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 is 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 want to challenge those emulators by creating our emulator on an FPGA to give our users a more realistic experience. Although there are some hardware emulators, they are incomplete and did not aim to perform as well as or better than the Game Boy. In addition to our emulator functionality, we want to be able to switch between games smoothly, which involves loading from and saving game state to persistent memory. To manage this, our System-on-Chip (SoC) will contain game switching logic that will allow the user to select the game they want to play and feed the FPGA the information it should load into its working memory. For user input, we will be using a NES controller because it maps nicely to the input controls required. To assure that we are cycle accurate, we will compare trace logs of our emulator to verified hardware and software emulators. For unit testing, to save us from having to write tests in Game Boy assembly, we will use Blarggs and Mooneye test suites, which are the gold standards in the emulator community. The test suites contain unit tests for memory and timing that assure accuracy and timing.

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 59.73 frames per second: This is the same framerate at which the original ran in. A framerate lower than this would severely impact the user experience due to lowered responsiveness.
- Audio is sampled at a rate of at least 44.1 kHz, the same frequency at which the Game Boy sampled its audio: Sampling at anything lower would lead to lowered audio resolution, affecting the user experience.
- The input latency should be lower than 55 ms: this is based on experiments done with the Game Boy Advance by some of the emulator community. High input latency makes games feel unresponsive, so this is very important.
- We want to have accuracy comparable to Verilog Boy, one of the better hardware emulators we found.

Qualitative Requirements

- Our emulator should be able to save game state and switch between games without having to reflash the FPGA.
- Audio and video should be in sync.
- 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

- The emulator community has spent countless man-hours working on getting the timings for instructions and execution of the Game Boy. Thus, we'll be using the test ROMs that they have come up with to test that the different timings work. These tests are Mooneyes-gb tests and Blarggs instruction tests, which are very comprehensive. These will prove that our emulator has reasonable cycle-accuracy and will allow us to compare the accuracy of our emulators with other emulators. We want to pass at least as many tests as Verilog Boy did, a reasonably accurate hardware emulator.
- 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.
- We will instrument the video code to count the number of frames per second.

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), the Audio Processing Unit (APU) and the timer. The specifics of the communication will be covered when discussing each of these devices. We will have a software component being controlled by the on-board SoC, which also manages game-switching and game-state saving. For our peripherals, the SoC will implement a controller driver and notify the FPGA of button presses through its GPIO pins. We will be using VGA as our display method due to its ease of use and availability. A VGA controller will take the PPU's video output and display it. Finally, we will use a digital-to-analog converter (DAC) to convert the digital audio from the APU and output it from a speaker.

We will upload ROMs to the memory region that is assigned to our CPU, which will then execute a boot sequence and start running the game. 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 will do a certain amount of setup every time a screen is displayed to prepare for the next one. This involves a combination of memory, APU, and PPU. Furthermore, there is also computation being done whenever there are controller or timer interrupts. The following processes are executed every audio and video output cycle.

The CPU will generate the audio that it wants to output for the current time frame by writing to the APU's registers (which are just a specific regions of memory). This audio will be in an encoded format which the APU will then decode it into a frequency/amplitude pair. There are multiple audio streams that will be generated, thus after being decoded the APU will mix them together to create the desired audio. Once mixed together, the audio will be controlled by three more registers, in memory, which dictate how the stream is outputted. This audio is in digital format so it cannot be outputted directly, so we will feed it into a DAC which will output it to our speaker through an audio jack breakout board.

For the video display, the CPU will write the desired image to a designated area in memory, which we'll call 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 will then render 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.

The SoC will not be too involved in this execution process other than generating the interrupts from the controllers. It will be involved with the beginning of execution, and also at the end. At the beginning of execution, it will write all of the game data to the SDRAM, this data is usually contained in a ROM file. We are planning on having some mechanism to allow the user to switch games, probably through one of the FPGA buttons. When a game switch happens, the SoC will 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. Afterwards, all the state will be cleared, and the new game state will be loaded in.

