The Cycle-Accurate Game Boy Docs

Version 0.X

by Antonio Niño Díaz (AntonioND)

Index

1. Introduction

Since nobody seems to care about documenting the Game Boy good enough to make a cycle-accurate emulator (the only “documentation” there is right now is the source code of the emulator Gambatte, and it requires a lot of time to understand how it works), I've decided to document it myself. The reason is that I'm making my own GB emulator and I just can't find any of the information I want, so I decided to document the GB while I code my own open-source emulator (GiiBiiAdvance) and check if my assumptions are correct. I started with the timer, but I plan to document the complete Game Boy and Game Boy Color.

I'm using information from the Pan Docs (a LOT from here, in fact), the source code of Gambatte and some other documents by other authors (all of them in the credits of this file), but the most important source of knowledge about the Game Boy is the Game Boy itself! That's why I've created lots of test ROMs to verify specific behaviours of the original hardware. I have an original GB, a GB Pocket, two different GB Color, a GB Advance and a GBA SP. While most of the test give the same results in every hardware, there are some things that are different, so I've tried to document them as detailed as I can.

Here are some abbreviations and nomenclature I'll use in this document:

- DMG/GB: Original Game Boy (Dot Matrix Game)
- MGB/GBP: Game Boy Pocket / Game Boy Light
- SGB/SGB2: Super Game Boy / Super Game Boy 2
- CGB/GBC: Game Boy Color
- AGB/GBA: Game Boy Advance
- AGS/GBA SP: Game Boy Advance SP
- Clock: Oscillator clock frequency is 4194304 Hz (8388608 Hz in double speed mode)
- Cycle: CPU cycle frequency is 1048576 Hz (2097152 Hz in double speed mode)

Whenever I write sample code I'll use RGBDS syntax, and I'll use the definitions of the file “Gameboy Hardware definitions” (hardware.inc).

2. Memory

General Memory Map

Addresses	Name	Description
0000h – 3FFFh	ROM0	Non-switchable ROM Bank.
4000h – 7FFFh	ROMX	Switchable ROM bank.
8000h – 9FFFh	VRAM	Video RAM, switchable (0-1) in GBC mode.
A000h – BFFFh	SRAM	External RAM in cartridge, often battery buffered.
C000h – CFFFh	WRAM0	Work RAM.
D000h – DFFFh	WRAMX	Work RAM, switchable (1-7) in GBC mode
E000h – FDFFh	ECHO WRAM	Mirror to C000h – DDFFh
FE00h – FE9Fh	OAM	(Object Attribute Table) Sprite information table.
FEA0h – FEFFh	HOAM	Unusable memory.
FF00h – FF7Fh	I/O Registers	I/O registers are mapped here.
FF80h – FFFEh	HRAM	Internal CPU RAM
FFFFh	IE Register	Interrupt enable flags.

Jump Vectors in ROM0

Some addresses in ROM0 are jump vectors:

0000h, 0008h, 0010h, 0018h, 0020h, 0028h, 0030h, 0038h – For RST instruction of CPU.
0040h, 0048h, 0050h, 0058h, 0060h – Interrupt Vectors (VBL, LCD, Timer, Serial, Joypad)

Unused vectors may be used for whatever purpose the program wants. RST are special instructions like CALL, but they are 1 byte long instructions (CALL are 3 bytes long) and need 2 cycles less.

Cartridge Header in ROM0

The data from 100h to 14Fh of the ROM bank 0 is the ROM header. Contains some information about the program and the cartridge used. Some games don't fill correctly this area, though. For more information, read the chapter about it.

External Memory and Hardware

Areas from 0000h – 7FFFh and A000h – BFFFh are mapped to the cartridge. The first area is usually used for ROM, and the second one is used for RAM and external registers (if any). Writes to ROM area are interpreted by the Memory Bank Controller (MBC) chip.

I/O Register Unreadable Bits

All unreadable bits of I/O registers return 1. In general, all unused bits in I/O registers are unreadable so they return 1. Some exceptions are:

- Some unknown purpose registers. Some bits of them can be read and written.

- The IE register (only the 5 lower bits are used, but the upper 3 can hold any value).

Boot ROMs

All Game Boy models have a small program that is run when the GB is powered on. This program initializes some registers and some memory areas (depending on the model) and perorms some checks with values in the ROM header. So far (2014/11) have been dumped:

- DMG: Mapped to 0000h – 00FFh.

- MGB: Mapped to 0000h – 00FFh. It has been dumped from a Game Boy Pocket, it is unknown if Game Boy Light has a different one. Probably not, since DMG and MGB are only 1 byte different.

- SGB: Mapped to 0000h – 00FFh.

- CGB: Mapped to 0000h – 00FFh and 0200h – 0900h.

3. CPU

Random notes:

- EI instruction enables IME the following cycle to its execution. RETI doesn't have this behaviour, it enables them right away. DI doesn't have any delay either.

- HALT, when it doesn't enter the HALT mode, needs 4 clocks to complete.

HALT Mode

...

