Gameboi: An FPGA-Based Gameboy Emulator

Author: Adolfo Victoria, Tess Chan, Pratyusha Duvvuri: Electrical and Computer Engineering,
Carnegie Mellon University

***Abstract*—A system capable of cycle-accurate emulation of most Game Boy games on a field programmable gate array (FPGA) for all non-illegal behavior. This means our goal is to have games running at the same speed as the original console, with the same graphics, framerate, audio, etc. The main difference will be the output and input peripherals, but our objective is to give a similar experience to what users experienced on the original console. We believe the level of documentation that Capstone entails will also help us contribute documentation for future hardware developers who want to create their own versions and iterate on our design.**

***Index Terms*—CISC, Computer architecture, Cycle-accurate, Emulator, FPGA, Game Boy**

I. Introduction

The first Game Boy was released in 1989 and was the first handheld console to use video game cartridges. It popularized handheld consoles and started a family of consoles that was manufactured until 2010. Our goal was to learn more about the game console that shaped gaming today by creating a cycle-accurate Game Boy emulator on an FPGA. Currently, there are numerous software emulators that are downloadable and playable on both your mobile device and laptop. We wanted to challenge those emulators by creating our emulator on an FPGA to give our users a more realistic experience. The work on hardware emulators is sparse and there are some areas where accuracy is lacking.

For user input, we will be using a NES controller because it maps nicely to the input controls required. To connect the NES controller to the FPGA our original plan was to use the SoC. Although the controller driver through the SoC was able to communicate with the FPGA, we did not have enough time to integrate it so we used a Raspberry Pi to convert the signals into digital signals for the FPGA and route them through the general-purpose input/output (GPIO). For testing, we used a combination of created unit tests and test suites. Our unit tests were written in Game Boy assembly, and then we used an open source compiler to convert it to a hex file so it can be loaded into memory. For more extensive testing we used Blarggs and Mooneye test suites, which are the gold standards in the emulator community. The test suites contain unit tests for instructions, memory, timing, and graphics that assure accuracy and timing. Within each test, each instruction and variation of the instruction was tested so all cases and combinations are accounted for.

II. Design Requirements

We outlined the following design requirements based on what has been achieved by previous software and hardware emulators and specifications of the original Game Boy (DMG versions), since we want to replicate or exceed its performance.

Performance Requirements

- Games should run at 59.7 frames per second: This is the same framerate at which the original Game Boy ran in. A framerate lower than this would severely impact the user experience due to lowered responsiveness.
- The input latency should be roughly 55ms: This is based on experiments made on the Game Boy Advance by some of the emulator community. We are aware that the Game Boy Advance is a newer console and probably has better input delay, but despite this, we think this is a reasonable goal given the speed of the FPGA. Furthermore, we would not mind exceeding the performance of the original since that improves the user experience.
- We should be cycle accurate in the games that use the constructs that we support (some games rely on obscure tricks specific to glitches in the original Gameboy and we do not believe those are worth pursuing). We want to have accuracy comparable to VerilogBoy, one of the better hardware emulators we found.
- Be able to play Tetris and Dr. Mario with no CPU or graphic glitches. These two games were selected because they do not require boot rom processes or memory bank switching making them the simplest games to load and run.

Qualitative Requirements

- All our code should be well documented, and it should follow a high standard of coding. We want to contribute to the emulator community by showing them a well-documented project which they can use as a reference for future projects.

Verification methodology

- We want to pass at least as many tests as VerilogBoy, a reasonably accurate hardware emulator. We will use the Mooneyes-gb tests and Blarggs instruction tests. They test all possible inputs to an instruction and check that each multi-cycle instruction follows the same sequence as the Game Boy in each of its micro-instructions.
- We will use a high-speed camera to measure the input delay, measuring the time it takes from a button press to a change registering on the screen.
- For the SoC, we will have the FPGA display the controller inputs and checksum of the SDRAM.

The emulator community has spent countless man-hours working on researching the timings for instructions and execution of the Game Boy. Thus, we used the test ROMs that they have come up with to test that our timings work. These tests are Mooneyes-gb tests and Blarggs instruction tests, which are very comprehensive. The test suites test every instruction and every variation of the instruction. That means it uses all combinations of sources and destinations and every input setting, like bit location or shift value. In addition to proving correctness, they also prove that our emulator has reasonable cycle-accuracy and allows us to compare the accuracy of our emulators with other emulators. Within the test suites, there are designed situations where being one cycle off will cause a failure.

III. Architecture and/or Principle of Operation

The overall architecture follows that of a simple computer system where the main CPU will be reading and executing the instructions of the current game from the on-board SDRAM. The CPU will communicate with the Pixel Processing Unit (PPU), memory controller and the timer. The specifics of the communication will be covered when discussing each of these devices. For our peripherals, the RPi will implement a controller driver and notifies the FPGA of button presses through its GPIO pins. Finally, we used VGA as our display method due to its ease of use and availability. A VGA controller takes the PPU's video output and displays it.

We uploaded ROMs to the memory region that was assigned to our CPU, which then executes a boot sequence and starts running the game. The boot sequence is when the Nintendo logo is displayed before the game begins to run and a check of the memory cartridge is conducted. The execution of games is mostly interrupt driven due to having to render an updated screen every time the previous one stops drawing. Thus, the CPU does a certain amount of setup every time a screen is displayed to prepare for the next one. This involves an accurate and well-timed communication between memory, CPU, and PPU. Furthermore, there is also computation being done whenever there are controller or timer interrupts. The following processes are executed every cycle.

The memory controller hides the complexities of accessing memory from the CPU. It manages a majority of the memory, excluding VRAM and OAM which are managed by the PPU. The memory controller has the memory bank controller that will dynamically adjust to the MBC setting of the game. In these cases, it will manage the logic needed for accessing the different banks so the CPU will only need to provide the address. From the perspective of our CPU, we have combinational reads and synchronous writes, but in our hardware, we have both synchronous reads and writes. On the FPGA, all inputs must be registered, which was a limitation we did not account for in our design. To work around this issue the CPU runs at a slower clock than the memory.

The timer runs at a different clock speed depending on what is specified for the game. It sets these options by loading values into designated areas. When enabled, it counts every cycle and once it overflows, it will trigger an interrupt that causes the CPU to load a new number to count from.

For every output cycle from the video display, the CPU will write the desired image to a designated area in memory, which is called VRAM. There are also special registers in the PPU which are used to set settings related to which parts of the image in VRAM to render. Other registers are used to enable interrupts when certain stages of rendering are reached or when a certain coordinate starts getting rendered, which are useful if the CPU plans to change something mid-render (developers use this to create graphical tricks or more intuitive interfaces). The PPU then renders only one area of the image on the screen because the size of the displayed image is smaller than the size of the in-memory image. This is useful for the scrolling effects in games such as Mario Brothers. Additionally, the PPU can initiate a direct memory access (DMA) transfer that copies data from various positions in memory into the OAM. During this transfer, any CPU access to areas outside of the OAM will be ignored. The outputs of the PPU will then write to a framebuffer, which is used to account for the differences in rendering timings between the PPU and the VGA controller. To cross from the PPU's clock domain to the VGA controller's clock domain we used an asynchronous FIFO.

The SoC or RPi will not be too involved in this execution process other than generating the interrupts from the controllers. It will convert the signals from the USB port into digital signals for the FPGA through the GPIO pins. To generate the interrupt, it writes to a register and once the CPU sees the attempt to write, it triggers the interrupt flag.

