

PAPIGB SPECIFICATION V1.0

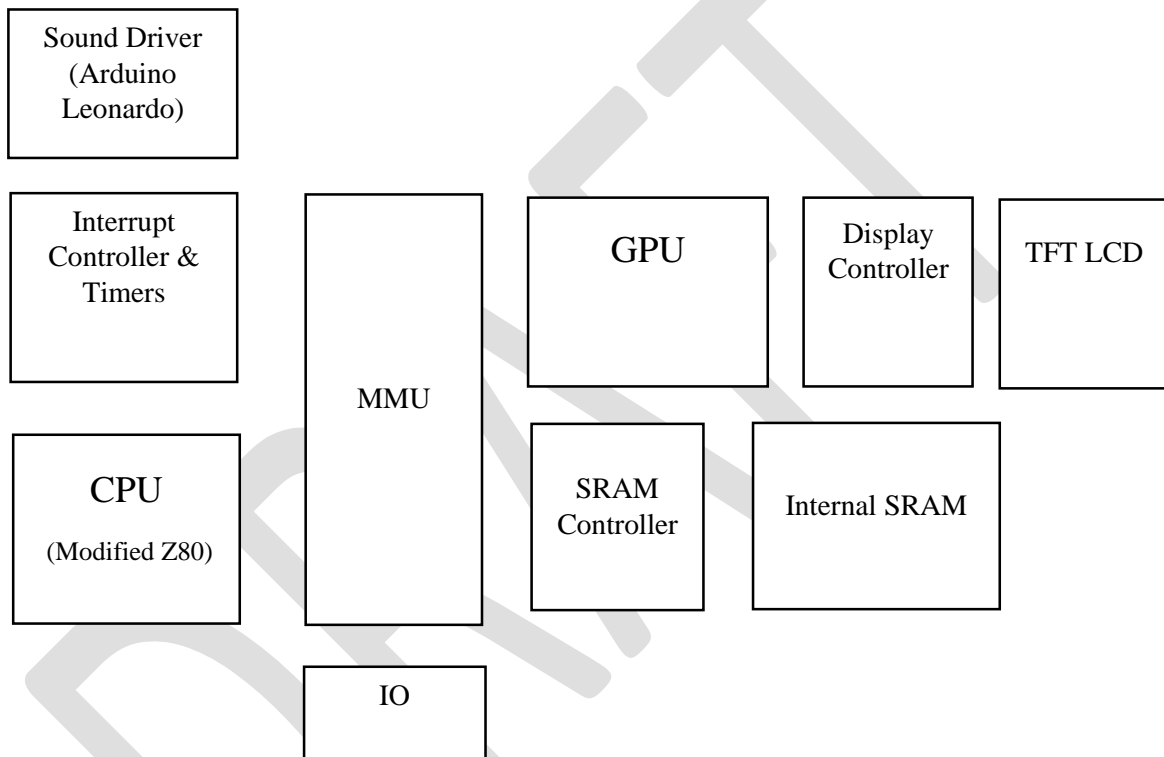


Revision History	Date	Author	Comments
Initial draft	Dec-10-2015	DValverde	Initial draft WIP

INTRODUCTION

SYSTEM OVERVIEW

WIP



ACRONYMS

pGB	papiGameBoy
GB	Game Boy classic
FPS	Frames per second
POR	Power On Reset

RTL	Register transfer level
TFT	Tiny Film Transistor
LCD	Liquid crystal display

1. DZCPU!



1. OVERVIEW

The original GB CPU was an 8bit Sharp LR35902, which is sort of a Z80 processor specially modified by Nintendo. Essentially, Nintendo took some opcodes out of the instruction set and added some opcodes of their own. The Z80 is very similar to an 8080 processor from the early 90s, in the sense that it has a CISC instruction set. In other words, it features lots of complex instructions, and each individual instruction does lots of things. Furthermore, decoding the Z80 instruction set may be a challenging task since the instruction width varies from single BYTE instruction up to 5 BYTE instructions. Also instructions take multiple clock cycles to execute and the time to fetch data from main memory (SRAM) adds additional latency to individual Z80 instructions.

The dZCPU! uses a micro-code approach to mimic the original Z80 CISC instruction set. The following diagram illustrates the overall processor architecture.

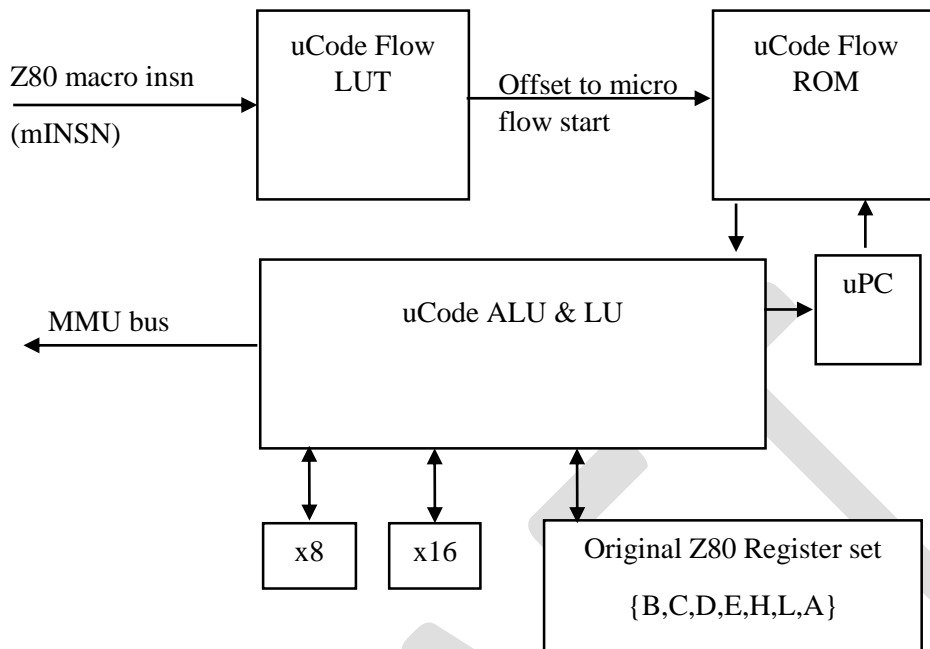


Figure 1 dZCPU! functional blocks

Figure 1 shows the basic blocks of the dZCPU!. Essentially, each time a Z80 macro insn is fetched from the MMU, the index to a micro-code is obtained from an uCode Look-up-table block. The ucode flow corresponding to that macro instruction is executed and the next macro instruction is fetched.

