AsymDeck
Final Report

Jonah Romero and Kenneth Collins

***Abstract*—In this paper, we present an implementation for a RISC-V32I video game console system and two proof-of-concept games to run on the system. The console includes multi-controller support, interchangeable game cartridges, and video support. We used the Xlinix Spartan-7 FPGA to implement a RISC-V32I processor, HDMI support, controller controllers, and a cartridge reader. Additional hardware included an 8-bit latch (SN54LS373), and 8kb EEPROM (AT28C64). The console is fully functional and can run two proof-of-concept games (chess and paint).**

***Index Terms*—Videogame Console, Xilinx, RISC-V32I.**

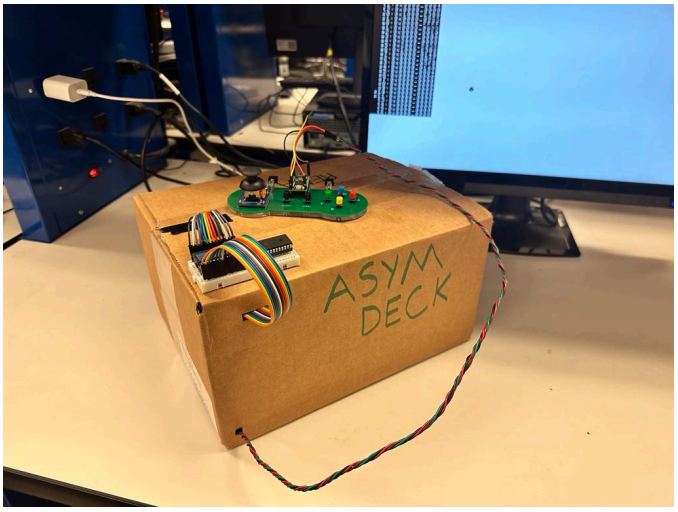

Fig. 1. AsymDeck running the paint game with the processor debugger on.

I. Physical Construction (Jonah)

THE video game console is constructed of several components that communicate to provide a full system experience. There are three essential features for a good generic video game console: a large and diverse supported game library, wide support of different controller inputs, and good graphics rendering. To implement these features, we utilized several pieces of hardware:

- A AT28C64 EEPROM containing two thousand RISCV-32I source instructions.
- A SN54LS373 8-bit latch to provide additional GPIO.
- An AMD Xlinix Spartan-7 FPGA.
- One Teensy 3.2 Controller managing user input

Jonah Romera is with the Department of Electrical Engineering and Computer Science, Cambridge, MA 02139 e-mail: jblt@mit.edu.

Kenneth Collins is with the Department of Physics and the Department of Electrical Engineering and Computer Science, Cambridge, MA 02139 e-mail: kengcol@mit.edu .

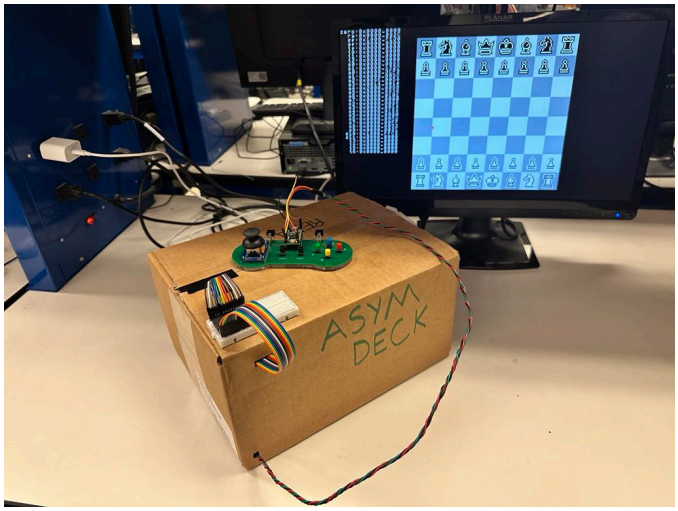

Fig. 2. AsymDeck running the chess game with the processor debugger on.

