

S.P.U.D.

Single-core Processing Unit and Display

Final Report Document

EEL 4914 - Senior Design I

Group C (27)



University of Central Florida

Department of Electrical and Computer Engineering

Authors:

Valentina Terry	Computer Engineering
Mohamed Moussa	Computer Engineering
Gabriel Saborido	Computer Engineering
Frank Laterza	Computer Engineering

Reviewers:

Dr. Sazadur Rahman
Dr. Suboh Suboh
Dr. Saleem Sahawneh

Advisors:

Dr. Mike Borowczak

Table of Contents

List of Figures.....	vi
List of Tables.....	viii
Chapter 1: Executive Summary.....	1
Chapter 2: Project Description.....	3
Chapter 3: Research.....	13
Chapter 4: Standards and Design Constraints.....	58
Chapter 5: AI Resources.....	69
Chapter 6: Hardware Design.....	78
Chapter 7: Software Design.....	93
Chapter 8: System Fabrication.....	103
Chapter 9: Prototype and System Testing.....	107
Chapter 10: Administrative Content.....	113
Chapter 11: Conclusion.....	119
Appendix A: References.....	121
Appendix B: Copyright Permissions.....	125

List of Tables

2.1 Project Goals.....	6
2.2 Specifications.....	8
3.1 Technology Comparison and Selection.....	21
3.2 Modules and Selection.....	23
3.3 FPGA Comparison & Selection.....	30

3.4 Level Shifter Technology Comparison & Selection.....	34
3.5 Level Shifter Module Comparison & Selection.....	36
10.1 Budget.....	115
10.2 Bill of Materials.....	115
10.3 Workload Distribution.....	117
10.4 Initial Project Milestones.....	118
10.5 End Project Milestones.....	118

List of Figures

2.1 Hardware Overview Diagram.....	10
2.2 Software Flowchart.....	11
2.3 S.P.U.D. Prototype.....	12
3.1 64x64 RGB LED matrix display.....	24
3.2 Arty S7 Spartan-7 FPGA Development Board.....	31
3.3 TXS0108E 8-channel logic level converter module.....	36
7.1 High-level Pipeline Structure.....	93
7.2 RISC-V Instruction Cycle.....	94
7.3 RISC-V Instruction Set.....	98
8.1 S.P.U.D. Peripherals PCB	103
8.2 S.P.U.D. Peripherals PCB.....	104
8.3 TXS0108EPWR Layout.....	105
8.4 S.P.U.D. Controller PCB.....	105
9.1 Simulation Waveform.....	10

Chapter 1: Executive Summary

SPUD. An approachable, hands-on demonstration of how computers think.

A Single-core Processing Unit and Display. An educational demonstration of the capabilities of a computer processor. Representing the culmination of our team's effort to bridge theory and practice, SPUD implements a functional CPU onto an FPGA that is able to utilize RISC-V instructions, run basic programs, and interact with a system of components.

The processor uses a fetch-decode-execute cycle and includes core components like the ALU, instruction memory, register file, control logic, and RAM. It is fully pipelined, with bypass and branch support. Usually, CPUs are fabricated using state-of-the-art technology and carefully integrated into a computer chip. However, this process is incredibly expensive and inaccessible to the public. Therefore, SPUD enables an affordable and interactive demonstration of a processor geared towards students and curious individuals. Rather than using a fabricated chip, SPUD programs an FPGA to behave like a processor.

FPGAs, or Field-Programmable Gate Arrays, are integrated circuits that can be configured after manufacturing to perform specific digital functions. They are widely used in education and prototyping because they are relatively low cost and reusable across different projects. They provide hands-on exposure to hardware-level programming and logic design, and their reprogrammable nature allows for quick iteration and experimentation, which is perfect for exploring digital logic behavior.

SPUD also utilizes a 64x64 RGB LED matrix to display its functionality. The matrix is interfaced directly with the CPU and serves to visualize the execution of programs in real time. Depending on the program, the matrix can display static or animated patterns, numerical output, or visual representations of internal states, such as register values or memory contents. More specifically, SPUD will use the matrix to run interactive games like Snake.

Thus, the scope of this project includes: a complete CPU implementation in Verilog that is synthesizable on an FPGA, a working toolchain to write and assemble programs for the CPU, several demonstration programs that run on the hardware, an interface for real-time output on the LED matrix, and finally, a custom PCB that connects all components into a cohesive system.

We are grateful for this incredible opportunity to showcase our work, grow as

engineers, and contribute something both educational and engaging to the world of digital design.

Chapter 2: Project Description

This chapter presents a comprehensive overview of our senior design project SPUD; an ambitious project to deploy a RISC-V CPU on an FPGA and integrate it with a dynamic visual interface. We begin by sharing our motivation and the academic background that prepared us for such a challenge, followed by a detailed review of individual team member contributions, and the unique direction our project takes. We provide our team's goals for this project that cover basic system functionality, as well as advanced and stretch goals like running interactive demos. This chapter defines what we aim to accomplish, how we plan to get there, and what makes our work meaningful.

2.1 Motivation and Background

Our motivation is simple yet bold: to create a CPU that runs RISC-V instructions and to become the best engineers we can possibly be! As a team of four passionate computer engineering students, we see our senior design project as a culmination of everything we have learned over the past four years. Given our team consists of only computer engineers, we feel empowered to take a dive deep into a project that is distinctly rooted in the principles of our field. We want to push the boundaries of what it means to be computer engineers by building something ambitious, innovative, and undeniably "us." That vision takes the form of designing a CPU from the ground up and utilizing open source technology with RISC-V to power our designed logic on an FPGA. We also want our project to have external functionality by connecting it to an LED matrix display allowing us to use our RISC-V software to display images, data, and run simple games like the game of life, pong, and snake. Also, our project will connect with a PCB to help demonstrate the inner workings of the CPU.

We are motivated by the opportunity to integrate hardware and software in a way that challenges us to apply the skills we've developed through our coursework and personal projects. For example, in Digital Systems, we dove into Verilog and programming FPGAs to calculate k-maps, truth tables, and digital logic. In Computer Organization, we learned about machine code, RISC-like assembly language, instruction types, and CPU schematics. In Computer Architecture, we went over different types of architectures, branch predictions, and cache. From the fundamentals of digital logic design to the intricacies of computer architecture, this project offers us a chance to synthesize everything we've learned while taking on real-world challenges. It's a bold, hands-on endeavor that highlights our passion for computer engineering: innovation, problem-solving, and the drive to create something extraordinary.

What excites us most is the potential to grow as engineers and push beyond our comfort zones. The basic goal of our project is to design SystemVerilog to run on an FPGA and design testbenches to confirm functionality. However, we wanted to find an interesting way to visualize our work. Using the RISC-V GCC compiler to generate machine code for our CPU, we plan to pair our logic with an interactive display to extend its functionality. If time allows, we would love to explore possible architectural improvements—such as security enhancements or out-of-order execution—and perhaps even running DOOM. These challenges fuel our motivation to build a CPU that is not only functional but also reflects our ambition to learn and innovate.

Ultimately, this project is a tribute to our journey as computer engineers. It's a chance to prove to ourselves and others that we are ready to take on complex, meaningful challenges. We aren't just designing a CPU; we are crafting an experience that combines our technical skills, creativity, and teamwork into a project that's as exciting as it is impactful. For us, this is more than a senior design project—it's a rite of passage and the first step toward becoming the engineers we've always aspired to be.

2.2 Review of Prior Work

Our team is composed of four inspired computer engineering students, each bringing unique skills and experiences that directly support the goals of this project. Together, we aim to leverage our diverse backgrounds to create a CPU that runs RISC-V instructions and powers an interactive screen interface.

Frank brings expertise in low level software development and PCB design. They have hands-on, industry experience with real time systems software development and multithreading design. Completing many projects like self balancing robots, embedded door locks, and flight avionics software he knows the ins and outs of computer functionality and implementation. Their ability to integrate low level software and solve tough problems will ensure thorough high level system logic and smooth integration of RISC-V implementation.

Valentina has extensive experience in ASIC design and verification. Having internship, research, and course experience, she has designed and verified certain designs using UVM and direct testbenches in SystemVerilog. Their familiarity with the language and their ability to debug hardware designs make them a valuable asset for implementing and testing the FPGA-based CPU.

Gabriel specializes in PCB design, embedded systems, and has a strong foundation in analyzing processor functionality. Their prior work involves designing a custom macropad using Arduino. This demonstrates their ability to

create hardware and implement software to complete specific tasks. These skills will be critical for ensuring our CPU supports the RISC-V instruction set via the GNU toolchain.

Mohamed has a strong background in circuit design, having successfully designed and simulated multiple circuits relating to a multitude of projects. PCB, electronic design, digital design, and verification will play a key role in developing the PCB and interfacing it with the FPGA and game display with his varied skill set.

As a team, we are motivated not only by our individual areas of expertise but also by our shared determination to design a cohesive, innovative project. By combining our knowledge of digital logic, computer architecture, embedded systems programming, and hardware design, we are confident in our ability to tackle the challenges of this ambitious project and create a meaningful contribution to the field of computer engineering.

Regarding similar projects, this project was inspired by various creative interpretations of the CPU, including those featured in popular video games that utilize built-in logic gates (Appendix A). We realized that with a sufficient supply of logic units, any digital logic circuit can be replicated. Building on these video game implementations, we also explored relevant literature on the process of designing such systems, which will serve as references as we incorporate our own unique twist (Appendix A).

2.3 Overview of Project Goals

Our basic goals are to get an FPGA to run digital logic to design our own CPU to run program instructions. Including verifying our design with direct testbenches (if time allows, we would ideally like to use formal verification methodologies, or UVM-based testbenches instead). While this sounds simple, this could potentially take up the bulk of our project, as it requires us to use all our knowledge from past courses involving digital systems and computer architecture. To extend the functionality of our project we also plan to add an LED matrix panel designed for FGPAs.

Once the CPU architecture is tested and functional we move onto our advanced goals. To take our project to the next level our goal is to start writing software programs for the CPU to crunch on. These programs will start small, but for demonstration day we'd like to make impressive demonstrations to put our work to the test! Since we'll be adding a screen to our project, we can utilize that to show code executions and display visuals.

At this point, the main goals for our project are complete. Moving forward we will reach the stretch goals to improve upon our machine by implementing new effectiveness. We plan to make improvements to the CPU architecture features that would benefit our project such as the out-of-order execution or security improvements. In terms of CPU power, we'd like to run DOOM and display it on the LED matrix with minimal issue.

Table 2.1 Project Goals

Goal Type	Description
Basic	Design and run simulations of CPU logic in modular architecture. Test through direct testbenches. Create example programs to confirm logic. Get the FPGA to run our CPU with precoded instructions. Design PCB to interface with FPGA for an LED matrix.
Advanced	Utilize the RISC-V-gcc compiler to create machine code to run on CPU, use UVM-based testbench to verify designs
Stretch	Make improvements to the RISC-V architecture. Improve CPU capabilities and implement DOOM onto the LED matrix display.

2.4 Overview of Objectives

For this project we have a set of objectives relating to multiple levels of design for the CPU. To break up the tough challenge we will delegate logical blocks into separate projects completed by members allowing us to work in parallel. The start of our design will be leveraged based on existing FPGA design. This will give us guidance to move forward with our design. We will focus on functionality and loading software then approach our own custom design as needed. After designing the CPU we will carefully calculate the inputs and outputs of each module so we can easily work on each section separately. This will also make running test cases on our design much easier. Our plan is to work from the high level design and adhere to the RV32IM instruction types to work with simple

lines of machine code. Understanding the architecture at a high level and making a good plan will increase workflow and focus our team.

We will start our journey with simulation of the digital logic to give us better insights into the behavior of our CPU and turn over rapid SystemVerilog code. After successfully confirming the logic and implementing it into the FPGA, we will move onto our advanced goal that will really make this project our own. Utilizing the RISC-V GNU toolchain, we can bring the project to the next level by compiling C code into machine code to run on the CPU.

We also plan to make our computer interactive to show our programs functionality. Creating a custom PCB for logic LEDs and User inputs and adding a 64x64 LED matrix connected to the FPGAs IO will allow us to extend the functionality of our CPU. We plan to map the physical bits in our 64 x 64 bit register architecture to the LED grid. Allowing us to physically see what's inside the CPU as well as any general purpose debugging interface. As a bonus this display can serve as a simple screen to run visual demonstrations to run complex programs or demos.

2.5 Project Features and Functionalities

Once our CPU has been completed, this opens lots of functionality and possibilities. A completed CPU can be fed instruction to run any program we desire. Developing in assembly, we can load data, store data, load rediters, use the ALU on registers and immediate values, branch on condition, jump, halt, and more. But using assembly is slow to develop with. To make development more rapid we can use a program to generate assembly/instructions for us. This is exactly what the compiler will do for us. Utilizing the RISC-V GNU tool chain we can use C code to further abstract resulting in accelerated development. This will take our CPUs functionality to the next level.

Crunching numbers is fun, but not always the most exciting way to show off what a CPU can do! To extend the functionality of our system and better showcase its capabilities, we're integrating a 64x64 LED matrix display. While low in resolution, it gives us a creative outlet to visualize all the computation occurring. Anything is possible; animations, games, or even interactive demos. This not only makes development more engaging, but also turns our project into something that's visually appealing and easier to connect with. Most importantly, this opens the door for our CPU to be used as an educational tool. Our goal for SPUD is for it to be something that can help others in the community better understand how CPUs work through clear and hands-on interaction.

We plan on running simple programs, simulations, and even games! Some examples include a perlin noise generator, a simulation of the game of life, Conway's game of life, and we can even use the buttons on the controller to play games. Thus, there are a plethora of possibilities in which we can showcase the capabilities of a processor.

2.6 Requirement Specifications

Below are all of the specifications that will guide the direction of our project. Laying out these details early on helps us stay grounded as we move through design and development. They don't just outline what the system should do, they also help us keep our goals realistic, focused, and aligned as a team.

By setting clear expectations now, we're giving ourselves a solid foundation to refer back to when things get complicated and decisions need to be made. These specs are essentially our blueprint because they keep us honest and on track. More importantly, getting into the habit of defining and following specifications is a core part of industry-standard engineering practice. It guarantees consistency between teams, reduces rework, and supports efficient collaboration. These are all skills that will be vital in our future careers.