2. dZCPU! REGISTERS

In addition to the original Z80 8bit registers, two more registers called x8 and x16 had been added to the dZCPU!. The x8 register is 8bits, while the x16 register is 16bits. These registers are actually hidden from the game code, and only used as temporary storage by the dZCPU! micro-code flows.

3. dZCPU! MICROFLOWS

In order to get around many of issues and complexity of mimicking the behavior of the original Z80, the architecture and instruction set of a minimalistic micro-flow based processor called dZCPU! is described in this section. The dZCPU! fetches the original Z80 instruction set macro-instructions, and maps each of these complex macro instructions into a flow

of dZCPU! micro-instructions. Each micro-instruction is a very simple operation which takes a single clock cycle¹.

Each uCode Insn is 12bit wide. The format of an uCode instruction is the following:

Field Name	Range	Description
IPC	11	Increment uPC: This uINSN increments the uPC. Note the value of the uPC will effectively increment in the next CC. The reason to have a dedicated IPC bit is because the size of the Z80 macro instructions is variable. You can have instruction sizes varying from 1Byte up to 5Bytes. Furthermore, the MMU bus is restricted to 1Byte only, so the only way for the CPU to figure out the actual macro instruction size is by actually fetching individual bytes from the MMU, and then deciding whether or not more bytes are required in order to fully execute the current instruction.
EOF	10	End of Flow. Marks the end of the ucode flow. A new mINSN can be fetched after this signal is set.
PRED	9:8	Reserved 2'b00 Conditional execution on Z 2'b01 Update CPU Flags 2'b10 Reserved 2'b11
UOP	7:4	The uINSN uOP as specified in table XXX
ARG	3:0	The operation argument, if any

Figure 2 dZCPU! uInsn format

The following table specifies the dZCPU! uInsn set. Note that the convention is to use UPPERCASE for mInsn and LOWERCASE for uInsn. This is done to avoid confusion since some uInsn and mInsn may share the same name.

Nemonic/Value	Syntax	Description
nop 4'h0	nop	The uop performs no operation
sma 4'h1	sma [reg]	mmu_addr = [reg] In the next cc, mmu_addr is presented with the value of [reg]. This value is subsequently used by

¹ With the exception of SRAM CPU IO

			<code>srm</code> for reading from MMU or by <code>smw</code> for writing into MMU.
<code>srm</code> <code>4'h2</code>	<code>srm [reg]</code>		<code>[reg] = mmu_in_data</code> In the next cc, <code>[reg]</code> latches the value of <code>mmu_in_data</code> .
<code>jcb</code> <code>4'h3</code>	<code>jcb</code>		<code>uPc = cb_lut[mmu_in_data]</code> <code>uFlow jump to cb_lut[mmu_in_data]</code>
<code>z801bop</code> <code>4'h4</code>	<code>Z801bop</code>		Performs a single cc Logic Z80 operation
<code>smw</code> <code>4'h5</code>	<code>smw [reg]</code>		<code>Mmu_data_out = [reg]</code> In the current cc, <code>[reg]</code> value is preset to of <code>mmu_out_data</code> .
<code>dec16</code> <code>4'h6</code>	<code>dec16 [reg]</code>		<code>[reg] = [reg] - 1</code> <code>Flags[zero] = [reg] == 0</code> <code>Flags[neg] = [reg] < 0</code> Decrements register <code>[reg]</code> . In case of an 8bit register, the register is sign extended before decrementing.
<code>bit</code> <code>4'h7</code>	<code>bit</code>		<code>Flags[zero] = mInsn[2:0] & (1 << mInsn[5:3])</code> Performs a bit set logical operation on <code>mInsn[2:0]</code>
<code>addx16</code> <code>4'h8</code>	<code>addx16 [reg]</code>		<code>x16 = x16 + [reg]</code> Adds to <code>x16</code> the contents of <code>[reg]</code> . In case of an 8bit register, the register is sign extended before incrementing.
<code>spc</code> <code>4'h9</code>	<code>spc [reg]</code>		<code>Pc = [reg]</code> In the next cc, sets the macro PC to <code>[reg]</code> . In case of 8bit register, the register's 8 MSB are zero extended.
<code>sx16r</code> <code>4'ha</code>	<code>sx16r [reg]</code>		<code>X16 = [reg]</code> In the next cc, the value of <code>[reg]</code> is copied to <code>x16</code> .

sx8r 4'hb	sx8r	<p>$X8 = [reg]$</p> <p>In the next cc, the value of [reg] is copied to x8.</p>
inc16 4'hc	inc16 [reg]	<p>$[reg] = [reg] + 1$</p> <p>In the next cc, the value of [reg] is incremented by 1.</p>
srx8 4'hd	srx8 [reg]	<p>$[reg] = x8$</p> <p>In the next cc, the value of x8 is copied to [reg].</p>
shl 4'he	shl [reg]	<p>$[reg] = [reg] \ll 1 + \text{flags}[\text{Carry}]$</p> <p>In the next cc, value of [reg] is shifted left. In case of overflow the C flag is updated.</p>
subx16 4'hf	subx16 [reg]	<p>$X16 = x16 - [reg]$</p>

Figure 3 dZCPU! micro code instruction set

So the Z80 has around 256 CISC instructions, and you are telling me that you achieve any of those CISC instructions with a proper combination of the 16 uInstruction from Figure 3? Yes, that's the idea, see the next table for some example ucode flows:

Z80 insn	dZCPU! mirco Flow	Description
LDSPnn	<pre> 1:{ `inc, `sma, `pc }; 2:{ `inc, `nop, `null }; 3:{ `srm, `spl }; 4:{ `inc_eof, `srm, `sph }; </pre>	<p>Loads 16bit stack pointer register {sph,spl} with the literal located in the second byte of the instruction.</p> <p>Note how the micro flow explicitly increments the Z80 program counter using the `inc prefix.</p>
CALLnn	<pre> 1:{ `inc, `dec16, `sp }; 2:{ `dec16, `sp }; 3:{ `srm, `x8 }; 4:{ `sx16r, `pc }; 5:{ `inc16, `x16 }; 6:{ `inc16, `x16 }; 7:{ `sma, `sp }; 8:{ `smw, `x16 }; 9:{ `spc, `x8 }; </pre>	<p>Calls a subroutine located at literal position 'nn'.</p> <p>Note how the micro flow used the x8 and the x16 registers to store intermediate values of the program counter and the stack pointer.</p>

