FP-GAme

Author: Andrew Spaulding, Joseph Yankel
Electrical and Computer Engineering, Carnegie Mellon University

***Abstract*—Software development on retro consoles is often expensive and difficult to understand for new developers. FP-GAme improves on this situation by providing a retro-style hardware environment in a safe and familiar software development environment. The device provides hardware comparable to the Game Boy Advance in power, with 60Hz video at 320x240, multi-layer sprite-engine-based graphics output, and 8-bit PCM audio output at 32KHz. Additionally, the device's price is greatly reduced from other development kits, to under $200.**

***Index Terms*—Design, FPGA-SoC, Gaming Console**

I. Introduction

At this time, application development for "retro" consoles, consoles which are commonly targeted by bare-metal software and use special hardware like sprite-engines, is difficult. Often, development kits for these consoles are expensive, and the process of writing software requires an intimate understanding of the details of the hardware. There are many developers who are interested in creating software for such devices, but who lack the experience or funds necessary to do so. Attempts to fulfil this need have been made in the past, through both software emulation and reprogrammable cartridges to aid in the development process, but these tools fail to provide full access to standard development tools, and often suffer from inaccurate representation of the target hardware. The device outlined in this document will fulfil the apparent need for an inexpensive retro console that provides a familiar and easy to use development environment. This allows prospective developers to obtain the experience they desire without being subjected to the low level details of the hardware. The design will provide hardware similar in style to the GBA to provide a retro experience; it will aim to provide video at 60Hz in 320x240 resolution using a sprite-engine pixel processing unit, and provide 8-bit PCM audio at 32KHz. It will also allow software to be developed and run on top of a Linux kernel, meaning that developers have full access to the standard suite of C functions.

II. Design Requirements

In order for a device to be useful both as a game console and as a development platform, while also capturing the distinct "feel" of a retro system, there are a number of requirements which must be satisfied.

A. Software Requirements

To ensure the most broad accessibility to developers which aspire to use the platform, it must provide a simple C interface to interact with the hardware of the console, and must also provide full access to the C standard. Additionally, a standard C development environment must be made available; specifically, compiling for the system must be at least as simple as cross compiling with gcc and a make file. A consequence of these requirements is that the processor chosen for the device must be of an architecture that C code can generate efficient assembly for.

B. Hardware Requirements

The system must provide hardware that is comparable in capabilities to the most well known retro consoles, thereby providing the developer with a retro experience. The defining feature of the most well known retro consoles is the use of a sprite engine for graphical output. This method of graphical output was used by many consoles, such as the Nintendo Entertainment System (NES)[1], the Super Nintendo (SNES)[2], and the Gameboy Advance (GBA)[3]. These sprite engines typically offered a background layer and a sprite layer, with the background layer being scrollable[1]. These background layers were composed of tiles aligned to a grid, which were typically bitmaps that were interpreted through a runtime-changeable pattern. Individual sprites were also composed of tiles, but could be placed on the screen in any location, though they were not scrollable. On the low end of the capability spectrum, the NES provided 1024 tiles (though only 256 were usable at a time for the background layer) and 64 total sprites with up to 8 sprites visible on a scanline[1]. From these, we form our requirements for a graphics device. It must support at least 256 tiles and 64 sprites total, with at least 8 sprites visible on a scanline at a time. There must be at least one background layer composed of these tiles, and said layer must support scrolling. Additionally, its output resolution must at least match that of the NES, which is 256x240 at 60Hz[1].

The audio hardware for retro consoles is much more varied than that of the graphics hardware. Some consoles used a variety of customizable wave generators[1], however these are difficult for the average programmer to understand and use to their fullest potential. On the other end of the spectrum, the GBA provided 8-bit PCM audio ranging from 8KHz to 32KHz[3]. This method of audio output meshes nicely with the requirements of an accessible and familiar development environment, though it is admittedly at the high end of what is considered retro. Therefore, we require that the audio output of the system be in the range of what the GBA provides, and must not surpass or undershoot it.

Next, like any console, the system must provide a controller for input. There are a large number of retro controllers widely available for purchase. To ensure ease of access to this component, it is a requirement of the device that it use some widely available pre-existing controller as an input method. Additionally, those playing games on a console expect updates in input to be received at least as often as the picture is updated. Since it is required that the video output run at 60Hz, then, the controller input must be processed at least this often as well.

Finally, in order to ensure developers using the device do not need to be excessively concerned with their applications memory footprint, the device must provide a moderate amount of working and program memory. The GBA offered around 300KiB of working memory, and cartridges varying in size from 4MiB to 32MiB[3]. As the GBA is one of the few retro consoles where development was done in C (rather than assembly), it is appropriate to require an amount of memory close to what the GBA offered. At a minimum, the device must have at least 4MiB of program memory and 512KiB of working memory.

C. Testing and Verification of Requirements

On the qualitative end of our requirement spectrum, we have the need for an accessible and easy to use development environment. This can be loosely verified by confirming that the process of preparing the environment on a new machine is brief and simple.

For our quantitative requirements, several tests have been prepared. Concerning the controller input, it would be sufficient to use an oscilloscope to inspect the signals leaving the device and ensure that the controller state is being sampled at at least 60Hz. Similarly, the audio sample rate can be tested by providing a signal whose frequency is exactly half that of the maximum frequency supported by the device and verifying that it doesn not alias. Another test for audio would be comparing the output of the system to a reference output generated by software and ensuring the integrity of the signal. As for graphics output, this can be tested creating and using a software tool which tests the full capability of the sprite and background tile engines by displaying a large number of sprites on screen (and on one scanline) and by scrolling the background layer independent of the sprite layer.