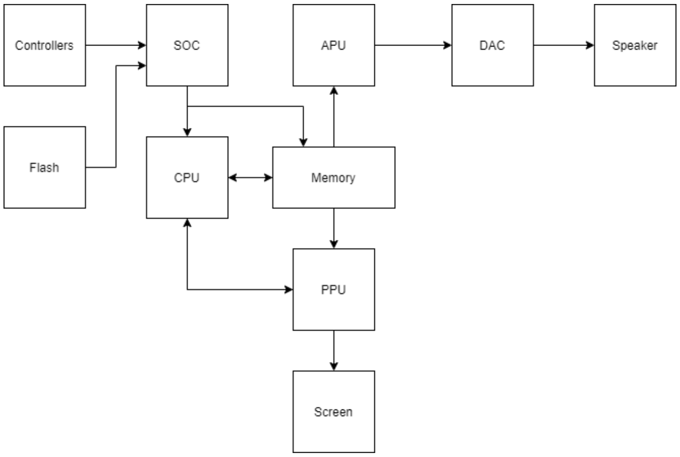


Figure 1. High level system architecture

IV. Design Trade Studies

A. CPU

The official Game Boy manual has never been released to the public, so all the documentation we followed was created by the emulator community based on their observations and tests. This meant that there was a clear set of requirements we had to follow to support the Game Boy, but it gave us plenty of design freedom. The first trade-off we made was using multiple FSMs instead of one. To manage multi-cycle instructions, we implemented 14 FSMs where each FSM was altered to support a specific family of instructions. This made it easier to implement instructions because we could focus on developing and testing one FSM at a time. Once we finished one, we were able to write our own unit test that specifically tested the instruction family without having to debug the other FSMs at the same time. When we moved to using the test suites, having a single FSM made it easier to trace through and identify the bug with ease since we only needed to trace through 5-10 states and a subset of the control signals.

With this design, there were two flaws. The first was we needed a manager to control the interaction between the FSMs and datapath. It needed to route the correct control signal packet to the datapath dependent on which instruction we had last seen. The second issue was some immediate values were the same value as an instruction, so a new FSM would get triggered while we were still servicing the previous instruction in another FSM. These issues were easy to solve with additional logic that did not impact the effectiveness of the FSMs. If we had used a single FSM, it would have contained at least 30 states and the code would not be modular and concise as it currently is. Therefore, using the multiple small FSMs was the best choice.

Another design decision we had to make was the timing for our memory. In our design, we assumed combinational read because in Introduction to Computer Architecture we were given magic memory that supported combinational read. Since the course taught us our foundation in CPU design, we assumed that was also a feature of our memory. When designing the system, it made timing and meeting cycle requirements easy because it gave us more flexibility about when we would get our data.

When we moved the system to the FPGA, we realized that memory could only physically support synchronous read. In Quartus' Megafunction Wizard, all inputs had to be saved by a register and then the access itself was combinational in the next cycle. Unfortunately, that meant for the CPU reads would have a cycle delay. To work around the situation, we gave memory a faster clock making it seem like it is combinational. Making the decision to assume memory had combinational read was the wrong decision in this case, although we were able to get the system to work on the board. If we had the correct assumption, the transition from simulation to synthesis would have been smoother and we would not have had to adjust our CPU to account for the different clocks.

B. SoC

To manage game switching, state saving, and controller drivers, we decided to use an SoC. For this reason, we decided to select the DE10-Standard rather than the other boards used in 18-240 and 18-341. We believed that with the SoC being on the development board with the FPGA it would be easy to integrate the two since the manual had a full description of their protocols. Additionally, the SoC has more storage and access to the FPGA than an external microprocessor so it would be able to support all the memory movement that we wanted for the game switching and state saving. The downside is no one on the team had experience with installing and setting up a SoC from scratch. This meant a large amount of time in the beginning of the project was dedicated to researching and learning through trial and error.

The alternative was to use an external microprocessor, like a RPi, to support this functionality. Overall, the team has more experience with a RPi and there is more documentation and example code for it as well. The downside is that it can not support the memory transfers we wanted for game switching and state saving because it does not have the persistent storage we needed. We believe our decision to use the SoC over the external microprocessor was the better choice for our requirements, though ambitious. Unfortunately, due to the remote access teaching and inability to meet in person, we were unable to get the SoC integrated fully with the FPGA. We initially used the RPi to manage the controller driver as we couldn't flash the FPGA and run programs on the RPi at the same time. Eventually, we were able to run programs on FPGA and SoC simultaneously.

V. System Description

A full, more detailed, system diagram is located at the end.

A. CPU

The CPU is a combination of the Intel 8080 and the Z80 CPUs. The ISA is the union of subsets from both ISAs, plus some extra instructions for array looping. The CPU has 8 (8-bit) general purpose registers: A, B, C, D, E, F, H, and L. Note that the CPU also allows programmers to merge the 8-bit registers into 16-bit registers for 16-bit arithmetic operations. Additionally, it also has two 16-bit registers: SP and PC. The memory has a 16-bit address space, so the special and merged registers are used for accessing it. The CPU has a single address line going into memory, which is not dual ported.

The Game Boy's main execution component, the ALU, handles all operations. Its inputs consist of two 8-bit values, a carry-in, current flags, a flags mask, and the operation to be executed. The ALU is capable of simple operations such as adding, subtracting, bitwise operations and logical operations, but it also supports more complex operations such as bit rotations and word swapping. When it performs an operation, it sets the processor's flags based on the result or other defined behavior (like set or reset) which are: Carry, Half-Carry, Zero, and Overflow. This is the same ALU that is used to support 16-bit operations, and to achieve this the CPU will break the upper and lower bytes into two distinct operations.

As mentioned, memory is 16-bit addressed and it is byte addressable. The CPU has no memory hierarchy, which means that there is only the main memory and registers for storage. There are no caching layers because execution of the instructions is driven by the memory's clock. Unlike more modern architectures there are also no separate instruction and data memories. Since memory is not dual ported, we can only either read instructions or perform memory operations on separate cycles. In our original design, we assumed we would have combinational read, which is why we placed memory accesses in the fetch and decode stage. When we moved to synthesizing the system on the hardware, we realized this was not possible so rather than changing our entire design, we have the CPU running at a slower clock rate than memory. Memory is the main tool for communication between the CPU, the PPU, and the timer. We have added a memory map at the end of the document to illustrate all the distinct memory regions.

In our CPU datapath, we have two stages: fetch and decode (DE), and execute and writeback (EX). In our first stage, we will present the address we want to access and decode the instruction. Our decode module will generate a packet of control signals that can be broken into three categories: DE, EX, and ALU. The DE control signals will impact the components in the DE stage during the current cycle. The EX control signals will be pipelined to the EX stage through a register and will be used to control the next cycle. The ALU control signals are specifically for controlling the ALU and were placed in a separate category to make it easier to organize. LIke other pipelined CPUs, there will be cases where DE and EX will be working on different instructions. The EX stage will execute the instruction decoded based on the EX control signals passed from the decoder. In the EX stage, the register file stores the 8 working registers and flags, and there is a separate register to keep track of the stack pointer.

