FPGA Plausible Gameboy Accelerator Final Report

Reng Zheng (rengz@mit.edu), Colin Clark (colclark@mit.edu)

***Abstract*—This paper describes Gameboy hardware behavior that the community has reverse engineered, our choices in emulating said hardware, and the motivations behind those choices. We hope this serves as a centralized repository of knowledge for future attempts to emulate the Gameboy in FPGA-space, which is useful if one wanted a portable Gameboy emulator that did not involve expensive microcontrollers.**

***Index Terms*—Digital systems, Emulation, Field programmable gate arrays, Archival systems**

I. Introduction

GAMEBOY emulation is one of the beginnings of a long tradition of game emulation; with the level of complexity present in a first-generation ROM-swappable gaming device being interesting enough to pose a challenge, but simplistic and more importantly well-documented enough to not pose issues of the more esoteric consoles like the Atari 2600 [1]. Additionally, the field of emulation as a whole is taking off due to the desire to preserve old games and other software running on now defunct software. Thus, we are pursuing emulating a Gameboy in FPGA to both get better at FPGAs while also providing a way to preserve Gameboy software in hardware. Below is our write-up to centralize knowledge of how the Gameboy works along with some implementation choices we made.

II. Components

The Gameboy architecture is a chimera of three components: the CPU, Pixel Processing Unit (PPU), and the Memory Bus. The CPU handles macro instructions, such as loading and unloading states from the ROM into the VRAM and processing the game's physics and non-playable character interactions. The PPU handles rendering the game, and communicates to the CPU through a series of memory-mapped status and control registers, known as LCD Control (`LCDC`) and LCD Status (`STAT`). Finally, the Memory Bus connects the CPU, PPU, and I/O like the buttons, game cartridge, and Link Cable [2].

A. The Central Processing Unit

The Central Processing Unit (CPU), compared to the upcoming PPU, has a relatively simple structure due to its lowered clock speed (evaluating instructions after each 1Mhz *M-Cycle* versus the PPU's reliance on 4Mhz *T-Cycles*) and its lack of Direct Memory Access (DMA). For this reason, it can be abstracted into a lookup table and a set of registers, along with one input and three outputs.

1) Instruction Lookup Table: The "DMG" CPU used to power the Gameboy uses an instruction set derived from both the Intel 8080 and Zerilog z80, comprising of 512 ($2 \cdot 16^2$) instructions or "opcodes".[3] As the Opcode needs to be able to be referenced in only eight bits, the width of a standard location in memory, a second set of 256 commands is locked behind the prefix `$CB`. For example, in order to set the 5th bit of register E to 0, otherwise referred to with the mnemonic `BIT 5,E`, the code `$CB` is followed by `$6B`. This command is just one of many, with other Opcodes ranging from basic arithmatic between registers (eg. `ADD`, `SUB`, `MOD`) to bit shifting a value in VRAM to the right (ie. `$38`). Furthermore, Opcodes can be followed by one or more additional bytes signifying parameters such as literals (both 8-bit and 16-bit) as well as memory locations (16-bit). As the instruction length grows longer, however, the need for optimization becomes apparent. The most straightforward of these includes overlapping the execution of commands, as otherwise a cycle would be wasted fetching the next command, when this could be done simultaneously with the final step of the previous command.

2) Registers: Many of the instructions supported by the CPU expect to have ready-access to data within the same clock-cycle, warranting the need for certain parts of memory to be held within the CPU as latch-based "registers" as compared to the Block RAM-based memory used elsewhere. In practice, these are implemented as six 8-bit registers (`B`, `C`, `D`, `E`, `H`, and `L`) and three 16-bit registers (`AF`, `SP`, and `PC`). The former may be accessed individually or together by instructions, while the latter serve specific purposes requiring 16 bits. `AF` holds various flags used to modify how data is accessed and processed. `SP` holds the stack pointer, needed to keep track of state during jumps. `PC` holds the stack pointer, which directly points to areas in memory holding both read-only instructions as well as working RAM (WRAM).