Final testing of all pieces of hardware working together will be done using a simple game which demonstrates all functionality of the device. The game must accept input from the user, use and scroll any non-sprite layers, and utilize the sprite layer to some extent. Said game must also support audio output, at the very least by playing music during gameplay.

III. Architecture and Principle of Operation

The console we will develop is split into 4 major components: CPU, Pixel Processing Unit (PPU), Audio Processing Unit (APU), and I/O Subsystem. Each component is dedicated to solving one or more of the user requirements.

The CPU should be implemented in a modern processor - one which is supported by a common C-toolchain. The PPU, APU, and I/O should be implemented in an FPGA.

The CPU hosts an embedded Linux kernel, under which the user's game-code will run and interact with the rest of the hardware via system-calls to our kernel modules. The kernel modules we write will access the PPU, APU, and I/O on behalf of the user. This is done for two reasons. The first reason is to abstract away hardware details from the user: Remember, our user is using our platform to write retro games on a system with retro console capabilities - not to write communication drivers for our specific hardware. The only details the user will need to know in order to interact with our hardware are the system calls used to interact with our kernel modules. The second reason to mediate hardware accesses via kernel modules is to provide a safe development environment for our user. By this, we mean that we protect the hardware itself from accidental bad accesses.

The PPU takes commands and data transfers from the CPU and uses them to construct 320x240 video output, which is then scaled up to 640x480@60Hz HDMI video output. These commands will allow the user's game code to modify VRAM and PPU control registers, which enables control of a tile-background, tile-foreground, and sprite engine: capabilities which capture the essence of retro-games development.

The APU takes a stream of raw (PCM) 8-bit 32KHz audio samples from the CPU and translates it to HDMI audio signal.

The I/O Subsystem samples the buttons pressed on a modified SNES controller connected via GPIO. The sampling frequency is 60Hz to maximize video/input responsiveness. This allows users to make games which utilize frame-perfect inputs.

Fig. 1 (next page) shows the system's operation from a high-level perspective. From the user's perspective, they will simply flash an SD card with our custom binaries (FPGA Configuration + Custom Embedded Linux) and their compiled game code. Once the game console boots, the FPGA will be configured, the Linux kernel will be booted, and the user's game code will be run. At this point, the user's game code is free to interact with the PPU, APU, and I/O Subsystem via our Linux kernel modules. With our system's capabilities, 2D retro games, such as single-screen games, platformers, or side-scrollers, are possible to build.

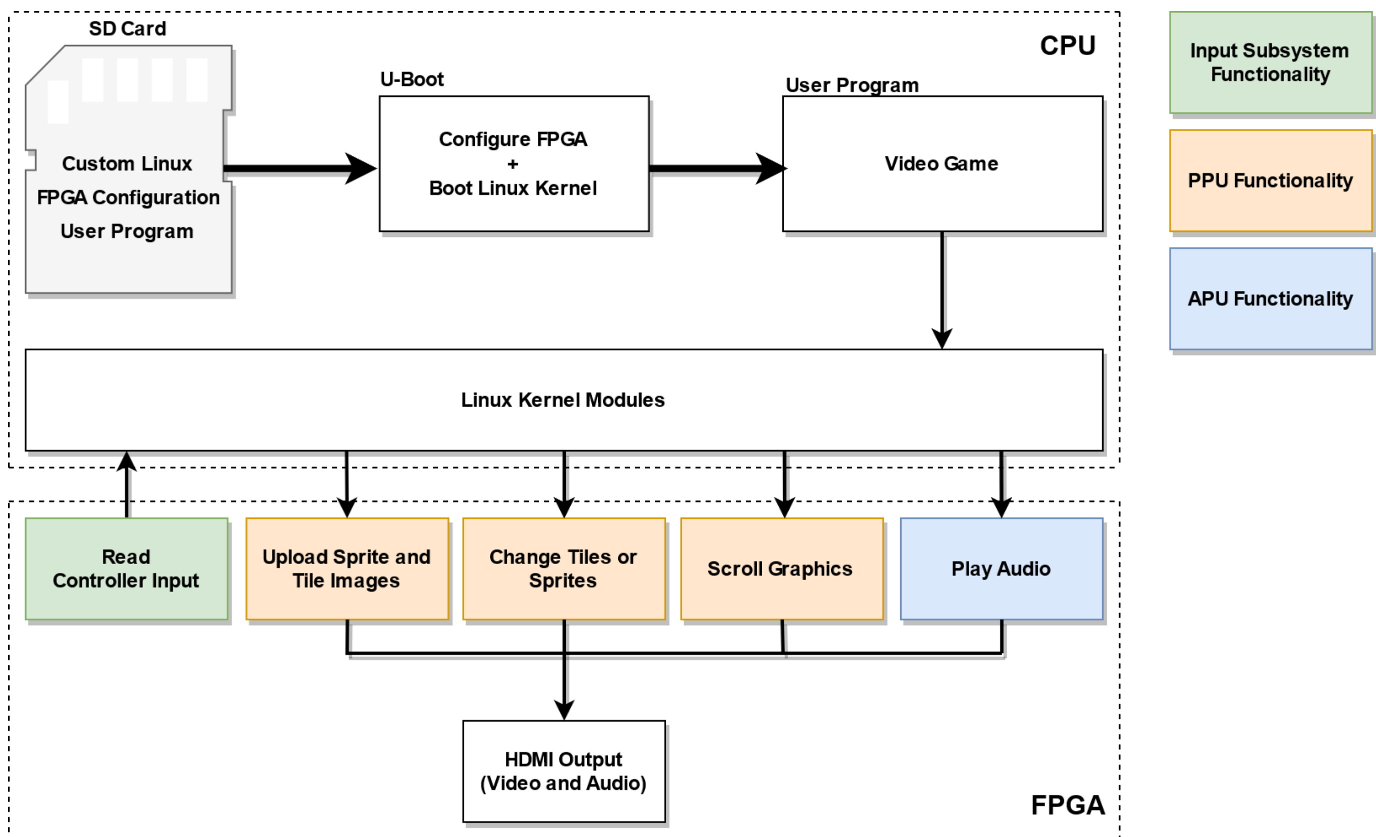

Fig. 1. System Overview Block Diagram

IV. Design Trade Studies

During the development process of our system, several designs were considered. In most cases, these designs were proposed at the module level; there was no overarching system design which was rejected, but rather individual module designs which were either revised or improved upon.