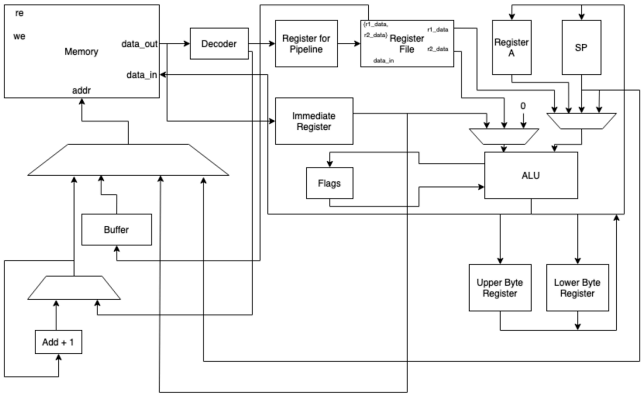


Figure 2. CPU sub-system diagram

The Game Boy's is a CISC, meaning instructions have varying lengths and cycle lengths. Some instructions do multiple operations over many cycles. This is due to the limited resources that the CPU has. As mentioned, it has a single 8-bit ALU, a single ported memory that is shared for both instructions and data, and only two pipeline stages. To handle this, complicated instructions are broken up into smaller instructions which are slowly processed by the processor. To break up these instructions, we used multiple FSMs that are based on the instruction's family. Each FSM is responsible for generating the new control signals for the current DE stage and next EX cycle because the system will have no recollection of the instruction it is in the middle of executing. Therefore, the signals that are generated are a combination of saved signals and signals that were updated to execute the next step of the multiple cycle instruction.

Finally, the CPU has multiple interrupt lines that come from the PPU, the timer, and the controllers. The CPU uses a vector table to service these interrupts and it is in a special area in memory for each of these interrupts to jump to. Interrupts can be enabled or disabled through special registers which are controlled by instructions serviced by the decoder.

B. Memory Management Unit (MMU)

Since the Game Boy has multiple different memory cards such as ROM, Work RAM, High RAM, cartridge RAM, VRAM and OAM which all have multiple readers or writers, there needs to be a bus arbitration unit to prevent bus conflicts. Having an MMU allows for accurate emulation behavior because what some emulators do is prevent the CPU from accessing any memory at all if, for example, the DMA controller is running. This is not how it works in hardware since it is possible to access memory areas that do not cause bus conflicts with the DMA controller. Thus, the MMU gives us higher emulation fidelity. The MMU was not involved with I/O. For I/O register writes, we have all the different I/O devices listen to the CPU's address bus for reads and writes that have their address.

As mentioned, the MMU has a Direct Memory Access controller which supports copy operations from any of the memory chips into OAM memory. Without the controller, this operation would take thousands of CPU cycles. With the controller, the operation takes a mere 160 CPU cycles to complete. The DMA controller has some more obscure behaviors that we decided not to pursue due to time constraints, but they are not required by any of the games we tested. In fact, a lot of the games act like none of the memory other than High RAM is accessible during a DMA operation.

The Game Boy also uses memory banking to make up for the limited address space since some games were too large to fit into the 16-bit address space. Thus, the Game Boy can programmatically change which part of physical memory it accesses, allowing it to support games that are bigger than its address space. There are 4 main types of memory setups for Game Boy games that we support: ROM0, MBC1, MBC3, and MBC5. ROM0 does not include memory banking since the games were small enough to fit in the original memory space. MBC1 and MBC3 have 128 ROM banks and 4 RAM banks. The difference between the two setups is that MBC3 has a real time clock, which is a timer that is continuously counting even when the Game Boy is turned off. This is used for games like Pokemon that adjust the game to match the time of day. MBC5 has 512 ROM banks and 16 RAM banks. In addition to having a varying number of banks, the size of the bank can change regardless of the MBC type.

Banks are stored in registers in the MBCs, these registers are mapped to different areas of the ROM (since the MMU prevents the CPU from writing to the ROM directly). Thus, the ROM memory areas are treated like write-only I/O registers. The logic of the MBCs is simple, some registers are used to construct the ROM bank number or the RAM bank number. These bank numbers are just the upper bits of the extended address space. Thus, the effective address for CPU reads and writes is calculated by using the address from the CPU as the lower bits and the bank numbers as the upper bits.

C. Timer

The timer is continuously counting regardless of what is happening in the CPU. There are four memory addresses that are used in this process and they have all been given names to make the explanation clearer. The divider register, DIV, is incremented at a set rate of 16384 Hz and cannot be disabled. If anyone attempts to write a value to it, it will automatically be set to 0. The timer modulo, TMA, holds the next value that we will begin to count from when our counter overflows. The timer control, TAC, determines the frequency we will increment our counter by and if we are counting. The timer counter, TIMA, is our counter. It will increment at the rate specified by TAC, and when it overflows it will load in the value from TMA and assert an interrupt. A value can be written to TIMA without TMA overflowing. The most challenging part of the timer was accounting for the different edge case scenarios. We needed to establish the correct priority order of changes to TIMA in the cases that multiple people were attempting to alter the value.

Originally, we used a clock divider to support this feature. Unfortunately, there were many edge cases we were missing so we instead opted for a high-fidelity reproduction of the timer in the Game Boy. This timer used a 4Mhz clock that was hooked up to a 16-bit counter. The DIV register is just the upper 8 bits of this counter. To obtain the different frequencies for the exported TIMA counter, the timer has an edge detector in different parts of the internal counter. When the selected edge detector triggers, the counter is incremented. There are some peculiarities with this design that lead to strange behaviors when writing the timer registers. Which is what prevented our initial approach from correctly emulating the Game Boy's timer.

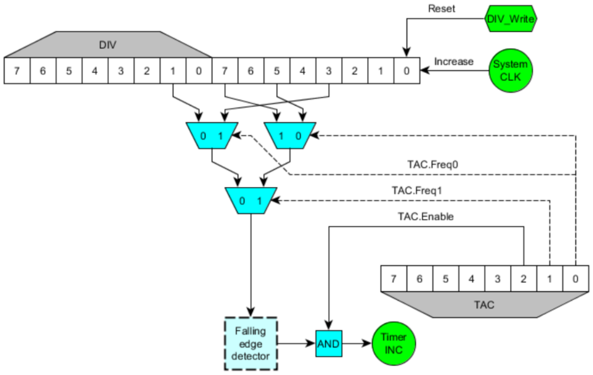


Figure 3. Timer Diagram, from [8]

D. PPU

The PPU (Pixel Processing Unit), handles the rendering of the game's frames. Frames are rendered in the same order as a VGA screen; pixels are drawn from left to right and from top to bottom where each row is called a scanline. Just like VGA, the rendering process is divided into stages based on this pattern. The PPU cycles through 4 stages: OAM search, Pixel Fetching and Drawing, H-Blank and VBlank. OAM search and pixel drawing happen in between every H-Blank stage.

We will discuss the different objects that are involved in a frame being displayed to create a background in the discussion that will follow. The basic building block for images is a pixel. The original Game Boy only supported 4 colors, or 5 if you count transparent as a color. As a result, pixels are encoded as two-bit values which are then translated using a look-up table or "palette". The next building block is tiles, tiles are 8x8 arrays of pixels. Tiles are identified by a unique number (their offset in memory) and make up the main display elements, background, windows, and sprites. When rendering, background is the default element to be displayed. There are special registers and memory regions that determine when a different kind of element must be rendered. There are window coordinate registers, which will make the PPU start rendering from the window memory instead of the background memory. A similar system is used for sprites, but the only difference is that sprites have their coordinates encoded in each one of them because we can have up to 10. Thus, the PPU has an array of comparators to check for this.

Sprites are treated differently from other elements. On top of the basic pixel encoding they have a variety of different attributes that are encoded in four bytes. The first two bytes are used for the X and Y coordinates. The third byte contains the tile number. Finally, the fourth byte contains extra attributes:

- **Priority with respect to the background**: this dictates whether the sprite will be rendered on top of the background (i.e. it will overwrite the background)

- **X and Y flip bits**: these bits mirror the sprite with respect to the X or Y axis so that developers don’t have to take up extra space for the sprite facing different ways
- **Palette number**: there are 2 different palettes that sprites can have in the Game Boy DMG, this bit chooses between them.

