1){
    // State 1
    while ((PINB & 1) == 0){}
    while ((PINB & 1) == 1){}
    PORTA = 0b10111001;

    // State 2
    while ((PINB & 1) == 0){}
    while ((PINB & 1) == 1){}
    PORTA = 0b11011001;

    // State 3
    while ((PINB & 1) == 0){}
    while ((PINB & 1) == 1){}
    PORTA = 0b10011101;

    // State 4
    while ((PINB & 1) == 0){}
    while ((PINB & 1) == 1){}
    PORTA = 0b10011011;
}
```

The output will likely *not* be perfectly changing between the 4 states for each line of graphics. I will be changing the data on the bus at every clock cycle regardless of what the Game Boy is expecting. The GameBoy can't practically read it that fast. At best, it will load one byte at every second cycle.

The code is made to first wait for rising edge – as it spinlocks on the clock-pin being 0. Then it waits for falling edge, as that is where the real Game Boy would expect the data to change after an address change. This can be seen in figure 16.

The result is a "ladder" structure as seen in figure 17. This proves that we can in fact change the data and as I reset the Game Boy multiple times, it shows that the Game Boy consistently reads the data from the bus as it changes. This means, that we are likely detecting the clock cycle correctly.

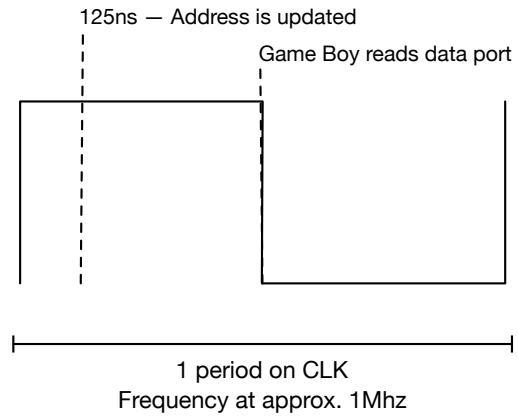

Figure 16: Illustration of read/write cycle

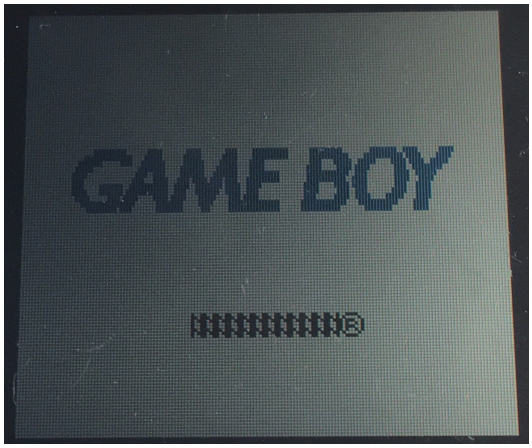

Figure 17: Showing some of the states for the test

We currently have a clear pattern of every second state being shown, which means the Game Boy uses a multiple of 2 clock cycles for its loading routine. If I was not matching the clock cycles, we would likely see a mixed or inconsistent pattern.

Although the code makes the output of the data bus change, I can not prove, that the data is changing on every clock cycle from the Game Boy. The code might take so long, that the overlaps several cycles. I cannot see how long it takes from the clock going high and until the Arduino registers it. For this, I will need some kind of logic-analyzer hardware, as the oscilloscope I have available cannot detect this correctly.

5.6 Reading the address

Now, I have proven that I can deliver a byte on the data bus, I can detect the clock

indirectly and I can change the data between clock cycles. The next step is to read the address and deliver the data, which the Game Boy is expecting.

I wrote a simple routine to read the address and send it to the PC. I'll look for the addresses 104_{16} to 143_{16} because this area contains the Nintendo logo. I'll start by not looking for the clock and just wait for occurrences of the correct addresses on the bus.

```
short a,b;

while(1){
    cli(); // disable all interrupts
    a = PinAddrHi;
    b = PinAddrLo;
    sei(); // enable all interrupts

    if (a == 0x01 && b >= 0x04){
        Serial.write(a);
        Serial.write(b);
    }
}
```

I want to verify the code, therefore I run the compiled code through `avr-objdump`. I especially want to examine the part of the code, which reads the address to see how many clock cycles it will be using. I've only included the important part below.