Table 2.2 Specifications

Specifications	
Processor Performance	> 10 MIPS
Arcade Button Latency	< 30 ns
LED Matrix Refresh Rate	< 200 Hz
Functions	arithmetic, jump, and memory
FPGA Board Power Consumption	< 36 W (12V, 3A supply)

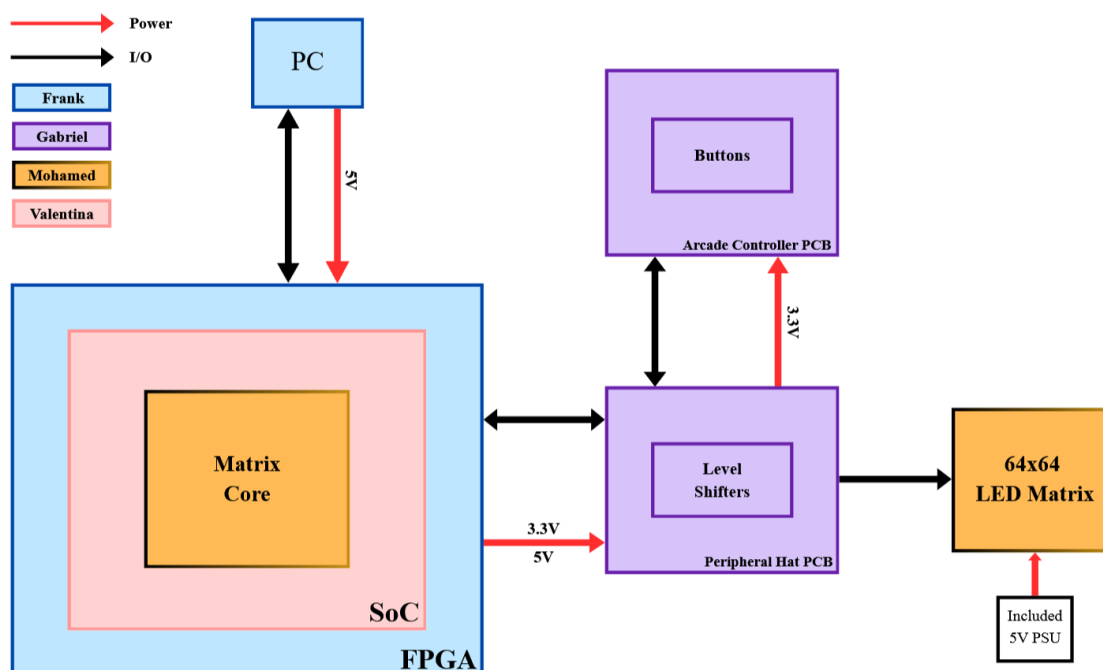
LED Matrix Power Consumption	< 20 W (5V, 4A supply)
Operational Conditions	Room Temperature: 68–77°F Humidity Level: 40-60%
Price Range	\$300-\$500

Based on Table 2.1, SPUD will be operating at a CPU frequency of 25 MHz. It will be delivering a processor performance of 10 MIPS or higher while supporting essential functions like arithmetic, jump, and memory. The system will interface with a 64×64 LED matrix, which is set to refresh at a rate of less than 200 Hz (fast enough to enable visual feedback for animations, games, and other real-time interactions). The power requirements are 36 W for the FPGA board (12V, 3A) and 20 W for the LED matrix (5V, 4A). Additionally, SPUD is being developed to function under normal indoor conditions, more specifically within a temperature range of 68–77°F and with a relative humidity percentage between 40% and 60%. Regarding verification, we are targeting at least 90% functional coverage to ensure the CPU behaves as expected across a wide variety of instruction types and edge cases. In terms of cost, the full setup is estimated to fall between \$300 and \$500.

2.7 Hardware Block Diagram

For our project, we determined a work distribution that aligns with each team member's strengths while also supporting the system's overall functionality. The diagram below (Figure 2.1) shows a simplified view of SPUD's architecture. We have highlighted the main components and the responsibilities assigned to each member. Each member also is assigned a unique color. For example, Gabriel is symbolized by the purple color, Valentina is symbolized by a light pink color, Frank is blue and Mohamed is yellow. This visual breakdown not only helps us clarify who is responsible for what, but also reinforces collaboration by showing how each part of our system connects. We have organized tasks based on our individual expertise (whether that's digital design, hardware integration, or software tooling).

Figure 2.1 Hardware Overview Diagram



The PSU provides power to the LED matrix, while the FPGA is powered by an external 12V DC that supplies the power to the custom PCB using the FPGA supply rails. These rails will also handle data communication between all components. Since the FPGA serves as the core of the system and manages most of the workload, all members will focus on refining their respective contributions to ensure that the FPGA functions as needed. Each member plays a crucial role in the success of SPUD as it cannot function properly without each piece coming together.

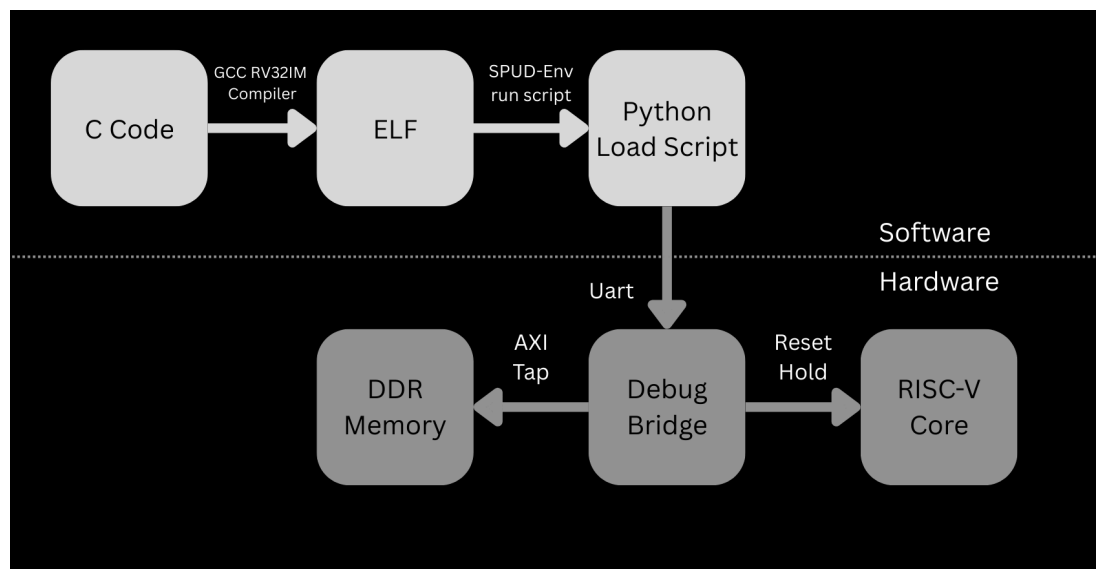
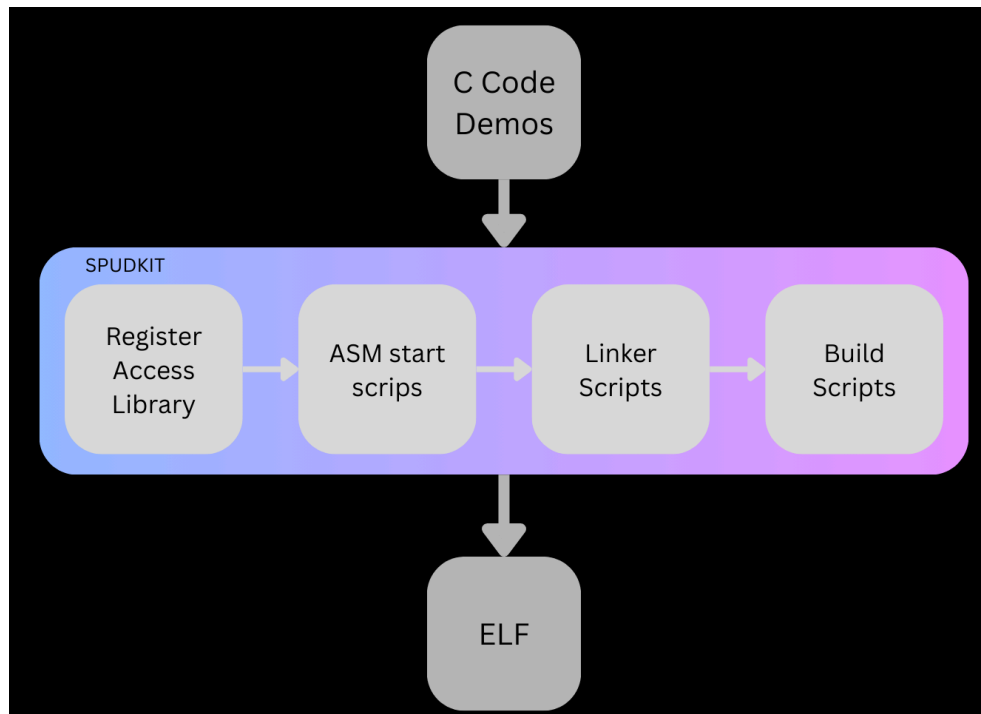
2.8 Software Flowchart

Spudkit is a bundled collection of tools and software that support compiling and running programs on SPUD. It includes a custom C library that simplifies hardware interaction, giving developers direct and convenient access to SPUD's components. Linker scripts define memory-mapped details such as the entry point and stack pointer, and the RISC-V cross-compiler toolchain builds the final ELF files. Each demo is written in C and compiled with the GCC RV32IM toolchain, producing code that conforms to SPUD's architecture.

After compilation, the linker scripts generate an ELF file tailored for the processor, and a custom Python script places the CPU into debug mode. The

debug bridge holds the processor in reset, takes control of the UART interface, and enables loading machine code directly into DDR3 memory. Through this Serial/UART connection, the program is transferred into RAM and prepared for execution on the SPUD hardware.

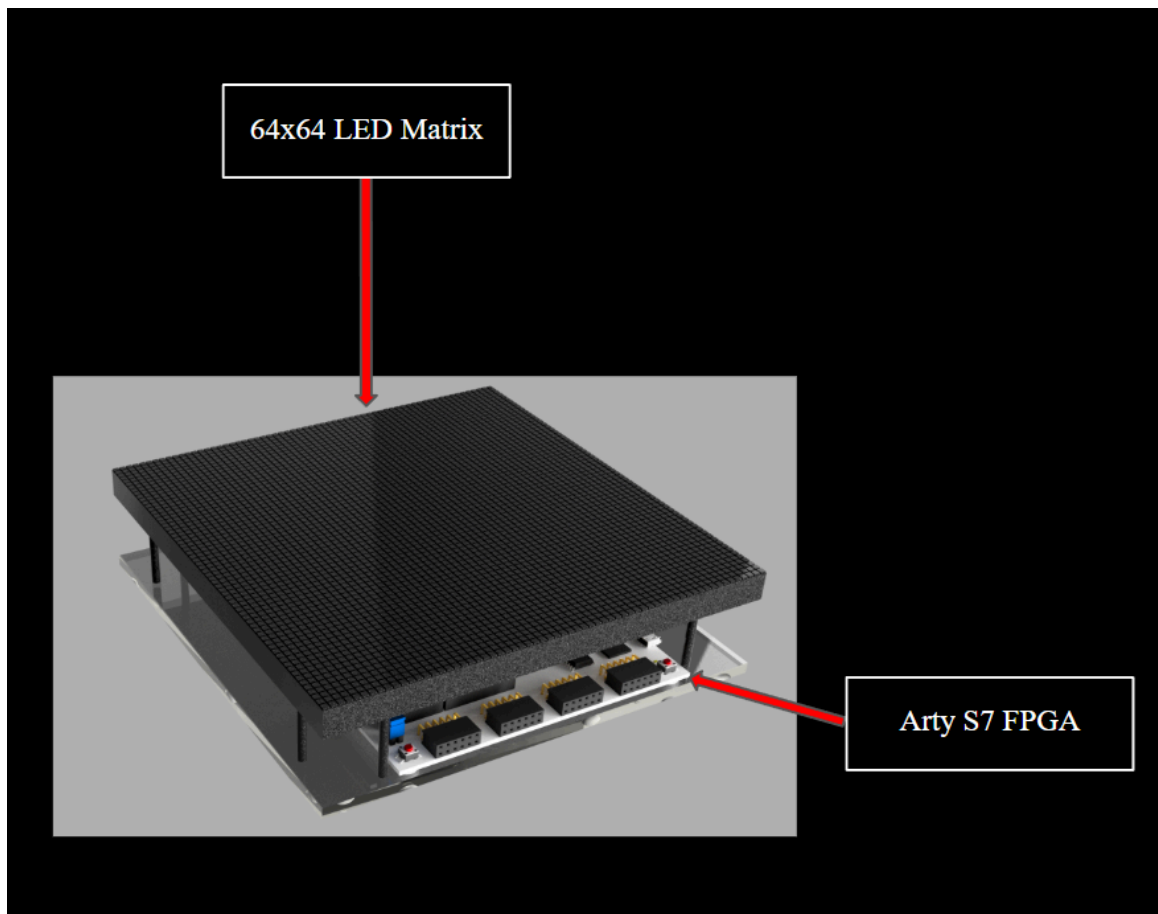
Figure 2.2 Software Flowchart

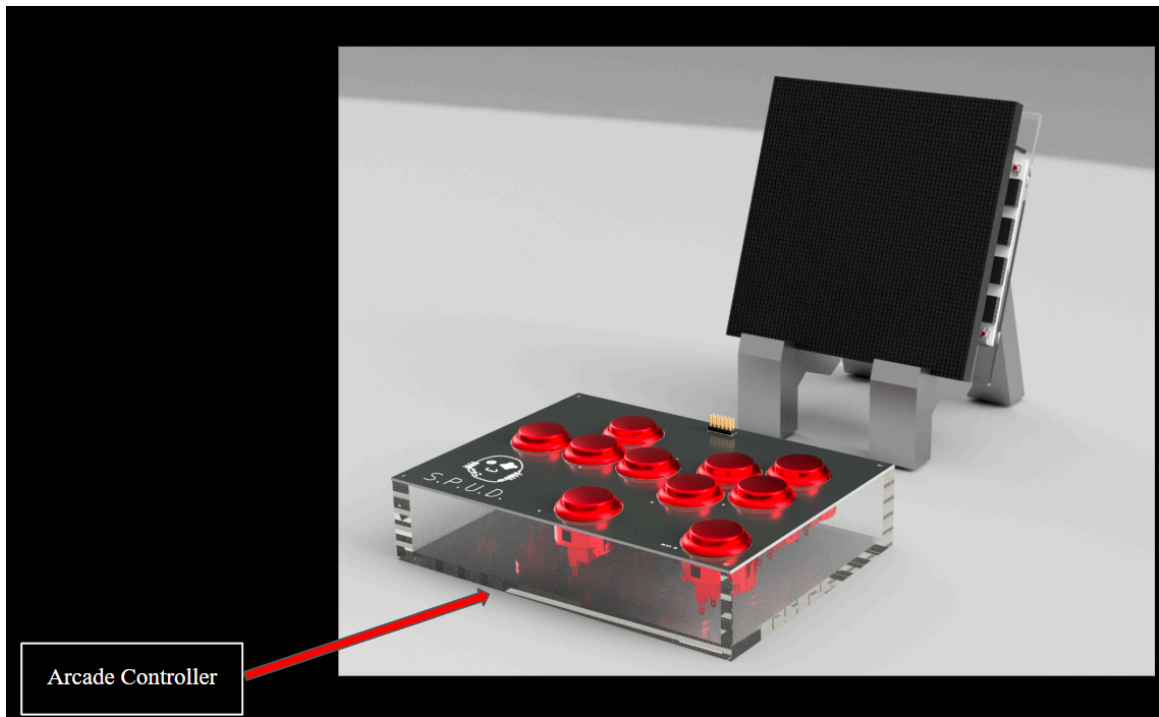


2.9 Prototype Illustration/Blueprint

Our prototype will include the following major components: an FPGA, a printed circuit board (PCB), and an LED matrix display. It will also include additional peripherals such as buttons to be used as a control interface. Our assembly will consist of two parts. One is the FPGA with the main display and I/O expansion PCB. The FPGA will sit on a backboard that will serve as the base. The matrix will go directly over the FPGA and screw into the base with standoffs, leaving some distance between them. This will allow flexibility for the ribbon cable connecting the FPGA and the matrix. The second assembly will serve as the interface for the interactive device. We will be using a 12-wire connector to pair the two together, which will allow us to control ten buttons. Although the number of wires is excessive and we could have reduced them by using a protocol like UART, since the button box won't be moved often, this will be acceptable for our purposes. The button box will use the connector on the outward-facing I/O connector.

Figure 2.3 S.P.U.D. Prototype created by authors.





Chapter 3: Research

Every design choice, from architectural layout to interface protocols, depends on a strong understanding of existing tools and standards. Because we are working with widely used components and standards, such as the RISC-V instruction set and FPGAs, there is a vast amount of documentation and community resources available to support our decisions. In this section, our team explores these resources and uses their information to select the appropriate technology and parts for SPUD. We break down this chapter into a hardware section (3.1) and a software section (3.2).

3.1 Hardware Technology Comparison and Selection

The hardware in our project has been carefully researched and selected to fit our needs. The most important piece of hardware for our project is the main FPGA that runs all of our logic/verilog software. All of our peripherals work directly with our FPGA utilizing IO. These peripherals include a 64x64 LED matrix that works directly with a level shifter to protect overvoltage on our signal lines. We are also utilizing a PCB to make our design more robust and expose an interface for the user to interact with the project.

3.1.1 Display Technology

In this section, we will look into different types of displays that were considered for our project SPUD. The display will be used to show the outputs of various applications and it is a crucial component of the project. The Major comparison factors that we based our selection on are as follows: resolution, implementation complexity, power consumption, portability/size, and display type.