STOP Mode

During STOP mode all the Game Boy is halted, even peripherals like the sound. The LCD behaves differently in each hardware.

LCD OFF:

- DMG/MGB: LCD OFF. No sound.

- GBC/GBA: Black screen. Sound.

LCD ON:

- DMG/MGB: White screen. No sound.

- GBC/GBA: White screen. Sound.

Entering STOP Mode correctly

In order to enter it correctly some of the the P1 select bits should be selected (to select buttons to exit STOP mode) and IE should be 00h. This is a sample code to do so:

```
ld  a,[rIE]
ld  b,a     ; Save IE.
xor a,a
ld  [rIE],a ; Clear IE.
ld  a,$00
ld  [rP1],a ; Select both joypad lines to exit with any button.

stop

ld  a,b
ld  [rIE],a ; Restore IE.
```

Speed Switch

CPU speed switch takes 128 × 1024 – 76 clocks (including the clocks to fetch and execute the STOP instruction). It needs the same clock amount when switching from single speed and from double speed. I've measured it with the timer, so the 76 clocks could be there because at some point

during the speed change the clock does weird things. Sample code to switch speed:

```
CPU_fast:
  ld  a,[rKEY1]
  bit 7,a
  jr  z,CPU_switch ; Check if CPU is in double speed mode.
  ret

CPU_slow::
  ld  a,[rKEY1]
  bit 7,a
  jr  nz,CPU_switch ; Check if CPU is in single speed mode.
  ret

CPU_switch:
  ld  a,[rIE]
  ld  b,a       ; Save IE.
  xor a,a
  ld  [rIE],a   ; Clear IE.
  ld  a,$30
  ld  [rP1],a   ; Disable joypad lines.
  ld  a,$01
  ld  [rKEY1],a ; Prepare CPU for speed switch.

  stop          ; Switch speed.

  ld  a,b
  ld  [rIE],a   ; Restore IE.
  ret
```

- If P1 register bits 4 or 5 are selected (set to '0') and any key is pressed during the speed switch, the CPU will hang, probably because of the STOP mode being cancelled.

- During speed switch, no HDMA is executed, even though all peripherals keeps running, because it depends on the CPU.

Undefined Opcodes

Undefined opcodes hang the CPU, but sound (even fade in/out and that kind of things) and video hardware keep working. Probably serial keeps working, but that has not been tested.

The exception is opcodes in the form 10h XXh where XXh is not 00h. That format of opcode (corrupted STOP) will switch the LCD on.

4. Interrupts

Vertical Blanking Interrupt – Vector 0040h

...

LCD STAT Interrupt – Vector 0048h

...

Timer Interrupt – Vector 0050h

This interrupt is requested when TIMA overflows. There is a delay of one CPU cycle between the overflow and the IF flag being set. For more information, read the timer chapter.

Serial Interrupt – Vector 0058h

This interrupt is requested when a serial transfer of 1 byte is complete.

Joypad Interrupt – Vector 0060h

This interrupt is triggered when there is a transition from '1' to '0' in one of the P1 input lines. For that to happen, one or two of the output lines have to be selected (with a '0' in that line). This interrupt happens in all Game Boy models. GBA SP seems to be the only model not affected by key bouncing because of the different buttons used in it.

IME – Interrupt Master Enable Flag

This flag is not mapped to memory and can't be read by any means. The meaning of the flag is not to enable or disable interrupts. In fact, what it does is enable the jump to the interrupt vectors.

```
0 = Disable jump to interrupt vectors.
1 = Enable jump to interrupt vectors.
```

IME can only be set to '1' by the instructions EI and RETI, and can only be set to '0' by DI (and the CPU when jumping to an interrupt vector).

Note that EI doesn't enable the interrupts the same cycle it is executed, but the next cycle:

```
di
ld  a,IEF_TIMER
ld  [rIE],a
ld  [rIF],a
ei
inc a ; This is still executed before jumping to the interrupt vector.
inc a ; This is executed after returning.
ld   [hl+],a
```

FF0Fh – IF – Interrupt Flags (R/W)

Only the 5 lower bits of this register are (R/W), the others return '1' always when read.

```
Bit 4 – Joypad Interrupt Requested (1=Requested)
Bit 3 – Serial Interrupt Requested (1=Requested)
Bit 2 – Timer Interrupt Requested (1=Requested)
Bit 1 – LCD STAT Interrupt Requested (1=Requested)
Bit 0 – Vertical Blank Interrupt Requested (1=Requested)
```

Each bit is set to 1 automatically when an internal signal from that subsystem goes from '0' to '1'. This is specially important in the case of LCD STAT interrupt, as it will be explained in the video controller chapter. The bits in this register can be set or reset manually too, the CPU will handle them the same way as when they are set by a real event.