	<code>A:{ `eof , `sma, `pc };</code>	
PUSHBC	<code>1:{ `dec16, `sp };</code> <code>2:{ `sma, `sp };</code> <code>3:{ `smw, `b };</code> <code>4:{ `dec16, `sp };</code> <code>5:{ `smw, `c };</code> <code>6:{ `inc_eof , `sma, `pc };</code>	<p>Pushes contents of 16bit register into the stack.</p> <p>Note how the 'b' register is pushed first, and only then the 'c' register is pushed</p>
LDHLDA	<code>1:{ `sma, `hl };</code> <code>2:{ `smw, `a };</code> <code>3:{ `sma, `pc };</code> <code>4:{ `eof, `dec16, `hl };</code>	<p>Fetches the contents of memory pointed by {hl} and stores this contents into the 'a' register.</p>

2. MMU

0x0000	ROM bank 0 (first 16kB of cartridge)
0x4000	ROM bank 1 (second 16kB of cartridge)
0x8000	Video RAM (8kB)
0xA000	Switchable RAM bank (8kb)
0xC000	Internal RAM (8kB)
0xE000	Echo of internal RAM (8kB)
0xFE00	Not used
0xFF00	I/O ports
0xFF4C	Not used
0xFF80	Internal RAM
0xFFFF	Interrupt register

Figure 4 - Memory Map

4. SD CARD CONTROLLER

The LCD Board comes with an integrated SD Card slot. The SD card slot data control lines uses a standard serial interface (SPI).

See this (not SPI)

http://opencores.org/project,sdcard_mass_storage_controller

Specification is here:

https://www.sdcard.org/downloads/pls/simplified_specs/archive/partE1_100.pdf

https://www.sdcard.org/downloads/pls/simplified_specs/archive/partE1_100.pdf

We will use section 7 of the physical layer specification:

https://www.sdcard.org/downloads/pls/part1_410.pdf

3. GPU

MMU Address	Local Address	Name	Size bits	Description
0xFF42	0	SCY	8	Background Scroll Y position
0xFF43	1	SCX	8	Background Scroll X position
0xFF44	2	LY	8	The LY indicates the vertical line to which the present data is transferred to the LCD Driver.
0xFF45	3	LYC	8	???

5. Rendering the Background

Recall from Figure 4 that the video area is located from address 0x8000 to address 0x9FFF. This is where the video information is stored. Now, consoles in the 80's did not have a lot of available memory so they relied on a "tiled" based system. The GB supports up to 256 8x8 pixel tiles for the game background rendering. The way it works is by using tile map in order to build the background image from the tiles.

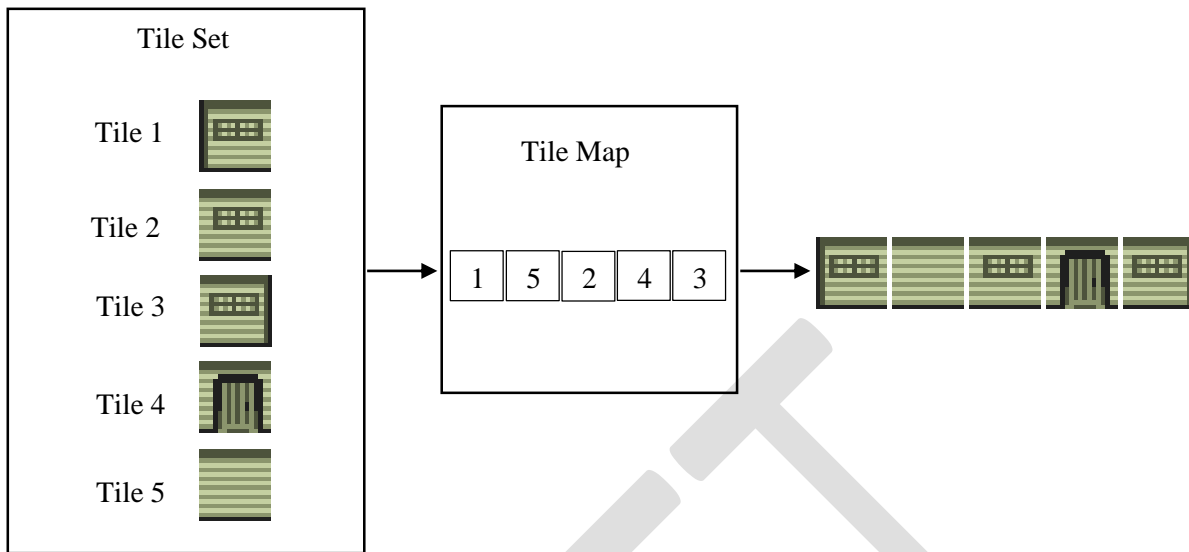


Figure 5 GB Tiling system

There are 2 tile set memory regions called TileSet0 and TileSet1. Each tile set can hold up to 32x32 tiles. Only one tile region can be accessed at a time. There are 2 tile map regions called TileMap0 and TileMap1.

GPU Region	Description
0x8000-0x87FF	TileSet1: Tiles 0 to 127
0x8800-0x8FFF	TileSet1: Tiles 128 to 255 TileSet0: Tiles -1 to -128
0x9000-0x97FF	TileSet0: Tiles 0 to 127
0x9800-0x9BFF	TileMap0
0x9C00-0x9FFF	TileMap1

Figure 6 GPU VRAM Memory regions

You noticed from Figure 6 that TileSet0 and TileSet1 are sharing memory at region 0x8800 to 0x8FFF, this is a workaround for games to overcome the limitation of accessing a single Tile Set at a time (they share some tiles).

Also note that TileSet0 is indexed from 0 to 255, while TileSet1 is indexed from -128 to 127.

6. RENDERING TILES

The GPU works on a tile by tile basis. It is wired to access a complete Tile Row (8 pixels) at a time. Each tile pixel has 2 bits, and can take the value of COLOR0 (2'b00), COLOR1 (2'b01), COLOR2 (2'b10) or COLOR3 (2'b11).