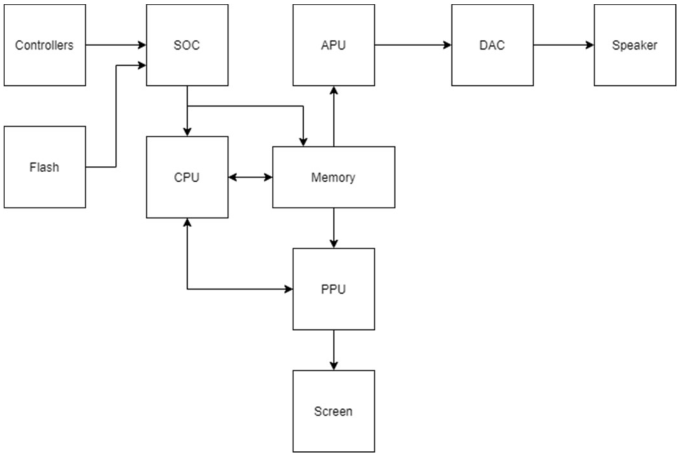


Fig. 1. High-level System Architecture

IV. 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, is very simple. It has two 8-bit input lines, one 8-bit flags input line and a 4-bit operation input. It is only capable of performing 8-bit arithmetic but supports a wide variety of operations. 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 processors flags based on the result which are: Carry, Half-Carry, Zero, and Overflow. The ALU is the main form of interaction between the registers and this same ALU is used to support 16-bit operations. To achieve such operations, the CPU splits the operations into two distinct operations over two cycles.

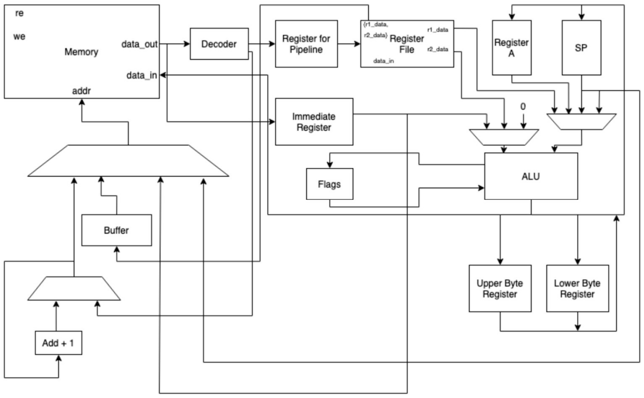


Fig. 2. CPU datapath

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. 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 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. Memory is the main tool for communication between the CPU, the PPU, and the APU. We have added a memory map at the end of the document to illustrate all the distinct memory regions.

To manage these elements of the datapath, we will use control signals to enable different parts of the datapath depending on the current instruction. These control signals are generated by the decoder module, which reads the instructions. The decoder module, along with the instruction fetching, is part of the first stage of the two-stage pipeline in the CPU.

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 will use multiple FSMs. We have divided the FSMs based on the instruction's family which will generate new control signals to be fed into the execute stage of the pipeline.

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.

B. PPU

The PPU (Pixel Processing Unit), handles the rendering of the game's frames for the Game Boy. 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 V-Blank. 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 16. 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 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 which is not able to commit pixels to the screen until it has at least 8 pixels in it. 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 sprites, the sprites at the smaller 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 interrupts that the CPU can enable, these 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). To make up for the missing fetching/rendering interrupts 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 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.

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.

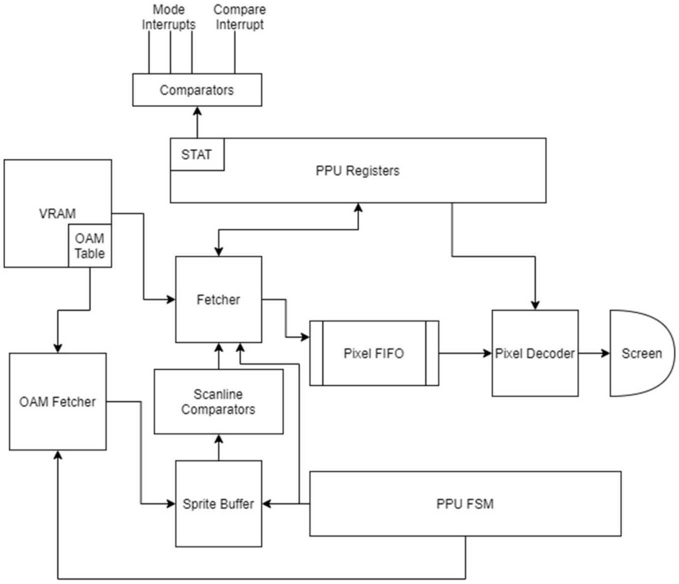


Fig. 3. The PPU system diagram.