```
19a:   96 b1           in  r25, 0x06   ; 6
19c:   80 91 09 01     lds r24, 0x0109
1a0:   78 94           sei
1a2:   91 30           cpi r25, 0x01   ; 1
1a4:   89 f7           brne    .-30
1a6:   84 30           cpi r24, 0x04   ; 4
1a8:   79 f7           brne    .-34
```

The first line shows `in r25, 0x06` which loads the sixth port into the register 25. This takes 1 clock cycle[2, 4-7]. The second line loads the SRAM address of 0x109 to the register 24. This instruction takes 3 clock cycles[2, 4-7]! Of the approximately $0.5\mu s$ of low on the clock cycle from the Game Boy, we are already using $0.25\mu s$ on loading the address. The AVR documention describes, that ports above a certain address will have to be accessed with the slower `lds` instruction [3, p. 5] I'll continue to run the test with this in mind. If it becomes a problem, I'll have to resolder the lower part of the address-bus to another port on the Arduino.

I run the test while logging the output through a simple Python script, that reads out the serial port as 16-bit hexadecimal values.

```
Ready.
0x104
0x105
0x106
0x107
0x108
[...]
0x133
```

The test shows perfectly, that we are able to read out the address from Game Boy. The fact that we read out all the addresses without skipping a single one, shows at the fact, that the boot-ROM of the Game Boy is not reading a new byte at every cycle. Thereby, the serial transfer has time to finish before the next byte is read.

5.7 Deliver the Nintendo logo

Now, I'll modify the routine to not send the address through the serial port back to the PC, but instead send the Game Boy the data for the Nintendo logo, which we have from our previous cartridge dumps.

```
short a,b,c;

cli(); // Disable interrupts
while(1){
    a = PinAddrHi;
    b = PinAddrLo;

    if (a == 0x01 && b >= 0x04){
        PortData = NintendoLogo[b];
        if (b == 0x43){
            break;
        }
    }
}

