RealBoy

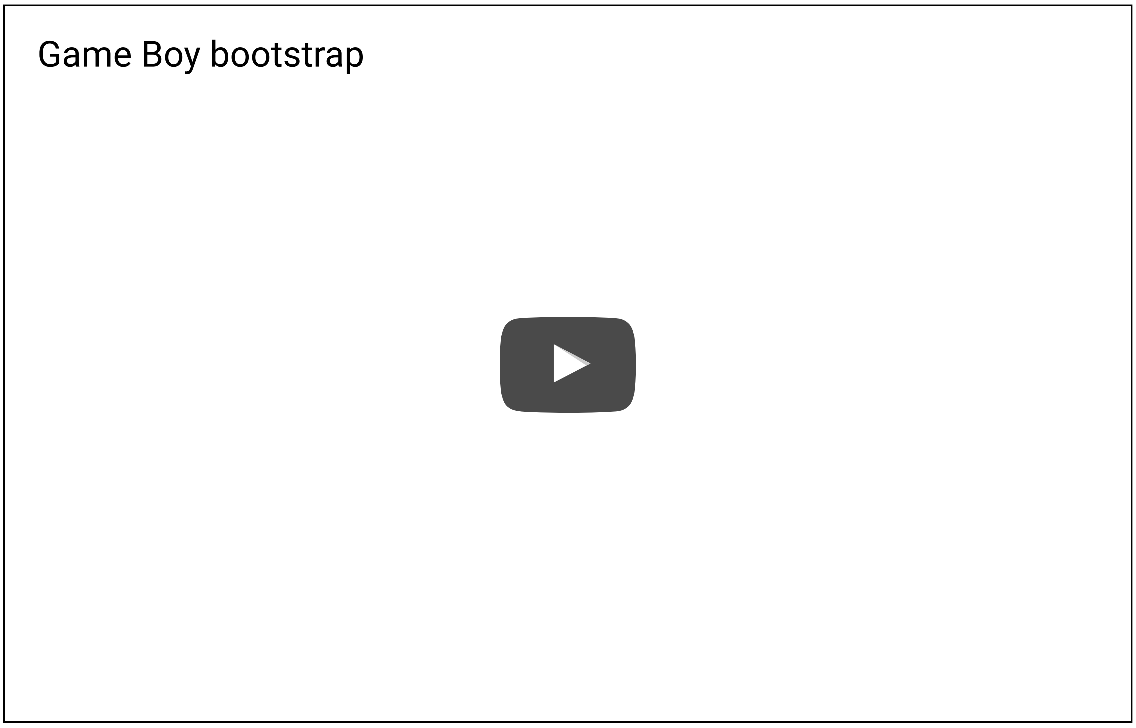
Complete, Fast, Accurate, Free, Game Boy/Game Boy Color/Super Game Boy Emulator for Linux/Unix.

A Look At The Game Boy Bootstrap: Let The Fun Begin!

Posted on **January 3, 2013**

THE WAY THE GAME BOY BOOTS

Let us study the Game Boy's bootstrap. Every time the console is turned on, this special, 256-byte program, is **mapped** to the beginning of the Game Boy's Address Space, the **Program Counter Register ('PC')** is set to **0x0000** and the 'real stuff' begins.



Booting the Game Boy with the RealBoy Emulator

The base of this work is done here. We will make further comments to the code because we want to be sure to capture the essence of this exercise. Although we will study the code at a considerable level of detail, at some point, too much detail would mean too much effort for little extra reward and a loss of the main goal of the exercise: **get used to the Game Boy's way of**

doing things. That is, don't worry if not everything is crystal clear.

The purpose of the bootstrap program is to scroll the Nintendo logo from the top of the screen and play a couple of 'beep' sounds. Besides this, the program does some other less obvious things.