FFFFh – IE – Interrupt Enable (R/W)

All 8 bits of this register are (R/W), but only the 5 lower ones are used by the interrupt handler.

```
Bit 4 – Joypad Interrupt Enable (1=Enable)
Bit 3 – Serial Interrupt Enable (1=Enable)
Bit 2 – Timer Interrupt Enable (1=Enable)
Bit 1 – LCD STAT Interrupt Enable (1=Enable)
Bit 0 – Vertical Blank Interrupt Enable (1=Enable)
```

Interrupt Handling

Interrupts are checked before fetching a new instruction. If any IF flag and the corresponding IE flag are bot '1' and IME is set to '1' too, the CPU will push the current PC into the stack, will jump to the corresponding interrupt vector and set IME to '0'. If IME is '0', this won't happen.

If 2 or more interrupts are requested at the same time and the corresponding IE bits are set, the vector with lower address has higher priority (vertical blank has the highest priority, joypad the lowest priority).

It takes 20 clocks to dispatch an interrupt.

The correct instruction to return from an interrupt vector is RETI, as it returns and enables interrupts in the same instruction. If the program needs to handle interrupts during an interrupt procedure, they can be enabled again with EI.

The HALT Instruction Behaviour

HALT instruction has three different behaviours depending on IME, IE and IF. It behaves the same way in all Game Boy models.

- IME = 1

    HALT executed normally. CPU stops executing instructions until (IE&IF&1F) != 0. When a flag in IF is set and the corresponding IE flag is also set, the CPU jumps to the interrupt vector. The return address pushed to the stack is the next instruction to the HALT, not the HALT itself.

- IME = 0

    - IE&IF&1F = 0

HALT mode is entered. It works like the IME = 1 case, but when a IF flag is set and the corresponding IE flag is also set, the CPU doesn't jump to the interrupt vector, it just continues executing instructions.

- IE&IF&1F != 0

  HALT mode is not entered. HALT bug occurs: The CPU fails to increase PC when executing the next instruction. This results on weird behaviour. For example:

One byte long instructions:

```
xor  a,a
halt
inc  a       ; PC fails to increase, it is executed twice.
ld   [hl+],a

The equivalent code with a non-bugged HALT instruction would be:

xor  a,a
halt
inc  a
inc  a
ld   [hl+],a ; [hl] is set to 2
```

With instructions that need more than one byte the results can be even worse:

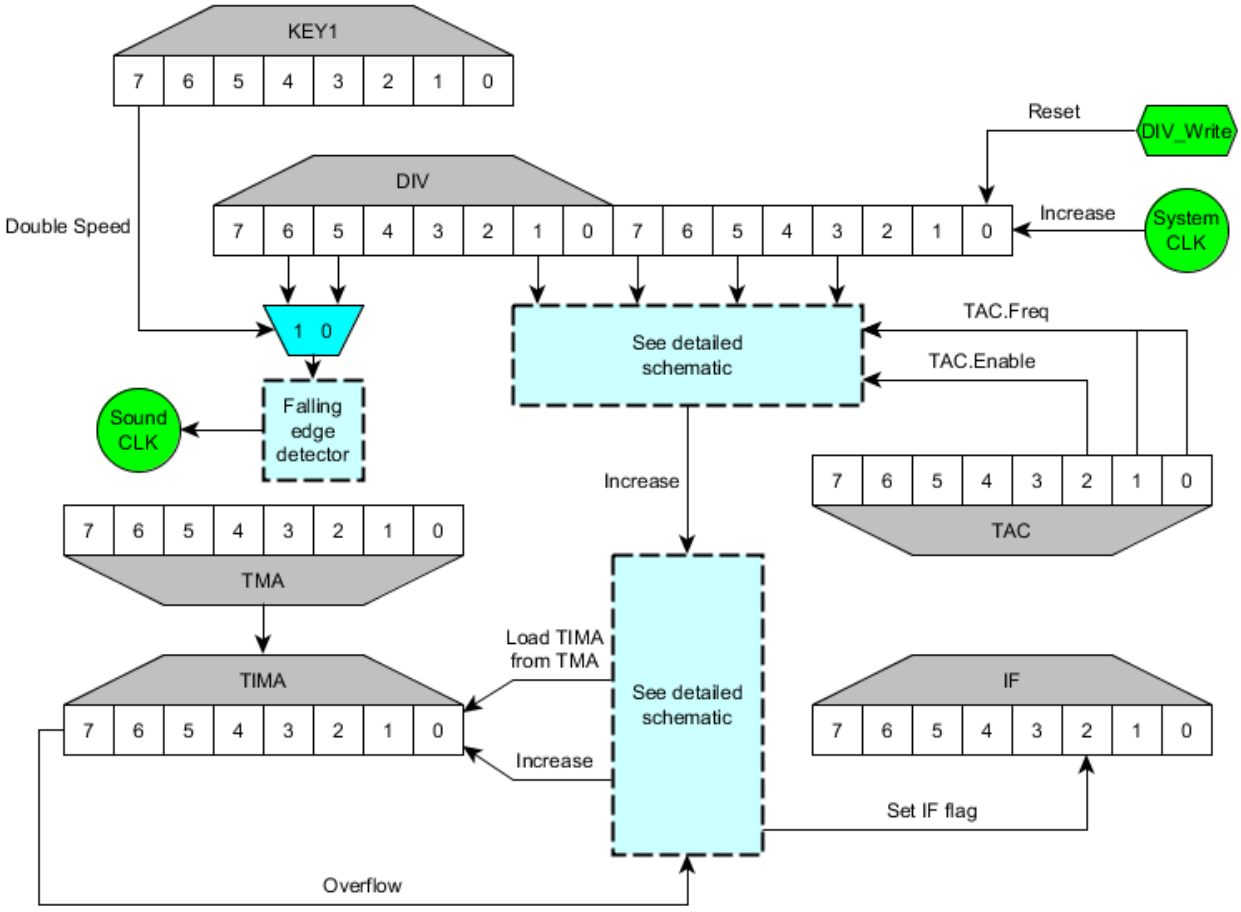
```
xor  a,a
halt
ld   a,$14   ; $3E $14 is executed as $3E $3E $14
ld   [hl+],a

The equivalent code with a non-bugged HALT instruction would be:

xor  a,a
halt
ld   a,$3E   ; $3E $3E
inc  d       ; $14
ld   [hl+],a ; [hl] is set to $3E, and register d is incremented
```