3) Operation: As the driver of the emulator as a whole, the CPU is hard-coded to read the instruction at `0x0000`, execute the instruction, and increment the program counter (`PC`) to move onto the next. To enable this functionality, the CPU module has an output wire `memory_address` which informs the Memory Controller what part of memory it needs to be access, routing the request to either a BROM or latch. The controller then returns this value in the next cycle via the 8-bit`data_in`. Oftentimes data will simply be moved between internal registers, but when the CPU needs to modify data in memory, it sets the `is_writing` wire high and sets `data_out` wire to the corresponding value needing to be

written.

B. The Pixel Processing Unit

The Pixel Processing Unit (PPU) is the dedicated hardware renderer of the GameBoy, and it interfaces with the rest of the Gameboy components through its VRAM and exposed registers [4]. These registers control the functionality of the PPU's rendering, which are described in further detail in section II-B4. Due to the constraining nature of emulation, the paper will first describe the intended behavior that is being emulated, before diving into how that emulation is achieved.

1) LCD Specifications: The PPU renders to a 160 wide by 144 tall monochromatic LCD. Due to a relic of the way CRTs work, the PPU communicates to the LCD using scanlines [5].

2) The Tile and Palette: The PPU's primary building block is the tile: an 8 by 8 pixel representation, which means the viewport is 20 tiles by 18 tiles. This representation is used to compress the amount of VRAM needed to render each frame [5], leveraging the repetition present in certain game elements. Each tile consists of 16 bytes, as each row of 8 pixels is encoded as 2 bytes, with the n^{th} pixel's value corresponding to the n^{th} pixel in the first and second bytes, with the first byte being the LSB. This creates 4 possible states the pixel can be in, which is interpreted by one of two specified palettes, OBP0 and OBP1, as some monochromatic value for the LCD to take. The specific palette used is determined by the source the pixel came from, described in II-B3.

Additionally, the tiles are stored together in memory at `$8000` to `$97FF` and referenced later by the logical layers comprising the PPU, explained in II-B3. There are two ways to find a tile in this space, the `8000` method, which uses `$8000` as the base pointer and an unsigned 8-bit integer representing the tile number to fetch a given tile through `8000+tile_num`, or the `8800` method, which is the same as the `8000` method but uses `$8800` as the base pointer and a *signed* 8-bit integer to represent the tile number. Whichever mode is chosen is defined by bit 4 of the LCDC register, explained in II-B4.

3) Layers: The PPU primarily consists of three logical rendering layers, the Background, Window, and Sprite layers. The Background layer is a wrapping 32 tile by 32 tile. There can be two backgrounds loaded at any given time, denoted in memory locations `$9800` to `$9BFF` and `$9C00` to `$9FFF`. Each background map has a byte pointing to a specific tile number, in the order of left-to-right, top-to-bottom, meaning the first byte corresponds to the top-left tile, and the 33rd byte corresponds to the tile below it. Which background is used is defined by bit 3 of the `LCDC` register, explained in II-B4. The palette associated with a background is defined through register `BGP` , mapped to `$FF47`. Every two bits forms a palette entry, with the lowest corresponding to Palette ID 0. Each entry maps that Palette ID to one of four colors: White, Light Grey, Dark Grey, and Black, in that order.

The Window layer is also a 32-tile by 32-tile grid like the background. However, the Window is non-wrapping, unlike the background, and if a Window pixel and Background pixel want to display at the same location on the screen, the Window pixel takes precedence. If a Window layer is to be displayed, as determined by bit 5 of the `LCDC` register, it exists in one of the aforementioned memory locations allocated to the background. The PPU knows which background memory location corresponds to the Window through bit 6 of the `LCDC` register. The top-left pixel of the window is placed in the global coordinate system according to the memory-mapped `WY` and `WX` registers. These registers are explained further in II-B4.