A. Software Design

At the software level, the primary subject of debate was the method by which the expectations of the C standard and the user would be supplied. Two implementations were considered: using a custom kernel to implement the most used functions of the C standard, or using the Linux kernel with custom kernel modules to provide access to our systems hardware. The benefits of using a custom kernel are mainly that of control. A kernel implemented with the specific needs of the system in mind would ensure the simplest implementation of the C API for our device. Unfortunately, the time and energy required to implement a custom kernel made this option especially risky, given the time constraints of our device's development cycle. On the other hand, using Linux would be a low risk option. It would negate the need to understand the lowest level details of the processor, and could be implemented much more quickly. Additionally, the C standard is already fully supported by the Linux kernel, and so we need not provide it under such an implementation. The drawbacks of this approach are that the Linux kernel has its own set of expectations and requirements. Using Linux means confining the device drivers to the rules of a Linux kernel driver, and it also means that those developing for our device must share resources with any Linux processes running in the background. Due to the high risk nature of developing a custom kernel, it was decided that our implementation would use Linux.

B. Hardware Platform Selection

In selecting the hardware platform to build our system upon, our primary concerns were in selecting modules which can be used to meet our design requirements and in ensuring that any communication between these modules would not undermine the performance of the system. Two hardware platforms were considered: The DE10-Nano, and a Teensy 4.0 connected to a TinyFPGA and various other hardware modules.

The primary advantages of using the Teensy and TinyFPGA include their inexpensive cost and extensive support for hardware extensions. Unfortunately, we found many drawbacks with this approach. The barebones nature of the Teensy and TinyFPGA meant that time would need to be dedicated to finding, wiring, and bringing up the video and audio devices themselves. Additionally, communication was a major area of concern in this setup. To expand on this, calculations were performed under the assumption that a user for our platform would send video updates during the vertical blanking period of each frame (which is typical for

retro consoles) and that the user would wish to update roughly 2KiB of data (enough to scroll a full tile, update the new tiles, and rewrite sprite attribute memory) during said period. Since one vertical blank period lasts 1.4ms, this means the user would need to be able to transfer data from the CPU to the PPU at around 11Mbps. This far exceeds the max data rate for I2C, though it may have been achievable under SPI. Yet, this was not the only area of concern. The bare metal nature of the Teensy would have mandated a custom kernel, including a driver that allows user data to be read in from an SD card, as the Teensy only has 2MiB of program storage (which does not meet the requirements). This would have added a non-trivial amount of work to the already difficult task of implementing a custom kernel. Lastly, the TinyFPGA's logic element count is relatively low, which was another area of concern given the number of modules it would need to implement.

Given the sub-optimal prospects for the usage of a Teensy and TinyFPGA, other options were considered. We selected the DE10-Nano as an alternative for several reasons. First, the FPGA on the DE10-Nano is a Cyclone V, which has a built-in HPS unit with an ARM core. Since the ARM core is on the same chip as the FPGA, high speed communication over AXI/Avalon is possible, negating the communication speed risk present with the Teensy. Additionally, the Cyclone V has a large number of logic elements, more than 10 times that of the TinyFPGA. In addition to the increase in logic elements, the Cyclone V has a large number of M10K blocks, giving even more freedom in the design of the PPU/APU, as on-chip caching and memory can be implemented with these. The board itself also features HDMI, meaning no additional devices would need to be connected to achieve both video and audio output. An extensive number of GPIO pins are also present on the board, which can be used as a simple point of connection for the controller. Additionally, the DE10-Nano provides UART over USB, which can be used as a simple method for debugging the software on the device. The platform also features a SD card port, which can be used to boot operating system software, such as Linux, and 1GB of DDR3 memory. Access to both of these means the device far exceeds our working and program memory requirements. Among all these benefits, the only notable drawback was the increased cost. Using the DE10-Nano would double the expected cost to the end users of our system. Yet, in light of the benefits of its use, we ultimately decided on the DE10-Nano.

C. Hardware Design

The design space for the hardware of our device consisted of three main areas: the APU, the PPU, and the controller module.

For the controller module, two controllers were considered for use with our device. These were the SNES and the Sega Genesis controllers; both of which would have met all the specified requirements for controller input. However, the Genesis controller's communication protocol was quite a deal more complicated than that of the SNES controller. Additionally, the Genesis controller used parallel output to provide the current state of the controller, meaning its use would require a sizable amount of wire traffic. The SNES controller, on the other hand, used a simple serial protocol, with the expected clock rate of the controller being an even division of the clock rate of our selected FPGA. Using a serial protocol also means that the SNES controller uses half as many wires as the Genesis controller. For these reasons, the SNES controller was selected for use.

The design variations for the APU center around the method of transferring samples from the ARM core to the APU. One possible method for doing this is to send the address of a sample buffer to the APU over MMIO, and then DMA that buffer into local memory for the APU to send over I2S to the HDMI controller as necessary. Alternatively, the Avalon connection allows the FPGA to directly read the memory of the ARM core, which can be used to pull samples directly from the DDR3 memory.