C. 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 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.

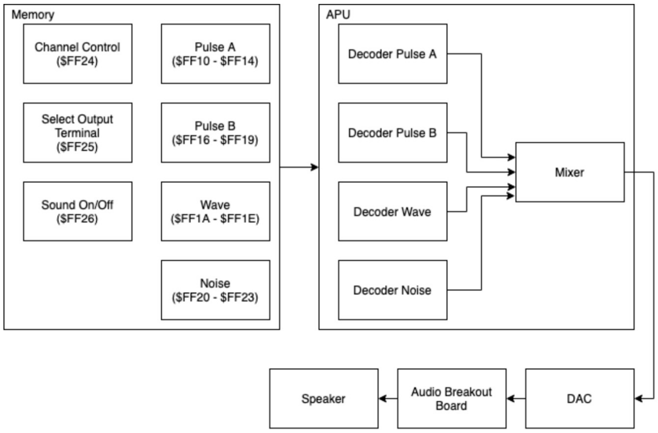


Fig. 4. The APU system diagram.

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.

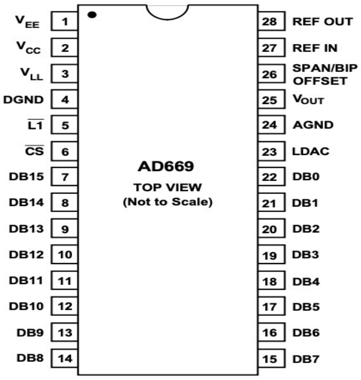


Fig. 5. AD6699 DACPORT chip pinout

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.

D. 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 and load the relevant game memory into the SDRAM. 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. Once we receive the signals through the OTG controller on the SoC, we will choose 8 of the 32 GPIO signals provided to the FPGA, which are controlled through registers in the FPGA Manager on the SoC.

We could have also used 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, and the protocol is not too complicated. However, we think it will be easier to have the SoC receive the signals and transmit them through the GPIO pins to the FPGA since we already plan on using the SoC for Game-switching and state saving, as described below.

The methodology for game switching is simple. We want to load game information being used into SDRAM, and when the player wants to switch games, the SoC will let the current screen finish rendering and then stop the CPU’s execution. Once the execution is stopped, all of the CPU’s state and the memory will be saved for later re-execution in persistent storage (flash memory). Afterwards, all of the state will be cleared, and the new game state will be loaded in from persistent memory to SDRAM.

We plan 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 micro-seconds (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.

To load a game after boot, the SoC will obtain the data from the flash 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. Once we transfer the data to the SDRAM, we will use interrupts or GPIO pins to signal the FPGA to start the new game.

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

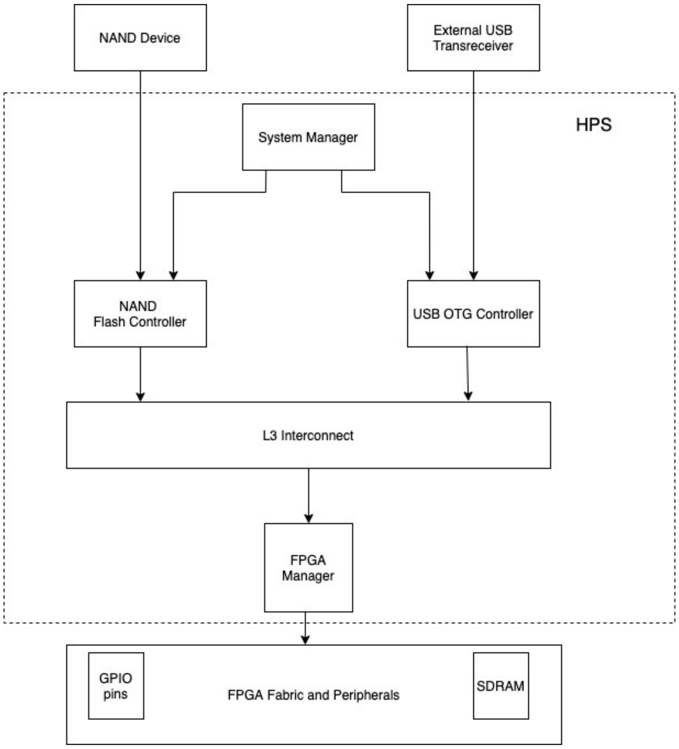


Fig. 6. SoC Sub-System Diagram

V. Project Management

A. Schedule

See last page for our full Gantt chart.