A. Controller

The controller uses a custom PCB that connects a Teensy 3.2 to a joystick and a total of 8 different buttons. The teensy's job is to constantly poll these inputs and maintain a record of which buttons are pressed and where the joystick is. At the same time, this information is constantly streamed out of two spi lines (MOSI and SCLK). This data is read into the FPGA through its GPIO ports and stored in some internal registers. These registers will be memory mapped, allowing a user program to access the controller state.

One issue encountered while interfacing with the controller is that the Arduino switches the data line very close to where the FPGA samples. Combining this with the noise makes it difficult to latch onto the correct values correctly. As a result, the original SPI code written in the lab had to change, and a slightly modified SPI was used.

Additionally, there were difficulties synchronizing the data transfer between the FPGA and the Teensy. We needed to transfer three bytes in total (two for the joystick's x-tilt and y-tilt, and one byte for a bit mask of all the buttons). It often happened that the FPGA would interpret an x-tilt as a button mask, or a y-tilt as an x-tilt, or some other combination. To fix this issue, another module was created to synchronize the two devices, and a fixed order that bytes were sent in. The easiest solution was to create a start sequence, but we did not want to tolerate any false positives. For example, if we use a starting delimiter 'A', it is possible to press the 8 buttons in a way that mimics this starting character. To fix this, we used the start character 0xFF, and we sent each button as its own