In considering the method where the audio buffer is transferred to the APU over DMA, our primary concern was with memory usage. This transfer would cause a burst of demand on DDR3 at an undefined time, which may interfere with the users program or with any possible transfer being made by the PPU, which could be problematic. The primary benefit of this method, however, would be the guarantees it places on memory access time when the APU needs its next sample.

The alternative method of using the Avalon bus would place more consistent and spaced out demands on DDR3 memory, however it has different drawbacks. This method requires the CPU to maintain a full copy of the buffer in memory for the duration of its playing time, instead of the duration of the DMA transfer. This, however, is largely not an issue as the APU MMIO interface will be abstracted from any developers using the system. Another potential issue is the delay in accessing DDR3 memory. However, assuming that the APU acquires 4 8-bit samples at a time for 32KHz audio, and that the Cyclone V is clocked at 50MHz, the APU will have 6250 cycles to acquire the next four samples, which far exceeds any possible DDR3 access delay. For these reasons, the Avalon bus method was selected for our APU.

Finally, we considered possible designs for the PPU. Two designs analogous to those considered for the APU were considered for the PPU: one where graphical data is fetched as needed from DDR3 memory, and one where the CPU instead transfers graphics data to internal double-buffered VRAM via DMA once per frame.

When initially designing the PPU, the plan was to directly fetch graphical data from DDR3 memory. This design was made under the assumption that, due to DDR3 memory being on the same physical board as the Cyclone V, the access times would be relatively fast. Unfortunately, our Joseph's optimistic estimates put the lower bound of data access time at 10 cycles, which proved to be an

unacceptable latency for our PPU design. This is made worse by the inconsistent locations in memory in which the PPU will need to pull graphical data from. Moreover, due to VRAM itself being difficult to access, there is no clear safe failure state for if the PPU cannot obtain the tile data in time. For these reasons, a different design was drafted.

Under the revised design, graphical data is instead transferred to memory internal to the PPU once per frame via DMA. The destination of this DMA is not the PPU's primary VRAM for two reasons. First, using a secondary buffer means that if the transfer takes too long the previous frame can simply be repeated. Next, if the transfer was sent directly to VRAM, it would have to take place during the vertical blank period of the frame, which is infeasible due to the short length of this time. Instead, the proposed solution will send data to an internal buffer, which will be transferred to VRAM during the vertical blank period. The drawbacks of this design are that the PPU may not take requests during the vertical blanking period, and that twice the size of VRAM must be used in M10K memory. However, given its great advantages over the other proposed design, this is the one which was selected for implementation.