EXECUTING THE PROGRAM

Let's take a look at the first instruction executed by the program:

- **0x0000** – *LD SP, $0xFFFE*

This instruction, located at memory address **0x0000**, means: load the value **0xFFFE** to the register '*SP*' (the stack pointer register). This simple process is known as **initializing the stack**. The only sure value upon powering on the Game Boy is the **Program Counter Register ('PC')** which is set to **0x0000**, and, although we can't be sure that the other registers are not initialized to a predefined value at this point, we can for sure affirm that the boot program **doesn't expect this to happen**; it explicitly initializes whatever as needed. Early initialization of the stack is always among the first things to be done in a boot program (boot programs for Linux and FreeBSD for the x86 architecture, for example, complies with this 'tradition'). Of course, initializing the stack on an x86 architecture is not as simple as on the Game Boy, among other things, because a special stack segment register must be set along with the stack pointer. Fortunately for the Game Boy, **setting the stack pointer suffices**.

With the stack pointer initialized, **it is now safe to call and return from routines**, among other things that need a working stack. Now let's continue execution:

- **0x0003** – *XOR A*
- **0x0004** – *LD HL, $0x9FFF*
- **0x0007** – *LD (HL-), A*
- **0x0008** – *BIT 7, H*
- **0x000A** – *JRNZ .+0xfb*
- **0x000C** – *LD HL, $0xFF26*

(The values preceding an instruction refers to the address in memory where the instruction resides; effectively, the value of the Program Counter Register ('PC') at that point in execution).

The purpose of these instructions is to clear (set to zero) all of the Video RAM (memory area from 0x8000 to 0x9FFF). This is a special area in memory that describes the pixels to be drawn on the screen. Clearing it is necessary before turning on the display, because this space contains random data when the Game Boy is turned on. If it were not cleared, random pixels would appear upon turning the display on. The mechanism is simple:

1. First, do a **bitwise exclusive 'or'** between the register 'A' and itself; this effectively sets 'A' to zero.
2. **Load** the value **0x9FFF** to the register pair '*HL*', which will serve as a pointer to clear the VRAM.
3. **Load** register '*A*' to the memory address pointed to by '*HL*' (write **0** to **0x9FFF**), and then decrement the value of '*HL*' (from **0x9FFF** to **0x9FFE**).
4. The following part is somewhat tricky, and admittedly it requires some practice to get use to

this kind of code. When **0x9FFF** was loaded to '*HL*', indeed **0xFF** was loaded to '*L*' and **0x9F** to '*H*'. Therefore, '*H*' is equal to the binary number **10011111**. The instruction *BIT 7, H* tests for the most significant bit of '*H*'. This means it just checks whether the bit is **0** or **1**. According to the result, the **zero flag** of the '*F*' register is **set** or **cleared**. Because a value of **0x9F** means that the most significant bit is **1**, the zero flag is cleared. This bit twiddling is used to know when to stop looping.

5. The next instruction reads: Jump if not zero to the address **0xFB relative to the current address**. Because the zero flag **was cleared**, the 'not zero' condition is met, so a jump is performed. The **current address** here would be the address for the instruction **following** *JRNZ .+0xfb,* that is, address **0x000C**. The value **0xFB** is **interpreted** as a signed byte, so *JRNZ .+0xfb* effectively translates to: **"jump if not zero to address 0x000C-0x0005=0x0007"**.

Now, the process is repeated: **0** is written to **0x9FFE**, and '*HL*' is decremented to **0x9FFD**, the most significant bit of '*H*' is tested again, and a jump is performed to address **0x0007** if 'not zero'. This loop repeats **as long as the 'not zero' condition is met.** So a value of **0** is actually written to addresses **0x9FFF**, **0x9FFE**, **0x9FFD**, and so on. How long does the loop last? Well, we see that as long as '*H*' is 0x9F the most significant bit is 1, and the 'not zero' condition is met, so **at least** the loop will write 0 to addresses **0x9FFF**… all the way to **0x9F00**. When '*HL*' is **0x9F00**, the next decrement 'HL' will be **0x9EFF**. We see that the 'not zero' condition is actually met all the way to '*HL*'=**0x9000**. But then, upon decrement, '*HL*' will be **0x8FFF**, and '*H*' will be **0x8F**. This is binary **10001111,** and we see that the most significant bit is **still** not zero. With this reasoning, we conclude that the 'not zero' condition is met until '*HL*' becomes **0x7FFF**. Then, '*H*' will be binary **01111111,** and the most significant bit (bit 7) will finally be zero. The loop indeed cleared all of VRAM, from **0x8000** to **0x9FFF**.