Finally, the Sprite layer exists outside of the grid system the Background and Window layer share, along with the two sections of memory. The Gameboy stores actively used sprites in the Object Attribute Memory (OAM), which is assigned to addresses `$FE00` to `$FE9F`. Each sprite is represented by 4 bytes, meaning the `OAM` can hold 40 sprites simultaneously. The sprites are stored as follows: byte 0 is the Y-Position of the sprite, but to support on-screen scrolling, 0 on the global y-axis is 16 for the sprite. Byte 1 is the X-Position; similarly, 0 on the global x-axis is 8 for the sprite. Byte 2 is the tile number corresponding to the sprite, and sprites always use the `8000` addressing method. The final byte corresponds to flag sprites, with the least significant 4 bits corresponding to Gameboy Color functions, Bit 4 corresponding to the palette used (0 for OBP0, 1 for OBP1), Bit 5 (X-Flip) corresponding to rendering the sprite horizontally flipped if set to 1, Bit 6 (Y-Flip) corresponding to rendering the sprite vertically flipped, and Bit 7 (OBJ-to-BG Priority) determining whether the sprite is rendered above the background. For Bit 7, if it is set to 0, the sprite is always rendered above the background, but if it is set to 1, Background colors 1-3 supersede the sprite, while the sprite supersedes Background color 0 [4].

Additionally, the sprite layer has "Tall Sprite Mode," where each sprite consists of two tiles on top of each other rather than one. The tile number of the top tile is the tile number in the OAM entry with the LSB set to 0. The tile number of the bottom tile is the tile number in the OAM entry with LSB set to 1. This mode is enabled by Bit 2 of the `LCDC`, described in II-B4.

4) Memory-Mapped Items: The PPU interfaces with the rest of the Gameboy through its memory-mapped components. Noted above, VRAM and OAM memory is already mapped to spots in the memory, allowing the CPU to access and thus command what the PPU displays. Additionally, the PPU contains registers the CPU can modify through a memory mapping to control the behavior of the PPU. As mentioned in the section on layers, some of these exposed registers are the `WY` and `WX` registers, mapped to the memory address `$FF4A` and `$FF4B`, respectively, which control where the top-left of the Window layer is. The `WX` register is strange, as to place the window at the far left border of the screen, the value 7 must be written to the register, so the effective position of the Window layer on the x-axis is `WX` $-$ 7. It is noted that strange hardware bugs occur when `WX` takes on values less than 7 and 166, specifically [6]. However, they are inconsequential for most games.

Additionally, the OBP0 and OBP1 palette registers for objects are also mapped to memory locations `$FF48` and `$FF49`, which works exactly like the `BGP` register, except that the lowest 2 bits are ignored, since Palette ID 0 on a sprite

corresponds to that pixel in the sprite being transparent. The last of the PPU rendering-related registers are SCYand SCXat $FF42 and $FF43 respectively, which controls the top-left position of the viewport with which to render.

The PPU is also controlled through the LCDC register, mapped to $FF40. The 0th bit is the Background/Window Enable Bit. If this bit is 0, no background nor window is drawn and all pixels are set to white (Color 0) except for the sprite pixels, which are unaffected. The 1st bit is the Sprite Enable Bit, if it is set to 0, all sprites are hidden, otherwise they are rendered as normal. The 2nd bit is the Sprite Size Bit, where setting it to 1 enables "Tall Sprite Mode." The 3rd bit is the BG Tile Map Select bit, where 0 sets the Background layer to use the section starting at $9800 and 1 the section at $9C00. The 4th bit is the Tile Data Select bit, where 1 corresponds to using the 8000 method of reading tiles, and 0 8800 . Bit 5 is the Window Display Enable bit, where 0 disables Window rendering, ignoring the layer entirely. Bit 6 is the Window Tile Map Select, which corresponds to which memory section the Window layer uses in the same fashion as the BG Tile Map Select bit. Finally, Bit 7 is the LCD Display Enable bit, if it is 1, the PPU runs as normal, but if it drops to 0, the screen goes blank, the PPU immediately stops all operation, and then the PPU resets itself.