Our schedule is very detailed which goes through 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. Additionally, we are all debugging together because it would be too challenging to try and debug each other's code. Our schedule has remained very similar to our proposal except we have updated the deadline for different tasks and minimized the date granularity.

B. Team Member Responsibilities

Adolfo - Adolfo is primarily in charge of the designing the CPU datapath and overall system layout. He will work with Tess and Pratyusha to implement and debug the CPU and integration. Additionally, he and Pratyusha will work on implementing and verifying memory mapping and communication between the SoC and FPGA. Adolfo is also responsible for researching, designing, and implementing the graphics.

Pratyusha - Pratyusha is primarily in charge of learning and designing how the SoC will need to interact with the FPGA. Once she is done with her design, she will inform the rest of the team so all of us can work together on designing the interaction between the SoC and FPGA. Additionally, she will research, design, and implement the game-switching, controller and controller driver.

Tess - Tess is primarily in charge of designing and implementing the FSMs for the CPU. She worked with Adolfo to design the initial FSM layout, and then designed most of them. Once all the FSMs are completed, she will be responsible for implementing them with the necessary control signals. Additionally, Tess is responsible for researching, designing, and implementing the sound circuitry and APU.

All - As a team, we will work together to implement the CPU. Additionally, we will all work on the testing infrastructure and the integration.

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 need this reader because the micro SD will act as our persistent memory.

For our audio, we needed a DAC to convert the digital signal into an analog signal so we could play the sound through a speaker. Since the DAC is a through-hole chip, we needed an audio jack breakout board to connect the DAC to the speaker through a 3.5 mm audio cable.

For our controller, we needed 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.

Part	Anticipated Cost	Status
DE10-Standard Board	$355*	Received
NES Controllers	$15	Ordered
Micro SD Card	$20	Ordered
AD669 DACPORT	$50	Received
Audio Jack	$20 (for 3)	Ordered
Speaker	$40	Ordered
3.5 mm Audio Cable	$7	Ordered

* 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, but from the emulator community developed documents since Nintendo has not released an official manual. 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 each datapath and design handlers to the best of our ability based on what we have learned. To mitigate this risk, we have reviewed numerous GitHub repositories, watch tech-talks of industry leaders breaking down how the Game Boy works, and assuring our logic supports the necessary cycle count for instructions.

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 have to 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.

VI. Related Work

There are multiple related works that we looked into 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 are 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