The following fragment makes some writes to the audio device; suffice for now that it turns on the device and writes to some of the devices' registers:

- **0x000C** – *LD HL, $0xFF26 # load 0xFF26 to HL*
- **0x000F** – *LD C, $0x11 # load 0x11 to C*
- **0x0011** – *LD A, $0x80 # load 0x80 to A*
- **0x0013** – *LD (HL-), A # load A to address pointed to by HL and Dec HL*
- **0x0014** – *LD ($0xFF00+C), A # load A to address 0xFF00+C (0xFF11)*
- **0x0015** – *INC C # increment C register*
- **0x0016** – *LD A, $0xF3 # load 0xF3 to A*
- **0x0018** – *LD ($0xFF00+C), A # load A to address 0xFF00+C (0xFF12)*
- **0x0019** – *LD (HL-), A # load A to address pointed to by HL and Dec HL*
- **0x001A** – *LD A, $0x77 # load 0x77 to A*
- **0x001C** – *LD (HL), A # load A to address pointed to by HL*

Let's continue. The following is perhaps the most demanding piece of code in this post:

- **0x001D** – *LD A, $0xFC # A represents the color number's mappings*
- **0x001F** – *LD ($0xFF00+$0x47), A # initialize the palette*
- **0x0021** – *LD DE, $0x0104 # pointer to Nintendo Logo*
- **0x0024** – *LD HL, $0x8010 # pointer to Video RAM*
- **0x0027** – *LD A, (DE) # load next byte from Nintendo Logo*
- **0x0028** – *CALL $0x0095 # decompress, scale and write pixels to VRAM (1)*

- **0x002B** – *CALL $0x0096 # decompress, scale and write pixels to VRAM (2)*
- **0x002E** – *INC DE # advance pointer*
- **0x002F** – *LD A, E # …*
- **0x0030** – *CP $0x34 # compare accumulator to 0x34*
- **0x0032** – *JRNZ .+0xf3 # loop if not finished comparing*
- **0x0034** – *LD DE, $0x00D8 # …*
- **0x0037** – *LD B, $0x8 # …*
- **0x0039** – *LD A, (DE) # …*
- **0x003A** – *INC DE # …*
- **0x003B** – *LD (HL+), A # …*
- **0x003C** – *INC HL # …*
- **0x003D** – *DEC B # …*
- **0x003E** – *JRNZ .+0xf9 # jump if not zero to 0x0039*
- …
- **0x0095** – *LD C, A # load A to C*
- **0x0096** – *LD B, $0x4 #* ***…***
- **0x0098** – *PUSH BC # push BC register on the stack*
- **0x0099** – *RL C # rotate left register C through carry flag*
- **0x009B** – *RLA # rotate left accumulator (register A) through carry flag*
- **0x009C** – *POP BC # pop word from the stack to register BC*
- **0x009D** – *RL C # rotate left register C through carry flag*
- **0x009F** – *RLA # rotate left accumulator (register A) through carry flag*
- **0x00A0** – *DEC B # decrement register B*
- **0x00A1** – *JRNZ .+0xF5 # jump if not zero to 0x0098*
- **0x00A3** – *LD (HL+), A # load A to address pointed to by HL and Inc HL*
- **0x00A4** – *INC HL # increment HL*
- **0x00A5** – *LD (HL+), A # load A to address pointed to by HL and Inc HL*
- **0x00A6** – *INC HL # increment HL*
- **0x00A7** – *RET # return from call*

So let's examine this intimidating piece of code:

First, let's note that the comment to each instruction does not describe the instruction's semantics; instead, the comment refers to what the instruction **actually** accomplishes. For example, instead of commenting *"load A to address 0xFF47"* we will comment *"initialize the color palette"*. Also, we have considered only the parts relevant to the current discussion; when we arrive at **0x003E**, for example, **we just skip all the irrelevant portion** until the instruction at address **0x0095**.