Finally, the PPU's internal state and interpretations of that state is controlled through the STAT register, mapped to $FF41. Bits 0 and 1 encode which mode the PPU is in: H-Blank, V-Blank, OAM-Search, and Drawing in that order. Bit 2 corresponds to whether the LY register internal to the PPU, tracking which scanline the PPU is on, equals LYC, a register mapped to $FF45. Bit 3 controls "Mode 0" STAT interrupts, or an interrupt when the PPU enters H-Blank when Bit 3 is set to 1. Like Bit 3, Bit 4 and 5 control "Mode 1" and "Mode 2" STAT interrupts, respectively, which corresponds to V-Blank and OAM-Search. Bit 6 controls LY=LYC STAT interrupts, which is an interrupt when LY=LYC, when Bit 6 is set to 1. Bit 7 is unused and always set to 1.

5) Modes: The PPU has 4 modes, associated with an enum, associated with H-Sync, V-Sync, OAM-Scan and Draw. H-Sync and V-Sync serve the same function as in CRT pixel-drawing: to sync the frame drawing with other states. H-Sync pads each scanline to 456 T-Cycles (see II-B6) and V-Sync provides 10 additional scanlines for the CPU to interact with VRAM and OAM, for reasons we describe later in . The OAM-Scan consists of 80 T-Cycles, where it spends 2 T-Cycles per sprite seeing if it is going to be drawn on the current scanline. It can note up to 10 sprites to be drawn per scanline, and puts them into a buffer.

The drawing phase is the most complex of the modes. Every T-Cycle, it must push a pixel to the LCD. Every M-Cycle (see II-B6) a new tile is ingested for this scanline. The ingested tile is pushed into the Pixel FIFO, which is used to buffer the LCD from the tile ingester. Here, in the Pixel FIFO, there is a Sprite FIFO and a Background FIFO. The Sprite FIFO ingests its data from the corresponding sprite that will be immediately drawn, while the Background FIFO loads from either the Background or the Window layer, depending on if the Window is being displayed and overlays the Background at a given point. Finally, the Pixel FIFO decides from which underlying FIFO it wraps to output from by determining first if sprites are being drawn (see II-B4), and then determining if a Sprite is in the foreground or background. If it's in the foreground, the sprite FIFO output always trumps the background FIFO output, except when it maps to Palette ID 0, where it is transparent. Otherwise the sprite is in the background, only trumping the Background FIFO in places where the Background FIFO is transparent.

6) Achieving Emulation: Notice how above, a lot of the commands assume sub-cycle/instantaneous memory read speeds, which is impossible to achieve on our given architecture, with BRAM reads occurring every 2 cycles and register updates taking one cycle. However, we can start leveraging the fact that our FPGA runs at a whopping 100MHz clock cycle, 25x faster than the 4MHz time cycles (T-Cycles) that serve as the functional quantized amount of time, or the 1MHz machine cycles (M-Cycles) which serve as the time it takes for an internal CPU state update to occur. This notation is a carry-over from the Z80 the Gameboy CPU is based on [7], and is due to the multi-T-Cycle spanning memory reads the Z80 needs to perform for certain opcodes.

This means that even though our FPGA cannot achieve 1-cycle reads from BRAM in its clock domain, when we start downscaling the clock output, we can do sub-T-Cycle memory reads and pretend that we are doing it instantaneously as the Gameboy software expects. While one may expect us to forgo clock accuracy together, this is infeasible as it both controls the rate at which games run but also because certain games rely on cycle-accurate interrupt states to function correctly [8], which makes it easier to support a wide array of games without issue if cycle-accurate emulation was done.

Also note that the palette outputs written to the LCD are still monochromatic, making them incompatible with modern HDMI monitors. Luckily, Gameboy pixel outputs can still be stored in DRAM and then later upsampled for HDMI consumption, bypassing this compatibility issue. This should now give us a fully functional transformation of the hardware PPU to our FPGA-emulated PPU.