The rest of the bits in the attribute byte are unused by the original Game Boy.

As mentioned, sprites are drawn on top of the background, unlike windows which are drawn instead of the background. There are also special considerations for what happens when multiple sprites overlap. The PPU uses a FIFO queue for pixels because it is not able to commit pixels to the screen until it has at least 8 pixels to commit. This is for mixing purposes because when a sprite is about to be rendered its pixels are “mixed” with the background pixels. This is done based on the sprite’s priority over the background. If the sprite has a transparent pixel then the background pixel will be displayed no matter what. For sprites being drawn over other sprites, the sprites at the earlier x-coordinate get picked instead.

The image that is being displayed is 160x144, but the actual image in memory is 256x256. Games with scrolling effects are a good application of this fact. To select the area of the 256x256 image the scrolling registers are used (SCX and SCY). These registers say which region of this image should be drawn. When rendering, pixels at coordinates outside of the range established by these registers are discarded. To achieve a scrolling effect, the CPU can adjust these registers based on the user’s position. To be able to change these things users can make use of the interrupts and special registers that the PPU offers.

The PPU communicates with the CPU through memory and the special “registers”. These registers are what control the palettes, the window region, whether background, windows or sprites are being drawn, and interrupts. There are multiple kinds of interrupt lines coming from the PPU that the CPU can enable, which are all set in the PPU’s STAT register. An interrupt can be set for the beginning of each of the PPU’s stages (except for the fetching and rendering). There is also the LYC interrupt where users can set the LY register with a value and then the LYC interrupt will trigger an interrupt when the LYth scanline is reached. These interrupts are used for a variety of reasons. They help developers modify the PPU’s state at certain points of execution to achieve various visual effects. For example, to make sprites not draw over windows, they can set LY = WY so that they can turn off sprites when we are rendering a window, and then turn them back on when we are done. They can also use this to dynamically scroll an image as it is being rendered to achieve a “warping” effect.

VRAM is divided into three blocks of 128 tiles each, the three blocks are block 0 ($8000-$87FF), block 1 ($8800-$8FFF) and block 2 ($9000-$97FF). Sprites can only go in block 0 from $8000 to $8FFF. There is a special area in memory called the Sprite Attribute Table (or OAM - Object Attribute Memory) which spans $FF00 - $FE9F. This table can only hold up to 40 sprites, which means that only up to 40 sprites can be displayed on the screen at once (without tricks) and only 10 sprites can be rendered in each scanline. This memory is special because the PPU can access 2 bytes at a time rather than one.

For writing to VRAM and OAM, there is a contract that the CPU must follow to avoid crashing the system. For VRAM, the CPU is only able to access it during H-Blank, V-Blank or OAM search, and the status of the rendering can be polled through the PPU’s STAT register or by enabling interrupts in that same register. The OAM table has two ways of accessing it. It is often recommended that it is accessed using the built in DMA functionality, which lets the user read it at any time. To use DMA, the CPU must write a target address, must be divisible by 0x100, from ROM or RAM to write or read from, respectively, to the DMA transfer register. Afterwards a DMA transfer will start; note that during this timeframe, the CPU can only access a special region in memory called HRAM. Thus, the function that is used for DMA should be located in this memory region. Other than using DMA, the CPU can directly access OAM during H-Blank and V-Blank.

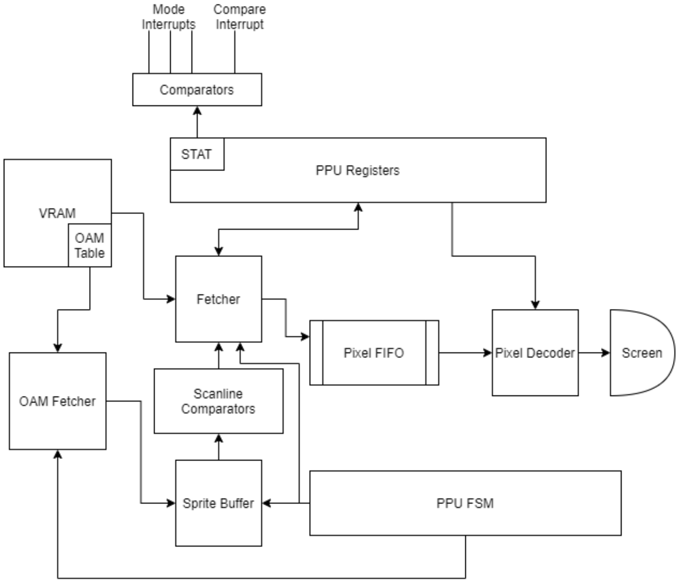


Figure 4. PPU Sub-System Diagram

E. APU

The APU is a read-only subsystem. There are four voices that make up the sound: Pulse 1, Pulse 2, Wave, and Noise. Pulse 1 and Pulse 2 are the tones, Wave dictates the shape, and Noise is white noise. Each is allocated five 8-bit registers in memory where the CPU will change the values depending on the sound that is needed. This was done because when the Game Boy was designed, having pre-built sound files took up too much space in the cartridge. To compensate for this, engineers developed this four-voice system to have a real-time synthesizer.

Although the voices have the same number of registers that correspond to similar parts of the sound, the bits are in different parts and bits in the same location have a different meaning. For our APU, we will have four decoders, one for

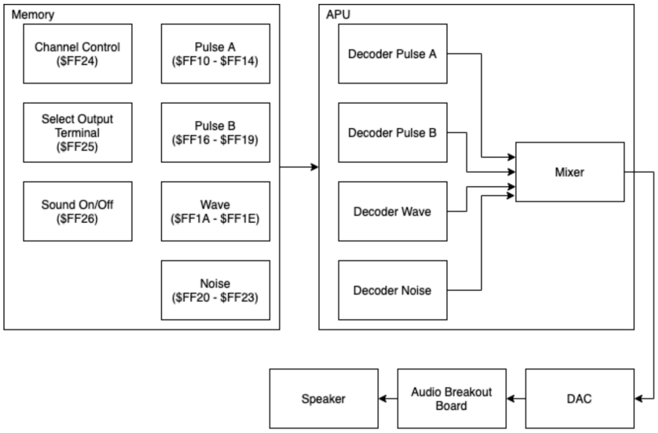


Figure 5. APU Sub-System Diagram

each voice, that will translate the information in memory to standard values that can be combined to make one sound. Once translated, a mixer will combine the four voices using the information set by the three sound controller registers (also in memory). The sound controller registers are the masters that control if the sound is outputted, which speaker it is outputted to, and which channels are used.

Once a final audio signal is created, it will be sent to the DAC through the GPIO. We need a DAC to convert our digital audio signal into an analog audio signal so it can be outputted by a speaker. Our DAC is 16-bit wide with parallel load, which means we do not need to establish a serial protocol to send over our data. Rather, we need control signals that will control which rank we are writing to. The DAC has a two-stage rank system to create a double buffer organization in order to prevent spurious analog output values. Therefore, in addition to the digital audio signal, we will also need to send signals to control the DAC. CS_n and L1_n will control the first rank, and then LDAC will control the second rank. We will be sampling at a rate of 50 kHz, which is better than the Game Boy sampling rate of 44.1 kHz.