The big picture is the following: As we know, the boot ROM we are examining is mapped to addresses **0x0000** to **0x00FF** (the first 256 bytes of the address space). Following it, the cartridge's ROM is mapped. Every cartridge contains, at a predefined location, special information that must follow some conventions; this is the **header** of the cartridge. One such convention is that at addresses **0x104** to **0x133** the cartridge must present the following values:

0xce, 0xed, 0x66, 0x66, 0xcc, 0x0d, 0x00, 0x0b, 0x03, 0x73, 0x00, 0x83, 0x00, 0x0c, 0x00, 0x0d, 0x00, 0x08, 0x11, 0x1f, 0x88, 0x89, 0x00, 0x0e, 0xdc, 0xcc, 0x6e, 0xe6, 0xdd, 0xdd, 0xd9, 0x99, 0xbb, 0xbb, 0x67, 0x63, 0x6e, 0x0e, 0xec, 0xcc, 0xdd, 0xdc, 0x99, 0x9f, 0xbb, 0xb9, 0x33, 0x3e

These values correspond to the Nintendo Logo that scrolls from the top of the screen every time the Game Boy is turned on. That's why, when no cartridge is present, what scrolls from the top of the screen is a black rectangle (all values are 0) instead of the Nintendo Logo. Also, if you ever inserted a cartridge with different values at these addresses, strange things would appear on the screen, and the Game Boy would then lock up at some point (we'll see that later). The part we are examining right now copies the Nintendo Logo to Video RAM so it can be displayed when the screen is turned on.

Let's see in detail:

1. First, '*A*' is loaded with the value **0xF3**, and then copied to address **0xFF47**. This memory location is mapped to a **special register** of the LCD Display device. The register does the mapping between the four possible values of the Game Boy's colors (numbers 0, 1, 2 and 3) to actual colors (white, light gray, dark gray and black); that is, it initializes the **color palette.** The register at **0xFF47** is divided as follows:

– Bits 7-6 – defines color number 3
– Bits 5-4 – defines color number 2
– Bits 3-2 – defines color number 1
– Bits 1-0 – defines color number 0

Each pair of bits can hold a value from 0 to 3. These values are interpreted as follows:

0 is white
1 is light gray
2 is dark fray
3 is black

The register was written with value 0xF3, which is binary **11110011**. This means that color number 0 is assigned black, as well as colors number 2 and 3; color number 1 is assigned white. You can see that any color number can be mapped to any **actual** color; if you, for example, wrote 0xFF (binary **11111111**) to the register, every pixel drawn to screen would actually be black. This flexibility is used to do some video effects.

2. Next, '*DE*' is loaded with the address where the logo starts; *DE* will indeed be used as a pointer to copy the individual bytes of the Nintendo Logo to Video RAM. Also, '*HL*' is loaded with **0x8010**, so '*HL*' points to a portion of Video RAM. Now, '*A*' is loaded with the value at address **0x104** (the start of the logo), and a call is issued to address **0x0095**.

3. The function at 0x0095 is not worth right now looking into much detail. The individual instructions are fairly easy, but what they accomplish is not (at least not for our introductory level). It occurs that the values found at **0x104** that correspond to the Nintendo logo aren't written directly to Video RAM; instead, the function at **0x0095** manipulates each individual byte before writing it to Video RAM. The manipulation actually **decompresses and scales 2x** the image both horizontally (through the rotate instructions) and vertically (by copying twice the value to Video RAM). This function returns at **0x00A7** and is called again immediately at **0x002B** to finish scaling the current byte. When all the bytes (really the pixels) are **decompressed**, **scaled** and **copied** to VRAM, execution continues at address **0x0034**.

4. Instructions from **0x0034** to **0x003E** consists of a loop that copies an extra 8 bytes to VRAM; this corresponds to the little 'R' next to 'Nintendo' in the Nintendo Logo.

We sure left out lots of details, particularly the way the Nintendo logo is decompressed and scaled before writing it to VRAM. Let's not worry too much about this and continue with our analysis.

The following piece does some further writing to VRAM; in this case, it writes to a special area known as the **tile map**. We will be familiar with the special structure presented by the Video RAM in following posts; let's just point out the basics: As we have seen, the Game Boy, and pretty much every console of the era, suffered from **memory limitations**. The Game Boy, for example, had to address the **whole system** (which includes cartridge's RAM and ROM, I/O devices, Video RAM, Working RAM, etc) in just **64KB** of **address space**. In particular, video systems always need considerable amounts of **dedicated** memory. In the Game Boy, this dedicated memory is known as **Video RAM** (VRAM). But the Video RAM is relatively small; the Game Boy assigns addresses **0x8000** to **0x9FFF** for VRAM (that is, **8KB**). So, the consoles of the era used a system known as **tile system**. The essence of this is the **tile**. Consider a tile to be a **block** of pixels; in the Game Boy, **each tile consists of 64 (8×8) pixels**. These blocks of displayable pixels were mapped into a region called, as we saw earlier, the **tile map**. So, instead of the pixels b 'directly used', **indirect** pixel access was achieved through this tile m could be **reused**, and a single copy of the tile would suffice. With no block of pixels would have to be copied the amount of times it was r considerable memory savings. The code is the following:

- **0x0040** – *LD A, $0x19 # …*
- **0x0042** – *LD ($0x9910), A # …*
- **0x0045** – *LD HL, $0x992F # 'HL' pointer to tile map*
- **0x0048** – *LD C, $0x0C # …*
- **0x004A** – *DEC A # …*
- **0x004B** – *JRZ .+0x8 # jump if zero to 0x0055*
- **0x004D** – *LD (HL-), A # load "HL' with A and then decrement 'HL*
- **0x004E** – *DEC C # …*
- **0x004F** – *JRNZ .+0xF9 # jump if not zero to 0x004A*
- **0x0051** – *LD L, $0x0F # …*
- **0x0053** – *JR .+0xF3 # jump to 0x0048*

The next chunk of code is responsible for actually **scrolling** from the top of the screen the Nintendo logo and **producing** the two sound 'beaps'. Let's remember that, upon power on, the Game Boy's LCD display is off, so all the writings to VRAM up to now didn't actually display anything; it merely prepared everything so that, when turning on the display, the Nintendo logo can be shown in a controlled manner.

- **0x0055** – *LD H, A # H=0 is taken as the 'scroll count'*
- **0x0056** – *LD A, $0x64 # …*
- **0x0058** – *LD D, A # D=0x64 is taken as a 'loop count'*
- **0x0059** – *LD ($0xFF00+$0x42), A # set the vertical scroll register*
- **0x005B** – *LD A, $0x91 # …*
- **0x005D** – *LD ($0xFF40+$0x40), A # turn on LCD Display and background*
- **0x005F** – *INC B # B=1*
- **0x0060** – *LD E, $0x02 # …*

- **0x0062** – *LD C, $0x0C # …*
- **0x0064** – *LD A, ($0xFF00+$44) # wait for vertical-blank period*
- **0x0066** – *CP $0x90 # value at 0xFF44 used to determine vertical-blank period*
- **0x0068** – *JRNZ .+0xfa # jump to 0x0064 (loop) if not at vertical-blank period*
- **0x006A** – *DEC C # …*
- **0x006B** – JRNZ .+0xf7 # …
- **0x006D** – *DEC E # …*
- **0x006E** – *JRNZ .+0xf2 # …*
- **0x0070** – *LC C, $0x13 # …*
- **0x0072** – *INC H # increment scroll count*
- **0x0073** – *LD A, H # …*
- **0x0074** – *LD E, $0x83 # …*
- **0x0076** – *CP $0x62 # when scroll count is 0x62, play sound 1*
- **0x0078** – *JRZ .+0x06 # jump if zero to 0x0080 (skip play sound)*
- **0x007A** – *LD E, $0xC1 # …*
- **0x007C** – *CP $0x64 # when scroll count is 0x64, play sound 2*
- **0x007E** – *JRNZ .+0x06 # …*
- **0x0080** – *LD A, E # …*
- **0x0081** – *LD ($0xFF00+C), A # …*
- **0x0082** – *INC C # …*
- **0x0083** – *LD A, $0x87 # …*
- **0x0085** – *LD ($0xFF00+C), A # …*
- **0x0086** – *LD A, ($0xFF00+$0x42) # …*
- **0x0088** – *SUB B # …*
- **0x0089** – *LD ($0xFF00+$0x42), A # adjust vertical scroll registe*
- **0x008B** – *DEC D # …*
- **0x008C** – *JRNZ .+0xd2 # …*
- **0x008E** – *DEC B # …*
- **0x008F** – *JRNZ +.0x4f # jump if not zero to 0x00E0*
- **0x0091** – *LD D, $0x20 # …*
- **0x0093** – *JR .+0xcb # jump back to 0x0060*

So, the previous lines of code **scrolls from the top of the screen the Nintendo logo, plays a couple of sounds, and then jumps to 0x00E0, which is the "Nintendo Logo check" routine**.