V. System Description

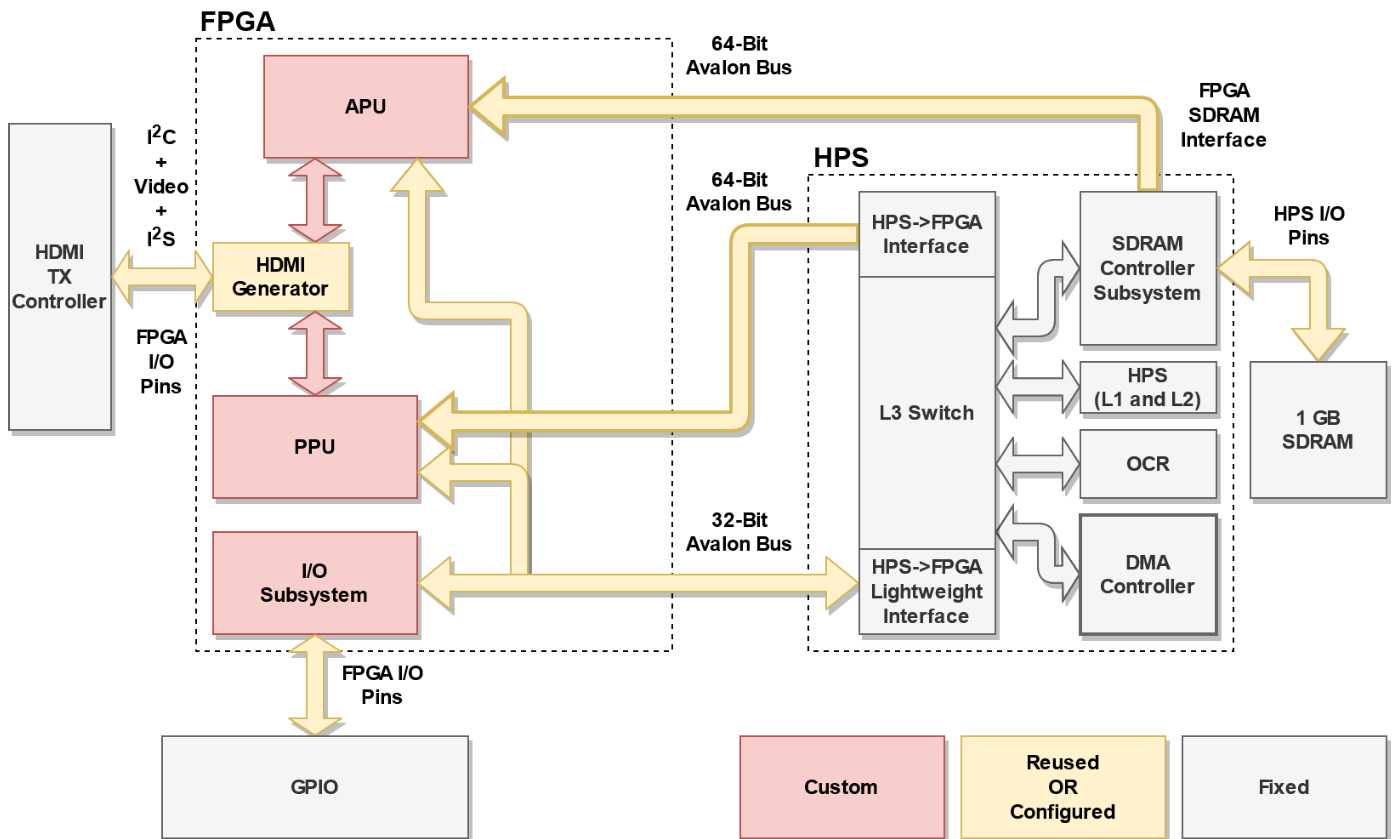

Fig. 2. System Interconnect Diagram

A. System Interconnect

Our system utilizes the DE10-Nano's Cyclone V SoC. The PPU, APU, and I/O are to be implemented in FPGA fabric, while the CPU is to be implemented using the HPS.

Our system interconnect is detailed in Fig. 2. The red blocks are custom logic that we will develop on our own. The grey blocks are fixed or physical devices which we do not implement. The yellow blocks consist of logic from one of three categories:

1. Logic reused from DE10-Nano demo code provided by Altera.
2. Logic generated by Quartus/Platform-Designer.
3. Interconnect which needs to be configured/enabled in Quartus/Platform-Designer.

The major interconnects in our system utilize Intel's Avalon bus architecture due to its simplicity and how easily HDL can be generated and written for it.

The CPU uses a 64-bit Avalon bus connected to the high-speed HPS to FPGA Interface. This bus is used to transfer sprite, tile, and pattern data into the PPU's VRAM. The APU takes a different 64-bit Avalon bus, which is more directly connected to the on-board SDRAM.

A 32-bit Avalon bus is used by the CPU to write to APU and PPU control registers. Altera recommends that this Lightweight HPS to FPGA Interface be used for low-bandwidth requirements such as this [5, section 9.11]. The CPU also uses this interface to read the SNES

controller state from the IS.

To generate HDMI video/audio output, the PPU and APU communicate with a physical HDMI TX Controller chip (ADV7513) via a modified HDMI Generator from Terasic's HDMI_TX demo code. More specifics are included in the HDMI Generator section below.

Not shown in the above diagram are interrupt signals from the FPGA to the HPS. These signals are used to provide the CPU's Linux kernel modules timing signals for the PPU and APU. More detail regarding the specific signals will be covered in the APU and PPU sections below.

B. *HDMI Generator*

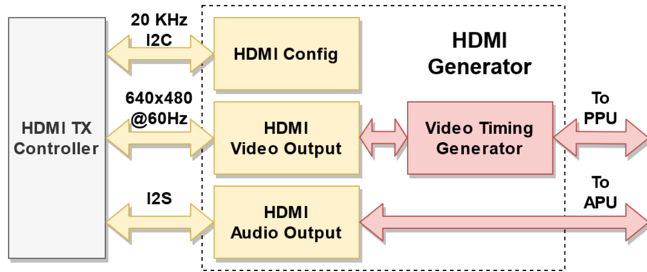

Fig. 3. HDMI Generator Block Diagram

The HDMI Generator consists of mainly reused logic from Terasic's HDMI_TX demo (included in an online CD for the DE10-Nano [6]).

The HDMI Config block uses 20KHz I2C to program the control registers of the ADV7513 HDMI TX Controller IC. the DE10-Nano's external HDMI transmitter chip. These control registers are used primarily to configure the HDMI video and audio formats.

A small FSM loops the I2C write protocol to send out address-command-data strings to the ADV7513. We adapt this HDMI Config block to our needs by customizing the list of commands to send out. In particular, we program the ADV7513 to output 640x480@60Hz video, and 16-bit 32-KHz audio. To meet these formats, our PPU scales up its 320x240@60Hz video output, and our APU scales up its 8-bit 32-KHz PCM audio output.

The HDMI Video Output block is partially implemented by the Terasic HDMI_TX demo. It currently draws a static debug pattern to the screen. We will change it to accept pixel output from our PPU. A notable feature of the HDMI TX Controller is that the video signals it requires as input are very close to standard TV/Monitor video standards such as VGA. The timings for signals such as VBLANK and HSYNC are copied directly from VGA timings. This familiar video signal is automatically translated to HDMI by the ADV7513 IC and sent to the physical HDMI output port on the DE10-Nano.

The Video Timing Generator will bridge the PPU and HDMI Generator by handling precise timings and forwarding the pixel color inputs to the HDMI Video Output when they are needed. The Video Timing Generator will send a signal to the PPU before the first pixel of a frame is needed so that the PPU has enough time to do its memory fetches and calculate the pixel.

The HDMI Audio Output block is also partially implemented by the Terasic HDMI_TX demo. The demo only plays a sine-wave from a look-up table, so we will need to adapt this block to accept PCM samples from the APU's sample buffers.

C. *Pixel Processing Unit (PPU)*

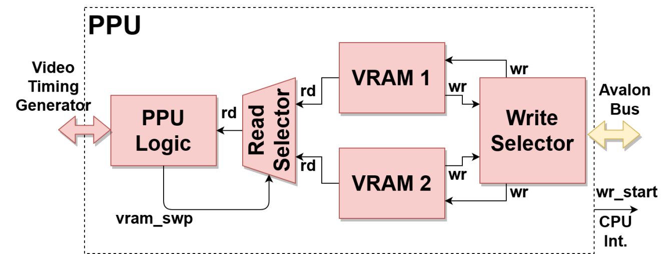

Fig. 4. PPU Block Diagram

At a high level, the PPU is implemented as a MMIO-accessible double-buffered VRAM and set of control registers. Each PPU VRAM is implemented as a collection of smaller, parallel M10K memories, organized by the type of data they store. The types of data and how they are organized in VRAM is explained in Table II below. Separating the VRAM as we have enables the PPU Logic to perform parallel reads to different VRAM sections, which allows for parallelization of pixel-color calculations.