Resolution defines an important metric for our display: what is the resolution of the image to be displayed on our screen. Resolution (in the context of display) is simply the number of pixels available in a display to construct an image, which can simply be calculated by the number of pixels in the width multiplied by the number of pixels in the height of the display, it is primarily the metric that directly defines the displayed image's "quality", which is how clear and less-segmented the display image looks; for our project, the quality requirement is light due to how our planned applications will not have too complex of images to be displayed, as adding more pixels to a program's interface display might cause it to be more complex to implement and heavier to run, Therefore, we focused on choosing a display that had a good enough resolution without making it the main focus of search metrics.

Implementation complexity is how difficult would it be to implement the desired drivers and execute the display related codes using the selected display device, for example, display A is straightforward, only requiring us to use a premade driver that is compatible with our main board selection, while the code is already written by the manufacturer, only waiting to be copied and pasted into the board's program memory by the end user, as for board B, we would not only need to implement display driver on our own, but we would also have to write the algorithms for the display from scratch. When considering implementation complexity, we have to consider the fact that a more difficult to use display will not only add to our labor hours in the project, but it will increase the likelihood of an error in our implementation due to the complexity of the design process.

Power consumption is how much power the display requires to run without issues. Although the issue of Power consumption might appear to be a simple, one-dimensional issue of just finding whichever supply/cable is strong enough to feed our display, in reality, it is a multifaceted problem. Power consumption affects three important factors relating to the power supply and the display itself: the physical size of the power supply/power circuitry, the heat emitted by both the display and the power supply, and how much power will be consumed and what accommodation is required to supply said power. Firstly, the size of the power supply will affect the portability of the circuit, its weight, and how easy will it be to "compress" everything into a neat package; we would like to avoid using a display that requires hauling around a brick-like power supply block or at least have the flexibility to separate it from the main package. Secondly, the more power the display consumes, the more heat it will produce; we can choose a display that might align with all of our other requirements, but if its cooling solution is too large/ too expensive to implement in our package, it will simply be unusable. Finally, how much power will be supplied will also decide the size of the power supply and how easy it is to include in our package.

Portability/Size are the metrics that describe how easy it is to move around the display (or to just include it in our main package) and how large said display is. If the display size is too big, it might cause issues with implementation in our package and might just make it awkward overall. Portability would also directly be correlated with size and is also considered as hauling around a PC monitor might be undesirable and will be avoided if possible.

Display type is the type of technology that will be used to implement our display functionality. The most common display types for our scale of projects would be LED matrices, LCD (TFT LCDs included), OLED displays, E-Paper Displays, and segmented displays.

LED Matrices are simply a grouping of LEDs in a grid that allows the user to individually control each “pixel” (i.e. one of the LEDs) to display a specific image at specific color profiles. The color scheme for the LEDs is usually displayed by using a built in RGB LED package in each “pixel” , typically, we display a specific color by using pulse width modulation to adjust each one of the RGB LEDs brightness to achieve a wide variety of colors, some matrices use addressable RGB LEDs instead and let a microcontroller control the color mixing.

When it comes to resolution, LED matrices fall behind as they usually have a resolution in the order of magnitude of 10s(36 x 36, 64x64), this might be debilitating for some applications but for most embedded/ small scale projects, this is good enough for displaying certain images that don’t require high resolution (like applications with small text or minute details). Despite the lower resolution, LED matrices boast very bright lights, making it easier to display in brighter locations.

As for the complexity of the implementation, LED matrices are fairly complex, as the LED packages require multiplexing, the coloring requires PWM to control the color of each pixel, and it will require a drive to be implemented in order to control the aforementioned functionalities without much issue, however, the limited number of pixel does put a cap on how complex the implementation gets as the smaller numbers generally mean less things to account for, and the implementation process of the LED matrix when compared to other display types is more friendly to lower level designs, making it a better fit for lower level projects that utilize FPGAs.

The power consumption for LED Matrices is on the higher end due to very bright lights, higher power consumption due to how inefficient LED technologies are, and due to how inefficient the display update methods are (multiplexing) compared to other display types. The previously listed inefficiencies in the technology and high power draw from updating the display prove to be too costly of a combination, making it borderline impossible to power the board without an external power adapter, this means that alongside the screen (which is larger than other display options) we will need to haul around a large power adapter that probably won’t be packaged neatly with other components of the circuit, heavily reducing the portability of the overall project. An example of the power consumption would be a 64x64 LED matrix panel, the required current for a single panel is around 4A and the voltage needed is 5V, making the power requirement around 20W which, realistically, would be very difficult to include in the PCB’s power supply.

Overall, although the LED matrix might not be the best in all categories (and the worst in some), it was our winning selection for many reasons, the complexity, although up there, was something that was fairly within reach for us and was better than the competing options, the power consumption was manageable despite how inconvenient it would be to have to deal with the large power supply block, the resolution fit our application perfectly: not too high in quality as to add unnecessary complexity and computational power to our project, and not too low quality as to be unusable, the colors available in the display were vibrant and competed with other options with less color variety, and the retro aesthetic that is given off by the low resolution larger display fit our vision excellently. The high brightness given off by the display was also considered as it would help with demonstrating our project regardless of the location. We also were interested in LED matrices as their implementation process

LCD stands for “Liquid Crystal Display”, it is a display technology that utilizes a backlight that lights behind a layer of shifting liquid crystals, voltage passes through the crystal and depending on the strength of the signal, it partially or fully blocks the backlight, allowing for a control of the brightness of the light. The colors are displayed through the use of three RGB sub-pixels in each pixel that depending on the voltage to each one of them, the intensity of each color is modified and a color is derived from the combination. There are multiple types of LCD technologies including monochrome LCD, character LCDs, and TFT LCDs, we will mainly focus on TFT and character LCDs.

The resolution for LCDs is a bit higher usually, ranging from a basic 2*16 characters up to 128 *64 characters for character LCDs and ranges including 320 * 480, 240 * 240, 240 * 320 (i.e. ranges in the magnitudes of hundreds of pixels), what makes character LCDs less desirable (despite the lower resolution count which might be easier to program) is the fact that it is restricted to a discreet, character based display, which heavily limits the possible applications; However, for the TFT LCD, it is a full fledged display system that can realistically display most possible images with its only limit being the resolution which can be increased if needed.

The Implementation complexity varies greatly between both versions of the LCD, with character LCDs being one of the simplest to implement as it is only capable of displaying a set number of character alongside a set grid, the steps to doing so is simply driving pins high or low using SPI or I2C (or a parallel interface) and updating the text as needed, the trade off for such simplicity is that what is displayed can only be purely in character form, no images or graphics are possible. A TFT LCD solves the problem of the limited display possibilities of the character LCD by having the possibility of setting every individual pixel instead

of only being limited to discrete positions for a limited set of characters, However, this exponentially increases the complexity which is further compounded by the need to control the color schemes of each individual pixel. The pixel data is usually stored and updated in a type of memory called framebuffer memory, which adds to the complexity of the implementation and its required resources, this technology requires more processing power than the character LCD, causing potential bottlenecks and performance issues. In higher level applications (a microcontroller), many of these implementation complexities are eased by pre-existing libraries, however, in the case of an FPGA, custom HDL modules are needed to implement most if not all of the functionalities, this however is with advantage of the high speed of the FPGA.

The power consumption for both types of LCD is on the lower side, as both technologies can realistically be implemented onto a breadboard without the need of an external power supply block, the character LCD is a low-power display as it can be powered on with a 5V power supply and typically requires just 1mA for the LCD and 20mA for the backlight, which is around 105mW, easily implementable on a breadboard. The TFT LCD requires more power draw, when using a 5V supply for example, it would require around 100mA-120mA total current draw on one of the smaller display sizes (1.8"), equalling 500mW-700mW, which increases substantially with the screen size; However, this is still easy to power on using a PCB friendly power supply. In either case, both types of tech should be easily portable as their inclusion in the project's package should not add much space beyond the screens themselves.

In conclusion, although character LCDs are easy to implement and have low power requirements, their overly limited display is the biggest limiting factor for its usability, it would be ideal for displaying readings from a sensor or output data from calculations, but attempting to display anything beyond character based representation of information is impossible and therefore it would not fit our application which will require a proper display to present our applications. The TFT display was a viable contender as its power requirements were fair and its resolution, although on the higher end, was somewhat acceptable for our application; However, we opted to not use them due to implementation complexity of its platform and how we preferred a lower resolution display with a larger screen size for our application.

OLED stands for organic light emitting diode, a type of display that differs in certain aspects from a typical LED, mainly, its semiconductors use organic materials instead of inorganic materials. A typical OLED is composed from a substrate (a plastic or glass component that is there for support), an anode, a cathode, and organic layers. The organic layers in an OLED are the main light

source, whenever they get excited, they emit light; this process does not require a backlight. One method OLEDs control their color is in a similar manner to normal LEDs, where they have red, green, and blue sub pixels in a single package, and based on the input the brightness of each subpixel is adjusted to achieve a desired color, another method for color control is having a fully white OLED display that is covered by a color filter that determines the color (this method has less efficiency as some light is absorbed into the filter), the second method is more common alongside embedded applications. Another thing worth noting for OLEDs is that they achieve better contrast than most displays as they do not leak light in darker colors.

When it comes to available resolutions, OLEDs boast a high resolution for their size, with displays as small as 1" having resolutions around 128*32, and bigger displays around 3" having resolutions around 480*272, making it high in resolution in comparison to more primitive display types like character LCDs and RGB matrices, and comparable to LCDs, compared to LCDs however, it excels in how it presents color, having better contrast, more vibrant colors, and true blacks with minimal light leakage.

The implementation complexity for an OLED is on the higher end, similar to non-character LCDs in requiring a frame buffer and in using similar communication protocols (I2C and SPI), a driver with proper initialization and implementation within the system is needed in order for the display to function properly. This usually is eased with high-level libraries that present important functionalities with minimal work, however, in our case, we will need to write custom modules to implement these functionalities.

For the power consumption, OLEDs boast one of the best performances in the category, as not only do they not require any back lights, greatly reducing passive power consumption, but OLED's blacks are created by turning off the LEDs, meaning that darker scene reduce power consumption, causing the average power consumption to experience an overall drop. Organic LEDs power consumption is also lower than older/more common variants of LEDs in general as the technology itself is designed with better power efficiency (at the cost of brightness), finally, OLEDs are driven with matrices rather than multiplexing techniques, further reducing the power draw from switching and updating the display compared to more primitive display update methods.

Due to the low power consumption OLED displays can easily be fit into a pcb with minimal added overhead for the power supply, another factor easing the addition of OLED displays onto the main board is how flexible the display is, making it easier to add to the board while opening up a wider array of possible

positions of placements, making OLEDs one of the easier displays to add to the board.

In conclusion, OLED displays are one of the best options available at our disposal for a display. Not only does it excel in image clarity with vibrant colors, great contrast, and minimal light leakage in darker scenes, but it also boasts very efficient power consumption. It also provides ease of implementation physically as it is flexible and can be positioned in more ways than other more rigid displays, however, its software complexity might prove to be too difficult for our project, therefore, it was discounted for that.

3.1.1.1 64x64 RGB LED Matrix Panel - 2mm Pitch from adafruit Product ID - 5362:

The specs of this specific module is as follows: a screen size of 128mm x 128 mm with a resolution of 64 x 64 and a pitch (distance between each individual LED) of 2M, and the brightness is 1,000cd/m². The possible refresh rates range from 1.9kHz and 3.8kHz. The recommended power supply is a 4As of current through a 5 volt power supply, meaning that the display can use up to 20 W of power. The display comes with 5 addressable pins and requires around 1.6kBytes of ram to buffer the 12 bit color image, the processing needed for this can be easily satisfied with an FPGA.

We decided to use a 64x64 RGB LED for multiple factors: it has large enough resolution: small enough to be easily controlled, and large enough to display all our lower level applications without losing much quality, the colors are good enough for our application and the display is very bright to withstand any external lighting while demoing the functionality of the processor, finally, the screen is large despite the smaller resolution which fits perfectly with lower resolution applications without losing any quality and improving visibility.

The processing power needed for this display is also easily fulfilled by our choice of FPGA, allowing us to drive the display with no loss in performance for either the display or the FPGA itself. The larger than normal screen size might pose some difficulties when it comes to situating the screen within the board (or deciding to have it separate from the board) and the large power adapter that is needed to power on the screen might cause some issues with the physical design, but the overall positives outweigh these negatives.

3.1.1.2 5.0" 40-pin 800x480 TFT Display without Touchscreen from adafruit Product ID - 1680 :

The specs of this display are as follows: a screen size of 5.0" with a resolution of 800x480 and a brightness of around 200-400 cd/m². The refresh rate for the display is 60 Hz. The power supply recommended for this display is a constant-current mode boost converter with a supply max of up to 24V, meaning there might be more physical space needed to power the display. The display comes with a 40 pin connector with 8 pins dedicated to each primary color (red, green, and blue), and although the processing requirements are not explicitly stated, a companion driver board with ~700kB of RAM is recommended to drive this display.

Although this display does fit our size requirement, the power supply needed to power it is not that much different from the LED matrix's, meaning that it suffers from the same physical space constraint; another issue with this display is the large number of processing power needed to run it, as it takes almost a third of our available RAM, and finally, the amount of pixels in this display makes it much more complex to implement software wise compared to the smaller 64x64 resolution from our LED matrix, and downsizing to a smaller display to have a small enough resolution to not substantially increase the complexity of the software is also not an option, as the screen size would be too small then. All these discussed factors conclude that a TFT LCD would not fit our previously discussed criterias.

3.1.1.3 1.91 inch Color OLED Module with 176x176 Resolution from New Haven Display International Model - NHD-1.91-176176UBC3 :

The specs for this display are as follows: a screen size of 1.91" with a resolution of 176x176 and a brightness of up to 80 cd/m². The refresh rate of this module goes up to 120 Hz. The needed voltage for this module is 3.3V and it can be powered on using the board's power supply without extra requirements for physical space, however, depending on the use, it might need a level shifter. The display offers the option of either 8-bit parallel communication or 4 bits SPI/I2C, making it easier to implement than the LCD.

Although the OLED display does offer a lot of upsides, its small size, which is difficult to scale due to added complexity due to the increase in the number of display pixels, and the larger OLED displays usually being monochromatic, is a large enough downside to disregard this display as a viable display option. Additionally, the limited viewing area makes it less effective for audience-facing demos or interactive visualizations, which are a key part of our project. While its

lower power requirements and higher refresh rate are attractive, they don't outweigh the need for a display that's both visible and impactful in real-time demonstrations. Ultimately, the trade-offs in visibility, resolution scalability, and usability led us to pursue the LED matrix instead. The OLED display may still be suitable for compact, internal status indicators or secondary output in other designs, but for SPUD's goals of accessibility and educational outreach, a larger and more immersive display is essential. We want users and observers alike to engage with what's happening on screen—and the OLED simply didn't meet that need.

3.1.1.4 Technology Comparison:

In our project, we selected an LED matrix display, but it is important to understand how it compares to LCD and OLED technologies. LCDs offer high resolution and brightness with a backlight but have limited contrast and slower response times. OLEDs provide superior contrast with true blacks, wide viewing angles, and extremely fast response times but can have lower peak brightness. LED matrix displays, while limited in resolution due to their grid design, provide sufficient brightness, moderate contrast, and adequate response times for our application. They also offer a straightforward driving method suitable for FPGA control. While OLED and LCD technologies have their advantages, the LED matrix aligns best with our project's needs, balancing performance, power efficiency, and ease of integration.