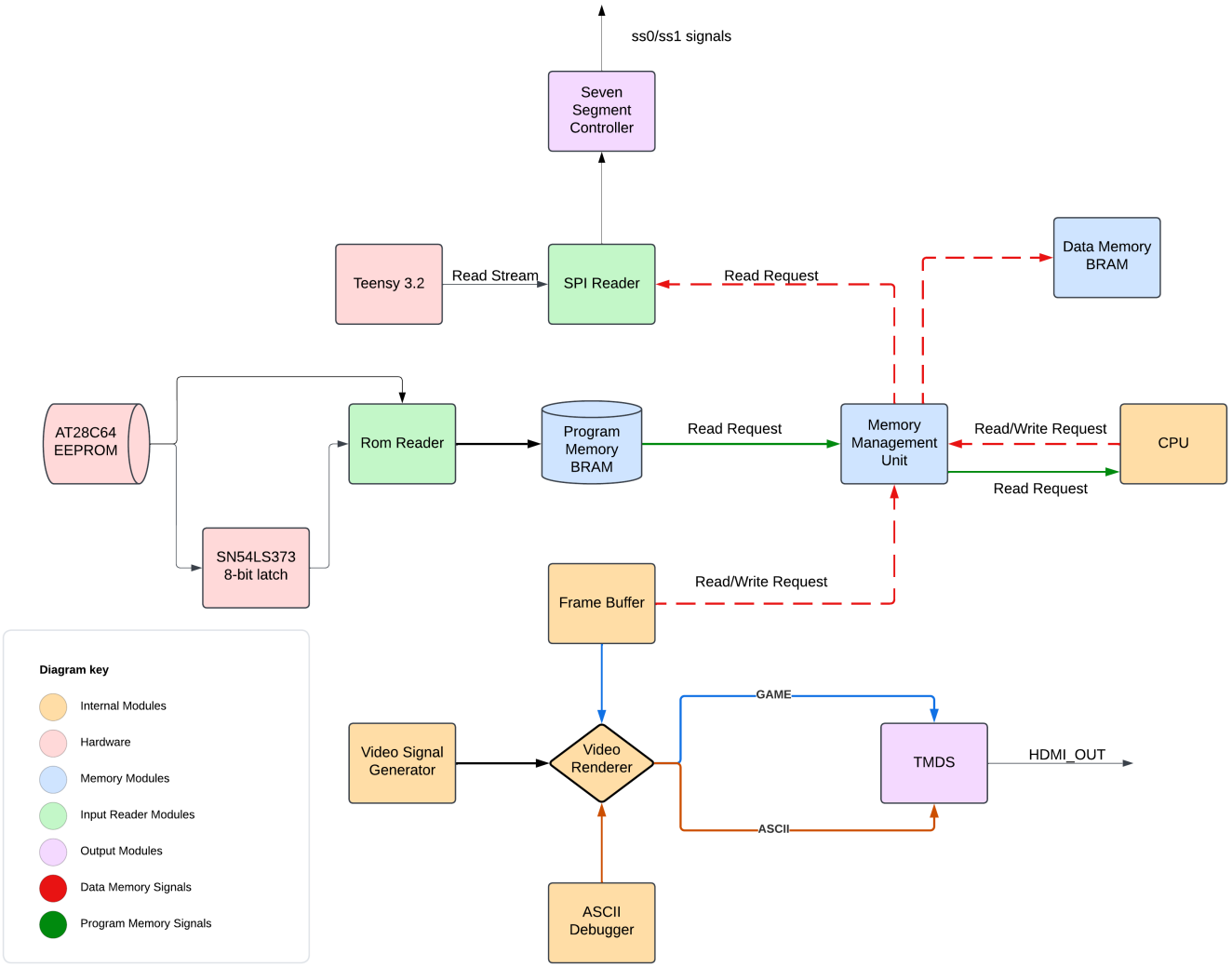

Fig. 3. A block diagram of the AsymDeck that shows the memory request signals between each module or hardware. The red dashed lines represent the communication signals between each module and the Data Memory BRAM. The green line represents the Program Memory signals. The decision block representing the Video Renderer illustrates how it takes input from the frame buffer and the built-in debugger and sends one of them to the TMDS, depending on the current rendering state.

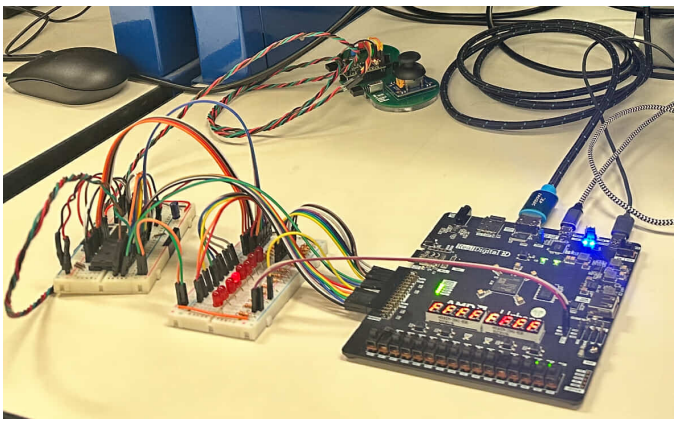

Fig. 4. AsymDeck Internals

byte, making it impossible to misinterpret the communication stream.

B. External EEPROM

The AsymDeck does not save the game internally, and instead loads game data from an external rom. Hence, we can implement the classic game cartridges we're all so used to on consoles.

The AT28C64 EEPROM is an 8Kbit storage that can easily be flashed with new video games. The IC is a cheap alternative to more expensive memory storage devices, only costing $2. One problem with using this IC is that our FPGA only has 22 GPIO pins, and our EEPROM has a 16-byte address bus and an 8-byte data bus. Moreover, we are already using two GPIO pins to receive controller input. To reduce this pressure, we introduce a latch that reduces the number of GPIO pins we need by 7. When we want to read an address from the EEPROM, we set an 8-bit address bus to the lower 8-bits of our target address. We then send a signal to our external latch

to latch the values, and finally, we set the 8-bit address bus to the high bits of our address. In this process, we let the latch hold onto some of the values for us. With this major problem solved, we are able to iterate through all the addresses and load two thousand instructions onto our board.

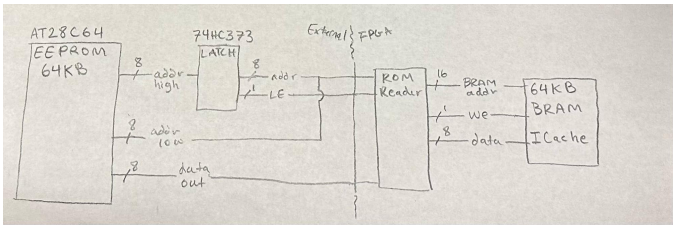

Fig. 5. ROM Latching Diagram

Difficulties here lie within the timing of all our signals. Similar to the latching logic in class, these chips also have setup times, hold times, and propagation delays. A special FSM was implemented to take all of these variables into account. This ensures that we correctly load the external program into an internal instruction cache that the processor can use. During debugging, an LED array was set up to watch the inputs being received by the FPGA. This allowed us to do single steps and watch the latching protocol in real time. This LED array can still be seen in the final product despite it not having a functional use.