Next comes the "Nintendo logo check" routine and a special 'checksum' of values at memory addresses from 0x0134 to 0x014C. First, the "Nintendo logo check" routine checks that the cartridge has the correct values at addresses 0x0104 to 0x0133; indeed, it checks that the cartridge contains the Nintendo Logo at that addresses. It does this by **comparing** the individual bytes in the cartridge with the correct values, which the boot ROM stores at addresses **0x00A8** to **0x00D7**. If a single byte does not match the corresponding value, the Game Boy **locks up**. If the cartridge passes the "Nintendo Logo check", then, a special 'checksum' routine adds the bytes at addresses **0x0134** to **0x014C** to register *A*. Lastly, the 'header checksum' value at **0x014D** is added to *A*. The whole sum must be **0**; otherwise, the 'checksum' fails and the Game Boy **locks up** at **0x00FA**.

- **0x00E0** – *LD HL, $0x0104 # point to Nintendo Logo in cartridge*
- **0x00E3** – *LD DE, $0x00A8 # point to Nintendo Logo in boot ROM*
- **0x00E6** – *LD A, (DE) # load next byte to compare to*

- **0x00E7** – *INC DE # point to next byte*
- **0x00E8** – *CP (HL) # compare bytes*
- **0x00E9** – *JRNZ .+0xfe # jump to 0x00E9 if no match (lock up)*
- **0x00EB** – *INC HL # point to next byte*
- **0x00EC** – *LD A, L # …*
- **0x00ED** – *CP $0x34 # test if end of comparison*
- **0x00EF** – *JRNZ +.0xf5 # if not the end, jump back to 0x00E6*
- **0x00F1** – *LD B, $0x19 # counter; prepare for checksum*
- **0x00F3** – *LD A, B # A=0x19*
- **0x00F4** – *ADD (HL) # add byte pointed to by HL to A*
- **0x00F5** – *INC HL # point to next byte*
- **0x00F6** – *DEC B # decrement counter*
- **0x00F7** – *JRNZ .+0xfb # loop*
- **0x00F9** – *ADD (HL) # add last byte; the 'header checksum'*
- **0x00FA** – *JRNZ .+0xfe # if bad checksum, lock up here*

Finally, with the cartridge passing the "Nintendo Logo" and the "checksum" tests, the next co… of instructions are executed before control of execution is at last passed to the cartridge's RO…

- **0x00FC** – *LD A, $0x01 # …*
- **0x00FE** – *LD ($0xFF00+$0x50), A # disable boot ROM*

Writing the value of 1 to the address 0xFF50 **unmaps** the boot RO… the address space, where it effectively was mapped, now gets map… cartridge's ROM. Because the instruction at **0x00FE** is two bytes lon… executed is at **0x0100**. This is the first instruction executed that lie… execution is indeed now under control of the cartridge.

Now, this sweet little program is history, and the game's universe begi…

Continue with: Emulating The Core, Part 1: The Fetch-And-Execute Cycle.

HELP US IMPROVE!

Did you like this post? Do you have any suggestions? *Please rate this post or leave us a comment so we can improve the quality of our work!*

RATE THIS:

8 Votes

SHARE THIS:

Twitter Facebook

Like

One blogger likes this.

This entry was posted in **RealBoy Project** and tagged **Boot**, **Docume** **Boy**, **Game Boy Bootstrap**, **Game Boy Color**, **Game Boy Color Em** **Emulator**, **Gameboy**, **Gameboy Bootstrap**, **Gameboy Color**, **Game** **Gameboy Emulator**, **How Emulators Work**, **Programming** by **sergi** **permalink [https://realboyemulator.wordpress.com/2013/01/03/a-l** **bootstrap-let-the-fun-begin/]** .