As mentioned earlier, a single row of a tile is 2 BYTES. Now, the width of the GPU memory is 1 BYTE (since the GB is an 8bit processor). This means that a single tile row is spread across two tile memory addresses. Now, Nintendo chose a funny way of encoding the tile row bits, the less significant byte has the least significant bits of each color index, while the most significant Byte has the most significant bits of each color index. It looks like this:

0x8000	0	1	1	0	1	0	1	1
0x8001	1	0	1	1	0	0	1	0
	2	1	3	2	1	0	3	1

Figure 7 Tile Row pixel color index encoding

7. EXAMPLE OF GPU OPERATION

7.1 Fetch next tile index from VRAM tile Map memory.

The VRAM tile map area is a 32x32 matrix where each entry has the index to of an 8x8 tile. There are two VRAM memory areas reserved to tile maps, 0x9800-0x9BFF called TileMap0 and 0x9C00-0x9FFF called TileMap1. Let's take a look at a memory dump of TileMap0 for the 'Nintendo' logo background.

```
00009800: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
*
00009900: 00 00 00 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 19 00 00 00
00009920: 00 00 00 00 0d 0e 0f 10 11 12 13 14 15 16 17 18 00 00 00 00
00009930: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
*
00009BE0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

As you can see each entry represents an index to an 8x8 tile, most entries have index 0 (which happens to be a blank tile) and then we have entries 1,2,3,4 and so on.

Another way to see it is like this:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	0	0	0
0	0	0	0	D	E	F	10	11	12	13	14	15	16	17	19	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

So, from the previous figure, tile 0 is simply a blank tile, tile 1 is the upper left 'N', tile 2 is the upper right 'N', tile 3 is the top of the 'i' and so on.

7.2 Fetch next tile data from VRAM tile memory.

The tile map memory from the previous section only had indexes to the tiles, but not the actual 8x8 tile pixel data. The next thing that the GPU has to do is, once it obtained the tile index, go and fetch the actual tile data. Let's take for example the tile pixel data corresponding to the tile with index '1'. For tile index '1', the VRAM TileSet memory are looks like this:

```
00008000 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00008010 : f0 00 f0 00 fc 00 fc 00 fc 00 fc 00 f3 00 f3 00
00008020 : 3c 00 3c 00 3c 00 3c 00 3c 00 3c 00 3c 00 3c 00
00008030 : f0 00 f0 00 f0 00 f0 00 00 00 00 00 f3 00 f3 00
```

Ok, now if we convert {f0,00,f0,00,fc,00,fc,00, fc, 00, fc,00,f3,00,f3,00} into an 8x8 tile, then it will look like this:

8010 : f0 11110000 8011 : 00 00000000								
8012 : f0 11110000 8013 : 00 00000000								
8014 : fc 11111100 8015 : 00 00000000								
8016 : fc 11111100 8017 : 00 00000000								
8018 : fc 11111100 8019 : 00 00000000								
801A : f3 11110011 801B : 00 00000000								
801C : f3 11110011 801D : 00 00000000								
801E : f3 11110011 801F : 00 00000000								

Ok, so what is that funny looking shape? Well it is the upper left tile of the 'N' character, like so:

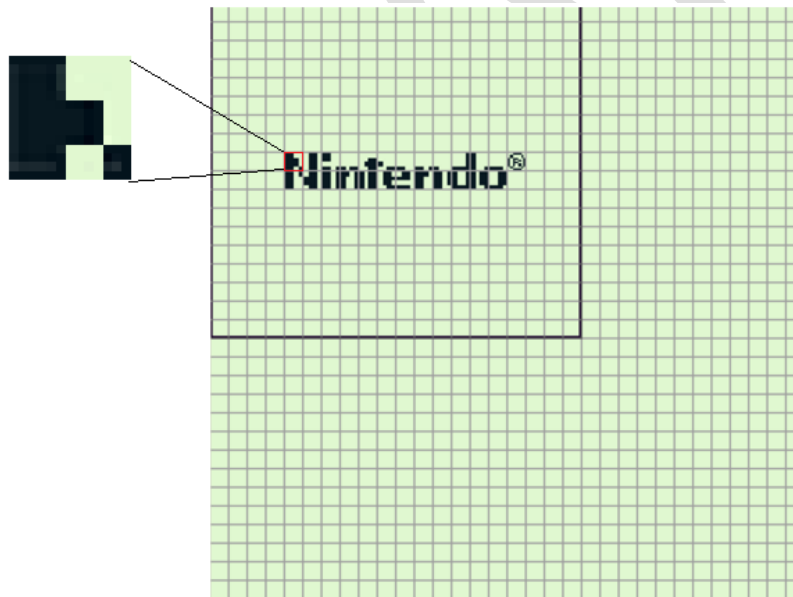


Figure 8 Background Tile Memory

8.1 Apply color palette to background Tile.

The Color palette, is configured by the CPU, by setting the BGP Palette register (0xff47) in this case, the BIOS set the following palette by default:

7:6 Color3	5:4 Color2	3:2 Color1	1:0 Color0
11	11	11	00

8.2 Repeat for all background tiles in the current buffer row.

Remember that the Background consists of 32x32 tiles, so we need to repeat the following steps for all of the 32 tiles in a row (since the GPU works on a tile by tile basis, and so does our LCD controller as we will see later on)

8.3 Fetch OAM to get next sprite in current buffer row.

WIP

8.4 Render sprite into current buffer row.

WIP

8. Scrolling the Background

Most GB games in the day were simple 2D scrolling games. How is the scrolling done? Recall that GB is 160x144 pixels, however the video memory consisted of 32x32 tiles (8x8 pixels each) this gives us an internal canvas of 256x256 pixels. The way programs scroll the background is by using two special registers called SCX (scroll X) and SCY (scroll Y), the next figure illustrates this.

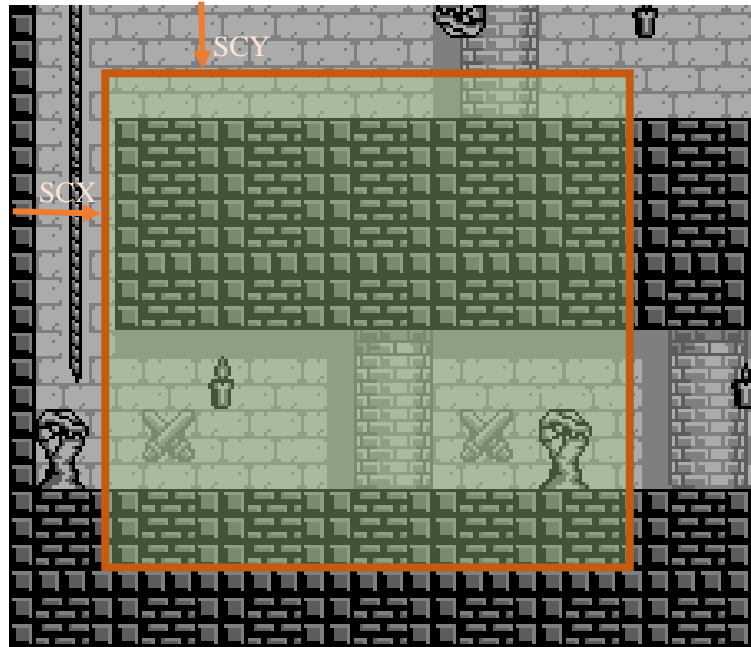


Figure 9 Using SCX and SCY to scroll the background

9.The GPU micro code

I like micro code so much that the GPU also has it. Seriously, it is easier to change it this way, so we can support GameColor or GameBoyAdvanced in the future.

The GPU micro code instruction set is described next:

Nemonic/Value	Syntax	Description
gwr1 5'd1	`gwr1 reg literal	reg = literal Set register with 10bit literal
gwr2 5'd2	`gwr2 reg1 reg2	reg1 = reg2 Assigns reg2 to reg1
gadd 5'd3	`gadd reg1 reg2 reg3	reg1 = reg2 + reg3 Adds two registers
gsub 5'd4	`gsub reg1 reg2 reg3	reg1 = reg2 - reg3 Subtracts two registers
gaddl 5'd5	`gaddl reg literal	reg += literal Adds literal to contents of register
gjnz 5'd6	`gjnz literal	

		Jumps if Zero flags was not set by previous instruction
gwbg 5'd7	`gwbg	Writes contents of special registers {Bh,Bl} to the video buffer
gsubl 5'd8	`gsubl reg literal	reg -= literal Subtracts literal to register
grvmem 5'd9	`grvmem	<code>vmem_data = [vmem_address]</code> Reads contents of special register <code>vmem_address</code> into special register <code>vmem_data</code> .
gshl 5'd10	`gshl reg literal	reg <= literal Shifts left register
ggoto 5'd11	`ggoto literal	Jumps to address
gjz 5'd12	`gjz literal	Jumps if zero flags was set by previous instruction

10. The GPU registers

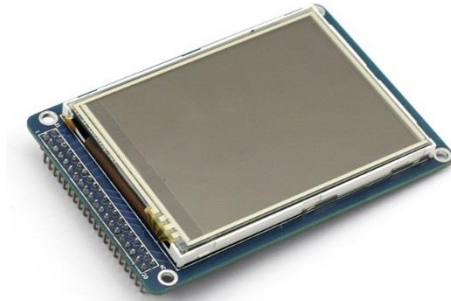
The GPU instruction set from the previous version might suggest yet another general purpose machine, but the secret sause is really in the registers. There are two flavors of registers in the GPU: Special registers and general purpose registers.

Register Name	Size	Description
bh	8	Tile Row High Word Stores the 8 most significant bits of the current tile row.
bl	8	Tile Row Low Word Stores the 8 least significant bits of the current tile row.
bg_buffer_block_sel	8	Not used....
state	8	The current state of the GPU. WIP
cur_tile	8	The index of the current tile. Valid values go from 0 to 255.
r1	16	General purpose register
r2	16	General purpose register

r3	16	General purpose register
tile_row	16	Holds the current tile row. Recall that each tile has 8 rows.
vmem_data	8	Holds the data latched from VMEM from the last `grvmem` instruction.
bgmoffset	16	Has the offset to the back ground map memory, this register already takes into consideration the configuration values from LCDCONFIG.
bg_row_offset	16	Has the offset of the current background row.
bgtoffset	16	Has the offset to the back ground tile memory, this register already takes into consideration the configuration values from LCDCONFIG.
ly_mod_8	16	LYC % 8
vmem_data_shl_4	16	Has the value of <code>vmem_data</code> << 4

4. Display Controller

Port Name	Type	Size bits	Description
iClock	In	1	32Mhz Main Clock
iReset	In	1	Global Reset signal
iRCS	In	1	Register bus chip select from MMU
iRWE	In	1	Register bus write enable from MMU
iRAddr	In	5	Register bus Address
oRData	Out	8	Register bus read data
iRData	Out	8	Register bus write data
iColorIndex	In	2	The color index. Possible values are COLOR1, COLOR2, COLOR3 or COLOR4



11. Overview

The pGB emulates the GB 160x144 LCD display using a 3.2'' TFT LCD 320x240 Display by SainSmart ². The TFT LCD display features SSD1289 driver. The pGB and the SSD1289 communicate using an 8080 16bit parallel interface as shown in Figure 10. The 16bit parallel interface was preferred over SPI serial interfaces in order to preserve original GB 60FPS playability.



Figure 10 pGB SSD1289 - 8080 16bit parallel interface

² <http://www.sainsmart.com/arduino-compatibles-1/lcd-module/sainsmart-3-2-tft-lcd-display-touch-panel-pcb-adapter-sd-slot-for-arduino-2560.html>

http://sainsmart.com/skin/frontend/base/default/document/3_2%20inch%20TFT%20with%20SD%20and%20Touch%20Quickstart.pdf

Pin	Description
/RD	Read Enable
/WR	Write Enable
D/C	Data (0) command (1) selector (a.k.a RS in Sainsmart pinout)
/CS	Chip select
D[15:0]	Data bus

Figure 11 TFT LCD pinout ³

The SSD11289 essentially works as follows⁴:

You can write either commands or data it in the D[15:0] pins.

Writing commands:

The SSD11289 has a set of special registers R00-R24. Each register represents a function that the LCD can do. In order to select the register that you want to write to, you first use a special register called the index register (IR). For instance, if you would like to write to R07 with the value 0x21 then you would do:

a- Set IR to 0x7.

/WR	D/C	DB[15:0]
0	0	0x7

b - Set R07 to 0x21

/WR	D/C	DB[15:0]
0	1	0x21

Writing data:

The SSD11289 features a 172800Byte internal GDDRAM (i.e. the Frame Buffer). For the purposes of pGB, writing data will essentially mean writing into the GRAM. The first thing that you do, is setting the initial GRAMX (R4E) and GRAMY (R4F) registers to specify the initial pixel position that you want to write. Remember that in order to select a Register, you first use the IR.

1. Select initial {X,Y} coordinates using R4E and R4F. (Note that Y coordinate is 9 bits and X coordinate is 8 bits)

$$R4E[7:0] = \text{Coordinate X}$$

³ <http://www.sainsmart.com/zen/documents/20-011-918/3.2LCD+UNO.pdf>

⁴ Diego: Or at least that's how I understood it

R4F[8:0] = Coordinate Y

2. Write Data to GRAM using R22 (this also updates address automatically)
R22[15:0] = {R,G,B}

Since the TFT LCD features a 65k color palette, the RGB mapping into R22 looks like this:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R4	R3	R3	R1	R0	G5	G4	G3	G2	G1	G0	B4	B3	B2	B1	B0

So, we previously mentioned that the next data address is updated automatically each time you write to R22, how is this done? Essentially there are two registers called I/D and AM.

I/D is the increment direction, it specifies if the internal address counter is incremented or decremented by 1 position. AM specifies if address goes up or down (vertically).

The following diagram illustrates this concept:

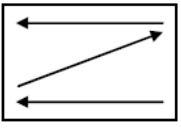
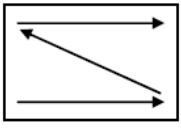
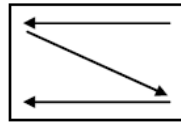
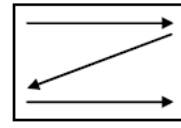


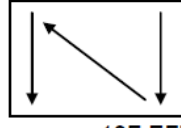
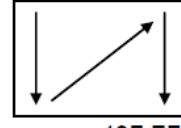
	ID[1:0]="00" Horizontal: decrement Vertical: decrement	ID[1:0]="01" Horizontal: increment Vertical: decrement	ID[1:0]="10" Horizontal: decrement Vertical: increment	ID[1:0]="11" Horizontal: increment Vertical: increment
AM="0" Horizontal	00,00h  13F,EFh	00,00h  13F,EFh	00,00h  13F,EFh	00,00h  13F,EFh
AM="1" Vertical	00,00h  13F,EFh	00,00h  13F,EFh	00,00h  13F,EFh	00,00h  13F,EFh

Figure 12 GRAM address auto-increment operation

Color Format

The original GB featured a 2-bit grayscale palette. This essentially means it had 4 shades of gray (or olive green to be more accurate). Since the SSD11289 supports 16bit color, a mapping to the original palette is required such that the ROM games work properly.

The next table illustrates the color palette.





Display Color Palette	RGB value	pGB 2-bit value
	{156,189,15}	COLOR_WHITE
	{140,173,15}	COLOR_LGRAY
	{49,98,48}	COLOR_DGRAY
	{15,56,15}	COLOR_BLACK

Figure 13 GB 2-bit original color palette

Note from Figure 13 that we use the convention COLOR_WHITE, COLOR_LGRAY, COLOR_DGRAY and COLOR_BLACK. This is because the actual game code does not directly use these colors but instead uses a “color game palette”. This is essentially a mapping between the game colors COLOR1, COLOR2, COLOR3 and COLOR4 and the actual 2-bit pGB colors.

The register at location 0xFF47 is called the “Display device color palette map”. It maps the game colors COLOR1, COLOR2, COLOR3 and COLOR4 to actual colors from Figure 13.

[7:6]	[5:4]	[3:2]	[1:0]
Color3	Color2	Color1	Color0

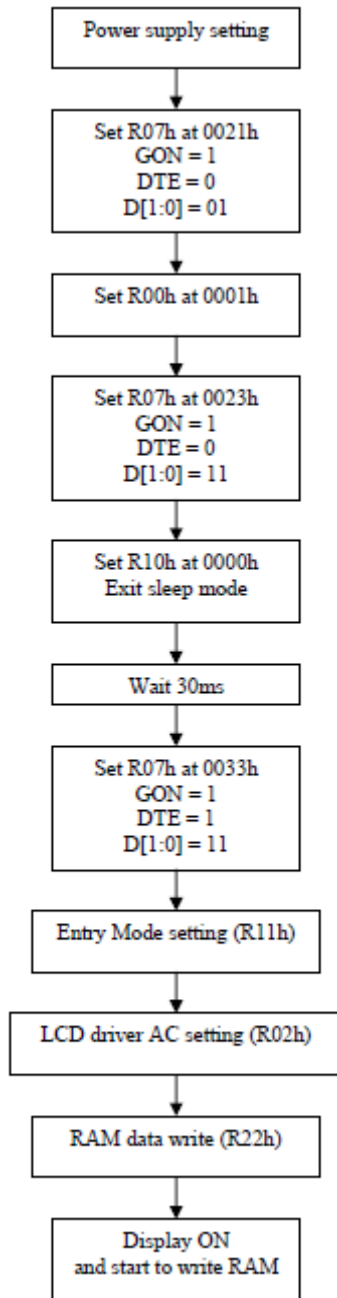
Figure 14 Display color Map - Address Local address 0, MMU address 0xFF47

The games can map any color index to any actual color, they do this achieve certain video effects. For instance, the ROM bootstrap code sets this register to 0xF3. This is COLOR0 = COLOR_BLACK, COLOR2 = COLOR_BLACK, COLOR3 = COLOR_BLACK, COLOR1=COLOR_WHITE, which simply means that the Nintendo boot logo is solid black and white.

The TFT LCD has a 40pin parallel interface, but since we want to save as many of the FPGA GPIO pins as possible, an alternative greyscale mapping is suggested. We have to be careful on which pins we trim since the commands may require those.

12. Display-On Sequence

- 1- The POR sequence of the SSD11289 is described next.



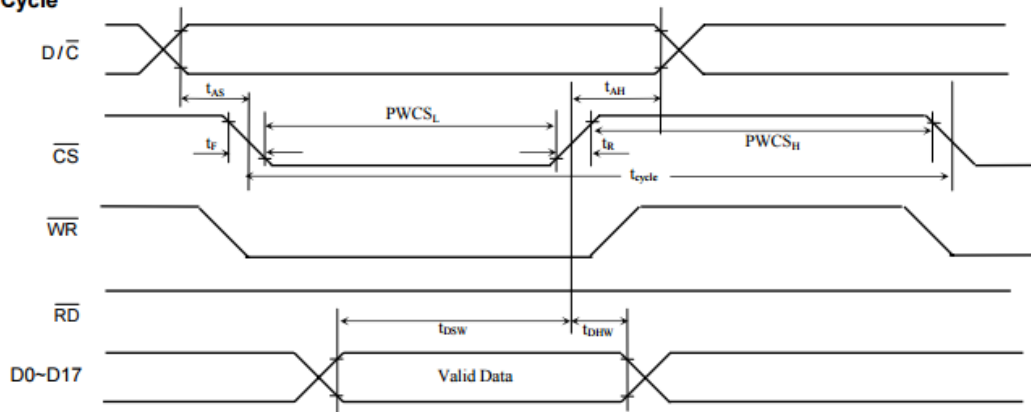
13. Write Cycle timing

Note that generally speaking we don't do reads from the GRAM only writes. This is because of the way the LCD has been wired, need to investigate if this is an issue for GB.

Table 13-2 – Parallel 8080 Timing Characteristics

(T_A = -20 to 70°C, V_{DDIO} = 1.65V to 3.6V, V_{DDEXT} = 1.65V to 1.95V, REGVDD = 'L')

Symbol	Parameter	Min	Typ	Max	Unit
t _{cycle}	Clock Cycle Time (write cycle)	100	-	-	ns
t _{cycle}	Clock Cycle Time (read cycle)	1000	-	-	ns
t _{AS}	Address Setup Time	0	-	-	ns
t _{AH}	Address Hold Time	0	-	-	ns
t _{DSW}	Data Setup Time	5	-	-	ns
t _{DHW}	Data Hold Time	5	-	-	ns
t _{ACC}	Data Access Time	250	-	-	ns
t _{OH}	Output Hold time	100	-	-	ns
PWCS _L	Pulse Width /CS low (write cycle)	50	-	-	ns
PWCS _H	Pulse Width /CS high (write cycle)	50	-	-	ns
PWCS _L	Pulse Width /CS low (read cycle)	500	-	-	ns
PWCS _H	Pulse Width /CS high (read cycle)	500	-	-	ns
t _R	Rise time	-	-	4	ns
t _F	Fall time	-	-	4	ns

Write Cycle

5. Bootstrap ROM

The first milestone of the project is getting the bootstrap code to run. This 256byte code gets executed once the GB is turned on and is located from 0x00 to 0xff. Once this is done the GB jumps to execute the first game ROM instruction that is located at address 0x100.

It essentially does this:

- Initializes the stack.
- Clears video memory areas 0x8000 to 0x9FFF
- Initializes the audio device
- Renders the scrolling Nintendo logo
- Plays the funny beep
- Jumps to the first ROM game instruction.

A very comprehensive guide of the code does can be found here:

<https://realboyemulator.wordpress.com/2013/01/03/a-look-at-the-game-boy-bootstrap-let-the-fun-begin/>

Please note that in order for the bootstrap code to work properly, a valid ROM has to be loaded from the SD card flash into the corresponding internal SRAM area (starting at address 0x100).

The game ROM has special header that must follow some conventions; in particular at addresses 0x104 to 0x133 the ROM has these 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.

The actual contents of the bootstrap code in HEX look like this:

```
0x31, 0xFE, 0xFF, 0xAF, 0x21, 0xFF, 0x9F, 0x32, 0xCB, 0x7C, 0x20, 0xFB,
0x21, 0x26, 0xFF, 0x0E,

    0x11, 0x3E, 0x80, 0x32, 0xE2, 0x0C, 0x3E, 0xF3, 0xE2, 0x32, 0x3E,
0x77, 0x77, 0x3E, 0xFC, 0xE0,

    0x47, 0x11, 0x04, 0x01, 0x21, 0x10, 0x80, 0x1A, 0xCD, 0x95, 0x00,
0xCD, 0x96, 0x00, 0x13, 0x7B,

    0xFE, 0x34, 0x20, 0xF3, 0x11, 0xD8, 0x00, 0x06, 0x08, 0x1A, 0x13,
0x22, 0x23, 0x05, 0x20, 0xF9,

    0x3E, 0x19, 0xEA, 0x10, 0x99, 0x21, 0x2F, 0x99, 0x0E, 0x0C, 0x3D,
0x28, 0x08, 0x32, 0x0D, 0x20,

    0xF9, 0x2E, 0x0F, 0x18, 0xF3, 0x67, 0x3E, 0x64, 0x57, 0xE0, 0x42,
0x3E, 0x91, 0xE0, 0x40, 0x04,

    0x1E, 0x02, 0x0E, 0x0C, 0xF0, 0x44, 0xFE, 0x90, 0x20, 0xFA, 0x0D,
0x20, 0xF7, 0x1D, 0x20, 0xF2,

    0x0E, 0x13, 0x24, 0x7C, 0x1E, 0x83, 0xFE, 0x62, 0x28, 0x06, 0x1E,
0xC1, 0xFE, 0x64, 0x20, 0x06,
```



```

0x7B, 0xE2, 0x0C, 0x3E, 0x87, 0xF2, 0xF0, 0x42, 0x90, 0xE0, 0x42,
0x15, 0x20, 0xD2, 0x05, 0x20,

0x4F, 0x16, 0x20, 0x18, 0xCB, 0x4F, 0x06, 0x04, 0xC5, 0xCB, 0x11,
0x17, 0xC1, 0xCB, 0x11, 0x17,

0x05, 0x20, 0xF5, 0x22, 0x23, 0x22, 0x23, 0xC9, 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, 0x3c, 0x42, 0xB9,
0xA5, 0xB9, 0xA5, 0x42, 0x4C,

0x21, 0x04, 0x01, 0x11, 0xA8, 0x00, 0x1A, 0x13, 0xBE, 0x20, 0xFE,
0x23, 0x7D, 0xFE, 0x34, 0x20,

0xF5, 0x06, 0x19, 0x78, 0x86, 0x23, 0x05, 0x20, 0xFB, 0x86, 0x20,
0xFE, 0x3E, 0x01, 0xE0, 0x50

```

```

LD SP,$fffe          ; $0000 Setup Stack

XOR A                ; $0003 Zero the memory from $8000-$9FFF (VRAM)
LD HL,$9fff          ; $0004
Addr_0007:
LD (HL-),A           ; $0007
BIT 7,H              ; $0008
JR NZ, Addr_0007     ; $000a