C. The Memory Controller

The Memory Controller in the Gameboy is essentially a series of lookup tables that routes data to and from the game cartridge (ROM), I/O, and various RAMs. In the I/O section, there exists control registers the CPU issues to the PPU and output registers the PPU emits so the CPU knows its status. Figure I shows the associated memory addresses at a high level, with the specific memory mappings for the PPU controls mentioned in II-B4. The I/O also includes the button states mapped to registers at $FF00.

The Memory Controller further serves as the gateway between the CPU and VRAM and OAM: due to weird hardware destruction issues from simultaneous access capabilities between the VRAM and OAM [2], the Memory Controller blocks CPU's access to the VRAM during the PPU's Draw mode is technically disabled. Similarly, the OAM is disabled in both the Draw and OAM Search modes. Technically, this

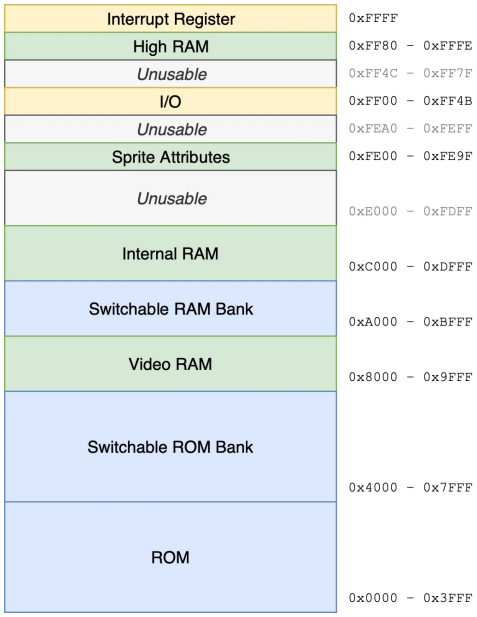

Fig. 1. A diagram of the memory-mapped structure of a Gameboy, courtesy of [9].

behavior seems to be not relied upon except for memory destruction reasons from hobbyist emulation forums [10], and it's recommended to start development without it due to the emulation completely breaking from the PPU blocking CPU inputs to VRAM.

Outside of that, the Memory Controller is responsible for its multi-delivery capabilities: its able to deliver PPU data in 1 T-Cycle and the CPU in 1 M-Cycle (4 T-Cylces), and needs to be able to do so simultaneously. In the actual Gameboy, this occurred flawlessly with multiple memory blocks, but in our Xilinx, we only have 2 BRAM blocks. Luckily, due to the blocking behavior that resulted from OAMs and VRAMs breaking, we can have one BRAM block dedicated solely to VRAM and OAM while the other is dedicated to General Purpose RAM the CPU uses. Then, we can leverage the 100MHz operating frequency of our FPGA to emulate 1 T-Cycle reads to achieve the desired characteristics of our memory controller. As the Urbana boards have 4.2 Megabits of storage, and the Gameboy can only address 2^16 bytes, we well exceed our ability to do this by a factor of safety of 8x.

As a final note, other FPGAs may choose to do this differently due to different memory architectures, but as a MVP this will work for our boards specifically, with more modularization to come if we have time.

D. Miscellany

1) Joypad Input: The goal of emulation is to preserve not only the software and/or game itself, but also the experience of playing it. To this end, we determined that a novel addition to the project was a physical controller input, specifically using the faceplate of an original Gameboy. To accomplish this, the device's casing was fully removed, and while consulting a schematic of the Gameboy motherboard [11], the eight buttons comprising the "Joypad," as well as the device's ground, were

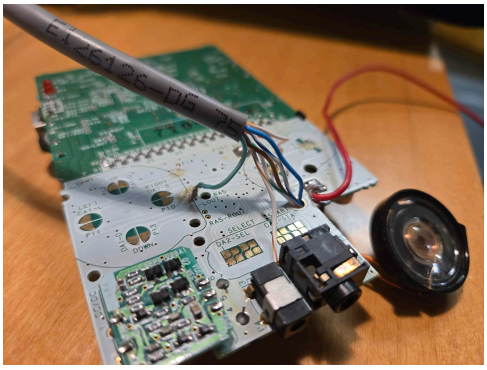

Fig. 2. A picture of a the button pads on a Gameboy motherboard being connected to an Ethernet cable.