Table 3.1 *Technology Comparison and Selection.*

Feature	LCD (Liquid Crystal Display)	OLED (Organic Light-Emitting Diode)	Matrix Displays
Resolution Potential	High, dependent on pixel size and TFT technology	High, with fine control over individual pixels	Limited by matrix grid design
Brightness	Moderate to high (200-400 cd/m ² with LED backlight)	Lower peak brightness compared to LCD; true blacks enhance perceived contrast	Limited brightness (~1/10 of segmented displays)

Contrast Ratio	Moderate (limited by backlight)	Infinite (self-emissive pixels can turn off completely for true blacks)	Moderate
Viewing Angles	Narrower than OLED (varies by LCD type: TN, IPS, VA)	Wide 180 degrees	Moderate
Response Time	Slower (~6-16 ms for LCDs; faster for TN panels)	Extremely fast (<0.01 ms), ideal for motion clarity	Moderate
Power Efficiency	Requires backlight, consumes more power	Highly efficient; only active pixels consume power	Dependent on the driving method

3.1.1.5 Part Selection:

When deciding between LED matrices, we removed options like the 16x16 custom LED matrix, because the resolution is too small for our project and would significantly limit the types of programs we could showcase. When choosing between the Adafruit 64x64 LED matrix and the Waveshare equivalent, although they are similar in many aspects, the Adafruit 64x64 LED matrix performed better with higher intensity applications and handled power and refresh demands more gracefully. It was also more accessible, with better documentation, tutorials, and community support, which played a key role in helping us integrate it smoothly into our workflow. These advantages made it the most suitable choice for SPUD. Additionally, the Adafruit matrix's compatibility with our existing development tools reduced setup time and potential integration issues. Its reliability during extended testing sessions gave us confidence in its long-term performance, especially for interactive demos that require stable and consistent display output. Although it was slightly more expensive than other options, the added value in support, quality, and ease of use justified the cost.

Table 3.2 Modules and Selection.

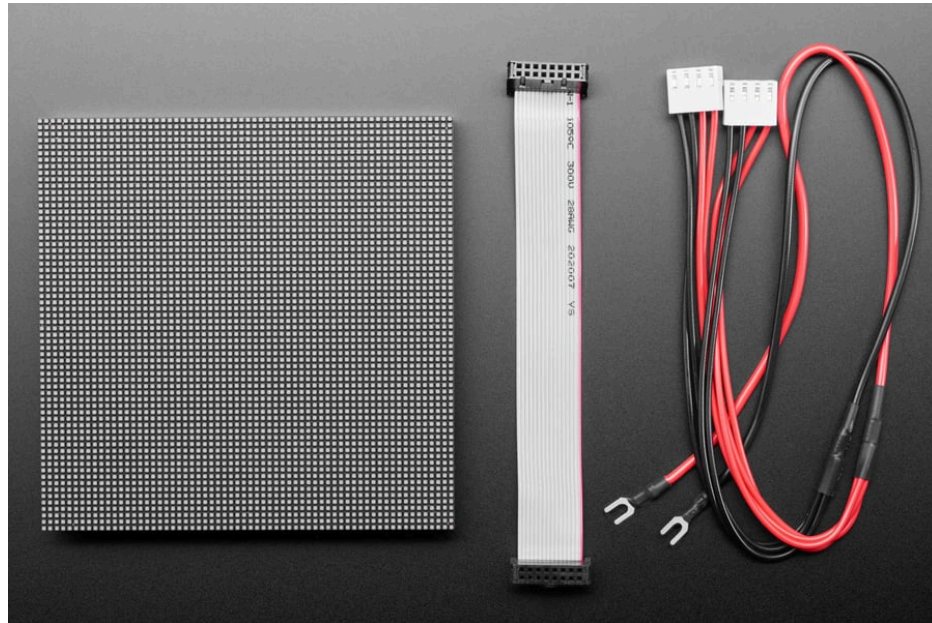
Feature	Custom LED Matrix (Merkury Innovations)	Adafruit 64x64 RGB LED Matrix	Waveshare RGB Matrix P2-64x64
Resolution	16x16	64x64	64x64
Brightness	Moderate (suitable for indoor use)	High (1,000 cd/m ²)	High (1,000 cd/m ²)
Physical Dimensions	~6 inches	7.13 inches	7.13 inches
Interface	Bluetooth app control	HUB75 interface	HUB75 interface
Power Requirement	USB-powered	4A 5V power supply	5V power supply
Best For	Beginners or simple decorative projects	Advanced projects requiring high resolution and brightness	Flexible and affordable projects needing high resolution

Considering everything listed in the table above, we eventually selected the **Adafruit 64x64 RGB LED Matrix** due to its perfect resolution; not too high to be computationally complex, and not too low to limit the clarity of our displays. The brightness also makes it easier for us to demonstrate regardless of the setting, ensuring our visuals are clearly visible even in well-lit environments. The screen size is big enough to enhance the visual presence of our project during our demos, while still being manageable in terms of integration.

The image shows the full kit contents for the 64x64 RGB LED matrix display. Included in the kit are the 64x64 RGB panel itself, a 16-pin IDC ribbon cable used for data transmission, and a power cable designed to connect the panel to

a 5V power supply. These components are essential for setting up and running the LED matrix with our FPGA system.

Figure 3.1 64x64 RGB LED matrix display. Retrieved from Adafruit (n.d.).



3.1.2 FPGA Platform

In this section, we will explore multiple FPGA platforms suitable for our SPUD project, which involves implementing a CPU on an FPGA. We will compare the Arty S7: Spartan-7 FPGA Development Board (which is the board we selected) with two other popular FPGA families: the Xilinx Artix-7 and Intel Cyclone V. The key factors for comparison are categorized into three main areas: Hardware & Performance, Connectivity, and Cost & Applications.

The **Hardware & Performance** category includes logic cells, flip-flops, DSP blocks, Block RAM, clock frequency, and power consumption. Logic cells serve as the building blocks of digital circuits within the FPGA and determine its capability to implement complex designs. A higher number of logic cells allows for greater design flexibility and scalability, which is crucial for implementing our custom RISC-V CPU architecture. Since our CPU needs to execute a variety of instructions and interact with external devices, we will need a sufficient number of logic cells.

Flip-flops are essential for sequential logic operations because they store binary state information that updates on clock edges. This makes them vital for

registers, pipelining, and finite state machines, all of which are required to properly execute CPU instructions. In regards to our specific CPU, implementing pipelining is a stretch goal and is not guaranteed. However, it is useful to know that it can be supported should we choose to incorporate it into our CPU design in future optimizations.

Block RAM serves as fast memory on the chip. This is crucial for storing any temporary data, registers, caches, or data for the LED display. Since our system could potentially rely on an external flash memory for program storage, having a sufficient amount of BRAM allows us to use cache for frequently used instructions. Cache is extremely beneficial as it improves execution speed and reduces memory access time.

Clock frequency directly impacts the instruction throughput of our CPU. It defines how quickly it can execute operations. A higher frequency improves game performance by reducing execution delays. In the case that we adjust our design to be pipelined, however, it would also introduce timing constraints that must be met to avoid errors. Additionally, our design must be synched with the refresh rate of the LED matrix to ensure that the frame updates occur without any sort of flickering or issues.

Lastly, power consumption is an important criteria for selecting an appropriate FPGA because it covers both thermal management and overall system efficiency—two important factors that allow for our FPGA to properly fit in with the rest of SPUD's devices. FPGAs with high power consumption will generate more heat, which would require additional cooling mechanisms and affect the safety of our project. Since SPUD involves continuous execution while a game is running, we want to minimize the risk of overheating as much as possible (especially because our design will be compact). By selecting an FPGA with lower power requirements, we can reduce those risks and keep our design energy efficient.

The Connectivity category includes communication protocols and memory support, which are directly associated with how the FPGA interacts with external devices in our SPUD system. If the FPGA has high-speed I/O capabilities, then it can improve data transfer rates, which could help improve communication between all the devices.

Communication protocols such as SPI, I2C, and UART play a major role in system integration. For SPUD, SPI is specifically important because it is a common protocol for interfacing with external flash memory. Having reliable SPI

will make sure that we efficiently access memory and reduce any potential bottlenecks when we fetch game instructions or graphing data. An FPGA that supports high-speed serial interfaces will provide greater flexibility in connecting with any additional components.

Regarding memory support, the FPGA must have the ability to interface with external RAM or flash memory. Since our game program is stored externally, we must ensure that the selected FPGA can communicate with the flash memory without increased delay. The ability to integrate with high-bandwidth memory solutions could improve the performance of our CPU by allowing faster retrieval of frequently accessed data. If our design later incorporates more complex game logic, having strong memory support will be incredibly useful.

The Cost & Applications category in Arty S7-50 FPGA includes board price, design software price, and typical board applications. These help determine the feasibility and practicality of each FPGA for SPUD. Board price is a key factor because of limited funding. Higher-end FPGAs usually come at a higher cost because they offer more resources. Thus, the board we chose (the Arty S7 Spartan-7), is both affordable and capable, making it the reasonable choice for SPUD. We continue to compare it to alternatives like the Artix-7 or Intel Cyclone V, because we still need to consider whether their additional features justify a higher cost.

The cost to use design software is another vital criteria for deciding on the ideal FPGA. Board development requires vendor-specific tools. For instance, Xilinx FPGAs (Spartan-7 and Artix-7) use Vivado, which has a free WebPACK edition. Intel FPGAs (Cyclone V) use Quartus Prime, which also has a free Lite edition.

Vivado's WebPACK edition is entirely suitable for our needs. However, higher-end features such as advanced synthesis optimizations or formal verification tools may require paid licenses. In our stretch goals, we mentioned verifying the CPU design using formal verification methodology, so if we want to use a design suite for this we may need to factor in additional cost. Regardless, making sure that the free tool versions support all the features we need (i.e. debugging and hardware simulation) is important for keeping costs at a minimum.

Typical board applications are the final criteria for determining which board is best suited for our needs. Spartan-7 FPGAs are often used for embedded computing, digital signal processing, and hardware prototyping, all of which align well with our CPU implementation and game-processing requirements.

Usually Artix-7 FPGAs are designed for higher-performance applications, such as video processing and high-speed networking, which may not exactly be necessary for our project but could be useful if we wanted to focus more on improving our graphics quality. Finally, the Cyclone V board is commonly used in low-power embedded systems. If power efficiency was our highest priority, this board would be ideal. By evaluating how each FPGA family is typically used, we can properly evaluate which board mostly aligns with SPUD's goals and constraints.

3.1.2.1 Arty S7-50: Spartan-7 FPGA Development Board

After considering all the criteria mentioned above, our team concluded that this would be the best fit for SPUD. With 52,160 logic cells, 65,200 flip-flops, 120 DSP slices, 2.7 Mb BRAM, and a 450 MHz max clock frequency, we felt that this would be sufficient to implement our custom CPU architecture design. It also has communication interfaces like USB-UART, USB-JTAG, Quad-SPI Flash, Pmod ports, and an Arduino/chipKIT shield connector.

The USB-UART and USB-JTAG interfaces are useful for communication between the FPGA and the computer. USB-UART can be used for serial communication, which is ideal for debugging and logging information from the board to the computer or vice versa. USB-JTAG is useful because it can be used for debugging the FPGA directly. We want to have the ability to constantly optimize the CPU design, therefore these interfaces will be just what we need. The Quad-SPI Flash will be vital for running our game programs, and the Pmod ports and Arduino/chipKIT shield connector will help us if we ever need to implement additional devices (such as I/O devices, sensors, or additional memory).

Regarding the connectivity ability of the S7-50, the 256 MB DDR3L with a 16-bit bus @325 MHz (650 MT/s) and 128 Mbits Quad-SPI Flash support can handle the memory-intensive nature of running a game. The DDR3L provides high-bandwidth memory access, which will be useful for storing and quickly accessing data needed by the CPU. While the DDR3L memory is used for high-speed operations, the Quad-SPI Flash can be used for long-term storage of the game files. Given that our design needs to efficiently manage both the game and CPU's data, the combination of these two memory options will properly balance the speed and storage we need.

The software for the S7-50 (Vivado Design Suite WebPACK Edition) provides a free, fully supported development environment for designing, simulating, and

testing FPGA logic. Having full access to this software is one of the prime reasons why we chose this FPGA.

Another prime reason why we went with the S7-50 is because of its affordability. The price of the Arty S7-50 FPGA ranges from \$150-\$200, depending on the website you buy it from. Compared to the other FPGAs we reviewed, this is the least expensive but still provides all the hardware we need to fully implement our CPU.

3.1.2.2 Xilinx Arty A7-100T: Artix-7

The Arty A7-100T FPGA from the Xilinx Artix-7 Family was also a strong candidate for SPUD. It provides a significantly higher capacity (basically double the S7-50), with 101,440 logic cells, 126,800 flip-flops, 240 DSP slices, and 4.86 Mb of BRAM, supporting a maximum clock frequency of 450 MHz. Like the S7-50, the A7-100T is a low-power FPGA and is equipped with the exact same communication protocols: USB-UART, USB-JTAG, Quad-SPI Flash, Pmod ports, and an Arduino/chipKIT shield connector.

As for the memory capabilities, the A7-100T has 256 MB of DDR3L memory with a 16-bit bus operating at 667 MHz (1.33 GT/s), which is much faster than the S7-50's DDR3L memory. This faster memory would significantly improve the performance of our CPU and thus improve gameplay (tempting). Additionally, the A7-100T includes 128 Mbits of Quad-SPI Flash memory (same as the S7-50).

The A7-100T is also compatible with the Vivado Design Suite WebPACK Edition which is another huge advantage. In terms of cost, however, the A7-100T is priced at approximately \$299, making it about double the price of the S7-50. We believe that, while the A7-100 has many valuable resources that would improve our project, the additional cost of it outweighs its benefits. We simply do not need it, especially if the S7-50 can provide everything we need at half the cost. Nonetheless, the A7-100T would most definitely be considered in the future should we choose to expand on the project further.

3.1.2.3 Intel Cyclone V GX Starter Kit

From the Intel Cyclone V Family of FPGAs, our team considered the Intel Cyclone V GX Starter Kit for our project. The Cyclone V is similar to the A7-100 in terms of flexibility and performance with 77,000 logic cells, 56,832 flip-flops, 342 DSP blocks, and 4.88 Mb of BRAM. This FPGA supports a maximum clock

frequency of 800 MHz, which is a much higher clock speed than the previous options, which could significantly boost performance.

The Cyclone V also includes a wide range of communication protocols, such as USB 2.0, UART, I2C, SPI, and LVDS, which gives us plenty of options for connecting to other devices. Additionally, it includes six transceivers supporting data rates from 614 Mbps to 3.125 Gbps and PCIe Gen2 support which would be more useful in the case that we want to expand SPUD in the future.

Regarding memory, the Cyclone V supports DDR3, DDR2, and LPDDR2 memory configurations. It also comes with 1 GB of DDR3 SDRAM which is good for more complex designs compared to the 256 MB in the A7-100T and S7-50 FPGAs. The Cyclone V also includes 64 MB of Quad-SPI Flash memory, similar to the other FPGAs, for storing game data.

Due to the Cyclone V FPGA belonging to Intel, it requires a different design suite. Thus, it is compatible with Intel's Quartus Prime software. The free Lite edition offers limited support for FPGA design, simulation, and debugging. Despite it offering fewer features than the full version, it can still provide us with the tools we need without additional costs. This keeps the board competitive with the other Xilinx boards that utilize Vivado Design Suite.

In terms of cost, the Cyclone V GX Starter Kit is priced at \$270. While it is more expensive than the Arty S7-50, it offers additional resources like higher clock speeds, more I/O pins, and the support for DDR3, DDR2, and LPDDR2 memory types. Similar to the A7-100, the Cyclone V is extremely powerful and would enhance our CPU greatly. However, the additional resources it provides are not desperately needed, and therefore are not worth the cost.