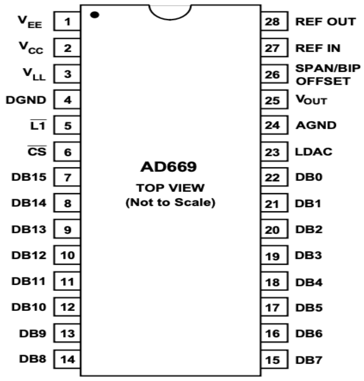


Figure 6. AD6699 DACPORT chip pinout

From the DAC, we will attach an audio jack breakout board to the V_{out} line. The breakout board will allow us to plug in a 3.5mm audio cable to connect the speaker.

Due to moving to remote access for the remainder of the semester, we had to remove the APU subsystem because we did not have access to the needed tools.

F. SoC

The SoC has two main functions. Players will use the NES controller to input joypad instructions, which go through the SoC to the FPGA. To save game state and switch between different games, the SoC will keep track of the regions of persistent memory that contain game information, save it when the game should be saved, and load the relevant game memory into the SDRAM if the user is switching between games. The loaded game resumes as usual.

The ARM-based hard processor system (HPS) provides two instances of the USB On-The-Go (OTG) Controller. It supports high speed, full speed, and low speed transfers in both the device and host modules and will be programmed to support data movement over the USB protocol between device and host. The two OTG controllers are independent of each other, and we will be using one of them to receive signals from the connected NES controller.

The four directions on the “direction-pad” (dpad), two buttons for “Select and “Start”, and the ”A” and “B” buttons make up for a total of 8 signals to be sent to the FPGA. To be able to read signals from the NES controller, we installed the relevant linux driver of the type USB Human Interface Device (HID) Configuration. Once we recognize and receive the signals through the OTG controller on the SoC, we choose 8 of the 32 GPIO signals provided to the FPGA, which are controlled through registers in the FPGA Manager on the SoC.

When the player wants to switch between games, the SoC would have let the current screen finish rendering and then stop the CPU’s execution. Once the execution is stopped, all the CPU’s state and the memory will be saved for later re-execution in persistent storage (flash memory). Afterwards, all the state would be cleared, and the new game state will be loaded in from persistent memory to SDRAM.

We planned on using external flash storage for persistent memory. NAND flash would be fast enough for game switching since we plan on doing bulk reads of data that has high locality. We considered using NOR flash, whose random-access time is 0.075 micro-seconds per read. However, for NAND Flash while the first byte is read at 25 microseconds, the remaining bytes would be shifted out at 0.025 microseconds (resulting in a bandwidth of 26 MB/s for 8 bit I/O and 41 MB/s for 16 bit I/O). We will use the NAND flash memory controller, which directly corresponds to NAND flash persistent storage. However, since the SoC boots from flash, we decided it would be best to use USB storage to aid with game switching, to avoid potential booting issues with memory sharing.

To load a game after boot, the SoC will obtain the data from the drive and transfer it to the SDRAM on the FPGA via the HPS-FPGA bridge. The SDRAM Controller on the FPGA provides an interface to the 64 MB SDRAM on the board, which is organized as 32M x 16 bits. It is accessible by the HPS on the SoC using word (32-bit), halfword, or byte operations, and is mapped to the address space 0xC0000000 to 0xC3FFFFFF by default7. Once we transfer the data to the SDRAM, we will use interrupts or GPIO pins to signal the FPGA to start the new game.

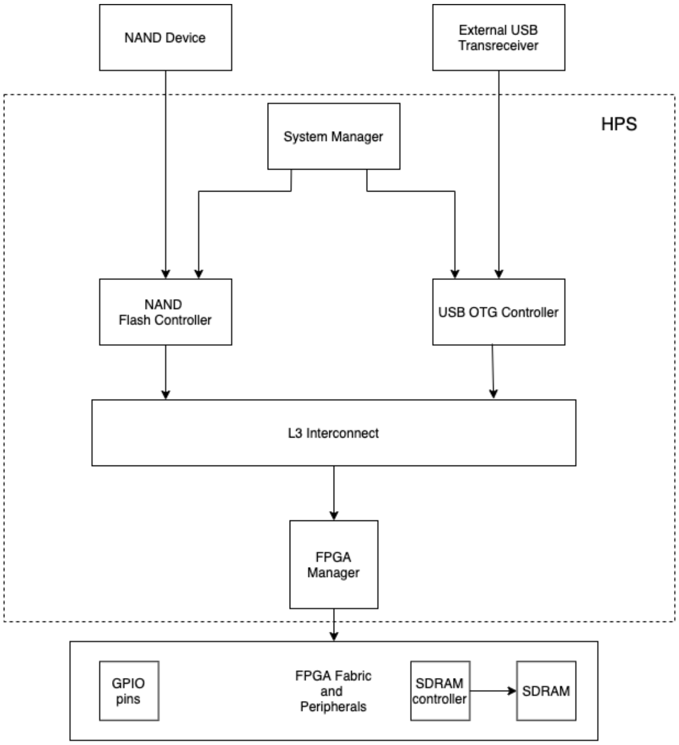


Figure 7. SoC Sub-System Diagram

An alternative would be to use the HPS's DDR3 memory and have the FPGA read it, since at the time we hadn't figured out the HPS-FPGA bridge. We found the above described approach to be better since the SDRAM is directly connected to the FPGA. It was more intuitive to use SDRAM directly connected to the FPGA to perform memory operations since it uses a simpler interface for both the FPGA and the HPS.

If another game were in progress, we would first save the game state by copying it into USB storage. We will have a Lookup Table stored in a reserved part of the USB Storage which would give us the addresses to find the required data to load and store.

During integration, we faced many issues, one of which was being unable to run programs on the SoC and the FPGA simultaneously. Running one would cause the other to stop. We did not anticipate this until near our final presentation. Due to moving to remote access for the remainder of the semester, which made integration harder, we considered transferring responsibility of SoC onto RPi, or using a USB/PS2 adapter and have the FPGA directly communicate with the NES-controllers connected through the USB port, since there is a PS/2 input for the FPGA. We chose to use the RPi, as elaborated below. However, we managed to simultaneously flash the FPGA while running the SoC with the help of a platform integration tool - Qsys and can run programs on the FPGA and SoC simultaneously.

Unfortunately, due to the current extenuating circumstances, we were unable to implement game switching and state-saving on the SoC and using GPIO to transfer data from the RPi to the FPGA proved to be too troublesome to implement near the deadline.

G. Joypad

The RPi uses the joypad Linux library, which supports the joypad we choose. With this, we wrote a Python script which listens for events from the joypad using the inputs API. Each of these events toggle the GPIO pins which are connected to the FPGA. The FPGA then grabs those inputs directly, maps them to buttons and feeds them into the joypad module. The joypad module has a sampling clock to sample from the button inputs to the module. The module then shows the sampled values in the I/O register that is exported for the CPU to read. There is only one 8-bit register for 8 possible buttons. Unfortunately, this register seems to have some reserved bits in it, which makes it not possible to fit all button presses, so it has 2 writeable bits which control which buttons from directional keys or button keys must be sampled. Furthermore, for the non-polling case, there is an edge detector that triggers the interrupt lines whenever there is a change in any of the inputs. This module is very flexible, since any source of inputs could be used, which made testing easy.

VI. Project Management

A. Schedule

See the last page for our complete Gantt chart.

Our schedule is very detailed because it lists each necessary step for each part. We have added a lot of slack at the end because of the integration issues and small fixes that we will need to make once we are able to load games. Our schedule has remained very similar to our proposal except we have updated the deadline for different tasks to account for having to refocus the project. Debugging was a challenge because we were unable to meet in person, but we were able to debug at the same time since each member had a board shipped to them.