tapped. These eight inputs lent themselves well to the use of an Ethernet cable for establishing the connection, as well as a separate red grounding cable, as seen in 2. The other end of the cable was connected and processed by the FPGA board via the built-in PMOD ports. By connecting the ground cable to the 3V source, a pressed button would cause a voltage offset in one or more of the eight input ports. A simple wrapper module reads from each of these, converting it to a 2x4 matrix that is fed to the Memory Controller II-C, and is finally mapped to the address `$FF00` to be read by the CPU when needed.

III. Conclusion

Emulation is an important part in preserving the utility of software that runs on specialized, now-defunct hardware. It often involves large-scale reverse-engineering efforts as the original documentation is lost. Because of this, hardware emulation should be done as quickly as possible once it is established a hardware is defunct, if emulation is desired, as otherwise working models to reverse engineer slowly disappear, as well as the original creators and potential release of the source code [2]. Efforts like these to emulate and document efforts to emulate the Gameboy serve as important references for the longevity of the software reliant on this emulation, as they provide an open, centralized repository of knowledge and its providence where it is often lacking.

IV. Qualitative Self-Evaluation

The pursuit of a Gameboy is mostly a binary success-failure proposition. According to the metric of having a working Gameboy, FPGABoy abysmally failed. Hardware issues were abundant due to parallelism and the fact clock cycles are not abstracted away: features just not found in software emulation and therefore were poorly documented in comparison. However, reverse engineering software solutions for intended behavior was still helpful for testbenching in that regard.

Likewise, our first attempt at trying to integrate in testbench effectively exploded immediately, we were not even able to reach this stage in any meaningful way due to the CPU imploding and effectively needing a rewrite. Most of the literature surrounding CPU timing fails to mention the need for overlapping executions, as described in II-A3, a feature that many programs will fail without, as it adds an

unexpected clock cycle otherwise. With this and other pitfalls associated with the lack of official documentation, along with the 500+ opcodes needing to be implemented, each uncovered "quirk" of the system added hours and days to the CPU's development cycle. Eventually, the need for Python to generate SystemVerilog clauses became apparent, but by that point there was not sufficient time. A future implementation would greatly benefit from acknowledging the power of generators early on in the process, as it would have greatly shortened the iteratition delay.

However, with our resources, we think that this was an academic exploration of SystemVerilog and project management, and we did have some success. We conducted independent research on the target at hand and sketch design requirements, which, although proved incomplete as we delved deeper into the project, led us in the right direction. While our test benching leaves much to be desired, the fact our code was modular enough that it is possible to testbench each component separately is a big win. We believe knowing what we know now, had we had an opportunity to start from the top at week 4, we could have completed this project by avoiding the knowledge pitfalls that caused significant setbacks in the forms of redoing relations between modules, accompanying refactoring, and general chaos and confusion that hinders progress. Starting the CPU from scratch, for example, in the form of our rewrite, would allow us to efficiently add and test everything we have if we were to have enough time.

We also learned a great deal about reverse engineering a product to specification. Unlike many other projects, emulation is one where you have limited latitude in your design choices - you do not control the high-level behavior, nor can you ensure that the user does not use features in a particular way because the real hardware promises them they can do that. Thus, you must use your resources to provide these features as best as possible, of which knowledge is often incomplete [12]. Unfortunately, because of the one-month time horizon, this became rather infeasible. Many Reddit [13] and blog posts [**?**] suggest an experienced dev at the matter takes anywhere from 150 hours to implement a software solution to 2 years on-and-off for a hardware one. Assuming they put only 2 hours per week on that, it is still over 200 hours of work from someone with more experience.