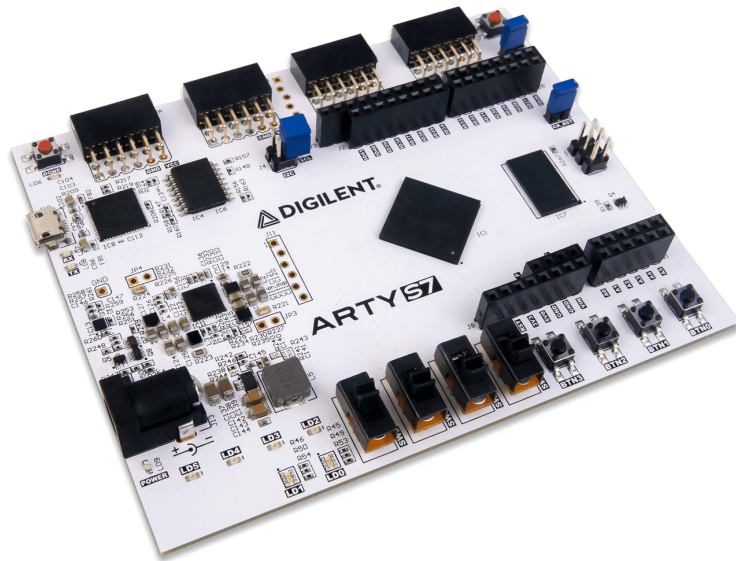
3.1.2.4 Comparison & Selection

The following table compares the key features of the Xilinx Arty S7: Spartan-7 FPGA, Xilinx A7-100T: Artix-7 FPGA, and the Intel Cyclone V GX Starter Kit. This comparison highlights the strengths and trade-offs of each platform based on our project's requirements. We have also highlighted the FPGA board that our team eventually decided on. Factors were carefully decided based on three categories. Some include available logic resources, such as logic cells and RAM. Others pertain to ease of development, power efficiency, and cost. were all considered during our evaluation. In the end, the Arty S7 stood out to us for its balance of capability, accessibility, and compatibility with our design constraints and toolchain preferences.

Table 3.3 FPGA Comparison & Selection.

Feature	Xilinx Arty S7-50: Spartan-7	Xilinx Arty A7-100T: Artix-7	Intel Cyclone V GX Starter Kit
Logic Cells	52,160	101,440	77,000
Flip-flops	65,200	126,800	56,832
Block RAM	2,700 Kbits	1,800 Kbits	1,532 Kbits
Clock Frequency	100 MHz	100 MHz	programmable (10 - 810 MHz)
Communication Protocols	- USB-UART - USB-JTAG - Quad-SPI Flash - Pmod ports - Arduino/chipKIT shield connector	- USB-UART - USB-JTAG - Quad-SPI Flash - Pmod ports - Arduino/chipKIT shield connector	- PCIe Gen2 - USB 2.0 - UART - I²C - SPI - LVDS
Memory Interface Support	- 256 MB DDR3L with a 16-bit bus @325 MHz (650 MT/s) - 128 Mbits Quad-SPI Flash	- 256 MB DDR3L with a 16-bit bus @667 MHz - 128 Mbits Quad-SPI Flash	Supports DDR3, DDR2, LPDDR2
Development Tools	Vivado Design Suite, WebPACK Edition	Vivado Design Suite, WebPACK Edition	Quartus Prime, Lite Edition
Board Price	\$150-\$200	\$299.00	\$270.00
Applications	Embedded systems, education, prototyping, hobbies	High-performance systems, communications	Industrial, automotive, telecommunications

Figure 3.2 Arty S7 Spartan-7 FPGA Development Board. Retrieved from Digilent (n.d.).



3.1.3 Level shifter

In this section, we'll discuss the use of level shifters on our PCB and compare the most common level-shifting methods: resistor-based, op-amp-based, and transistor-based. For our LED matrix, we want to ensure that the large amount of logic signals from our Spartan-7 FPGA will properly translate into the LED matrix. Depending on the flash chip, we'll likely need two level shifters to operate SPUD without any component issues.

In terms of requirements to consider for choosing the type of level shifter, there are a few that are necessary for our project, these include: 3.3V to 5V conversion, low propagation delay, directionality, and durability.

Proper **voltage conversion** will be an absolute necessity when choosing the right level shifter for SPUD. Because we are using components that have different voltage requirements (i.e. FPGA, LED Matrix), we must ensure that they can communicate thoroughly without damaging each other with their corresponding voltage levels. The Spartan-7 uses 3.3V logic levels for its pins and the matrix will require 5V signals to communicate effectively with our FPGA. Our level shifter needs to be able to translate the Spartan-7 output into a 5V input for the matrix without risking damage to the matrix or the data being sent to it. The same is true vice-versa, applying the 5V to a 3.3V pin risks permanent damage to the Spartan-7 and can destroy the SPUD entirely. Without correct

voltage conversion, we risk the potential for damage to our components when translating signals from the FPGA to the matrix.

We want a **low propagation delay** so that we can achieve a higher clock frequency, allowing for stronger signal speeds, leading to higher quality image rendering on our matrix. Having a low propagation delay ensures that our large matrix can reliably handle the data being transmitted without risking misalignment, which could ruin our expected output of the matrix. More importantly, when using a flash memory chip, we absolutely cannot risk having a signal timing issue, as this could cause problems with our data.

Directionality of a level shifter is a crucial feature that will ensure the correct flow of signals. If we want a simple and quick shifter that flows in one direction, we can go for a unidirectional level shifter. We mentioned earlier that applying a 5v to a 3.3V pin could risk damage, we can prevent this by using a unidirectional shifter, as we wouldn't have to worry about that voltage returning from the matrix to the FPGA. When discussing the directionality of a level shifter for the flash memory, we must consider the operations that the flash memory needs to complete. If our FPGA needs to read/write memory, then we must be able to shift in both directions for proper communication between the two components.

Lastly, when considering the **durability** of our level shifters, we want something that can handle the continuous data traffic to our 64x64 matrix. Because of how fast we want our data to reach the matrix, we'll want to opt for a level shifter that can move quickly while also handling the heat. If we use a cheap, undurable level shifter, we risk having it overheat while trying to send data to our matrix, which in turn, could result in a malfunction that would damage the components.

3.1.3.1 Resistor Design

Resistor-based level shifters are simple and typically aren't used often. There was a resistor-based design that we considered when looking at level shifters to use, specifically, voltage divider design. Other designs such as open-drain with a pull-up are more of a mosfet design rather than resistors. For this section we are focused on passive designs and their capabilities and constraints.

The simple voltage divider design will only decrease the voltage rather than increase. It's a basic circuit with 2 resistors that will pull down the voltage level, meaning it's capable of shifting to a lower voltage only. A voltage divider level shifter does meet the 3.3V-5V requirement, but only from 5V to 3.3V. That alone eliminates the chances of us using this design for our project. If that wasn't enough, the propagation delay for resistor-based is the highest compared to the

other level shifter designs. This won't be ideal when we're working with a high-speed LED matrix that will be requiring fast signals. Directionality is another crucial feature that the voltage divider resistor design lacks. Resistor-based design is only unidirectional, which of course means we'd have to stay locked into a single direction data transfer design.

Resistor-based level shifters would be a good fit if we needed a unidirectional shifter that could lower the voltage rather than increase it. In general, resistor-based design is unreliable and very slow, two things that we cannot risk if we want our LED matrix to function correctly. While its simple design might be useful for certain projects, this is a practically useless design for SPUD.

3.1.3.2 Op Amp Design

When considering op amp based level shifters, there were a couple variations of design that we had considered for SPUD, specifically, summing op amp design and comparator design. Both designs are fairly simple, yet capable in terms of basic level shifting.

The summing op amp design utilizes an op amp with two input voltages. Because of the nature of summing amplifiers, we'd need both the 3.3V input and a separate input voltage that would add into the 3.3V. Then, depending on the resistors ratio, we could scale the output to a 5V output voltage. While this sounds like a simple, easy design, we realized that the signal can only flow in direction. Unidirectionality is less of an issue in this case when compared to the resistor-based design, since we could still level shift up and down based on how we scale component values in the op amp. The problem lies in the speed of op amps as the LED matrix will need to be communicating at high speeds to function properly.

Using an op-amp design for the level shifter is a more viable option when compared to the resistor-based level shifters, but it still comes with its problems. On one hand, using an op-amp level shifter would be a simple design, the main issue with the op amp design is the speed. When working with our matrix, we'll be using shift registers for communication, which would be slowed down by the use of op-amps.

3.1.3.3 Transistor Design

When considering what kind of level shifter, we looked at MOSFET and BJT-based level shifters most, as they better meet our needed requirements

than resistor or op-amp based level shifters. But, we must first compare the two types of transistor level shifters.

MOSFET level shifters utilize a N-channel MOSFET alongside pull-up resistors to shift the voltage. This allows for our 3.3V input from the FPGA to be shifted up to 5V for the matrix. MOSFET-based level shifters also allow for unidirectional or bidirectional use, both which would serve useful to SPUD, depending on whether it is used for a parallel flash chip and/or the LED matrix. MOSFET-based shifters are much faster than both resistor-based and op-amp based and due to MOSFETs having higher switching speeds, they typically have a propagation delay in the low nanoseconds.

BJT level shifters typically utilize NPN transistors and resistors, thus allowing for the 3.3V to 5V conversion. BJT shifters would be a better choice for SPUD if we needed to have a continuous high amount of power to be handled. Otherwise, a BJT level shifter would only slow down our speeds. When compared to MOSFET-based shifters, BJTs are slower in both switching speed and propagation delay, which are both crucial to the communication and functionality of our flash memory chip and LED matrix.

3.1.3.4 Technology Comparison

We decided to use the transistor design for our level shifters, as they are the most stable and suitable for SPUD. Using a transistor-based level shifter will ensure that all of our needs are met when building the PCB for the peripherals. Not only are they the fastest type of level shifter available to us, but they are also capable of voltage conversion and are the most durable of the three. This will allow us to work on SPUD worry free when testing the LED matrix.

Table 3.4 *Level Shifter Technology Comparison & Selection.*

Feature	Resistor	Op Amp	Transistor
3.3V to 5V conversion	N/A Only 5V to 3.3V	Yes	Yes
Propagation delay	~50-200ns	~10-100ns	~5-50ns

Directionality	Unidirectional/ Bidirectional	Unidirectional/ Bidirectional	Unidirectional/ Bidirectional
Durability	Low	Medium	High

3.1.3.5 Part Selection

After deciding on the type of level shifter we needed for SPUD, we considered several widely available specific modules. Our short list includes the TXB0108, PCA9306, 74LVC245, and the TXS0108E. Each one offers unique trade-offs; all of the shifters differ slightly in terms of speed, directionality, voltage range, and integration complexity. The TXB0108, while also supporting bidirectional shifting and fast performance, was found to be less reliable for slower or heavily capacitive lines. The PCA9306 is commonly used in I²C communication but is limited to two channels and lower speeds, making it impractical for our multi-line design. The 74LVC245 is a unidirectional level shifter with good propagation delay and voltage range, but its lack of automatic direction sensing adds design complexity and makes it less flexible overall.

We have decided to use the TXS0108E level shifter by Texas Instruments. The TXS0108E simply meets all of our needs for a level shifter with an automatic bidirectional sensor, a maximum speed of 110 Mbps for push-pull and 1.2 Mbps for open-drain. Most importantly, it has a min/max input voltage of 1V/5.5V and a min/max output voltage of 1.4V/5.5V, meaning it can shift 3.3V to 5V easily. The capabilities of the resistor-based and op-amp-based level shifters are not suitable for SPUD as they are either unable to do proper voltage conversion or simply not as powerful as a transistor-based level shifter. The TXS0108E's compact design and reliable performance make it ideal for managing signal integrity between our FPGA and high-voltage peripherals. Its low power consumption and ease of integration also help simplify our overall circuit design and layout.

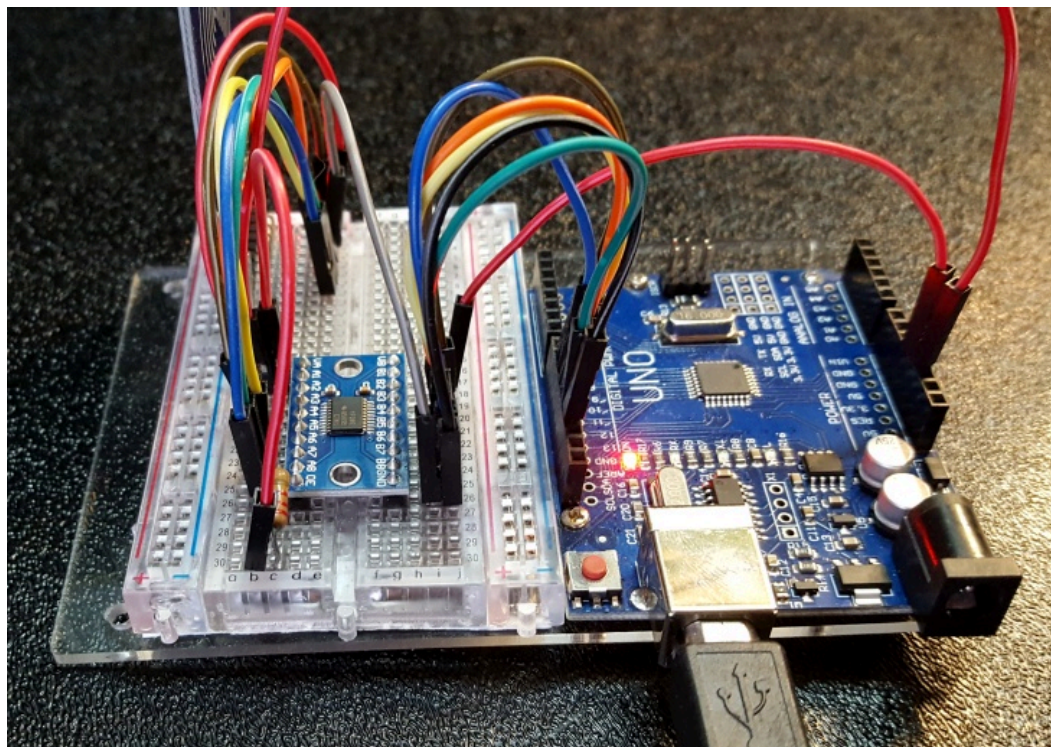
Table 3.5 Level Shifter Module Comparison & Selection.

Feature	TXS0108E	TXB0108	PCA9306	74LVC245
Propagation delay	1-4 ns	<1 ns	10-20 ns	6-8 ns

Directionality	Bidirectional	Bidirectional	Bidirectional	Unidirectional
Voltage Range	1.2V to 5V	1.2V to 5V	1.2V to 5V	1.65V to 5V
Channels	8	8	2	8

The image below demonstrates the TXS0108E. This module is particularly useful in systems like microcontroller projects where components with different logic levels need to interface. The module features 8 channels, allowing it to convert signals bi-directionally. It provides the necessary bridge between different components in mixed-voltage systems.

Figure 3.3. TXS0108E 8-channel logic level converter module. Retrieved from ProtoSupplies (n.d.).



3.1.4 Program Memory Chip

Memory is easily one of the most important components of our design. Computers essentially read, manipulate, and write memory. Non-Volatile Memory allows us to store programs for our CPU to fetch and compute. Because this chip will directly interact with our processor we will need to make

sure certain requirements are met. Our memory technology will need to be fast, reliable, and reprogrammable. We've chosen the key attribute we need our chip to support. It's also important to note here that our memory access will have a direct impact on our processing speed.

This **size of the memory** is not the most important feature, but bigger size will give us larger programs we can run. We'd like to choose a chip that's greater than 4Mb, While also allowing us to read and write quickly.

We choose to research a widespread array of **memory technologies** to make sure our project is well thought out and to minimize as many drawbacks as possible. The three types we chose to research are MRAM, Flash, and EEPROM. Each has their own advantages and disadvantages. The MRAM offers virtual unlimited endurance and fast access times, making it ideal for high speed applications. Flash is widely used for its cost effectiveness and ease of integration while EEPROM provides the most simple yet lower power solutions for small data storage. Our project will need a happy medium between these technologies that offers fast mem access and ease of implementation.

One of the most important parts is the transfer of data and the protocol used to do it. A protocol is any set of standards and rules that, when followed, result in the correct behaviors. For our **Write/Read protocol** on our memory devices we will be sending over bytes of information. Specifically we will be storing the machine code instructions for our processor. The protocols we will be researching are I2C, SPI Parallel. SPI and I2C are more suitable for low-speed applications but offer versatility and save IO while also allowing multiple devices to be on the same bus. Serial data transfer protocols mostly use a system where first a data address will be sent and after the mem device will send back the data at the address. It works by clocking out data one bit at a time. Parallel offers much faster speeds but requires much more IO. It needs pins for the address lines and also pins for the data lines at that address. This is what makes the protocol so fast as it has instantaneous read and write. When compared against each other, parallel superior in speed while serial protocols save lots of IO.