LD HL,$ff26          ; $000c Setup Audio
LD C,$11             ; $000f
LD A,$80             ; $0011
LD (HL-),A           ; $0013
LD ($FF00+C),A       ; $0014
INC C                ; $0015
LD A,$f3             ; $0016
LD ($FF00+C),A       ; $0018
LD (HL-),A           ; $0019
LD A,$77             ; $001a
LD (HL),A            ; $001c

LD A,$fc             ; $001d Setup BG palette
LD ($FF00+$47),A     ; $001f

```

```

        LD DE,$0104      ; $0021 Convert and load logo data from cart into Video RAM
        LD HL,$8010      ; $0024
Addr_0027:
        LD A,(DE)        ; $0027
        CALL $0095       ; $0028
        CALL $0096       ; $002b
        INC DE           ; $002e
        LD A,E           ; $002f
        CP $34           ; $0030
        JR NZ, Addr_0027 ; $0032

        LD DE,$00d8      ; $0034 Load 8 additional bytes into Video RAM
        LD B,$08         ; $0037
Addr_0039:
        LD A,(DE)        ; $0039
        INC DE           ; $003a
        LD (HL+),A       ; $003b
        INC HL           ; $003c
        DEC B            ; $003d
        JR NZ, Addr_0039 ; $003e

        LD A,$19         ; $0040 Setup background tilemap
        LD ($9910),A     ; $0042
        LD HL,$992f      ; $0045
Addr_0048:
        LD C,$0c         ; $0048
Addr_004A:
        DEC A            ; $004a
        JR Z, Addr_0055  ; $004b
        LD (HL-),A       ; $004d
        DEC C            ; $004e
        JR NZ, Addr_004A ; $004f
        LD L,$0f         ; $0051
        JR Addr_0048     ; $0053

        ; === Scroll logo on screen, and play logo sound===

Addr_0055:
        LD H,A           ; $0055 Initialize scroll count, H=0
        LD A,$64         ; $0056
        LD D,A           ; $0058 set loop count, D=$64
        LD ($FF00+$42),A ; $0059 Set vertical scroll register
        LD A,$91         ; $005b
        LD ($FF00+$40),A ; $005d Turn on LCD, showing Background
        INC B            ; $005f Set B=1
Addr_0060:
        LD E,$02         ; $0060
Addr_0062:

```

```

        LD C,$0c          ; $0062
Addr_0064:
        LD A,($FF00+$44) ; $0064 wait for screen frame
        CP $90            ; $0066
        JR NZ, Addr_0064 ; $0068
        DEC C             ; $006a
        JR NZ, Addr_0064 ; $006b
        DEC E             ; $006d
        JR NZ, Addr_0062 ; $006e

        LD C,$13          ; $0070
        INC H             ; $0072 increment scroll count
        LD A,H            ; $0073
        LD E,$83          ; $0074
        CP $62            ; $0076 $62 counts in, play sound #1
        JR Z, Addr_0080  ; $0078
        LD E,$c1          ; $007a
        CP $64            ; $007c
        JR NZ, Addr_0086 ; $007e $64 counts in, play sound #2
Addr_0080:
        LD A,E            ; $0080 play sound
        LD ($FF00+C),A    ; $0081
        INC C             ; $0082
        LD A,$87          ; $0083
        LD ($FF00+C),A    ; $0085
Addr_0086:
        LD A,($FF00+$42) ; $0086
        SUB B              ; $0088
        LD ($FF00+$42),A ; $0089 scroll logo up if B=1
        DEC D             ; $008b
        JR NZ, Addr_0060  ; $008c

        DEC B             ; $008e set B=0 first time
        JR NZ, Addr_00E0 ; $008f ... next time, cause jump to "Nintendo Logo check"

        LD D,$20          ; $0091 use scrolling loop to pause
        JR Addr_0060      ; $0093

        ; ==== Graphic routine ====

        LD C,A            ; $0095 "Double up" all the bits of the graphics data
        LD B,$04          ; $0096 and store in Video RAM
Addr_0098:
        PUSH BC           ; $0098
        RL C              ; $0099
        RLA               ; $009b
        POP BC            ; $009c
        RL C              ; $009d

```

```

RLA                ; $009f
DEC B              ; $00a0
JR NZ, Addr_0098  ; $00a1
LD (HL+),A        ; $00a3
INC HL            ; $00a4
LD (HL+),A        ; $00a5
INC HL            ; $00a6
RET               ; $00a7

```

Addr_00A8:

```

;Nintendo Logo
.DB $CE,$ED,$66,$66,$CC,$0D,$00,$0B,$03,$73,$00,$83,$00,$0C,$00,$0D
.DB $00,$08,$11,$1F,$88,$89,$00,$0E,$DC,$CC,$6E,$E6,$DD,$DD,$D9,$99
.DB $BB,$BB,$67,$63,$6E,$0E,$EC,$CC,$DD,$DC,$99,$9F,$BB,$B9,$33,$3E

```

Addr_00D8:

```

;More video data
.DB $3C,$42,$B9,$A5,$B9,$A5,$42,$3C

; ===== Nintendo logo comparison routine =====

```

Addr_00E0:

```

LD HL,$0104        ; $00e0 ; point HL to Nintendo logo in cart
LD DE,$00a8        ; $00e3 ; point DE to Nintendo logo in DMG rom

```

Addr_00E6:

```

LD A,(DE)          ; $00e6
INC DE             ; $00e7
CP (HL)            ; $00e8 ;compare logo data in cart to DMG rom
JR NZ,$fe         ; $00e9 ;if not a match, lock up here
INC HL            ; $00eb
LD A,L             ; $00ec
CP $34             ; $00ed ;do this for $30 bytes
JR NZ, Addr_00E6  ; $00ef

```

```

LD B,$19           ; $00f1
LD A,B             ; $00f3

```

Addr_00F4:

```

ADD (HL)           ; $00f4
INC HL             ; $00f5
DEC B              ; $00f6
JR NZ, Addr_00F4  ; $00f7
ADD (HL)           ; $00f9
JR NZ,$fe         ; $00fa ; if $19 + bytes from $0134-$014D don't add to $00
; ... lock up

```

```

LD A,$01           ; $00fc

```

```
LD ($FF00+$50),A ; $00fe ;turn off DMG rom
```

References

<http://www.righto.com/2014/10/how-z80s-registers-are-implemented-down.html>