The fact that we were willing to tackle the project, despite the unsuccessful result, have something that could work if we just had more time seems like as much of a success as is possible given the circumstances and time constraints.

V. Work Breakdown

A. Reng

Reng worked on the PPU in both research and implementation and attempted to maintain timing synchronicity. He wishes to credit Pandocs, The Ultimate Gameboy Talk, the SameBoy emulator, the r/EmuDev community, and Colin for being willing to work on a project that we knew from the outset would be difficult and being an interlocutor for working out the Z80's antiquated multi-clock system. Much of the work was only possible due to the thorough research of all these people, although as much of the information was contradictory, some reverse engineering of working emulators was required. He wishes to also credit funnyplaying's Gameboy for inspiring the project and Eli Lipsitz's blog, which, in retrospect, we should have looked for before, as it would have told us an FPGA Gameboy emulator would take an experienced dev over two years of on-and-off work to complete and testbench.

B. Colin

Colin researched and developed the CPU and the Memory Mapper, trying to develop scalable Python generation code for the former to mixed results while having completing much of the Memory Mapper. He wishes to thank Reng first and foremost for being able to put up with various delays and failings due to unforeseen intricacies, all the while remaining a ready source of information and expertise. He'd also like to thank Stephen Kandeh for the personalized advice concerning OpCodes and their timing. Finally, he'd like to thank the PanDocs project for providing much of the structure of the modules, the RGBDS linker forums for direction on implementing OpCodes, and Joonas Javanainen for the verbose wiring schematics enabling the use of the physical JoyPad.

References

[1] T. Jacobs, "Why you should write a gameboy emulator," Jun 2022. [Online]. Available: https://media.ccc.de/v/emf2022-200-why-you-should-write-a-gameboy-emulator

[2] media.ccc.de, "The ultimate game boy talk (33c3)," Dec 2016. [Online]. Available: https://www.youtube.com/watch?v=HyzD8pNlpwI

[3] GBDev, "gbz80(7) — cpu opcode reference — rgbds," Jun 2018. [Online]. Available: https://rgbds.gbdev.io/docs/v0.8.0/gbz80.7#ADD_SP

[4] A. Hacktix and B. Jia, "The ppu," Apr 2021. [Online]. Available: https://hacktix.github.io/GBEDG/ppu/

[5] R. Engineering, "The insane engineering of the gameboy," Mar 2024. [Online]. Available: https://www.youtube.com/watch?v=BKm45Az02YE

[6] PanDocs, "Lcd position and scrolling," Dec 2023. [Online]. Available: https://gbdev.io/pandocs/Scrolling.html

[7] A. Weissflog, "Getting into way too much detail with the z80 netlist simulation," Dec 2021. [Online]. Available: https://floooh.github.io/2021/12/06/z80-instruction-timing.html#m-cycles-and-t-states

[8] endrift, "Revisiting "holy grail" bugs in emulation - mgba," Mar 2018. [Online]. Available: https://mgba.io/2018/03/09/holy-grail-bugs-revisited/#the-phantom-of-pinball-fantasies

[9] R. Stäbler, "Memory and memory-mapped i/o of the gameboy — part 3 of a series," Jan 2020. [Online]. Available: https://raphaelstaebler.medium.com/memory-and-memory-mapped-i-o-of-the-gameboy-part-3-of-a-series-37025b40d89b

[10] Aug 2021. [Online]. Available: https://www.reddit.com/r/EmuDev/comments/oxn2pf/is_this_how_the_gameboy_cpu_and_ppu_works/

[11]

[12] J. Javanainen, "Github - gekkio/mooneye-test-suite: Mooneye test suite is a suite of game boy test roms," Oct 2021. [Online]. Available: https://github.com/Gekkio/mooneye-test-suite/tree/main

[13] r/EmuDev, "Emulation project "gameboy" - where/how to start?" Mar 2020. [Online]. Available: https://www.reddit.com/r/EmuDev/comments/fbuwxa/emulation_project_gameboy_wherehow_to_start/

APPENDIX

Block Diagrams