The **program technique** influences how efficiently data can be written and erased. Direct write technologies MRAM eliminate need for erase cycles which enhances performance in applications requiring frequent updates. Page programming with sector erase, used in Flash, requires more steps but is most cost-effective and widely supported. Self-timed erase/write cycles in EEPROM ensure reliability but may be slower overall.

Access speed is vital for applications requiring fast data retrieval or updates. Technologies with fast access times, like MRAM's 35ns, are ideal for high-performance systems. SPI Flash can operate at high frequencies, making it suitable for applications needing rapid data transfer. Slower technologies like EEPROM are better suited for low-speed applications where data is infrequently accessed.

Endurance refers to how many times data can be written before the memory degrades. Technologies like MRAM offer virtually unlimited endurance, making them suitable for applications with frequent updates. FLASH and EEPROM have limited endurance typically 100,000 cycles, which can be a constraint in high-write environments.

Data retention is critical for ensuring that stored data remains valid over time. Technologies like MRAM and EEPROM offer long-term persistence, with MRAM being particularly robust. FLASH typically retains data for over 10 years, which is sufficient for many applications but may not be enough for long-term archival purposes.

The **voltage range** affects the compatibility and reliability of the memory chip in different power environments and needs to be specific to our application. Some chips with a wide voltage range, like 1.7V-5.5V, offer flexibility in various power supply conditions, making them suitable for battery-powered devices or systems with variable voltage supplies. Narrower ranges, such as 2.7V-3.6V, require more stable power conditions but can be more energy-efficient. For our application we need ICs that support 3.3V data and power.

The **temperature range** determines the environmental conditions under which the memory chip can operate reliably. Industrial-grade chips typically operate from -40°C to +85°C, while automotive-grade chips can handle up to +125°C.

3.1.4.1 M3004316

The **M3004316** offers several key advantages using non-volatile MRAM technology. It provides high speed with read and write times of 35ns. MRAM offers virtually unlimited write cycles with degrading. When compared to other storage technologies like Flash or EEPROM it is infinitely better. It also consumes low power, especially while doing write operations to make it energy efficient. MRAM is ideal for applications that require fast data access and frequent updates. This chip also uses parallel data access to further increase its speed when compared to serial data interfaces like SPI or I2C. However one of

the main challenges are the size of the cells. They are hard to scale down and make dense memory options which limits widespread adoption.

3.1.4.2 AT25DL081

One of the most common non volatile memory ICs use SPI Flash storage technology. Such is true for the **AT25DL081**, offering flexible erase architecture and high operating frequencies. Its main data bus is utilized with the serial SPI protocol and offers features like sector protection with one time programmable security register. This makes it suitable for secure applications. A strong advantage of the ICs is its cost effective and easy integration. However it has limited endurance and requires a separate erase phase before writing to a sector. This can slow down write operations. Additionally, data retention is good but not as good as MRAM.

3.1.4.3 Quad-SPI Flash on Arty S7

The most convenient option of non volatile memory is the onboard flash chip that's already installed on our FPGA. The Arty S7 provides Quad-SPI Flash interface that's already connected to IO on our FPGA. Since it uses the Quad-SPI protocol, it makes it a great option for faster data transfer compared to the standard SPI interface. This leverages cost effective flash technology while also transferring data at high speeds. However since the chip is Flash it also has limited endurance and requires sector erase before writing.

Ultimately this is the technology we are choosing to use since our FPGA development tool allows us to add sections to the onboard flash chip to upload programs alongside the bitstream needed for the FPGA configuration data. The main drawback of using the serial SPI interface is that it needs far more time to get memory which will have a direct impact on our CPU. This is a compromise that we are willing to take for the ease of development and the lack of OI we would need to integrate our own parallel memory chip. If we would like to speed up the serial SPI interface we could invest our R&D into prefetching instructions since our programs will be stored on the chosen Flash device.

3.1.4.4 Comparison & Selection

After researching and discussing the solution with the team, It's in our best interest to select the pre-flash existing chip already attached to our FPGA. Though the drawbacks are unavoidable, the ease for development and implementation for the scope of this project justify our decision to use this technology.

Table 3.6 Flash Memory Comparison & Selection.

Feature	M3004316	AT25DL081	24LC16B (Arty S7)
Size	4Mb	8Mb	16 MB (128 Mbits)
Mem Technology	MRAM	FLASH	FLASH
Write/Read Protocol	Parallel	SPI	Quad-SPI
Program Technique	Direct write (no erase needed)	Page programming with sector erase	Page programming with sector erase (similar to standard SPI)
Access Speed	35ns read/write	~100MHz clock, sequential access	Up to 50 MHz data rate for Quad-SPI programming
Endurance	Virtually unlimited	100,000 cycles (typical)	Typically 100,000 cycles (similar to standard SPI)
Data Retention	Long-term persistent	>10 years (typical)	>10 years (typical)
Voltage Range	2.7V-3.6V	1.65V-3.6V	1.65V-3.6V (for S25FL127S/S25FL128S)
Temperature Range	-40°C to +105°C	-40°C to +85°C	-40°C to +85°C

3.2 Software Technology Comparison and Selection

The software in our project has also been thoughtfully developed and integrated to fit our needs. At the base of our software stack is the RISC-V Instruction Set Architecture (ISA), which serves as the foundation for our processor design. In order to test our source code, we research simulation and verification tools for the testbench code. The software toolchain, which includes synthesis and implementation tools, allows for the translation of our high-level design into a working hardware model. Additionally, we explore PCB software to finalize our circuit design and properly interface the processor with the other components. Each component of the software stack has been carefully selected for our project.

3.2.1 ISA

The selection of an instruction set architecture (ISA) forms the foundation of any processor design. This choice will influence both hardware implementation and software toolchain development. The ISA will determine how instructions are decoded into control signals, shape compiler complexity, and mold that instruction type. Choosing an ISA will directly affect the balance between computational efficiency and design simplicity. For our FPGA based processor we require an ISA that prioritizes extensibility, decode efficiency, and minimal resource overhead while retaining compatibility with modern development workflows and toolchains.

The development of RISC-V was motivated by the need for a more simple instruction set. Specifically one that's flexible and extensible that could support both research and enterprise applications. Unlike the big names during RISC-V's infancy, RISC-V goal was designed to be open and modular while supporting a wide range of implementations. Applications like embedded systems, microcontrollers, and high performance, efficient multi core systems.

The creators of RISC-V sought to address the limitations of existing ISAs at the time. Particularly the challenges posed by licensing restrictions and the lack of extensibility in commercial offerings. The open source aspect of this ISA enabled a vibrant ecosystem of academic and industrial contributors, accelerating innovation.

In specific the modularity of this ISA in its foundation. There are many different versions of RISC-V that can be easily adapted to.

Three dominant architectures we are choosing as candidates are x86, Arm, and RISC-V. x86 is a CISC used in high-performance computing while Arm is a RISC based ecosystem typically used in mobile and embedded devices. While the only open source options being RISC-V as a RISC architecture designed for modularity. While x86 offers dense instruction encoding for general purpose workloads and Arm provides a mature commercial ecosystem, we select RISC-V for its configurability, royalty free licensing, and streamlined base specification.

The choice of **architecture type** is between Reduced Instruction Set Computing (RISC) and Complex Instruction Set Computing (CISC) that fundamentally shape SPUD's design. RISC-V, introduced in 2010 as an open-source ISA, emphasizes modularity and simplicity, enabling customizable implementations optimized for embedded systems and smaller processors. Its modular model contrasts with Arm's proprietary architecture, which evolved from 1980s Acorn computers to dominate mobile and embedded markets through energy efficient designs and standardized toolchains. x86, a CISC architecture developed by Intel in 1978, prioritizes backward compatibility and dense computational power for general purpose computing. For our applications x86 would certainly hinder our development time as it's far more complex for our application.

Fixed **length instructions**, as seen in RISC-V and Arm, simplify pipelining and decoding logic, which could potentially reduce timing uncertainties and resource overhead. Variable length encoding in x86 complicates fetch and decode stages but improves code density for memory-constrained applications. For our implementation we must balance these factors. RISC-V's compressed extension RV32 halves instruction size for embedded workloads without sacrificing decode efficiency, while x86-style variable encoding demands complex finite-state machines that consume FPGA lookup tables (LUTs)

IEEE 754 **floating-point** units (FPUs) enable high dynamic range computations in scientific simulations and neural networks but consume significant hardware blocks and logic. Native floating-point workflows trade ~30% higher resource usage compared to fixed-point for simplified algorithm porting and reduced quantization errors. RISC-V's optional F or D extensions provide configurable precision, while x86's legacy x87 stack complicates FPGA microarchitecture.

Custom instruction **extensions** in ISAs allow domain specific tasks. RISC-V's modular design enables seamless integration of application specific instructions (ASIs) for cryptography operations without disrupting base ISA functionality. For project SPUD we could easily integrate extra opcodes to extend our functionality using RISC-V. Arm's coprocessor interface and x86's extensions offer less

flexibility, requiring hard macro integration. Not to mention Arm and x86 require licensing to use their architecture.

RISC-V's fixed-length 32-bit **instruction encoding** opcode, registers, and immediate into consistent bit fields, enabling efficient combinatorial decoding in FPGAs. x86's variable-length prefixes and ModR/M bytes necessitate multi-stage decoding pipelines that increase LUT consumption. FPGA-specific encodings can exploit unused opcode ranges for custom operations, such as adding tensor instructions between standard ALU and load/store opcodes.

Addressing flexibility impacts memory access patterns and the potential FPGA routing complexity. Immediate and register-indirect modes dominate RISC-V's load/store architecture, minimizing memory disambiguation logic. x86's complex addressing (base+index*scale+displacement) suits data structure traversal but requires multi-stage address generation units. FPGA designers often extend base ISAs with circular addressing for DSP buffers or bit-reversed modes for FFT accelerators, balancing mode richness against critical path delays.

3.2.1.1 RISC-V

RISC-V is a Reduced Instruction Set Computing architecture that has gained significant popularity in both academic and industry settings because of its flexibility, modularity, and royalty-free licensing model. The architecture was introduced in 2010 to emphasize simplicity and extensibility. One of its key features is its fixed 32-bit instruction length in its base specification, with the option for compressed 16-bit instructions for memory-constrained applications. The RISC-V ISA is designed to allow for easy customization; developers can add their own extensions. Additionally, RISC-V offers optional floating-point extensions (F and D) to provide flexibility to its precision for computing. RISC-V's simple instruction format makes it easier to implement in hardware, especially when working with FPGAs. Lastly, being open-source means there's a big community behind it that offers plenty of support.

Typically used in smaller more lightweight applications you'll find RISC-V used in products with simple microcontrollers or solutions needing customization. RISC-V common applications are used in IoT, edge devices, automotive, and more. It's also common to find RISC-V in implementations in FPGAs. Due to its simplicity and efficiency it is a very portable and versatile instruction set making it quite desirable.

3.2.1.2 ARM

ARM is another popular RISC architecture that is known for its role in mobile and embedded devices. It was originally created in the 1980s by Acorn Computers, and has grown into one of the most widely used ISAs in consumer tech. ARM processors are known for their energy efficiency and wide range of tools and software support available. It offers both 32-bit and 64-bit instruction sets, with the standard ARM mode using 32-bit instructions. Unlike RISC-V, however, ARM is less flexible because it is a proprietary system that requires licensing. Its instruction encoding and flexible addressing modes are efficient for general computing, but they may require more resources when implementing on FPGAs.

ARM is now the Dominate instruction yet in smaller devices. Specifically in embedded systems, microcomputers, smartphones, tables, wearables, due to its low power versus high performance.

3.2.1.3 X86

x86 is a CISC architecture and is the go-to architecture for most desktop, laptop, and server processors. x86 processors are designed to handle more complex instructions and use variable-length encoding. This helps with memory efficiency but makes decoding more complicated. This added complexity can make implementing x86 on an FPGA more challenging. Moreover, the architecture's complexity and its backward compatibility requirements, can make it resource-heavy when adapting to custom hardware. For SPUD, choosing x86 would add unnecessary complexity and licensing issues, making it much harder to work in than a modular and open-source environment.

3.2.1.4 Technology Comparison

For project SPUD, it makes the most sense to choose the ISA with the best extensibility, free licensing, and open-source development to implement into our project. The advantages that RISC-V provides perfectly benefit our project in a way that gives us freedom and easier development. Also, being open source gives us a large community to reach out to for help and feedback. This choice of a smaller, more simple ISA is also fitting for the scale of our project. RISC-V allows us to stay focused on building our processor from the ground up without the limitations or overhead of proprietary architectures. Because of its modular nature, we can build upon our base CPU later—adding custom instructions, debugging features, or performance upgrades without needing to completely redesign the system. In contrast, ARM and x86 would have introduced unnecessary complexity and required dealing with licensing issues that don't

align with the educational and exploratory goals of SPUD. RISC-V's hardware-friendliness and emphasis on readability make it much easier to test, verify, and implement on an FPGA. RISC-V, therefore, gives us just the right balance of control, performance, and simplicity needed for our goals.

Table 3.7 ISA Comparison & Selection.

Feature	RISC-V	ARM	X86
Architecture Type	RISC	RISC	CISC
Instruction Length	Fixed 32-bit (base), supports 16-bit compressed	Fixed 32-bit (ARM), 16-bit (Thumb)	Variable length (1-15 bytes)
Extensibility	Highly extensible	Limited extensibility	Limited extensibility
Instruction Encoding	Simple, fixed format	Fixed format	Complex encoding
Address Modes	Simple	Multiple addressing modes	Complex addressing modes
Open Source	Yes	No	No
Market Adoption	Growing	Widespread	Dominant in desktop/server

3.2.1.4 Technology Selection

The RISC-V instruction set has many interactions and versions that differ greatly from one another. For the scale of our project we are mostly interested in a solution that is powerful but simple and easy to implement. Too many tricks and complications will make the process more complicated and slow down development. For the needs of our project we have found that RV32IM will suit our project perfectly. We will be giving up decimals and floating point precision, but this isn't critical to our project. Just integers will suffice for the programs we are writing. Also we specifically chose the 32 bit architecture. This reason being is again its simplicity and will speed up our memory fetch.

Table 3.8 ISA Comparison & Selection.

Feature	RV32IM	RV64F	RVDFD
Instruction Set Type	Base integer instructions	Single-precision floating-point	Double-precision floating-point
Registers	32 integer registers	32 integer + 32 floating-point	32 integer + 32 floating-point
Advantages	Simple and compact for embedded systems	Enhanced arithmetic precision for scientific applications	High precision and fused operations for advanced computing

3.2.2 Simulation and Verification Software

One of the most important aspects of designing hardware is verification. As such, we have found three contending verification software tools that have the capability of testing and simulating a design as complex as SPUD's processor. These tools were evaluated based on several criteria, including accessibility, ability to handle complex designs, compatibility with Vivado, and support for Universal Verification Methodology (UVM, a stretch goal).

3.2.2.1 EDA Playground

EDA Playground is an online platform that supports writing, simulating, and debugging SystemVerilog designs. It is a free tool and does not require installation, which makes it extremely convenient for verifying our design in its early stages. The platform supports multiple simulators, including Synopsys VCS and Cadence Xcelium, which are the two other verification tools that we will be evaluating. However, the fact that EDA Playground can incorporate either VCS or Xcelium is a huge bonus because it allows for consistency between the tools.

To evaluate EDA Playground's suitability for SPUD, we looked into an existing RISC-V RV32I processor implementation, called DarkRISCV, which is available on the platform. This design demonstrates that EDA Playground is capable of simulating a RV32I processor that includes key components like the ALU, control unit, register file, etc. The testbench within the design is significantly simple, unfortunately. Therefore, we will need to do further research regarding

testbenches for our design so that we can sufficiently verify that the processor exhibits proper functionality.