TABLE I. VRAM Organization and Use

Section	*Size*	*Purpose*
BG Tile Memory	8 KiB	For each background tile, stores pattern address, palette address, and horizontal/vertical mirror controls.
FG Tile Memory	8 KiB	Same as BG Tile Memory, but only for foreground tiles.
Sprite Memory	320 B	Stores on-screen position, pattern address, pattern width and height, palette address, horizontal/vertical mirror controls, FG/BG priority.
Pattern Memory	32 KiB	Contains 4-bit color addresses (to index into a 16-color wide palette) for each pixel of every tile or sprite image.
Sprite-Palette Memory	2 KiB	Contains 32 16-color wide 24-bit depth palettes for Sprites.
BG-Palette	1 KiB	Contains 16 16-color wide 24-bit depth palettes for use by the BG tile layer.
FG-Palette	1 KiB	Contains 16 16-color wide 24-bit depth palettes for use by the FG tile layer.

In total, the expected M10K usage by both VRAMs in our design is around 15.03%.

There are two important features of our double-buffered VRAM design:

1. Allows the CPU more time to transfer video data every frame. Instead of being limited to the VBLANK (non-display) period like the single VRAM implementation would require, the CPU can now transfer to the VRAM not in use during

the frame's display time instead of in VBLANK.

2. In the case that the CPU does not finish the transfer the user's program requires in the allotted time, a double-buffer implementation provides a fall-back mechanism: Simply display the next frame using the previous VRAM and allow the CPU to finish the new VRAM transfer. This "drops the new frame", causing a visual pause or stutter, but avoiding visual glitches.

However, these features do not come for free. The double-buffered VRAM design introduces a mandatory sync period where the most-recently updated VRAM must sync its data with the previously used VRAM. It is mandatory for our design, because otherwise, the CPU would need to transfer double the data to keep both buffers up to date - which limits how much useful transfer can be done in between drawing frames. Therefore, when the CPU transfers to the PPU's VRAM, it must obey certain timings in order to prevent overwriting of the buffers while they are synchronizing.

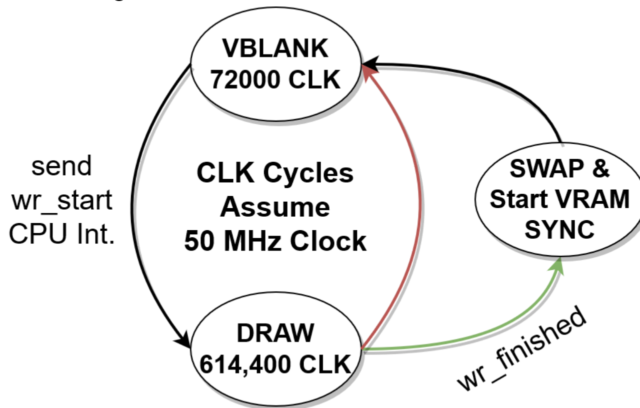

Fig. 5. PPU-CPU Communication Timings FSM

As shown in Fig. 5 above, there are two main states for the PPU concerning the CPU: The PPU is either currently drawing a frame (labeled DRAW) using VRAM, or it is waiting for the start of the next DRAW period (VBLANK). We choose to use this downtime between DRAW states to synchronize and swap the VRAMs.

The CPU can only write to the VRAM not currently in use by the PPU Logic during the DRAW state. We choose to let the CPU write to the other VRAM during the DRAW period (as opposed to the VBLANK period) since we want the CPU to have more time to transfer data over the Avalon Bus. The synchronization between VRAMs during VBLANK is expected to require a relatively shorter duration anyways, since writes between these VRAMs can be parallelized on FPGA-fabric.

Not shown in Fig. 4 is the Lightweight Avalon Bus which allows the CPU to write to control registers directly in the PPU Logic (see Fig. 6). These registers include:

- Layer Enable/Disable: Enable or disable rendering of particular layers independently (Foreground, Background, or Sprite).
- Universal Background Color: Set the default color for the PPU to use when overlapping pixels on all three layers are transparent.
- Foreground/Background Scroll: Independently set the scroll amount in both X and Y directions. This tells the PPU how many pixels to shift the final image by.

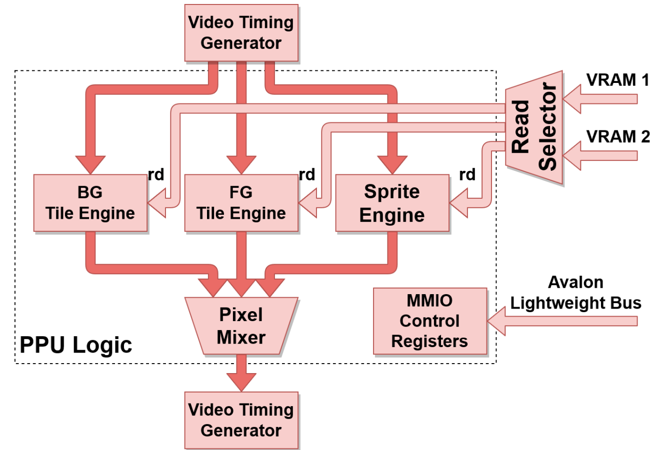

Fig. 6. PPU-Logic Block Diagram

The PPU logic consists of three main sources of pixel color: Two Tile Engines and one Sprite Engine. The two Tile Engines are identical in structure, but calculate the foreground and background tile graphics respectively. Given the current scan line (row), the Tile Engine fetches the tile data from Tile Memory for that row and uses the addresses within to obtain the pixel values needed from Pattern Memory. The Sprite Engine must read all of Sprite Memory and decide based on the current scan-line which 16 out of 64 total sprites to display, minimizing the number of fetches to Pattern Memory required. Once all three Engines pick a color for the given pixel, the Pixel Mixer decides which color to display based on implicit priority (FG takes priority over BG) and sprite priority (sprites can choose whether to appear above or behind the FG or BG).