while ((PINB & 1) == 0){NOP;}

while(1){
    PortData = 0x76; //HALT
}
```

The Game Boy didn't quite do what was planed, but atleast it seems to be going in the right direction. The Game boy now shows some of the data from the Nintendo logo, but it seems to be scrambled a bit. And it doesn't change between reboots, which is a good sign, because everything is stable.

To move on, I will need an oscilloscope, to measure the timings, to see where it fails. I didn't immediately have access to this, therefore I looked at the output on

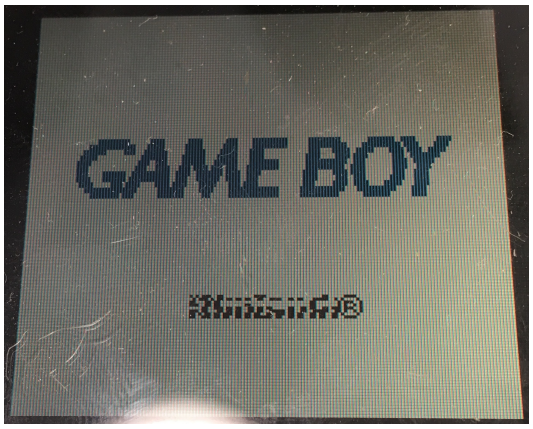

Figure 18: The transfer partially working

the screen, to see if I could find any hints. I loaded the image into GIMP and put some guiding lines on every 4×2 pixels – from 5.5 I found, that these are the pixels of one byte.

Figure 19: Guides showing the boundary between each byte

From the image above, it seems that some of the horizontal rows have been swapped. Looking closely, I realise, that there is a clear pattern. As illustrated in figure 20, the order of the bytes is clearly showing, that the data is being delivered one byte out of pace.

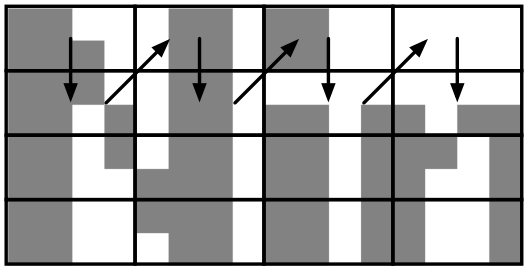

Figure 20: The order of the bytes being loaded onto the screen

After getting stuck on this issue, I searched online and found a blog post from 2011 which had hit the exact same issue with a pixel-perfect replica of the Game Boy screen[1]. By incredible coincedence, he had taken the exact same decision to just ignore the clock from the Game Boy and look for the correct addresses.

His way of solving the issue, is to initialize the Arduino with the first value of the Nintendo logo on the data bus. Then on purpose create an off-by-one in the array where he stores the Nintendo logo.

To try it out, I implement the same, and modify my Arduino code. This makes the data come out in the right order and the Nintendo logo gets displayed perfectly.

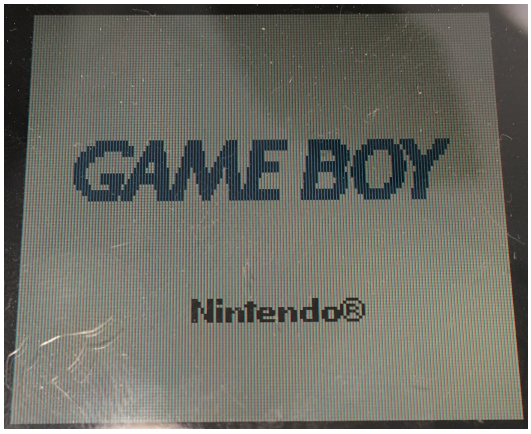

Figure 21: The Nintendo logo transferred correctly

Although this makes the logo appear, it does not solve the underlying issue, that the data is not delivered on the bus fast enough. From the symptoms I saw, I read the address and began to find the data for the bus, but by the time it is written, the Game Boy has moved on. When the next byte has to be read, the previous value will still be on the data bus, while the next one is fetched.

5.8 Improving the timing

If I want to be able to use this for testing code, I will have to deliver the data within a clock cycle from the Game Boy.

Before I can improve the performance, I will have to investigate the details of the timings and where the bottleneck will be. After setting up a test, I'll have to estimate the duration and see if it can be measured in practice.

5.8.1 Measuring the timing

As detailed in section 3, the cartridge connector has a pin, which outputs the clock from the Game Boy. Although it is not the direct clock frequency of 4Mhz. The clock pin only ticks at $\frac{1}{4}$ of the frequency, because the CPU of the Game Boy uses 4 clock cycles for each instruction.

I have disassembled two different games (Star Wars and Pokemon Silver), but both of them did not have the clock pin connected to any part of their circuits. I have not found any documentation on what the intended purpose of the clock pin is, but I assume, that I can use it to synchronize the operations between the Game Boy and Arduino.

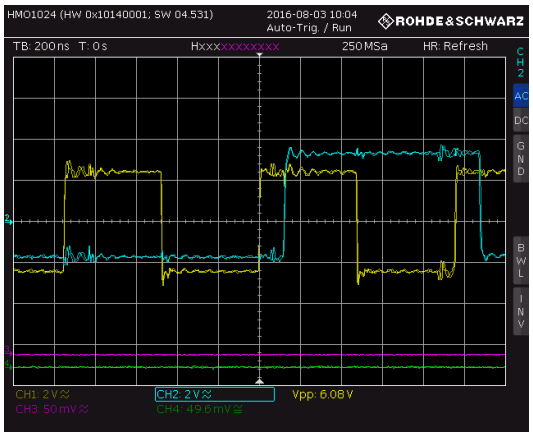

Figure 22: Oscilloscope showing the Game Boy's clock frequency and first bit of address line

When measuring the clock of the Game Boy (yellow line on figure 22), I was surprised to find, that the clock speed is actually quite a bit off of the previously noted 4 Mhz. From several measurings, I find, that the clock out to the cartridge has a duration of 960ns. Remembering the 4 clocks per instruction, I find the clock frequency to be $\frac{1}{960} \cdot 4 = 4.1667$Mhz. This gives me even less time to run the instructions I need. Assuming the data has to be on the bus at the mark of the falling edge, I will have to read the address and deliver the data within $960/2 = 480$ns.

Looking at the address line on figure 22, it seems evident, that the address changes around 120ns after the rising edge. This means, that I could move around the instruction on the Arduino to best utilize the first 120ns, but I cannot read the address before this point. This gives $480 - 120 = 360$ns to read the adderss and deliver the data. At 16Mhz, the duration of a clock cycle is 62.5ns. This gives us a total of $480/62.5ns = 7.68$ instructions in total and just $360/62.5ns = 5.76$ instructions for reading and writing the data.

5.8.2 Interrupt controlled

Looking for ways to optimize, I am looking at the assembly code on the Arduino and find any issue that I had overseen. I am spinlocking for the falling edge of the clock from the Game Boy. Having a simple while-loop waiting for a pin to become low would produce the following assembly:

```
sbis 0x03, 0; Skip next instruction if
            ; first bit of port is set.
rjmp .-4
```

The first instruction takes one clock cycle, but the jump takes two. If the clock goes high just after the Arduino has measured the pin, this routine will add a delay of almost three cycles. This gives a lot of uncertainty when we need to read the address at the 120th nanosecond, which is just two instructions after the clock going high.

To bring down the amount of cycles and correct the uncertainty, I try to implement another test, which utilizes an interrupt on the Arduino.

The Arduino Mega supports triggering interrupt-vectors on changes on a pin. The interrupt can be registered to call a function on rising edge, falling edge or any change. As I found in my tests and research above, we will likely want to look for a rising edge and read out the address shortly after. When the address is read, we will have to deliver the data in time for the falling edge.

The Arduino framework has made it quite easy to register the interrupt vector. During setup I specify which function that will be the interrupt vector.

```
void setup(){
    ...
```

```
    pinMode(ClockPin, INPUT);
    attachInterrupt(
        digitalPinToInterrupt(ClockPin),
        rising_clock,
        RISING);
}

void rising_clock(){
    // Read address and write data
}
```

The documentation for the Arduino's ATmega2560 CPU does not clearly state how fast the interrupt handler is. I therefore assume, that when the clock pin changes, that the Arduino will trigger the interrupt routine at the very next instruction. I will measure this at a later time, so if this assumption is off, I will have to correct for it.

5.8.3 Optimizing compiled code

To start off, I want to create the simplest possible example that will get me through the DRM routine of the Game Boy. This means, that I will have to serve the bytes in the 0104_{16} to 0133_{16} range. Given, that the higher byte is 01_{16} in the whole range, I will start by ignoring it, and add it back in, if it becomes needed. Given that I want to run my own tests, I can start off by not using more code than what I can keep in the lower byte of the address.

I have an array that defines the bytes in the Nintendo logo. To avoid subtracting the offset of 04_{16} from every look up address, I add four bytes to pad the data to align from 0100_{16}.

This boils the code down to a one-liner in C:

```
void rising_clock(){
  PortData = NintendoLogo[PinAddrLo];
}
```

Looking at the compiled code, I see a few instructions, that the compiler could have better optimized.

```
; avr-gcc generated code
lds  r30,  0x0109; Read lower address
ldi  r31,  0x00; Zero higher address
subi r30,  0x00; Calculate offset
sbci r31,  0xFE; Calculate offset
ld   r24,  Z; Read byte pointed by Z
out  0x02, r24; Write byte to data bus
ret
```

The Z is a special AVR name for using the `r30` and `r31` registers as a 16-bit pointer.

As the C code shows, it is completely redundant to calculate the higher part of the address, as it is constant. The compiler also wants to calculate the offset, but as I changed the array to compensate for this, it is also redundant.

As I seem to be able to optimize it better than the compiler, I change the code to inline the assembly with my corrections.

The compiler luckily aligns the array at 0200_{16} in the Arduino's program memory, so I don't have to calculate any offset. I set the `r31` register to contain the higher part of the address for the Nintendo logo, which is determined at compile-time.

Assuming, that the Arduino triggers at exactly the rising edge of the Game Boy's clock, I added a `nop` to get past 120ns before reading the address.