EDA Playground also has limitations when handling more complex verification tasks. Since it is a web-based environment, simulation performance could be constrained by resource availability. This could become a bottleneck when testing a bigger design or doing timing analyses. Regarding UVM support, one of us has experience using UVM with EDA Playground, therefore we know firsthand that the platform fully supports formal verification methodology. The only concern would be related to complexity. Verifying a much more complex design like SPUD's may not be feasible using this platform. Additionally, the debugging tools within EDA Playground are not as powerful as those available in Vivado, Synopsys VCS, or Cadence Xcelium.

Given its high accessibility, we still plan to incorporate EDA Playground into our verification process. However, it will not serve as our main verification tool. Instead, it will function as an initial verification measure for testing SPUD's CPU functionality. This will allow us to verify individual modules, such as ALU operations, control flow, and register interactions first. Verifying these lower-level modules is vital because it simplifies debugging and guarantees that smaller components are working correctly before advancing to a higher-level design.

3.2.2.2 Synopsys VCS

Synopsys VCS is a high-performance simulation tool that is widely used in the industry for verifying complex hardware designs. It offers significant support for UVM, making it well-suited for advanced verification methodologies. Since VCS is available on our school server (vlsi.eecs.ucf.edu) and integrates with Vivado, it is a strong choice for testing and debugging SPUD's CPU. Furthermore, VCS is known for its efficient handling of large designs, making it an extremely reliable choice for verifying the full processor design. While it does require a more thorough setup compared to EDA Playground, its expertise with functional verification makes it a strong contender for our final verification strategy.

3.2.2.3 Cadence Xcelium

Cadence Xcelium is another industry-standard simulator with strong support for SystemVerilog, UVM, and mixed-signal verification. Like VCS, it is accessible via UCF's VLSI server and is supported by Vivado. Xcelium is known for its advanced debugging capabilities and efficient parallel processing, which can speed up simulation times for large-scale designs. Its ability to handle complex testbenches makes it a powerful tool for verifying SPUD's CPU. We also have

some first hand experience with Xcelium, which is a huge plus. Given its flexibility, performance, and capability, Xcelium is a great alternative to VCS for ensuring the correctness of our design before we are ready for its implementation on the FPGA.

3.2.2.4 Comparison & Selection

After evaluating these options, we plan to use a combination of EDA Playground and either Synopsys VCS or Cadence Xcelium for simulation and verification. EDA Playground will serve as an initial step to verify the accuracy of our CPU RTL design before transitioning to FPGA implementation. This approach allows for quick testing and minimizes the need for resource-intensive simulations early on. Once the design is verified at a basic level, we have decided to utilize VCS to perform more thorough verification. However, should we feel the need to switch over to Xcelium instead, we know we can easily do so. We have simply chosen VCS due to existing familiarity with the tool.

Table 3.9 Verification Software Comparison & Selection.

Criteria	EDA Playground	Synopsys VCS	Cadence Xcelium
Accessibility	Free website; no installation needed	Requires installation, available on UCF's vlsi server	Requires installation, available on UCF's vlsi server
Complex Design Handling	Limited by web-based resources, struggles with large designs	Optimized for large-scale designs, highly efficient	Strong parallel processing, handles large designs well
Vivado Compatibility	No direct integration	Direct compatibility	Direct compatibility
UVM Support	Fully supports, but complex designs may be difficult to verify	Strong support	Strong support
Debugging Capabilities	Limited capabilities	Advanced capabilities	Advanced capabilities

Simulation Performance	Constrained by resource availability	High performance for functional verification	Fast parallel simulation for large testbenches
Application for SPUD	Initial verification for individual modules	Full CPU verification	Full CPU verification

3.2.3 Source and Testbench Code Selection

Developing a CPU from scratch is an extensive process that requires not only careful design and verification but also significant time for debugging and modifying the architecture. Given the constraints of our senior design timeline and the challenges faced by previous teams, we determined that building an RV32I processor from the ground up would not be feasible. The true difficulty of this project will not stem from the design process of a processor, but in implementing it on an FPGA and integrating the necessary components to demonstrate that it functions correctly. If time allows, we would love to enhance the CPU design further by adding certain features such as pipelining, floating-point support, multiple stages, branch prediction, etc. However, it is important to note that in our system implementation process, we may already need to modify the selected source code to account for game support. Regardless, due to all these factors, we emphasize the difficulty of the system implementation of a CPU on an FPGA. Without an existing processor implementation to start from, we would risk spending most of our time verifying that basic CPU operations work rather than focusing on how the processor interacts with memory, I/O, and external peripherals like the LED matrix.

To navigate around this, we explored publicly available RISC-V RV32I core implementations that align with our design goals. By selecting a well-structured, verified core, we can shift our focus towards integration, testing, and demonstration rather than low-level CPU design. Our priority is to make sure that the processor can be successfully synthesized onto an FPGA and interact with supporting components to run real applications like a game. We have compared three repositories that offer different trade-offs in terms of architecture, flexibility, and testing structure.

3.2.3.1 RISC-V Core with RV32IM Extension

The first repository [1] provides a SystemVerilog implementation of an RV32IM core with a five-stage pipeline architecture. The design follows the standard RISC-V pipeline structure with distinct instruction fetch, decode, execution,

memory, and write-back stages. Additionally, it includes a forwarding unit and a hazard detection unit. Both are important for fixing data dependencies and avoiding any stalls during execution. This design is more complex than our needs for SPUD, as our plan was to incorporate a single cycle design (no pipelining). However, we are open to increasing the complexity of our design if the optimizations are worth the risk (the more complicated the design, the greater the risk of implementation errors). One of the key advantages of this implementation is its inclusion of the 'M' extension, which provides hardware support for multiplication and division. While SPUD's initial focus is on the base RV32I instruction set, the presence of these additional instructions could be beneficial for game execution. However, the addition of the 'M' extension into our project would require further verification that these instructions do not introduce more issues in synthesis or FPGA implementation.

Despite the strong architectural foundation, the author of the repository states that this implementation has not been tested on actual hardware. Therefore its FPGA compatibility is uncertain. The author also notes that simulation was performed using ModelSim, which implies that the design has been functionally verified at a basic level but has not undergone any sort of synthesis or timing analysis for an FPGA. With SPUD, we are mostly focused on real-world execution rather than purely software-based simulation, thus additional modifications and debugging will most likely be required to make this core fully functional on an FPGA.

Another important aspect to consider is that this project was originally a school assignment with limited documentation. Compared to other available implementations, this repository offers strong architecture: solid pipeline structure and useful hazard-handling features. We will definitely take these optimizations into consideration, however these uncertainties make it a less favorable choice for our project.

3.2.3.2 Single-Cycle RV32I Processor with Simulation Focus

This repository [3] also represents a SystemVerilog implementation of an RV32I processor, but it lacks sufficient documentation and detailed organization. The design itself– while functional– does not come with any clear descriptions of its architecture or any useful instructions on how to use it effectively. This vagueness makes it challenging for us to integrate it into our project. The repository does provide testbenches, however, for individual modules, but it lacks a testbench for the top-level design. If we wanted to test the entire design, we would have to search elsewhere for a top-level testbench. This makes it

challenging to fully verify the processor's functionality in a complete system context. We may still be able to use these testbenches if we want to take additional verification measures. The lack of organized and detailed documentation, nonetheless, makes it difficult to understand the design fully. If we move forward with this repository, we may spend more time on reverse engineering than we can afford with the project's timeline.

Given these challenges, we have decided to not move forward with this repository. Whereas the core processor design may be functionally correct, there is insufficient documentation.

3.2.3.3 Minimalist RV32I Processor with Single-Cycle and Multi-Cycle Support

The final repository [2] provides a SystemVerilog implementation of two types of RV32I processors: a single-cycle processor and a multi-cycle processor. Since our current plan is to implement a single-cycle processor for SPUD, this repository stands out as a strong candidate because of its well-organized structure. By including both single-cycle and multi-cycle source codes, it allows for easy comparison between the two, giving us the option to explore adjustments if needed.

A key advantage of this repository is its clear separation between processor architectures. The single-cycle implementation processes each instruction in one clock cycle, making it a straightforward design well-suited for our FPGA-based implementation. However, if we run into timing issues or resource constraints, the multi-cycle implementation could be considered as an alternative that could reduce the use of hardware at the cost of execution speed. Regardless, having both implementations available makes this repository really useful because it provides a clear path for potential modifications or optimizations.

Another key aspect of this repository is that the author states that the design is purposefully written in SystemVerilog for FPGA implementations. This aligns perfectly with our project's goals. This repository also clearly provides us with its licensing rules regarding the use and modification of the code into our project. It has an MIT License, which states that we can use the code freely. This makes it the only repository we have found with explicit licensing.

3.2.3.4 Comparison & Selection

The source code is extremely vital for our project. SPUD needs an organized, functional, and testable code structure. Thus, we select the repository that best

matches this criteria. Taking these aspects into consideration, the repository that reflects our best interests is the RV32IM Core. This provides a well-organized starting point for our processor implementation while also being well known,

Selection	RV32IM Core	RV32I (Single & Multi-Cycle)	UltraEmbedded RV32IM Core
Architecture	Five-stage pipeline with forwarding and hazard detection	Single-cycle or multi-cycle implementations	Two/three-stage pipeline (configurable) with optional caches, MMU, and AXI bus support
Complexity	More complex than SPUD's needs, but includes useful optimizations	Well-balanced; single-cycle design with an option for multi-cycle	More complex than the other cores, but configurable for FPGA SoCs
FPGA Compatibility	Not tested on hardware, simulation only (ModelSim)	Designed for FPGAs	Verified on similar (but not the same) FPGA
Testing & Verification	Functional simulation performed, but no FPGA testing	Well-organized and has all the necessary testbenches	Includes extensive testbenches and a working SoC reference design (booting Linux!)
Documentation	Limited, originally a school assignment	Clear structure with licensing information (MIT License)	Well-documented repository
Extensions	RV32IM (includes multiplication and division)	RV32I only	RV32IM (M-extension for multiply/divide), optional caches, MMU, and peripheral support

3.2.4 Toolchain Selection

In this section, we will describe our selection of the right toolchain for SPUD. We need tools that not only support the RISC-V architecture but also provide the necessary features we need to create the bridge between the hardware and software.

The toolchain selection directly impacts our ability to design, implement, and test both the hardware and software components of our processor. In regards to SPUD, we need a well-integrated set of tools that allow us to synthesize the processor design, compile and run programs, and debug execution issues. Without the right toolchain, we'd risk running into compatibility issues, inefficient workflows, and difficulties in verifying that our CPU operates correctly.

3.2.4.1 Vivado

Vivado is an ideal choice for this project because it's specifically designed for FPGA development and it is essential for implementing our processor design onto the FPGA. As we are already using Vivado for the FPGA portion of the project, it provides a complete set of tools for synthesizing, implementing, and programming designs onto the hardware. Vivado's integration with other components of the development flow guarantees that we can efficiently manage every stage. Furthermore, Vivado supports simulation and integrates with tools like the GNU Toolchain.

3.2.4.2 RISC-V GNU Toolchain

The RISC-V GNU Toolchain is the open-source software development kit tailored for the RISC-V architecture. This toolchain includes several tools such as the GCC compiler, binutils, and GDB debugger. The GCC compiler will allow us to compile high-level code, including the games we will run on our processor, into machine code that our processor can execute. Binutils will assist with tasks like assembling and linking code, and GDB will allow us to debug the software running on our processor. This toolchain is cross-platform, meaning we can write software on a host system and compile it for the processor. There is also a huge open-source community behind this tool that increases its reliability.

3.2.4.3 OpenOCD

OpenOCD is a tool used for the debugging and flashing process. It allows for on-chip debugging and allows for interaction with the processor during execution to troubleshoot any issues. It also assists with programming the FPGA with the synthesized design and loading the compiled software onto the

processor. However, after further consideration, we have decided that OpenOCD introduces significant complexity that does not align well with our project. Setting up OpenOCD for debugging requires a lot of configuration. This includes writing custom scripts to interface with our specific FPGA and processor. Additionally, OpenOCD is mostly designed for systems with JTAG-based debugging hardware, which will add a lot of extra time to configure. Given our project timeline and the learning curve associated with properly configuring and using OpenOCD, we have decided not to move forward with this tool.

3.2.4.4 Comparison & Selection

Vivado excels at the FPGA-specific tasks such as synthesis, placement, and routing for hardware. On the other hand, the RISC-V GNU Toolchain provides the necessary support for software development, compiling, and debugging for the RISC-V processor. Together, they form a sturdy bridge between the hardware and software of SPUD. While tools like OpenOCD are available for debugging, they would introduce unnecessary complexity, which could hinder the project's progress. Therefore, we have chosen Vivado and the RISC-V GNU Toolchain as our primary tools in the development process.

Table 3.11 Toolchain Comparison & Selection.

Selection	Vivado	RISC-V GNU Toolchain	OpenOCD
Primary Purpose	FPGA design and simulation	Software development for RISC-V	Debugging and flashing
Features	- FPGA synthesis and implementation - Hardware programming - Simulation support	- GCC compiler for RISC-V - Binutils for assembling and linking - GDB for debugging - Cross-platform support	- On-chip debugging - FPGA programming support - Software loading onto the processor

Role in SPUD	Implements and verifies the CPU on the FPGA, manages design synthesis and simulation	Compiles high-level software (including games) into executable machine code for SPUD's CPU	Facilitates debugging and flashing the CPU design onto the FPGA, ensuring software and hardware interaction works correctly
Reason for Selection	Needed for FPGA implementation and simulation	Open-source and widely supported; provides all the necessary tools for RISC-V software development	Enables real-time debugging but at the cost of significant extra time and effort

3.2.5 PCB Design Software

When building the PCB for the SPUD, we had several options for the design software. We felt that Fusion360, Altium, and KiCad were our most comfortable options, as we've had experience with each. Below, we'll discuss the differences between each design software and how we decided which would be most suitable for the SPUD project.

3.2.5.1 Fusion360

Fusion360, better known as EAGLE CAD, would allow for a simple design, especially for integrating the level shifters, FPGA connections, and flash memory because of its versatile component library support. Not to mention, it is also free, which is good for us as we wanted to keep the budget low.

EAGLE uses a built-in renderer in 2D and even allows you to push your PCB into Fusion360 for a full 3D render of your design, this allows us to view our PCB and its layouts. It's very basic, but would serve us well since we're making a peripheral board for the rest of SPUD. In terms of component libraries, we'd be able to import pre-made libraries from 3rd party websites like Ultra Librarian or even community created libraries for existing components.

While EAGLE sounds like a sound choice, we wanted to step slightly outside of our comfort zone and use something a little more challenging than EAGLE. Because of this, we will not be using EAGLE for SPUD, but have heavily considered it.

3.2.5.2 Altium Designer

Altium Designer serves us best because it provides the tools need to properly integrate our components onto a PCB with the Arty S7 board. While the PCB serves as a simple peripheral board, it will require precise impedance-controlled tracing, and advanced signal/power integrity tools. These tools will ensure that we are properly handling voltage across the flash memory chip and level shifters, while also allowing us to simulate the behavior of our high-speed signals. While all of our options are able to provide these tools, Altium does them best.

Altium is the industry standard in terms of PCB design software, so all of its features are incredibly advanced and high quality. If our design was very compact and packed with components, we would be wise to pick Altium because of the signal integrity tools available to us. Altium 3D renderer has a built-in collision detection feature. This would be useful if our PCB had a lot more components. Altium also has access to many pre-made component libraries and can pull information such as pricing and availability.

Unfortunately, Altium is overkill for what we need done. Its high cost and high learning curve would only do more harm than good for the PCB and because of this, we will not be using Altium Designer.

3.2.5.3 KiCad

KiCad is an open source design software that is flexible and solid, but has a bigger learning curve when compared to EAGLE. KiCad serves as a strong alternative for creating the peripheral PCB even without the advanced tools such as SI (signal integrity) or EMI (electromagnetic interference) simulation. KiCad has its own features that make it a great choice for SPUD, as well as being completely free of cost.

Similar to EAGLE, KiCad has its own built-in 2D viewer of the PCB and a 3D visualizer for viewing the model of our PCB. We also have access to STEP file exports, which would let us work with our model in any MCAD software if we so choosed. Component libraries are available through the large open-source community. This would allow us to potentially work with existing libraries that have the same exact components that we want to use for our PCB.