II. MMU (Jonah)

The system already has several different memory banks: ROM, RAM, Controller IO, and frame buffers. This is a lot to keep track of, and we want the processor to be agnostic to this separation. To address this issue, we create a single memory management unit that accepts memory requests and forwards them to the appropriate memory system. A memory request consists of:

- A memory address
- A memory width (byte, half-word, or word)
- The memory operation (read or write)
- A value to write in the case of a write request.

For this project, we define the word as 32 bits and the half-word as 16 bits. The memory address received by the MMU maps to one of the memory banks, resulting in the following ranges:

Memory Sub-system	End Addr	Start Addr
ROM	0x0000_FFFF	0x0000_0000
RAM	0x1000_FFFF	0x1000_0000
Frame Buffer	0x201C_200F	0x2000_0000
Controller IO	0x3000_0004	0x3000_0000

The MMU is ideally a 0-cost abstraction over all the sub-memory systems it manages, reducing it to a bundle of combinational logic. However, one key detail limits this, and that is the fact that memory width is also a specification on the bus. This means memory requests of any size should be valid. However, by default, many of the memory sub-systems only support fixed width sizes. The memory system thankfully uses an FSM to correctly forward data in a size appropriate way. For example, since our RAM's width is only a byte, this takes four cycles. This is the best we can do. If we were to make our RAM's width 32 bits, writing to a byte or half-word would require 2 cycles to load an address and another cycle to write the new word back.

While writes are efficient, reading from memory does incur a cost with our MMU since transactions are synchronous and reading takes two cycles. This results in a missed opportunity to pipeline. In the future, it would be helpful to offer more memory functionality. For example, offering bulk memcpys would speed up several programs we were running on the console.

III. C Infrastructure (Jonah)

To enable a seamless game developer experience, we leverage the GCC suite for RISC-V32I to create game binaries compatible with our system. We define a custom linker script that considers our system's memory layout and builds binaries with the resulting .text and .rodata sections. We define several helpful headers for interacting with the device peripherals and a custom-build script that takes the developer's source code and produces an Intel Hex file for flashing to the game cartridge.

Additionally, a simple rendering library was created to render sprites, such as the chess pieces in our chess game. Accompanying scripts take images and convert them into the associated C arrays.

A. Proof Of Concept Games (Jonah)

To test our system fully, we developed two simple applications that can be tested: a painting simulator and chess. We originally had concerns about whether the two thousand instruction limit would be enough to demonstrate a game. However, both games are well within the instruction limit. The painting simulator takes 321 instructions to implement, and the chess game takes 1483 instructions. As mentioned later in the CPU section, the "M" extension could reduce these program sizes. Before we had developed this extension, the games would have to manually compute division, multiplication, and modulo with for loops, causing larger binaries.

IV. Video Game Graphics

We have designed and tested the AsymDeck to render games at 60 frames per second with a 720p resolution on a PLL2210W monitor. However, the frame rate does vary game-to-game. We used the HDMI port on the board and modules from HDMI lab assignments to do so. The HDMI signals are encoded using the Transmission Minimized Differential Signaling (TMDS) encoding scheme. We used the variable `hcount_hdmi` to denote the horizontal position of the pixel and `vcount_hdmi` as the vertical position of the pixel. In addition, we used the following timing dimensions for the sync, front, and back porches:

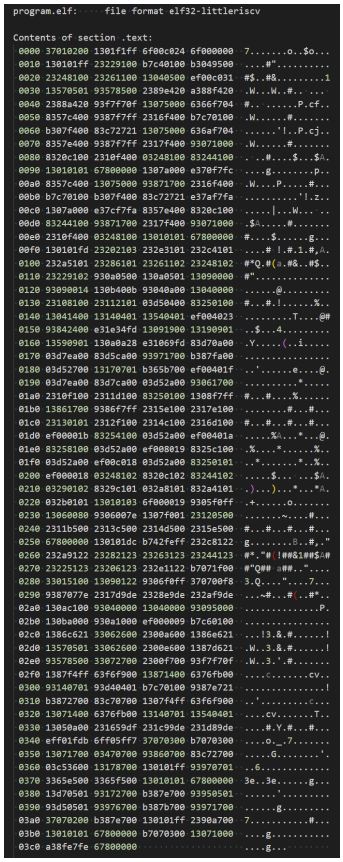

Fig. 6. Paint Game Binary.