One way to prevent this bug is to always add a NOP instruction after the HALT.

There are no extra clocks as a result of this bug, only the corresponding to the actually executed instructions.

5. Timer

The Game Boy timer subsystem is formed by the three timer registers (TIMA, TMA and TAC), and the DIV register. This is a simplified schematic:



FF04h - DIV - Divider Register (R/W*)

- It works by using an internal system 16 bit counter. The counter increases each clock (4 clocks per nop) and the value of DIV is the 8 upper bits of the counter: it increases every 256 oscillator clocks.

- The value of DIV is the actual bits of the system internal counter, not a mirror, not a register that increases with the system internal counter: The actual bits of the counter mapped to memory.

- DIV overflows at 64 Hz in normal speed mode, 128 Hz in double speed mode.

- It can be written, but its value is reset to 0 no matter what the value written is. In fact, the whole system internal counter is set to 0. DIV is just its the 8 MSB of that counter, so DIV is set to 0 too.

- Serial clock is not derived from this counter, it is not affected by DIV register.

Interaction with APU

- The APU uses the DIV to update sweep (channel 1), fade in/out and time out, the same way the timer uses it to update itself.

- In normal speed mode the APU updates when bit 5 of DIV goes from 1 to 0 (256 Hz). In double speed mode, bit 6.

- Writing to DIV every instruction, for example, will make the APU produce the same frequency with the same volume even if sweep and fade out are enabled.

- Writing to DIV doesn't affect the frequency itself. The waveform generation is driven by another timer.

FF05h - TIMA - Timer Counter (R/W)

This register holds an 8 bit value that is the timer counter, it increases at a certain frequency until it overflows. In that moment, the value in register TMA is loaded into TIMA and an interrupt is requested if the corresponding IE flag is set and IME is set. The interrupt vector address is 0050h.

FF06h - TIMA - Timer Modulo (R/W)

The 8 bit value in this register is loaded into TIMA when it overflows.

FF07h - TAC - Timer Control (R/W)

This register enables/disables the timer and sets its frequency.

```
Bit    2 - Timer Enable (0=Disable, 1=Enable)
Bits 1-0 - Main Frequency Divider
             00:   4096 Hz (Increase every 1024 clocks)
             01: 262144 Hz (   “       “     16 clocks)
             10:  65536 Hz (   “       “     64 clocks)
             11:  16386 Hz (   “       “    256 clocks)
```