D. Audio Processing Unit (APU)

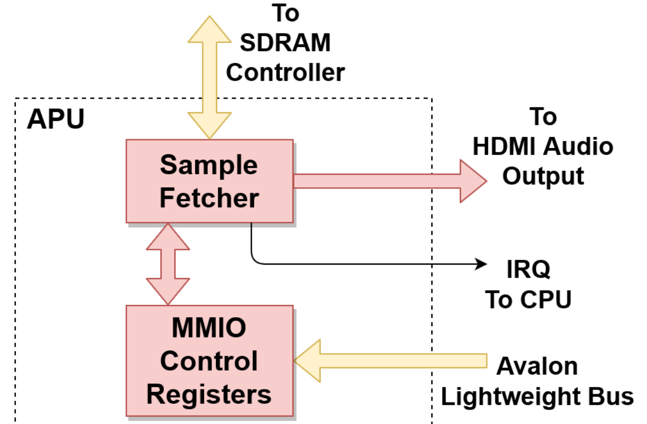

Fig. 7. APU Block Diagram

The APU handles fetching and storing of 8-bit PCM

32KHz sample buffers from SDRAM.

The Sample Fetcher tracks which buffer needs to be filled next and periodically sends the CPU an interrupt to request that a new buffer be placed in SDRAM.

The MMIO Control Registers in the APU allow the CPU to set the address of a newly prepared sample buffer in SDRAM

When the Sample Fetcher requires a new buffer, it reads from SDRAM via the FPGA SDRAM Controller interface, using the address supplied by the CPU in MMIO Controller registers. These samples are upconverted to 16-bit PCM (to adhere to the I2S standard) and are streamed out to the HDMI Audio Output block, which handles the transmission of PCM data to the HDMI TX Controller via I2S.

E. I/O Subsystem

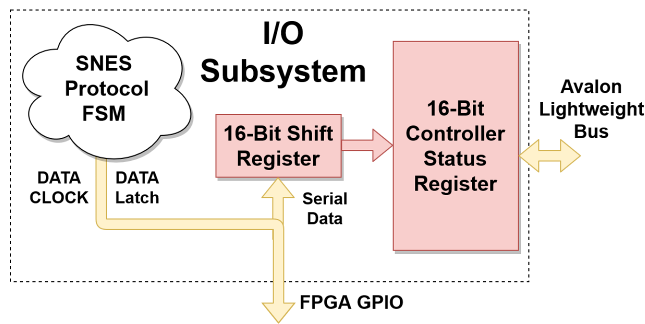

Fig. 8. I/O Subsystem Block Diagram

Our system uses a SNES controller connected to the DE10-Nano's GPIO pins. Reading the SNES controller's inputs requires interfacing with a very simple protocol [7]. The I/O Subsystem drives the Data Latch GPIO pin high every 1/60th of a second, which causes the SNES controller to output the currently pressed buttons in a 16-bit serial stream. These inputs are read into a 16-bit shift register, which is then made available to the CPU via an MMIO status register.

VI. Project Management

A. Schedule

The first major deadline is for the Interim Demo. Notably, the PPU's major features as well as the APU in its entirety are to be completed so that we can showcase the visual and auditory components of our system. Around the deadline, we will begin to develop a prototype game demo to show off these features. We do not expect to fully complete the game demo and user-facing documentation until the Final Presentation/Demo.

Our schedule is noticeably frontloaded, as we expect our other coursework loads to increase towards the end of the semester. We have included 2 weeks of slack time towards the end of our schedule for the purpose of finishing up any small tasks.

See Appendix B for the project schedule Gantt Chart.

B. Team Member Responsibilities

Andrew is working on the Input Subsystem, APU, and most Linux kernel modules. Before the interim demo, Andrew will write a demo game to showcase the functionalities the project has implemented at that point.

Joseph is working primarily on the PPU, and once finished, will work on the PPU driver kernel module. After the PPU work is complete, Joseph will assist in completing the demo game for the Final Report and Presentation.

Both team members will work together on class-related deliverables, such as presentations and reports. Additionally, team members are working together and providing feedback via internal design reviews.

See Appendix B for the project schedule Gantt Chart.

C. Budget

Our project is built to be cheaper than traditional development kits. For reference, the official development kit for the Nintendo Game Boy costs $4000 for a debugger and $3000 for an FPGA-based emulator [4].

There are only two main components to our project that our end users are likely to need to buy:

- The DE10-Nano development board was chosen due to the cost-effectiveness (in terms of features/price) that it offers. The cost for this board is $135.
- A SNES controller is modified by adding GPIO-compatible jumper cables to the controller's 5 output wires. These are attached to the DE10-Nano's GPIO pins, which allows for user input. The SNES controller costs $18.

More detailed information on our project budget and end-user costs is provided in Appendix A. Notably, we separate our personal project costs from those required of a typical user without hobbyist supplies (cables, electrical tape, etc.). Additionally, our personal costs incorporate both the need to order 2x the project materials (due to remote project development) and reuse/scavenging of some items.