Week 8 is intentionally blank to account for spring break. The additional changes that have been made were to account for additional collaboration when working on the CPU and the tasks required for DMA, MBC, and joypad that we did not originally account for.

B. Team Member Responsibilities

Adolfo - Adolfo was primarily in charge of the designing the CPU datapath and overall system layout. He worked with Tess to implement and debug the CPU. Once the CPU was complete, he worked with Tess to implement and integrate the timer and memory controller. Adolfo was also responsible for researching, designing, and implementing all the graphics. Additionally, he was responsible for researching and implementing the joypad through the RPi.

Pratyusha - Pratyusha was primarily in charge of learning and designing how the SoC would work. She was also in charge of setting up the SoC communication, software support, program toolchain and having it interact with FPGA peripherals. Additionally, she oversaw researching, designing, and implementing the controller driver and game-switching using SoC.

Tess - Tess was primarily in charge of designing and implementing the FSMs for the CPU. She worked with Adolfo to design the initial FSM layout, and then finished the design for most of them. Additionally, she worked with Adolfo to implement the datapath for the CPU and debug it. Once the CPU was complete, she worked with Adolfo to implement and integrate the timer and memory controller. Additionally, Tess was responsible for researching and designing sound circuitry and APU before it was removed from the project for refocusing reasons.

All - As a team, we worked together on designing and reverse engineering the datapath for the CPU, integrating components and completed all required assignments for the course.

C. Budget

We selected the DE10-Standard Development Board because it has an FPGA and SoC, so integration between the two would be easier, and it has a micro SD card reader. We needed this reader because we planned to boot the board through a linux console image on a micro SD card.

For our controller, we used a controller with a similar layout to the Game Boy and the closest we could find was the NES controller. We specifically chose one with a USB connector so the signals could be handled by the SoC.

Due to remote access, we ordered additional parts as backups in case we could not get our first-choice method to work.

Part	Anticipated Cost	Used
DE10-Standard Board	$355*	Yes
NES Controllers	$15	Yes
Micro SD Card	$20	Yes
AD669 DACPORT	$50	No
Audio Jack	$20 (for 3)	No
Speaker	$40	No
3.5 mm Audio Cable	$7	No
Raspberry Pi	N/A	Yes
PS/2 Port Keyboard	$60	No
USB to PS/2 Converter	$15	No

* Is not subtracted from team budget

D. Risk Management

For the project, the biggest risk is lack of official documentation for the Game Boy. Since we are building the emulator from scratch, we had to learn the Game Boy architecture through-and-through. Thankfully, the emulator community developed documents which were very accurate. Therefore, all of the manuals and articles that we have read were written by developers who have created their own emulators in their spare time. Although they have found many of the nuances, they may not have found all, and each developer has their own solution for the various parts within the Game Boy. This meant that we had to reverse engineer the layout of the datapath of each of the components to the best of our ability based on what we have learned. Additionally, not all of the sources are complete or included every detail. To mitigate this risk, we have reviewed numerous GitHub repositories, watched tech-talks of prominent people in the emulator community breaking down how the Game Boy works, and assuring our logic supports the necessary cycle count for instructions. When we had a bug and assured the logic, we implemented was doing what it was supposed to, we would use multiple sources to assure that we accounted for all required behavior.

The other big risk is the size of the system. The entire Game Boy system can be divided into the CPU, sound, video, memory accessing/assignment, and button input. On top of that, we have added an SoC component that will require memory mapping, memory transfers, and a controller driver to connect input signals from the USB of the controller to signals for the FPGA. We divided the components to play to each team member's strengths, but we each must learn how it is implemented in the Game Boy. Additionally, integration will be a significant portion of our debugging process because of all the interactions each unit will have with another and the different clocks throughout the system. To mitigate this risk, we plan on spending a significant time together in the beginning of the project to outline interactions to prevent these integration issues in the future. Additionally, we will all work together on the CPU to learn how the Game Boy system works overall, and in parallel we will conduct our individual research and implement our assigned section.

A new risk that was introduced was debugging. Since we were never able to meet, we all had to work remotely in parallel. Although this is time efficient and allows each member to focus in one test or bug, there are various issues that arise. One is version issues where members are working with different versions of the same code. This could lead to merge conflicts and overwritten code when updating the master code. Another issue is multiple members may be solving the same bug. Since we will be working remotely and not always communicating to each other in the moment, we may waste time debugging a bug another member has already encountered and/or solved. This can also lead to merge issues if each member has a different solution. To mitigate these risks, we decided to all stay on a call and work together. This would allow us to talk with each other at any moment as if we were together but was not distracting since each member could mute their microphone if they were just working. Additionally, we all followed good Git practices by constantly pulling and checking with others when resolving merge conflicts.

Another risk was working with the SoC. As mentioned previously, none of us had experience working with a SoC, and did not anticipate the software, equipment, and tools required to set it up. We also did not predict the need for a tool outside of Quartus and toolchain for cross compilation of code to integrate FPGA and SoC flow, as one cannot flash the

FPGA and run code on SoC without the help of another tool such as Qsys.

VII. Summary

A detailed breakdown of all the tests that we ran can be found at the end of the document.

Our FPGA system was able to meet and surpass all the requirements that we outlined in our refocused statement of work. In the statement of work, we removed the audio requirement and made the ability to play Tetris and Dr. Mario as our minimum goal. After implementing ROM0 for basic games, we had enough time to design, implement and test the MBC so we can support more games. Our CPU was able to pass all the emulator community's accuracy and timing tests, more than our original goal, VerilogBoy. Our PPU was able to run the dmg-acid2 test which tests a lot of the PPU behaviors that are needed to correctly display images. Additionally, it was also able to pass ⅓ of Mooneye's PPU acceptance tests, which tests the timings for all the PPU's different state changes. While playing the games, there are no graphical or logical glitches. The controller inputs have an estimated lag of 8.3 - 12 ms, which is significantly better than the original Game Boy's input lag of 55 ms. Integration of all FPGA based systems was effortless due to the good approach we took to modularity; it was also able to communicate with external systems such as the joypad and the different memories.

The SoC system passed the controller unit tests outlined in the refocused statement of work. While we could initially not run programs on the SoC and FPGA at the same time, for which we used RPi to process controller inputs, we were able to flash FPGA and run the controller driver on the SoC simultaneously towards the end. However, due to the COVID-19 situation and the inability to meet in person, we were not able to implement game switching and saving game state on the SoC.

There are three main limitation s with our emulator. The first is we cannot save the game state, so if the user turns off the board, they must restart the entire game. This is problematic for story-oriented games, like Pokemon. The second limitation is to switch games, the user will need to reflash the FPGA. The FPGA does not have enough memory to store more than one game, so to load a new game the user will need to resynthesize the project with a different hex file. The third limitation is we do not support audio, but this was purposely removed from the project. The last two limitations are not terrible, but they are not ideal.

VIII. Future Work

If we were to continue working on the project, the first feature we would implement/work on would be audio. Due to the lack of access to the correct tools, we were not able to work on audio in the given time limit. With more time, we could have implemented the design we planned out with the external DAC. Alternatively, if we still did not have access to the lab tools needed, we could attempt to use the on-board DAC that required the P2C protocol. We did not attempt this route because we did not do the proper research for it so it would have been unrealistic to attempt to research and implement this subsystem within the remaining time.