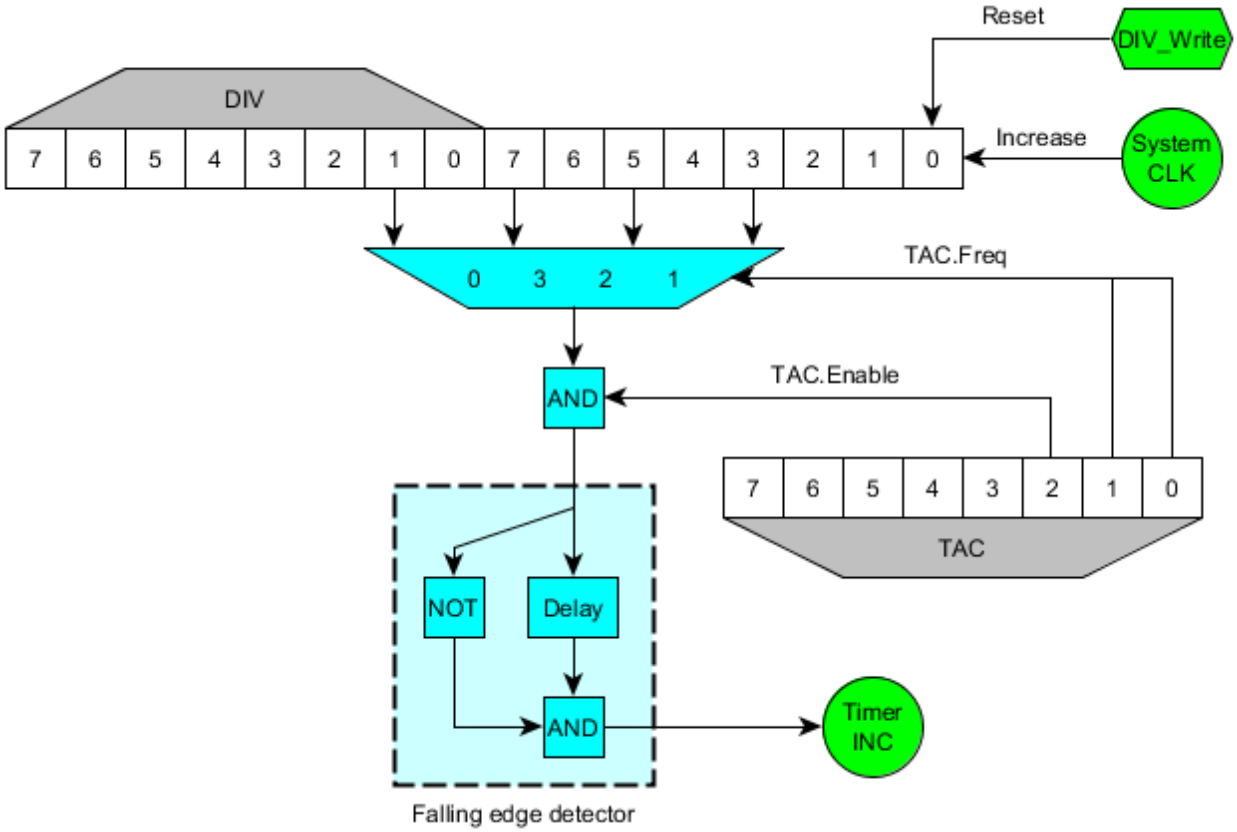
Only the lower 3 bits are (R/W).

Timer Obscure Behaviour

The signal used to increase the TIMA register is generated in a weird way. It selects the actual value of one bit of the system internal counter, and performs an and operation with the enable bit in TAC. This means that writing to the DIV register affects the timer too (writing to timer registers doesn't affect the DIV register).

- For example, if DIV register is written every 12 clocks, even in the fastest timer configuration, it will never increase. If the timer is configured to increase every 64 clocks but the program writes to it every 20 clocks, it will never increase (but it would if it was configured to increase every 16 clocks).

The next picture is a schematic of the relationship between the DIV register and the timer in a DMG or MGB for a better understanding of its behaviour:



As you can see, TAC register only selects the bit of the system internal counter that will be used to increase the timer counter. Then, it performs a logical AND operation with the enable bit in TAC and that signal goes to a falling edge detector.

For example, if timer is configured to increase every 64 clocks, TAC will select bit 5 of the system internal counter. As soon as it overflows from bit 5 and goes back to 0, the falling edge detector will trigger a pulse to increase TIMA register.

This circuit has three problems:

- When writing to DIV register the TIMA register can be increased if the counter has reached half the clocks it needs to increase, since the selected bit by the multiplexer will go from 1 to 0 (which is a falling edge, that will be detected by the falling edge detector).

- When disabling the timer, if the corresponding bit in the system counter is set to 1, the falling edge detector will see a change from 1 to 0, so TIMA will increase. This means that whenever half the clocks of the count are reached, TIMA will increase when disabling the timer.