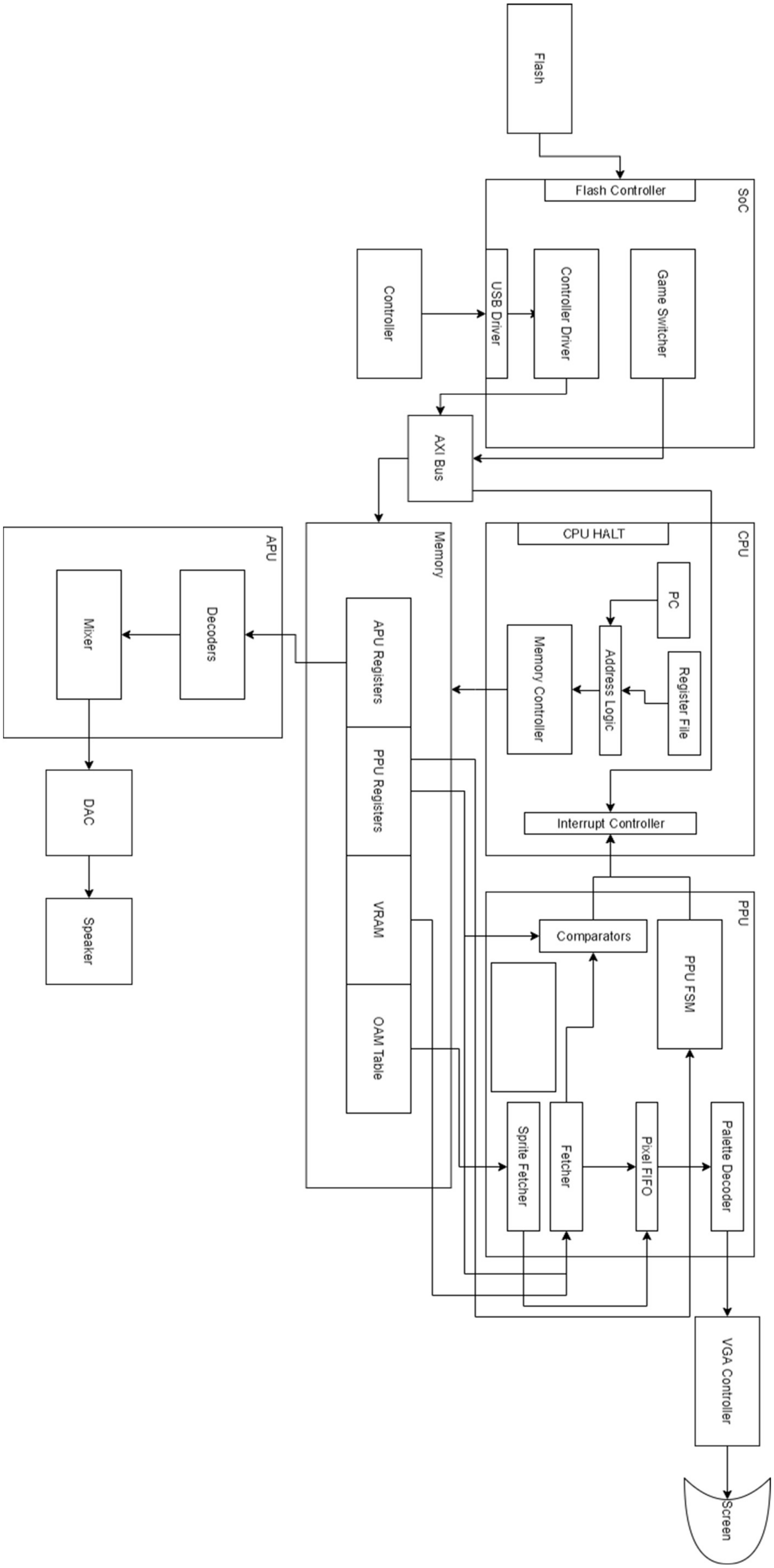
[5] Gekkio/mooneye-gb, https://github.com/Gekkio/mooneye-gb

[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

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: 320	Week 15: 4/27			
Documentation																	
Abstract	All	100%	🟨													🟨	All
Proposal Presentation	All	100%	🟨													🟦	Tess
Design Review Presentation	All	100%		🟨												🟥	Adolfo
Design Review Report	All	50%		🟨												🟩	Pratyusha
Final Presentation	All	0%											🟨				
CPU																	
Instruction Decoder	Tess	0%				🟦											
Register File + ALU	Tess	0%		🟦	🟦												
Memory Controller	Adolfo	0%		🟩	🟩												
PPU interconnect	Adolfo	0%				🟩											
Interrupts	Pratyusha	0%				🟥											
MCU interconnect	Pratyusha	0%		🟥	🟥												
PPU & Graphics																	
VGA Controller	Adolfo	0%		🟩													
Display RAM Reading	Adolfo	0%			🟩	🟩											
Tiles, Background and Window Display	Adolfo	0%					🟩										
Sprites	Adolfo	0%						🟩									
Full Frame display	Adolfo	0%							🟩								
CPU interconnect	Adolfo	0%								🟩							
Sound (APU)																	
Audio Research	Tess	100%		🟦													
APU design	Tess	50%			🟦	🟦											
Waveform Output	Tess	0%					🟦										
Full Stereo Ouput	Tess	0%						🟦									
Game Boi integration	Tess	0%							🟦								
Drivers and MCU																	
Controller driver	Pratyusha	0%					🟥										
Flash Driver	Pratyusha	0%						🟥									
Memory Mapped RAM	Pratyusha	0%		🟥					🟥								
Game State Saving	Pratyusha	0%															
Game Boi MCU interface	Pratyusha	0%			🟥	🟥											
Memory																	
Block RAM	Adolfo	0%						🟩									
ROM loading	Adolfo	0%							🟩								
CPU integration	Tess	0%							🟦								
Testing (and debugging)																	
Blarggs Tests	All	0%									🟨	🟨					
Mooneye Tests	All	0%									🟨	🟨					
Video Tests	All	0%									🟨	🟨					
Audio Tests	All	0%									🟨	🟨					
Input Lag Tests	All	0%									🟨	🟨					

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
Flash Controller
SoC
Controller
USB Driver
Controller Driver
Game Switcher
AXI Bus
CPU
CPU HALT
PC
Address Logic
Register File
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