Next, we would implement and integrate game switching and saving state on SoC with FPGA. Now that we have the SoC running memory management on its own, we would focus on creating the protocol between the FPGA and the SoC.

The final item we would work on is redesigning the CPU to account for synchronous read. This would require a complete overhaul of the CPU, but it would eliminate any potential timing issues when doing memory to memory transfer. Although we believe we have resolved these issues for the original Game Boy, if we decide to build on to support the Game Boy Color or any other version, we may run into new issues since they have different clocks.

IX. Lessons Learned

Throughout the project, we learned many lessons and new topics. For design, we learned that you should do your research for future stages, and not just the first stage. This became an apparent lesson when we realized combinational read is generally not supported by memories, so never assume that you will have it. Additionally, since synthesis is slow and hard to trace through, do as much testing in simulation as you can and have the simulation model match the settings on the hardware.

For debugging, having a correct model to trace through side-by-side with your faulty model is super-efficient and effective. Specifically, for emulators, BGB is a software emulator that has a debug mode so you can step through each instruction to check which instruction causes the failure and which registers or memory areas are affected.

From the SoC component, we learned that we must do more research initially into designing how to communicate between the SoC and FPGA. We did not anticipate the needed external tools that are required so there was a surge of workload at the end of the timeline that we could not handle.

We also learned some lessons that are specifically relevant for people who intend on creating a Game Boy emulator, or any emulator, in the future. The first is the graphics subsystem is much harder and tedious than it may seem, so dedicate more time than you initially thought. The second is make sure you have a solid CPU before trying anything else because if you cannot depend on your CPU being correct, then you will waste hours debugging after implementing the various processes the Game Boy needs, like DMA and timer interrupts. The third is there are various unique behaviors that you may not account for because one source does not list it. If there is a bug and you know the code is doing what it is intended to, double check that you are not missing one of these behaviors. Finally, we recommend joining the Game Boy emulator discord. It is a community of people who have extensively studied the Game Boy and know all of the unique behavior and how the tests run. We are proud of what we have accomplished given the time and the circumstances, we were worried we might have to import modules from other existing projects. In the end, we were able to do everything from scratch.

X. Related Work

There are multiple related works that we investigated and would also like to thank due to their comprehensiveness and insight. The Game Boy Pandocs, which were made to contain nitty gritty details about the timings of instructions, PPU, and APU. The Pandocs are the basis for many of the popular wikis. Another useful document that has a similar spirit is "Game Boy: Complete Technical Reference" by gekkio, it is a work in progress trying to give an approachable document that contains all the timings for the Game Boy instructions.

Other emulators that make accuracy their top priority is Gambatte and Mooneye-gb. These are software-based emulators, and based on the research we did, they seem to be the closest to matching the timings of the Game Boy. Note that they have added support for the Game Boy Color as well so there may be some slight differences in their documentation. For hardware emulators, we found Verilog Boy, which we are using as the point of reference for our performance. There are other hardware emulators that we found, but don't seem as polished.

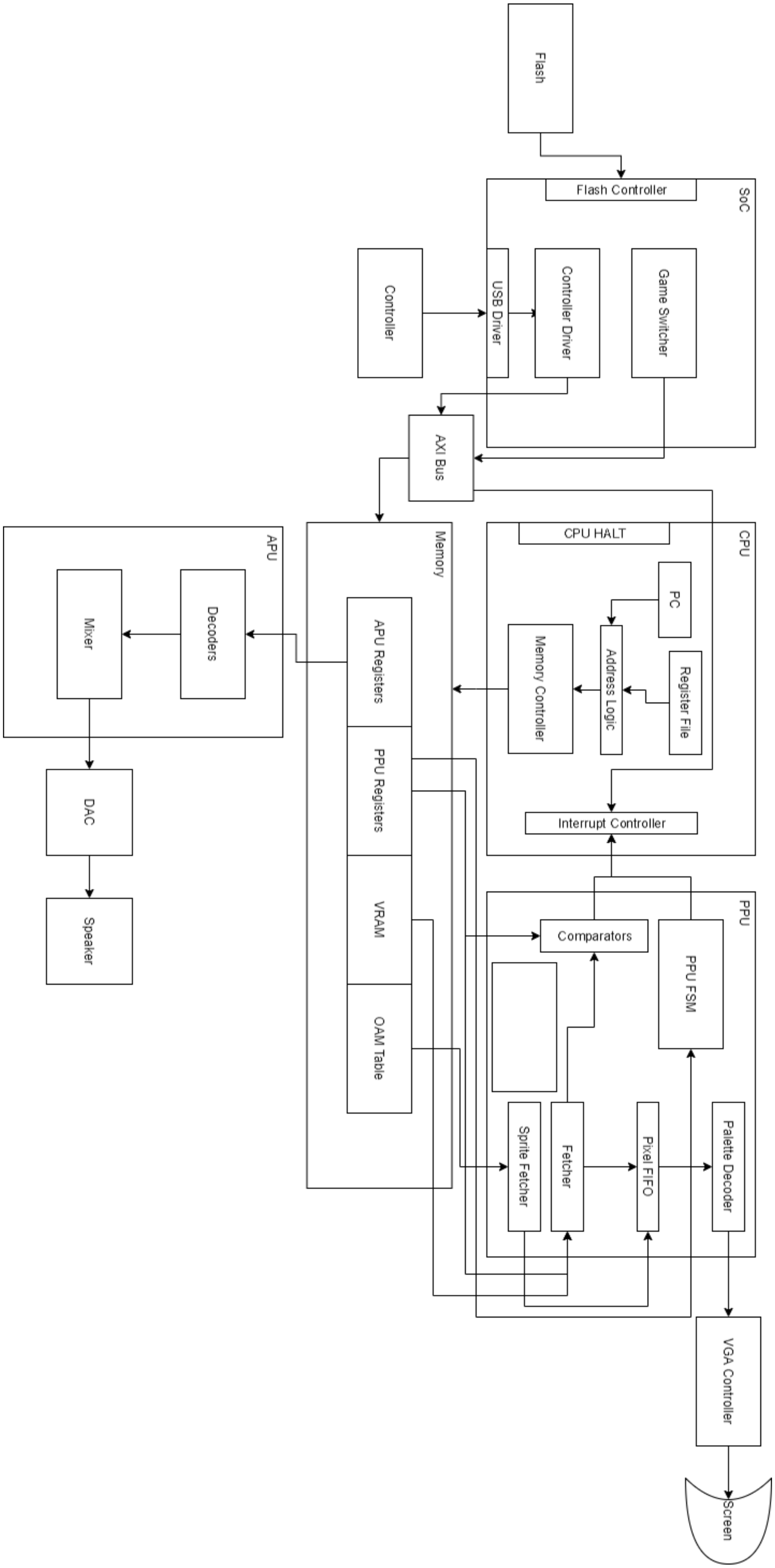
References

[1] Game boy Architecture: A Practical Analysis, https://copetti.org/projects/consoles/game-boy
[2] Game Boy CPU Manual, http://marc.rawer.de/Gameboy/Docs/GBCPUman.pdf
[3] VerilogBoy - GameBoy on FPGA, https://hackaday.io/project/57660-verilogboy-gameboy-on-fpga
[4] gbdev/awesome-gbdev, https://github.com/gbdev/awesome-gbdev
[5] Gekkio/mooneye-gb, https://github.com/Gekkio/mooneye-gb
[6] DE10-Standard User Manual, https://www.terasic.com.tw/cgi-bin/page/archive_download.pl
[7] Cyclone V HPS Manual: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/cyclone-v/cv_54001.pdf
[8] The Cycle-Accurate Game Boy Docs https://github.com/geaz/emu-gameboy/blob/master/docs/The%20Cycle-Accurate%20Game%20Boy%20Docs.pdf