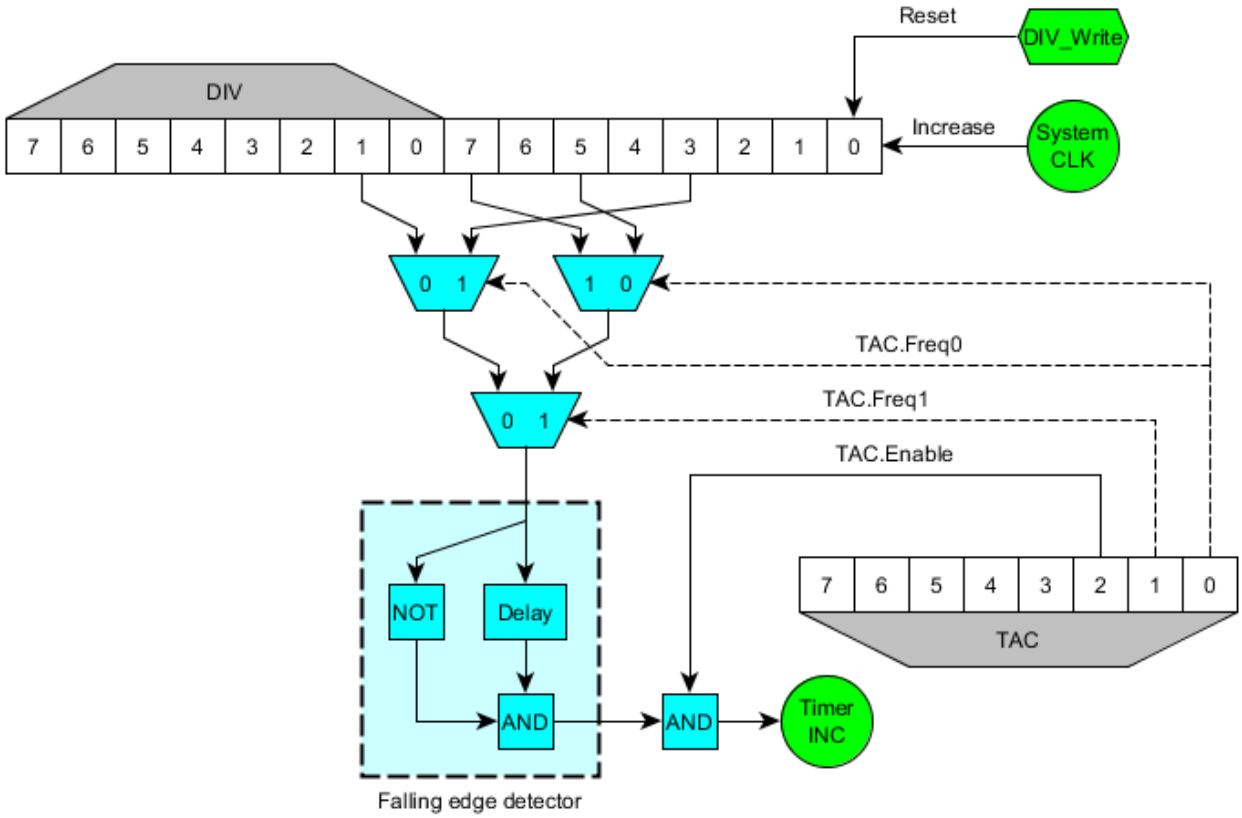
- When changing TAC register value, if the old selected bit by the multiplexer was 0, the new one is 1, and the new enable bit of TAC is set to 1, it will increase TIMA.

This behaviour is a lot glitchier in newer models. DMG and MGB behave the same way (SGB and SGB2 haven't been tested), but GBC, AGB and AGS (GBA SP) are completely different. In fact, different revisions of the GBC have a different behaviour. The DMG behaviour when writing to TAC is this:

```
IF old_enable == 0 THEN
     glitch = 0 (*)
ELSE
     IF new_enable == 0 THEN
          glitch = (sys_clocks & (old_clocks/2)) != 0
     ELSE
          glitch = ((sys_clocks & (old_clocks/2)) != 0) && ((sys_clocks & (new_clocks/2)) == 0)
     END IF
END IF
```

For example, if the old TAC value was 6, old_clocks is 64 and old_enabled is 1. If the written TAC value is 0, new_clocks is 1024 and new_enabled is 0.

The sentence marked with a (*) has a different behaviour in GBC (AGB and AGS seem to have strange behaviour even in the other statements). When enabling the timer and maintaining the same frequency it doesn't glitch. When disabling the timer it doesn't glitch either. When another change of value happens (so timer is enabled after the write), the behaviour depends on a race condition, so it cannot be predicted for every device. One possible explanation is that the AND gate corresponding to the enable bit is placed after the edge detector, but I'm not sure about this, so the next schematic shouldn't be trusted:



Because all of this, timer initialization should be done carefully. A safe way to set up the timer is:

```
ld  a,TACF_16KHZ    ; Change this for the desired frequency.
ld  [rTAC],a        ; Disable timer and load new frequency.
ld  a,__TMA_VALUE__ ; __TMA_VALUE__ is any byte value.
ld  [rTMA],a        ; Load modulo value in both
ld  [rTIMA],a       ; TMA and TIMA registers.
ld  a,TACF_16KHZ|TACF_START
```

```
ld  [rDIV],a        ; Synchronize system counter.
ld  [rTAC],a        ; Enable timer.
```

If the timer interrupt is going to be used:

```
ld  b,~IEF_TIMER
ld  a,[rIE]
and a,b
ld  [rIE],a         ; Clear IE timer flag.

xor a,a             ; a = 0
ld  [rTIMA],a       ; Set TIMA to 0
ld  a,TACF_16KHZ
ld  [rTAC],a        ; Disable timer and load new frequency.
ld  a,__TMA_VALUE__ ; __TMA_VALUE__ is any byte value.
ld  [rTMA],a        ; Load modulo value in both
ld  [rTIMA],a       ; TMA and TIMA registers.

ld  a,[rIF]
and a,b
ld  [rIF],a         ; Clear IF timer flag.
ld  b,IEF_TIMER
ld  a,[rIE]
or  a,b
ld  [rIE],a         ; Set IE timer flag.

ld  a,TACF_16KHZ|TACF_START
ld  [rDIV],a        ; Synchronize system counter.
ld  [rTAC],a        ; Enable timer.
```

Timer Overflow Behaviour

When TIMA overflows, the value from TMA is loaded and IF timer flag is set to 1, but this doesn't happen immediately. Timer interrupt is delayed 1 cycle (4 clocks) from the TIMA overflow. The TMA reload to TIMA is also delayed. For one cycle, after overflowing TIMA, the value in TIMA is 00h, not TMA. This happens only when an overflow happens, not when the upper bit goes from 1 to 0, it can't be done manually writing to TIMA, the timer has to increment itself.