```
void rising_clock(){
    asm volatile(
        "ldi r31, hi8(NintendoLogo)\n\t" //1clk
        "nop\n\t" // 1clk
        "lds r30, %1\n\t" //2clk
        "ld r24, Z\n\t" //2clks
        "out %0, r24\n\t" //1clk

        // Total: 7clks
        // Total without nop: 6clks
        :: "I" (_SFR_IO_ADDR(PortData)),
           "I" (_SFR_IO_ADDR(PinAddrLo)));
}
```

After running the compiler, this boils down to much more simple routine, than the one we say from the `avr-gcc` compiler.

```
ldi r31, 0x02 ; Higher address
nop
lds r30, 0x0109; Read lower address
ld  r24, Z ; Read byte pointed by Z
out 0x02, r24 ; Write byte to data bus
```

5.8.4 Moving port to optimize

As I found out in section 5.6, and as seen above, the lower part of the address is read from a port, which takes two clock cycles. Looking at the layout of the ports in appendix A, we can see port "F" is conveniently placed and has a relatively low port number. I use this port, as it will be accessible with the faster `in` instruction. I verify this with the compiler, and resolder the pins to use this port instead of port "L".

```
void rising_clock(){
    asm volatile(
        "ldi r31, hi8(NintendoLogo)\n\t" //1clk
        "nop\n\t" // 1clk
        "in r30, %1\n\t" //1clk
        "ld r24, Z\n\t" //2clks
        "out %0, r24\n\t" //1clk

        // Total: 6clks
        // Total without nop: 5clks
        :: "I" (_SFR_IO_ADDR(PortData)),
           "I" (_SFR_IO_ADDR(PinAddrLo)));
}
```

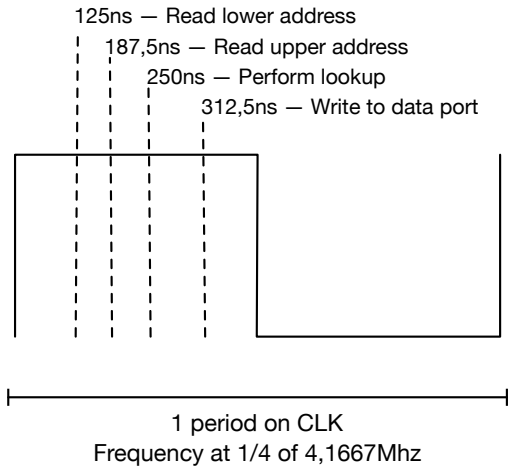

Figure 23: The timeline of reading address and writing data

The final code gives just six instructions, which is just below the threshold of 7.68 instructions that I calcuted above. Apart from this, I even had time for another instruction which I filled with a nop. This might come in handy, if I add the higher address to the routine.

5.8.5 Testing the new routine

With the new routine, I am ready to see if it has improved the reading and writing enough to deliver the Nintendo logo at the right pace. In theory, I have calculated the timings, but the only variable is how fast the Arduino triggers the interrupt vector. If it gets delayed too much, we will have to look for other alternatives. If the interrupt is triggered at the first instruction after the clock goes high, the optimized code I've written should have enough overhead to make it work.

The first time I turn on the Game Boy with the new code, the output does not seem to deliver even one byte correctly.

I suspect, that the error might be with reading the address correctly. I hardcode the address register to a constant and start the Game Boy again. This time I see the same byte show up repeatedly. This shows, that my code for look-up and data bus is working.

I change the code back and for a few restarts I am able to make the first byte appear, but the rest is scrambled as before. I must be close, otherwise it is an insane coincedence, that the random data perfectly aligned to be the first byte of the logo at the right place.

I reset the Arduino and the Game Boy, and now it is back to showing random data. I suspect the timing between the rising edge, the time the address becomes available and the time I read it is what is causing the issue.

I only have one oscilloscope which can work at this frequency, and it won't be possible to record and correlate the clock, address bus and data bus at the same time. I can see others who have tried projects similar to this are using what they call a logic analyzer to record a timeline of the pins going high and low[6].

I won't be able to acquire a logic analyzer in time. The Arduino is also not fast enough to send any debug information while running.

I reprogram the Arduino to output the address it reads onto the display by writing the address directly to the data bus. As the pixels on the screen each represent a bit on the data bus, I hope to see a pattern of the bits increasing like the address would do.

This does not seem to be the case, as the address should increase by exactly one at every part of the logo. The pattern the data is loaded can be seen on figure 20.

Still trying to pin point the issue, I create a small test that will see if the interrupt routine even gets triggered at the right pace and at the same time try to measure the deplay the interrupt might introduce.

The test is just a few lines of C:

```
byte ff = 0xff;
void rising_clock(){
    ff = ~ff;
```

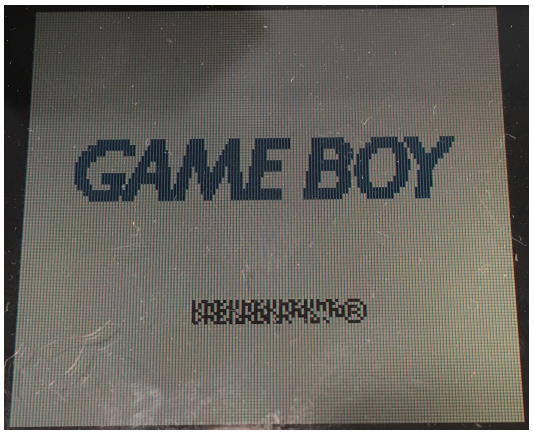

Figure 24: Test trying to output address on screen

```
    PortData = ff;
}
```

The code works flawlessly when run in an infinite loop. It creates a wave pattern of just a few hundred nanoseconds of wavelength – which is shorter than the time the clock is high.

I move the same code into the interrupt vector and check the clock is reaching the interrupt pin. This produces nothing. The oscilloscope, which I used for the while loop, shows no signal at all. Not even a small jitter.

I try out several different way to produce some output, and it would seem that the interrupt vector does get triggered when the Game Boy gets turned on, but the interrupt vector stop being called at some point. This can be seen, because I was able to load some code that oscillates, but from time to time, the logic signal stays high.

This is where I stopped. I conclude, that to continue, I will need to probe the set up with a logic analyzer to see which parts are working and what is out of synchronization.

5.9 Future work

It was not possible to pass the DRM check of the Game Boy. The only known issue holding this back, is whether or not the interrupt-routine is getting triggered correctly and on time. Without a logic analyzer, this will be time consuming and impractical to investigate.

If we assume the interrupt-routine started being triggered and was proven to do it within just one instruction on the Arduino, the loading routine should start working instantly and get the Game Boy past the DRM check.

After getting past the DRM, the Game Boy would begin to run code at the entry point of 0100_{16}. The current code will only be able to read the lower part of the address, but it would still be possible to have code prepared for the addresses from 0134_{16} to $01FF_{16}$. This would overlap a part of the cartridge's header data, but it wouldn't be needed to run the code.

If planned out well, the Game Boy could even send commands to the Arduino like it was an MBC to make it switch between several tests or page in small parts of code at a time.

When the test code has run on the Game Boy, it could make a write operation with the result to the Arduino. Then the Arduino could lock the data bus with 76_{16} for the `HALT` operation to pause the Game Boy and give it time to communicate with the PC.

6 Conclusion