- Horizontal Front Porch: 110
- Horizontal Sync Width: 40
- Horizontal Back Porch: 220
- Vertical Front Porch: 5
- Vertical Sync Width: 5
- Veritcal Back Porch: 20

Our new extensions from this preexisting design include using two frame buffers and displaying overlaid ASCII text for debugging.

A. Frame Buffer (Jonah)

In previous labs, while mentioned, screen tearing was never addressed. The camera was running at comparable speeds to the monitor rendering, so the effects were minimal, especially if there were not any fast-moving objects. We do not have such fortunate circumstances with the game console. For example, a complicated game will have more sprites it needs to draw, and it will not be running even close to the same speed as the HDMI protocol, and screen tearing will be painfully obvious. To fix this, we provide the user with two frame buffers. They're mapped to the same memory space (0x201C_200F - 0x2000_0000). However, only a single one is active at a single time. The console can toggle between frame buffers by writing to 0x2FFF_FFFF (Frame Buffer Swap). The inactive frame buffer is passively written to the monitor. The resulting C code would invoke: draw_frame(); switch_frame_buffer(); resulting in no screen tearing since we only render finished frames.

B. ASCII text buffers (Kenneth)

We implemented text buffers to display the contents of the processor's 32 registers, opening the possibility of text-based games and other games that use text. We control whether the game or ASCII text is written on the screen by transitioning between two states in the video renderer: GAME and ASCII.

We created a 5 x 7 bitmap of each alphabetical English character (uppercase and lowercase), the numerical characters, and a few special characters than the standard 8 x 8 bitmap to minimize board utilization. The FPGA would select an ASCII character by using the relevant ASCII Hex value. This Hex value would be generated from an ASCII interpreter that took in values and indices of unpacked arrays to determine the appropriate ASCII index. We used the index of the unpacked register file array to determine which symbolic characters to write on the monitor. Then, the values in each register are used to determine the appropriate Hex value to display at a specific point on the screen. We chose to display the register's contents in the hexadecimal format as we saw it as the most readable format while still allowing character compression.

Each character's starting point on the screen was determined by how much space each character would take on the screen and how many more characters were left to display. Since the original bitmap was too small to read, we doubled the bitmap and padded it with zeros to create a 16x16 bitmap. We chose to pad the bitmap with zeros to make determining the position of where to display the bitmap easier since the last four bits of `hcount_hdmi` and `vcount_hdmi` could be used to traverse the modified bitmap.

Then, we determined each ASCII character's starting point by right-shifting the current position by 4. This scheme allows us to avoid the issue of needing either a complicated module or long if-elseif-else statements to do the same task.

Due to the screen rendering line by line, the number of cycles to render one text line is 16 times the bottom right position of the last character in the text line.

As of now, the text buffer is implemented for the top left portion of the screen, but it would not be difficult to extend this logic to the rest of the displayed screen. Doing so would open the possibility of supporting text-based games since it would only require allocating space on the board and an ASCII interpreter that reads from it when signaled by the processor.

V. RISC-V32I CPU (Kenneth)

One of the challenges in developing a CPU was overcoming the issue of all of the combinational logic happening simultaneously. As a result, we had to think about balancing the number of states in the processor FSM with the number of flip-flops used. Since our proof-of-concept games did not require the fastest processor, we targeted a processor that handles one instruction at a time. This design would allow us to avoid the issues of data and control hazards that would present in a properly pipelined processor.