TASK NAME	TEAM MEMBER	PERCENT COMPLETE	Week 4: 2/10	Week 5: 2/17	Week 6: 2/24	Week 7: 3/2	Week 8: 3/9	Week 9: 3/16	Week 10: 3/23	Week 11: 3/30	Week 12: 4/6	Week 13: 4/13	Week 14: 4/20	Week 15: 4/27
Documentation														
Abstract	All	100%	🟨											
Proposal Presentation	All	100%	🟨											
Design Review Presentation	All	100%		🟨										
Design Review Report	All	100%		🟨										
Final Presentation	All	100%											🟨	
Final Report	All	100%												🟨
CPU														
Designing FSM and reverse engineering datapath	All	100%			🟨									
Instruction Decoder	Tess & Adolfo	100%				🟪								
FSM	Tess & Adolfo	100%						🟪						
Register File + ALU	Tess & Adolfo	100%						🟪	🟪					
Memory Controller	Tess	100%									🟪			
PPU interconnect	Adolfo	100%								🟩				
Interrupts	Tess & Adolfo	100%								🟪				
MCU interconnect	Pratyusha	50%							🟥					
PPU & Graphics														
VGA Controller	Adolfo	100%		🟩										
Display RAM Reading	Adolfo	100%			🟩	🟩								
Tiles, Background and Window Display	Adolfo	100%				🟩								
Sprites	Adolfo	100%						🟩						
Full Frame display	Adolfo	100%							🟩					
CPU interconnect	Adolfo	100%								🟩				
Drivers and MCU														
Setting up board communication and linux environment	Pratyusha	100%						🟥	🟥					
Controller driver	Pratyusha	100%								🟥				
Flash Driver	Pratyusha	100%								🟥				
Memory Mapped RAM	Pratyusha	75%							🟥	🟥	🟥			
Game State Saving	Pratyusha	0%									🟥	🟥		
Game Boi MCU interface	Pratyusha	75%				🟥					🟥	🟥		
Memory														
Block RAM	Tess	100%								🟦				
ROM loading	Adolfo	100%								🟩				
DMA Controller	Tess & Adolfo	100%							🟪					
Memory Bank Controller	Tess & Adolfo	100%										🟪	🟪	
CPU integration	Tess	100%									🟦			
Audio														
Audio research	Tess	100%		🟦										
APU design	Tess	100%			🟦	🟦								
~~Waveform output~~	Tess	0%						🟧	🟧					
~~Full stereo output~~	Tess	0%							🟧					
~~Game Boi integration~~	Tess	0%								🟧	🟧			
Testing (and debugging)														
Blarggs Tests	Tess & Adolfo	100%									🟪	🟪		
Mooneye Tests	Tess & Adolfo	100%									🟪	🟪		
Video Tests	Tess & Adolfo	100%									🟪	🟪		
~~Audio Tests~~	Tess & Adolfo	0%									🟧	🟧		
Input Lag Tests	Tess & Adolfo	100%									🟪	🟪		
Joypad														
Make a Joypad Module	Adolfo	100%											🟩	
Joypad RPI driver	Adolfo	100%											🟩	
Connect the RPI to the FPGA through Synthesis	Adolfo	100%											🟩	
Joypad CPU Interconnect	Adolfo	100%											🟩	

🟨	All
🟦	Tess
🟥	Pratyusha
🟩	Adolfo
🟧	Removed
🟪	Tess and Adolfo

Gameboy Memory Layout	
$FFFF	Interrupt Enable Flag
$FF80 - $FFFE	Zero Page (127 bytes)
$FF00 - $FF7F	Hardware I/O Registers
$FEA0 - $FEFF	UNUSABLE
$FE00 - $FE9F	OAM – Object Attribute Memory
$E000 - $FDFF	Echo RAM
$D000 - $DFFF	Internal RAM (Memory banks 1 – 7)
$C000 - $CFFF	Internal RAM (Memory bank 0, fixed)
$A000 - $BFFF	Cartridge RAM
$9C00 - $9FFF	BG Map Data 1
$9800 - $9BFF	BG Map Data 2
$8000 - $97FF	Sprite RAM
$4000 - $7FFF	Cartridge ROM – Switchable Banks
$0150 - $3FFF	Cartridge ROM – Bank 0
$0100 - $014F	Cartridge Header Area
$0000 - $00FF	Restart and Interrupt Vectors


Flash
SoC
Flash Controller
Controller
USB Driver
Controller Driver
Game Switcher
AXI Bus
CPU
CPU HALT
PC
Register File
Address Logic
Memory Controller
Interrupt Controller
Memory
APU Registers
PPU Registers
VRAM
OAM Table
APU
Decoders
Mixer
DAC
Speaker
PPU
Comparators
PPU FSM
Sprite Fetcher
Fetcher
Pixel FIFO
Palette Decoder
VGA Controller
Screen

Mooneye GB Test for instruction accuracy
Excludes tests that target or include features we do support.

Test	Status
add sp e timing	Pass
call timing	Pass
call timing2	Pass
call cc_timing	Pass
call cc_timing2	Pass
di timing GS	Pass
div timing	Pass
ei sequence	Pass
ei timing	Pass
halt ime0 ei	Pass
halt ime0 nointer_timing	Pass
halt ime1 timing	Pass
halt ime1 timing2 GS	Pass
if ie registers	Pass
inter timing	Pass
jp timing	Pass
jp cc timing	Pass
ld hl sp e timing	Pass
oam dma_restart	Pass
oam dma start	Pass
oam dma timing	Pass
pop timing	Pass
push timing	Pass
rapid di ei	Pass
ret timing	Pass

ret cc timing	Pass
reti timing	Pass
reti intr timing	Pass
rst timing	Pass
jp timing	Pass
pop timing	Pass
push timing	Pass
daa	Pass
ei sequence	Pass
ei timing	Pass
jp cc timing	Pass
ld hl timing	Pass
ret cc timing	Pass
ret timing	Pass
reti timing	Pass
rst timing	Pass
oam_dma/basic	Pass
oam_dma/reg_read	Pass
oam_dma/sources-GS	Fail (Not a supported feature)

Mooneye GB Timer tests

Test	Status
div write	Pass
rapid toggle	Pass
tim00 div trigger	Pass
tim00	Pass
tim01 div trigger	Pass
tim01	Pass

tim10 div trigger	Pass
tim10	Pass
tim11 div trigger	Pass
tim11	Pass
tima reload	Pass
tima write reloading	Pass
tma write reloading	Pass

Mooneye GB PPU tests

Test	Status
Hblank ly scx timing GS	Fail
intr 1 2 timing GS	Pass
intr 2 0 timing	Pass
intr 2 mode 0 timing	Pass
intr 2 mode 3 timing	Fail
intr 2 oam ok timing	Fail
intr 2 mode0 timing sprites	Fail
lcdon timing dmgABCmgbS	Fail
lcdon write timing GS	Fail
stat irq blocking	Fail
stat lyc onoff	Fail
vblank stat intr GS	Pass

Blargg cpu_instrs test

Test	Status
01-special	Pass
02-interrupts	Pass
03-op sp,hl	Pass
04-op r,imm	Pass

05-op rp	Pass
06-ld r,r	Pass
07-jr,jp,call,ret,rst	Pass
08-misc instrs	Pass
09-op r,r	Pass
10-bit ops	Pass
11-op a,(hl)	Pass