15 THOUGHTS ON “A LOOK AT THE GAME BOY BOOTSTRAP: LET THE FUN BEGIN!”

Dudeson

on **November 4, 2013 at 5:26 am** said:

Nice tutorial bro!

serginho89

on **November 4, 2013 at 12:54 pm** said:

Thanks!😀

Mostafa
on **November 29, 2013 at 7:05 pm** said:

According to the gameboy programming manual and the other resources I found, the syntax for relative jumps is JR cc, n.
cc can be Z, NZ, NC, C. So shouldn't in clearing the VRAM, the jump be 0x000A – JR NZ, 0xfb ??

Or will both be treated the same by the assembler??? I am confused about this. Also why does the jump cause the execution to jump to 0×0007.. I understand that PC = 0x000C, and when Z != 0. PC = PC + 0xfb, how will the addition yield to 0x0007?, shouldn't it be 0x107??..I'd really appreciate it if you can clear this up for me. And thanks a million for this very useful blog😀

serginho89
on **November 29, 2013 at 11:37 pm** sa

Hello.
It is actually very simple: The immediate byte
Relative instruction is *interpreted* as a signed
byte represents numbers from -128 to 127. T
in two's complement; you have to make the d
the number itself, and the number it *represen*
(http://en.wikipedia.org/wiki/Two%27s_comp
So, let's look for example at the following ins
JR NZ, 0x20 (decimal 32, represents number +32)
JR NZ, 0x7f (decimal 127, represents number +127)
JR NZ, 0x80 (decimal 128, represents number -128)
JR NZ, 0x81 (decimal 129, represents number -127)
JR NZ, 0x82 (decimal 130, represents number -126)
…
JR NZ, 0xa0 (decimal 160, represents number -96)
JR NZ, 0xfb (decimal 251, represents number -5)

So the jump is to byte 0xfb relative to address 0xc, where 0xfb is *interpreted* as a signed two's complement byte; 0xfb *represents* -5.
I hope this makes it clear.

Pingback: Sistema para prevenir la ejecución de memorias no autorizadas en Game boy | [This Side Out]

Daniel Mewes
on **June 7, 2015 at 7:19 pm** said:

Thanks for the article. Very interesting. I was wondering before if the scrolling Nintendo logo was somehow used as a check for the cartridge being connected properly. Now I have the answer.

> **serginho89**
> on **June 8, 2015 at 2:48 pm** said:
>
> Glad to help… 🙂

Glenn
on **June 18, 2015 at 7:36 pm** said:

Hey there, just noticed an error in your OPCode

0x001C: LD (HL-), A # load A to address pointed

^ This is actually a LD (HL), A
Because the opcode is 0x77, not 0x32.

This portion from the gameboy binary is: "3e77 773e fce0". where the 3e77 is 'LD A, 0x77', then the 'LD (HL), A' then finally the 'LD A, $0xFC' that follows.

> **serginho89**
> on **June 18, 2015 at 10:57 pm** said:
>
> Thanks a lot, Glenn.😀
>
> > **Glenn**
> > on **June 18, 2015 at 11:39 pm** said:
> >
> > Just another one!
> >
> > 0x0042 – LD ($0x9010), A # …

As per the line in the bootcode: 'ea10 9921'
EA = LD (a16),A

So the line at 0x0042 should be: 0x0042 – LD ($0x9910), A # …

I'm only finding these because I am debugging my game boy emulator and making sure all my right OP codes do the right thing! And therefore the progress is tracking the same as your tutorial does!

So thank you for this resource 🙂

serginho89
on **December 6, 2015 at 9:10 pm** said:

Thank you very much, Glenn. How is you going?

Abel Shields
on **October 17, 2015 at 12:34 pm** said:

Hey, I have a problem. I'm getting to 0x00A1 (JRNZ .+0xF5 # jump if not zero to 0x0098) and my emulator (or what there is of it so far…) is getting stuck in a loop. Using http://www.devrs.com/gb/files/instr.txt I found that the only two commands that reset the Zero flag (and so will break out of the loop) are ADD SP, n and LDHL SP, n and neither of these are in the loop. So how does it stop?

Abel Shields
on **October 17, 2015 at 12:47 pm** said:

EDIT: Okay, I fixed it. Turns out I was bit shifting by the wrong number (got confused between dividing by 0x100 and bit shifting right by two).

Pingback: Emulating The Core, Part 1: The Fetch-And-Execute Cycle | RealBoy

Pingback: Redesigning Pokémon for Mobile | Unnecessary Writing