A. Single-Instruction Processor

We divided the processor into four stage FSM:

- **Fetch-Decode Stage**: The processor sends a read request to the MMU using the program counter (PC) as the address. Since the instruction is stored in BRAM and has a double word length, the instruction has a two-cycle latency. The processor stays in this state until the instruction is returned, then it decodes the instruction and transitions to the Execute Stage.
- **Execute Stage**: The processor uses the decoded instruction to call the relevant non-memory modules and return the results in one cycle. It transitions to the MEM stage.
- **Mem Stage**: In this stage, the processor handles any memory-related instruction that can have varying latency, as the logic only occurs when the MMU is not busy. When it is not busy, it makes the relevant request and tells the MMU the length of the data it wants to read or write from.
- **Writeback Stage**: If the instruction is not a memory instruction, this stage has a single-cycle latency. If it is a memory instruction, it will also have a varying latency, taking 2 - 8 cycles before it can transition back to the Fetch-Decode Stage to process the next instruction.

Currently, the processor only handles RISCV-32I instructions, but we hope to extend it to include the instructions from the ”M” and ”F” extensions. The ”M” extension would enable a reduction in the instructions needed to program games that use multiplication and/or division. Therefore, a reduction would enable more complex and longer games to program as fewer instructions would be needed to code the game logic. We have implemented the ”M” extension, but we were not able to put it on the board before the deadline.

We also planned to program the ”F” extension in the console, as it is needed for some of the more advanced retro games, such as Quake, that we wanted to support. Furthermore, to support these games, significant upgrades to the processor would be required, such as handling more than a single instruction, handling data and control hazards, and other optimizations to run the games beyond the sub-1 frame per second.

While this implementation is not what we initially had in mind, it is sufficient for our purposes since the targeted games do not need much computational power.

B. Debugger

One of our biggest challenges in this project was simulating the AsymDeck due to the hundreds of instructions that needed to be checked to track an error properly. Normally, the way to check to make sure a processor is handling each instruction properly is to check the state of the registers. However, due to software specifications, it was hard to obtain a readable signal from the packed array.

Our solution to this issue was to display the registers file on the screen using the ASCII text buffers while the game was running. We created a 256 x 512 text buffer on the top left of the display since there were 32 registers to display and 16 characters needed to write the register's symbolic name and its current value. In addition, we added a mode to the processor that would hold an instruction until either given new user input or the mode was turned off so that the user can see the registers update with each instruction.

We also planned to add a feature allowing the user to halt the processor on a specific instruction by turning on the mode and inputting the PC via the switches on the board.

VI. Retrospective

We have learned a few lessons through development:

- We learn that not all hardware languages are designed the same. In our case, it resulted in the initial design of the processor being canned and creating a single instruction processor inspired by it. The main issue is that in translating the processor from MiniSpec (more software-oriented HDL) to SystemVerilog (more realistic HDL), we lost the nuance that combinational logic happens simultaneously rather than line by line. It cost us tens of hours of work as we spent time in office hours not realizing this issue. We only wished we had realized it sooner as the progress on the stretch goals since switching has been moving much faster and would have created a possibility of implementing the retro game support.
- It would have been helpful to use a RISCV-32I emulator or develop the debugger sooner, as debugging the processor was a long and arduous task. Testing the code on a working processor would have been much simpler than the process we used. One person would read the assembly instructions and mentally calculate the result, while the other person would check the simulated waveform to see if it matched.
- Several modules existed in our code base, with large buses that needed to be connected to multiple other modules. We opted to use SystemVerilog interfaces to minimize errors when copying these buses. These provided a very helpful abstraction. However, they were incompatible with the Cocotb and icarus setup we used in class. This forced us to use the Vivado GUI, while it was difficult at first setup. It ended up being very helpful. The waveform viewer felt better than GTKWave, and we were able to quickly add IPs and other FPGA configurations. For example, it made adding a configuration memory device very easy, which we wanted so that way the console could be power cycled without losing our build. In the future, it would be nice to try using other simulators that support interfaces with Cocotb since it was very pleasant to use the higher-level language features.
- Debugging a physical system with external hardware made debugging parts of the system difficult at times, such as the ROM and controller input. Debugging these systems included manual inspection of the code, oscilloscopes, and LED arrays. It would have been very helpful, however, to have known about the ILA (Integrated Logic Analyzer) IP sooner, as it was able to provide system signals in real-time as the system was interacting with real devices.

APPENDIX

The source code for the project is [redacted]