D. Tools

To develop for the DE10-Nano's Cyclone V SoC, we use two sets of tools: Quartus and Platform-Designer for building and configuring the FPGA components, and the ARM-none-gnueabihf cross compilation toolchain for code running on the HPS.

Quartus compiles our SystemVerilog and Verilog HDL into a bitstream we can flash onto the SD Card along with the Linux kernel and U-Boot bootloader.

Platform-Designer (formerly Qsys) is a sub-component of Quartus which automatically generates interconnects between IPs in a design. We use it to generate the Avalon Buses between the HPS and FPGA components for our design.

Additionally, for verifying the Input Subsystem's polling rate, Joseph will use his oscilloscope.

E. Risk Management

We have identified two major risks in the project so far:

1. Unreliability in timing requirements with the PPU and DDR3 memory access
2. Linux Kernel Module APU Uncertainty

The first major risk mitigation strategy we employed was to redesign the PPU to be less dependent on SDRAM timings. Our research into the Cyclone V SDRAM controller and the external SDRAM IC showed us that SDRAM contention between the CPU, PPU, and APU causes read latency to be inconsistent. Since the PPU's video output requires consistent timings, and we could not place a reasonable upper bound on the SDRAM read latency without major assumptions, we decided to rework the PPU's interaction with memory entirely. The PPU's design now utilizes an M10K FPGA-fabric memory, which provides consistent read and write timings. The CPU to PPU memory write timing requirements were relaxed with a double-buffer fall-back mechanism, ensuring no visible glitches would occur in the case that the CPU spends too long writing to PPU memory.

Another risk we have identified is in the APU system call interface. Our APU system call will require a callback function from the user to fill a new buffer. It may not be possible to write a Linux kernel module which can call a user mode callback function without also allowing the user to "escape" into kernel mode; creating an unacceptable security risk. If it proves impossible to directly force the user into a callback function from the kernel module, an alternative would be to use one of the C standards reserved signals (SIGUSR1 or SIGUSR2) as a means of executing a callback, and instead having the kernel module send a signal to the users process as appropriate.

References

[1] https://www.copetti.org/writings/consoles/nes/
[2] https://www.copetti.org/writings/consoles/super-nintendo/
[3] https://www.copetti.org/writings/consoles/game-boy-advance/
[4] https://www.retroreversing.com/gameboy-development-kit-hardware/#is-cgb-emu-nintendo-game-boy-color-emulator
[5] https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/cyclone-v/cv_5v4.pdf
[6] https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=205&No=1046&PartNo=4
[7] https://github.com/marcosassis/gamepaduino/wiki/SNES-controller-interface

VII. Appendix

A. *Project Bill of Materials (BOM)*

TABLE II. Development and End-User BOM

Development BOM			End-User BOM		
Item	**Quantity**	**Price**	**Item**	**Quantity**	**Price**
DE-10 Nano	2	$135	DE-10 Nano	1	$135
SNES Controller	1	$17	SNES Controller	1	$17
GPIO-Compatible Jumper Cable Set	1	$7	GPIO-Compatible Jumper Cable Set	1	$7
SNES Controller	1	**Reused**	Electrical Tape	1	$5
GPIO-Compatible Jumper Cable Set	1	**Reused**			
Total Price		294	**Total Price**		164

Links to purchase pages for project materials are included below.

DE-10 Nano FPGA Board ($135):
https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=1046

SNES Controller ($17):
https://retro-bit.com/ryu-dual-link-controller-snes-pc-mac.html

GPIO-Compatible Jumper Cable Set ($7):
https://www.amazon.com/Elegoo-EL-CP-004-Multicolored-Breadboard-arduino/dp/B01EV70C78/ref=sr_1_6?dchild=1&keywords=jumper+wires&qid=1615682122&sr=8-6

Electrical Tape ($5):
https://www.amazon.com/Wapodeai-Electrical-Temperature-Resistance-Waterproof/dp/B07ZWC2VLX/ref=sr_1_3?_encoding=UTF8&c=ts&dchild=1&keywords=Electrical+Tape&qid=1615682353&s=industrial&sr=1-3&ts_id=256161011

B. Project Gantt Chart

TABLE III. FP-GAME GANTT CHART

FP-GAme Schedule											
Task **Week**	**2/22**	**3/1**	**3/8**	**3/15**	**3/22**	**3/29**	**4/5**	**4/12**	**4/19**	**4/26**	**5/3**
CPU											
Define System Call Interface		A/J									
Implement PPU Driver							J				
Implement Audio Driver					A						
Implement Controller Driver			A								
Implement Full Test Game							A		A/J		
Slack										A	A
PPU											
Design PPU Interface		J									
HDMI - Video/Audio Bringup	J										
Tile Engine - Single-palette, Non-scrolling Prototype			J								
Tile Engine - Palette-Indirection					J						
Sprite Engine						A					
Tile Engine - Scrolling						J					
Slack				J			A/J			J	J
Audio											
Design APU Interface		A									
Research I2S	A										
PCM Sample Buffer Transmit over I2S				A							
APU -> CPU Interrupt on Buffer Empty					A						
Slack					A						
Input											
Design Controller Interface	A										
Decide on Controller	A										
Implement Input Protocol		A									
Expose to CPU (Bring up FPGA -> CPU Interface)			A								
Slack				A							
Class											
Design Presentation Slides		A/J									
Design Presentation Report			A/J								
Interim Demo								A/J			
Final Presentation										A/J	A/J
Final Video										A/J	A/J
Final Report										A/J	A/J