For example (SYS is the system internal counter divided by 4 for easier understanding, each increment of the graph is 1 cycle, not 1 clock):

Timer overflows:

				[A]	[B]		
SYS	FD	FE	FF	00	01	02	03
TIMA	FF	FF	FF	00	23	23	23
TMA	23	23	23	23	23	23	23
IF	E0	E0	E0	E0	E4	E4	E4

Timer doesn't overflow:

				[C]			
SYS	FD	FE	FF	00	01	02	03
TIMA	45	45	45	46	46	46	46
TMA	23	23	23	23	23	23	23
IF	E0	E0	E0	E0	E0	E0	E0

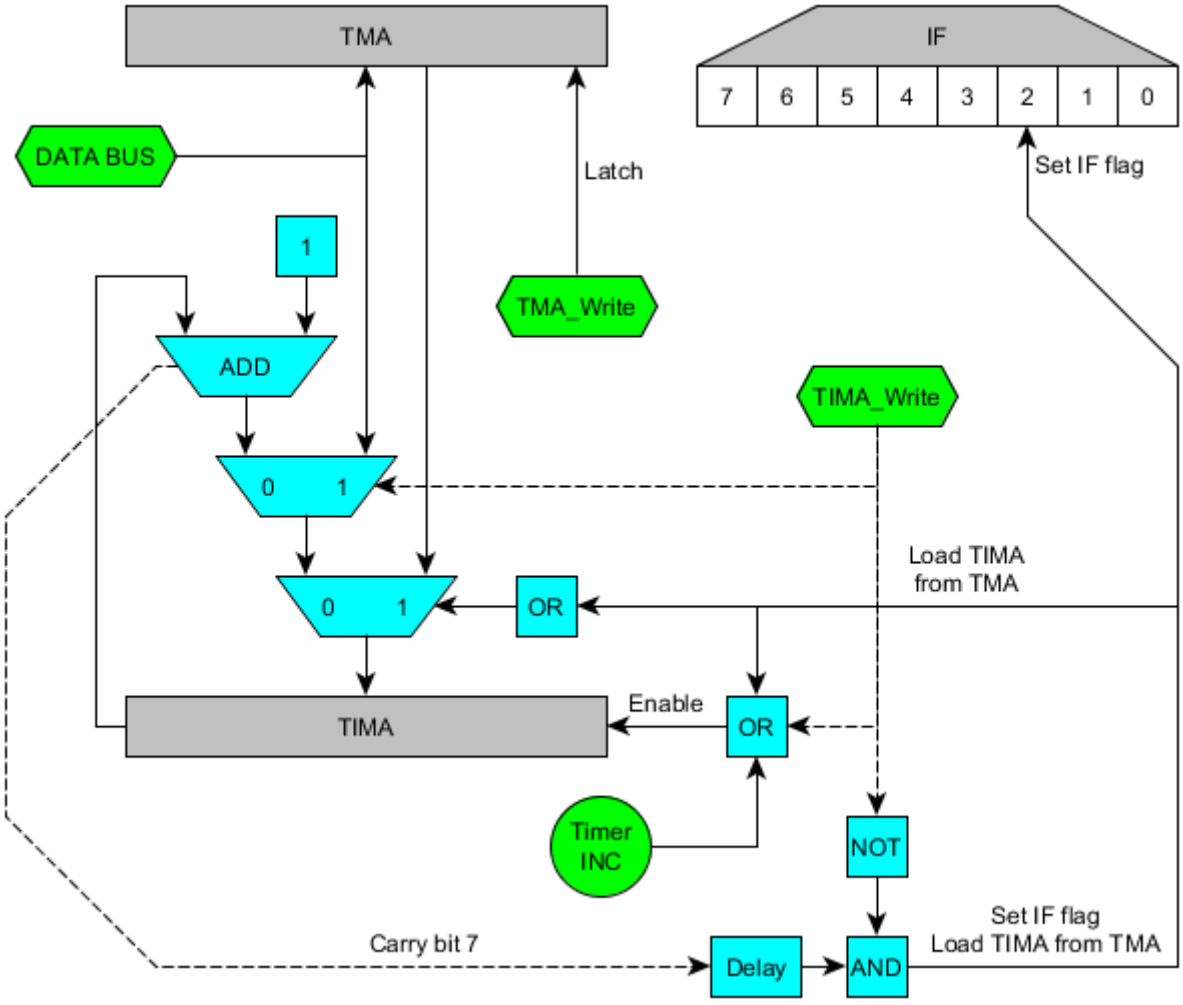
- During the strange cycle [A] you can prevent the IF flag from being set and prevent the TIMA from being reloaded from TMA by writing a value to TIMA. That new value will be the one that stays in the TIMA register after the instruction. Writing to DIV, TAC or other registers won't prevent

the IF flag from being set or TIMA from being reloaded.