To work on the PyBoy project, I needed a way to probe the Game Boy for details.

I did succeed in dumping several cartriges to my PC, that can be used to investigsate further. The dumped cartridges also proved to work on the PyBoy emulator, which fills in the missing link of the PyBoy emulator. It was also possible to implement checksum verification as a self-test to prove the process is working.

To emulate a cartridge on the Game Boy was a bigger challenge than expected. It was possible to send the Nintendo logo to the routine in the boot-ROM, but it doesn't seem, that the code passed the DRM check.

After several optimizations, it was possible to shrink the code into something that would theoretically work within the Game Boy's requirements.

The last step that held back the development was whether or not the interrupt-

routine was triggered continuously. Or if it was triggered with a large delay. To conclude what the cause of the issue was, I would have to acquire some equipment, that I did not have access to at the time of the project.

For the future of this project, the only known issue, is to trigger the interrupt routine at the right time. Then it would be possible to instantly load small tests, or tests that are confined to run without the Arduino reading the upper byte of the address. Including the upper part of the address, might require the Arduino to operate at a higher frequency.

References

[1] Alex. Emulating the nintendo logo on the gameboy, 2011.

[2] Atmel. Avr assembler user guide, 1998.

[3] Atmel. Conditional assembly and portability macros, 2008.

[4] Author information on page. GBATEK.

[5] Marat Fayzullin, Pascal Felber, Paul Robson, Martin Korth, nocash, kOOPa. Pan Docs.

[6] Dhole Wolfe. Emulating a gameboy cartridge with an stm32f4, 2014.

[7] Troels Ynddal, Mads Ynddal, and Asger Lund Hansen. Emulation of nintendo game boy (dmg-01), 2016.

Appendices

A Arduino Mega

Specifications from Arduino[1]

Microcontroller	ATmega2560
Operating Voltage	5V
Input Voltage (recommended)	7-12V
Input Voltage (limit)	6-20V
Digital I/O Pins	54
Analog Input Pins	16
DC Current per I/O Pin	20 mA
DC Current for 3.3V Pin	50 mA
Flash Memory	256 KB
SRAM	8 KB
EEPROM	4 KB
Clock Speed	16 MHz
Length	101.52 mm
Width	53.3 mm
Weight	37 g

Port and pin layout drawn by Fritzing from `http://fritzing.org/`.

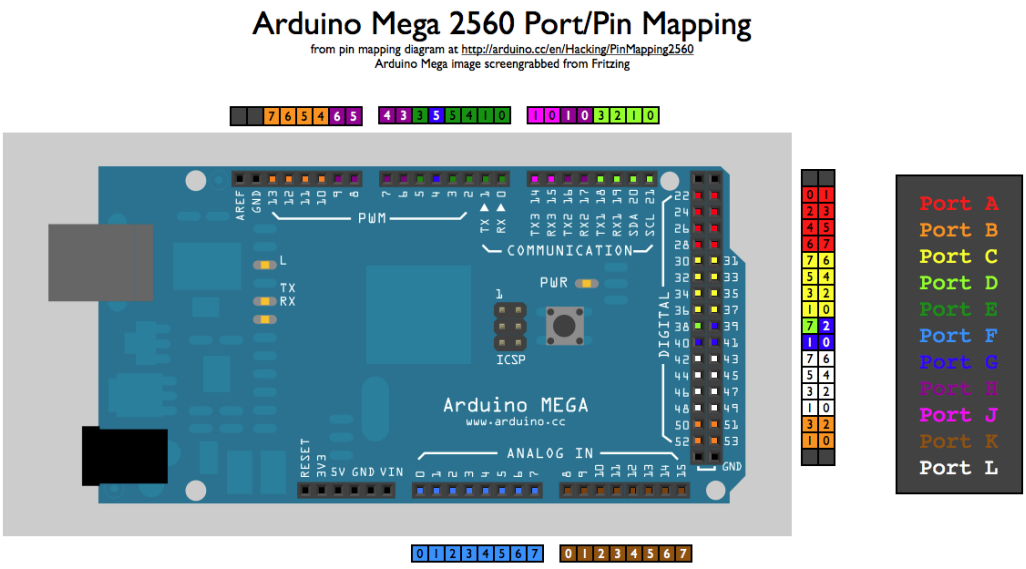

Figure 25: The pin and port layout of the Arduino Mega

[1] `https://www.arduino.cc/en/Main/ArduinoBoardMega2560`