- If you write to TIMA during the cycle that TMA is being loaded to it [B], the write will be ignored and TMA value will be written to TIMA instead.

- If TMA is written the same cycle it is loaded to TIMA [B], TIMA is also loaded with that value.

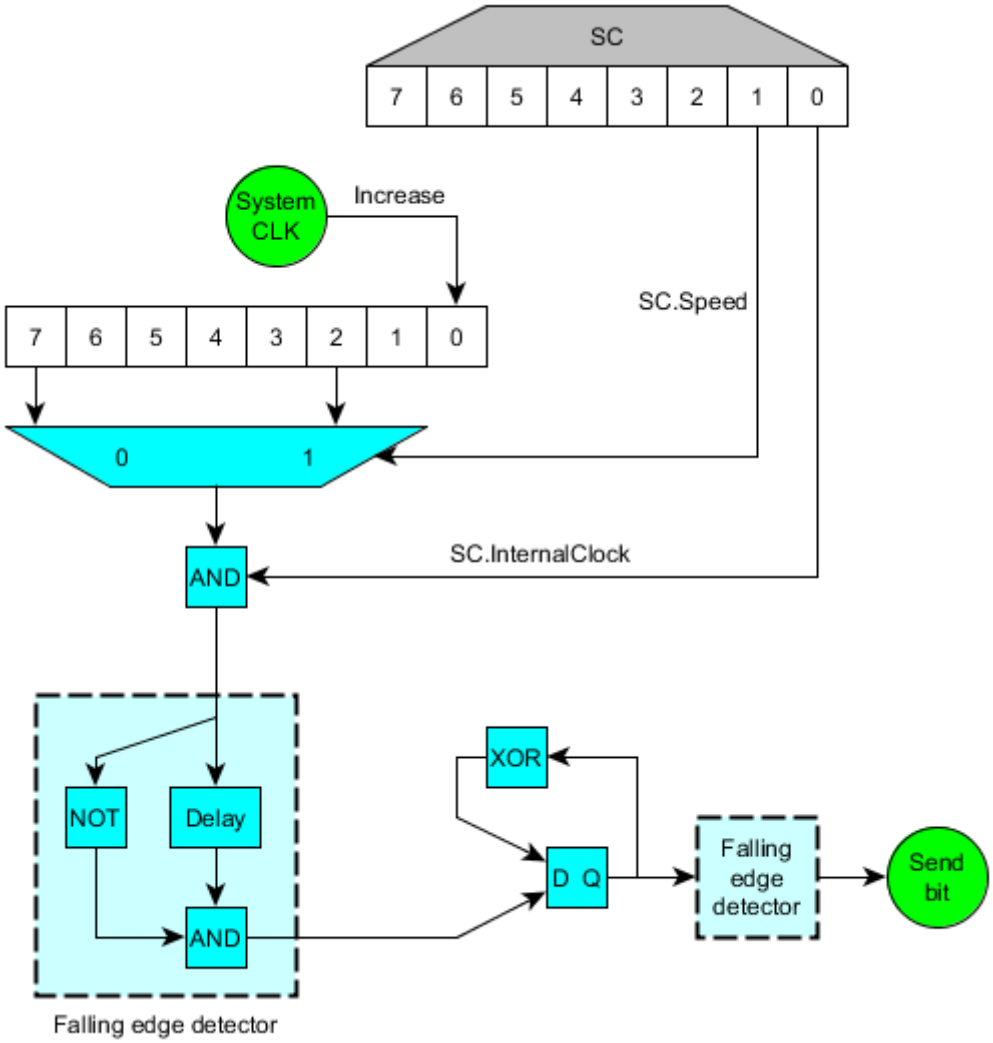
- This is a guessed schematic to explain the priorities with registers TIMA and TMA:



TMA is a latch. As soon as it is written, the output shows that value. That explains that when TMA is written and TIMA is being incremented, the value written to TMA is also written to TIMA. It doesn't affect the IF flag, though.

6. Serial

The serial hardware works with an internal timer that can't be reseted by any means. This is a diagram of the control part of the serial:



TODO: check, document... everything that goes here…

7. Video controller

The LCD

What happens when turning it on/off?

8. DMA

...

- During speed switch, no HDMA is executed, even though all peripherals keeps running, because it depends on the CPU.

….

9. Audio processing unit

10. The Game Boy Cartridge

11. Credits

- Pan Docs, by “Pan of Anthrox”, with contributions from Marat Fayzullin, Pascal Felber, Paul Robson, “kOOPa” and (a lof from) Martin Korth “nocash”.

- The people at the IRC channel #gbdev in EFnet. Specially beware, for BGB emulator, which was really useful when designing test ROMs before checking them in real hardware.

- Jonathan Gevaryahu “Lord Nightmare” for GBSOUND.txt.

- Carsten Sorensen for RGBDS and Anthony J. Bentley for maintaining it in Github.

- Sindre Aamås for Gambatte and its source code (Used for some details about sound emulation).

- GeeBee for the GB DEV FAQs.

- Otaku No Zoku for the Gameboy Crib Sheet.