While KiCad lacks some features that Altium offers such as the SI and EMI simulation, we felt as though it was our best option for our needs. KiCad is more than enough for the peripheral board and also allows for a stronger learning

experience as we are using software with a higher learning curve than we're used to.

3.2.5.4 Comparison & Selection

KiCad was our choice of PCB software simply because it has more than enough to offer for what we need. Not only is it a strong software that's completely free, because it has such a large community backing, we know we'll be able to work with the software to properly build our peripheral board. We can use community built component libraries to find the perfect parts we need to complete SPUDs board and this will serve as a learning experience for us as KiCad has a bit of a learning curve when compared to EAGLE, a software we're most comfortable with.

Table 3.12 PCB Design Software Comparison & Selection.

Feature	Fusion360	KiCad	Altium Designer
Cost	Free for students	Free	Subscription based
Ease of Use	Easy	Moderate	Intuitive
Routing Tools	Basic	Manual	Advanced & constraint-driven
Component Libraries	Good for common parts	Open-source	Large selection of verified parts
3D Visualization	Basic 3D support	Basic 3D support	Advanced 3D visualization
Version Control	Cloud based (Fusion360)	Git	Altium 365 (Built-in)

Chapter 4: Standards and Design Constraints

In the development of SPUD, both industry standards and design constraints must be considered. This chapter highlights the relevant standards that guide the design process and makes sure that the project meets the best practices in hardware design. In addition to that, we explore the practical constraints that must be considered (i.e. time, budget, power, and safety) to make sure that the project is feasible, reliable, and efficient. By following these standards and constraints, SPUD will be built to meet performance expectations.

4.1 Industry Standards

Industry standards are essential in ensuring consistency, reliability, and interoperability across various aspects of engineering and technology. By following established standards, we can create designs that are not only efficient and scalable but also compatible with existing technologies, making our work more accessible and future-proof. These guidelines help streamline development, reduce errors, and improve overall performance by providing a structured approach to design, manufacturing, debugging, and communication. In our project, adhering to these standards will be crucial for building a functional and reliable system. We will follow the RISC-V Instruction Set Architecture (ISA) Standard to develop a modular and flexible CPU, apply PCB Design and Manufacturing Standards to ensure a well-constructed and manufacturable board, use the JTAG (IEEE 1149.1) standard for debugging and testing, implement the SPI Communication Standard for high-speed data transfer, and rely on SystemVerilog Standards (IEEE 1800) for accurate hardware description and verification.

4.1.1 RISC-V Instruction Set Architecture (ISA) Standard

The RISC-V ISA is an open-source instruction set architecture of a CPU that is modular and free to use. By implementing a small base ISA (like RV32IM, the instruction set that we will be using for SPUD), the modular RISC-V ISA allows for a flexible complexity that can be modified with extensions. For example, the M in RV32M represents the extension for multiplication, and the F in RV32F represents the extension for floating point. Additionally, RISC-V is open-sourced and completely free, meaning that there are no royalties or costs involved when using the architecture.

The exact instruction set that we will be utilizing for SPUD is the RV32IM ISA with the multiplication extension. Regarding the registers of this ISA, they are 32

bits wide– with x0 hardwired to be always zero, x1 as the return address, and x2 usually as the stack pointer (this can be modified). There are also 5 unique instruction types such as R-type, I-type, S-type, B-type, and J-type. The following operations are supported: ADD, SUB, AND, OR, XOR, immediate value operations, SLT, SLTI, LW, SW, BEQ, BNE, BLT, BGE, JAL and JALR.

It is important to make sure that the source code we selected for our project accurately follows the RISC-V standard. The instruction behavior and way that they fetch, decode, execute, and write back must match the specifications. We also need to be able to verify that the source code is functional via testbenches and simulation.

This modular instruction set aligns perfectly with our project because we are designing SPUD to be as flexible as possible in terms of complexity. Our main goal is to develop a functional CPU on an FPGA that can drive a game on the LED display. Once we accomplish that, we will have the freedom to either enhance the architecture ourselves or integrate existing open-source extensions. Thus, RISC-V's structure allows us to start with a simple core (RV32I) and later introduce more complexity– either by adding custom instructions, optimizing performance, or incorporating extensions like RV32M and RV32F. By following this standard, we will create a scalable design that supports both our immediate objectives and any potential future enhancements.

4.1.2 PCB Design and Manufacturing Standards

An important aspect of this project is the manufacturing standards of our PCB for this project. Our design will start with a 4 layer pcb consisting of two routing plans, a power plan, and a ground plane to ease our design complexity. The SMD parts sourced must be big enough to easily work with while soldering. This includes passive components that must be bigger than 0805.

Our PCB orders must be from a reliable manufacturer with a reasonable shipping time. Our manufacturer of choice is JLCPCB as we've ordered a handful of boards from them before and are pleased with the results produced from this board house.

The board design needs to comply with the arduino hat standard to interface with our FPGA development kit as it also adheres to this header fitting. This will give us access to the available OI. Also needed is the fitting that works with the HUB75E interface to connect properly to the matrix display.

The PCB will be housing two level shifters, a HUB75E interface for the LED matrix, and headers for FPGA connection. Considering all of these are crucial components for the success of SPUD and nearly every pin will be utilized, there is little room for error. We will be following JLCPCB standards. Because we are designing a multi-layer PCB, when it comes to solder pads, we must have a minimum of 0.10mm between each pad.

Once we've received the board, we must apply solid standards to the construction of the board following arrival, specifically soldering the components. Poor soldering can lead to a poorly functioning board regardless of how well the board is designed/manufactured. We must ensure that we avoid any potential issues that can come from rookie mistakes such as cold solder joints. To accomplish this, we will control our environment such as the equipment we use to solder our board. Instead of using our own at-home equipment, we will utilize the lab resources and equipment provided by the university to ensure that our board is of a high standard.

4.1.3 JTAG (IEEE 1149.1)

The JTAG standard, also known as IEEE 1149.1, is a serial interface used for many purposes (debugging, testing, programming). This standard plays an important role in SPUD, as it will be used to program and debug the FPGA. JTAG will be used to make sure that the RV32IM ISA is executed properly and interacts with peripheral components (the LED matrix). Unlike traditional debugging methods that require direct access to internal signals, JTAG provides a non-intrusive way to test and program hardware through a dedicated Test Access Port (TAP), making it especially useful for verifying the CPU's operation and diagnosing hardware issues.

IEEE 1149.1 was developed to address the increasing complexity of multi-layer PCBs and densely packed integrated circuits, offering a standardized method for boundary scan testing. The TAP interface consists of five signals—TDI, TDO, TCK, TMS, and TRST—which allow communication with JTAG-compliant devices. This enables functionalities such as pin-by-pin connectivity checks, in-system programming, and real-time debugging. The standard also defines boundary scan registers, allowing each pin's state to be observed without physical probing. Additional features like instruction registers, bypass modes, and device identification (IDCODE) support help streamline testing and debugging, ensuring that JTAG-enabled devices can be efficiently verified during development.

For SPUD, JTAG will be the primary interface for programming and debugging the FPGA. Most modern Xilinx and Intel FPGAs support JTAG for direct configuration, allowing the CPU design to be synthesized and uploaded without requiring external flash memory. JTAG will also provide access to real-time debugging tools, such as Xilinx Vivado Hardware Manager, which can monitor and modify register values on the FPGA while it is running. Additionally, JTAG boundary scan techniques can be used to verify soldering and electrical connectivity, helping detect common assembly issues like short circuits or open connections.

4.1.4 SPI Communication Standard

The Serial Peripheral Interface (SPI) standard is a commonly used communication protocol designed for high-speed, full-duplex data transfer between a master device and multiple peripherals. SPI is commonly used in embedded systems, microcontrollers, and FPGAs to interface with components such as flash memory, sensors, displays, and other digital devices. In the context of SPUD, our design could possibly utilize SPI to be used for fast data transfer between the FPGA and external peripherals, ensuring efficient communication without the complexity of parallel data buses.

Unlike protocols such as I²C, which rely on shared communication lines and address-based communication, SPI operates using a dedicated set of signals: MOSI (Master Out, Slave In), MISO (Master In, Slave Out), SCLK (Serial Clock), and SS (Slave Select). The master device controls the clock (SCLK) and manages communication with selected peripherals by asserting the corresponding SS line. Since SPI is a synchronous protocol, data transmission occurs in lockstep with the clock signal, allowing for higher speeds and lower latency compared to asynchronous protocols like UART.

The SPI standard does not impose a strict data structure or addressing scheme, making it a flexible and efficient solution for high-speed communication. However, this also means that SPI devices require custom configurations, including clock polarity (CPOL), clock phase (CPHA), and data bit order (MSB-first or LSB-first), which must be correctly set to match the peripheral's requirements. SPI supports various modes of operation depending on these settings, allowing for compatibility across different devices.

In SPUD, SPI could be particularly useful for interfacing with external memory, display controllers, or other high-speed peripherals that require fast data access. Unlike I²C, which supports multi-master configurations, SPI is primarily a single-master, multi-slave protocol, meaning the FPGA will act as the master

and control communication with any connected devices. To integrate SPI into SPUD's hardware design, the PCB will need to include properly routed traces for SPI signals, ensuring signal integrity and minimizing noise, especially at high clock speeds.

The Universal Asynchronous Receiver/Transmitter (UART) standard is a widely used protocol for serial data exchange between two devices. UART is particularly valued in embedded systems, microcontrollers, and FPGAs for its simplicity and minimal wiring requirements. Unlike synchronous protocols such as SPI, UART operates asynchronously, meaning it does not rely on a shared clock signal. Instead, both communicating devices must agree in advance on a common baud rate, which defines the speed at which data is transmitted and received. This makes UART well-suited for straightforward, point-to-point communication where only two devices need to exchange data.

4.1.5 UART Communication Standard

Standard UART communication involves two wires. One for transmitting data and one for receiving data, along with a common ground to ensure proper voltage referencing between devices. Due to its asynchronous nature, UART uses its data lines independently of each other. Meaning it can transmit and receive at the same time no matter the specifics of the start and stop. Each data transfer is organized into frames, which consist of a start bit, a configurable number of data which can be 7 or 8 bits. An optional parity bit for error checking, and one or two stop bits to signal the end of the frame. The start bit signals the beginning of a transmission by pulling the line from its idle high state to low, while the stop bit returns the line to idle, allowing the receiver to recognize the boundaries of each data frame.

The absence of a shared clock in UART means that timing and synchronization are achieved solely through the agreed baud rate and the framing bits. This approach simplifies hardware design but requires careful matching of communication parameters such as baud rate, data bits, parity, and stop bits on both the transmitting and receiving ends to ensure reliable data exchange. UART does not inherently support addressing or multi-device communication, making it best suited for direct links between two devices rather than complex networks. In the context of SPUD, UART could be used for simple, low speed communication tasks such as debugging, configuration, or interfacing with devices that do not require high data rates or complex bus architectures. Integrating UART into the hardware design involves routing the TX and RX lines appropriately and ensuring that both devices share a common ground reference, especially when operating at different voltage levels.

4.1.6 Verilog Standards (IEEE 1364 - 2005)

Verilog is a hardware description language that's used to model digital systems at various levels (from gate-level to behavioral modeling). Its standard name is IEEE 1364, and the final version is known as IEEE 1364-2005. This standard defines how Verilog code should be written, how it behaves during simulation, and the key language features used in circuit design.

In the context of SPUD, Verilog serves as the foundation for defining the CPU's logic. One of the easiest ways to identify it is via its syntax and file extensions. Verilog files typically use the .v extension. The language also does not include any of the advanced object-oriented and verification features that are in SystemVerilog. For example, traditional Verilog does not support classes, interfaces, or constrained randomization. Instead, it relies on procedural blocks like *always*, *initial*, and *continuous assignment* (*assign*) for defining behavior.

Additionally, ports and internal signals must be declared with specific widths and types (*wire*, *reg*, etc.), and timing behavior is often defined with sensitivity lists in *always* blocks. Example blocks such as *always @(posedge clk)* and *assign out = a & b;* are common syntax that exemplify the Verilog style.

By following the Verilog standard (IEEE 1364-2005), we familiarize ourselves with industry standards and set a strong foundation for describing hardware.

4.2 Practical Constraints

The SPUD project is subject to several critical constraints that shape its design, development, and implementation. Time constraints require careful planning and prioritization to meet the academic semester deadline. Power and thermal considerations influence hardware efficiency and reliability, particularly concerning the FPGA's performance. Economic and sustainability constraints guide cost-effective component selection while minimizing environmental impact. Ethical and security factors ensure responsible development, protecting both users and data from potential risks. Safety constraints mandate adherence to electrical and operational guidelines to prevent hazards. Lastly, manufacturability constraints focus on ensuring that the hardware design can be efficiently produced within budget and technical limitations. Together, these constraints define the project's feasibility and drive design decisions to achieve a functional and reliable final product.

4.2.1 Time Constraints

Time constraints are a critical factor in the SPUD project, as the development of

the CPU and associated hardware must be completed within two semesters. Moreover, as full-time engineering students, we must be aware that we will be dedicating time into this project in addition to our other coursework. Some of us will be taking a total of four classes in the upcoming fall semester, others will be balancing part-time engineering jobs. Thus, we must manage our time wisely and carefully break down our tasks to successfully deliver a completed system by the end of the fall semester.

We also need to be especially careful about burnout, an underestimated symptom that has incredible consequences. Caused by an imbalance between work and proper rest, burnout can lead to severe consequences such as a decrease in attention span, motivation, and/or productivity. In some cases, stressful situations weaken the immune system and can lead to sickness (it's happened before). While it seems like a stretch, it is very valid to assume that students in their final semester of their undergraduate degree will most likely be undergoing some of the highest levels of stress than ever before. There is an immense pressure to meet major project deadlines and pass classes in order to meet team expectations and graduate. Our goal is to prevent mistakes as much as possible in order to deliver a quality, impressive system. We must, therefore, find the perfect balance between productivity and rest to prevent burnout.

Another key time constraint to consider is the fact that this is a group project that comprises busy, hardworking students. There are some tasks that require the entire group to work together at a given time, and finding a suitable time that aligns with all four team members' schedules can be difficult. We must each align our priorities and allot time to work together. Most importantly, we want to use that time wisely, which takes extra planning.

Lastly, it is vital to consider a realistic timeframe for each stage in the design and building process. For instance, debugging and testing phases must be carefully managed since we need to be prepared for any unexpected issues. The tight timeframe also means that any changes to the design, whether due to errors or new requirements, must be carefully evaluated for their potential impact on the project timeline. Tasks such as simulation and verification, bitstream generation, and peripheral integration can take a lot of time. We also would like to finish our project early enough to receive feedback, work on stretch goals, and make as many improvements as possible before the deadline.

With all these time constraints in mind, we emphasize how important it is to be aware of them in order to work effectively and deliver a successful product.

4.2.2 Power and Thermal Constraints

Power consumption and thermal management are essential considerations for SPUD's hardware design, especially since the project utilizes an FPGA as the core processing unit. The FPGA's power requirements depend on factors like clock speed, logic utilization, and external components connected to the system. To manage power consumption, efficient clocking techniques and low-power design practices will be used, aiming to keep the system within acceptable power limits. Thermal constraints also play a crucial role; excessive heat can damage components and reduce system reliability. Therefore, proper thermal dissipation methods must be integrated, such as ensuring adequate airflow and considering heat sink placement for critical components like the FPGA. These constraints guide the optimization of power usage and thermal management to maintain system performance and reliability.

4.2.3 Economic and Sustainability Constraints

Economic constraints dictate that the SPUD project must be developed within a limited budget, which affects the selection of components, materials, and tools used throughout the design process. Given the financial limitations, cost-effective solutions must be chosen without compromising the project's functionality or performance. This includes selecting components with an optimal cost-to-performance ratio and avoiding unnecessary over-specification of materials. Additionally, sustainability is a consideration in the project's design, where components are selected based on environmental impact and ease of disposal. Choosing energy-efficient parts and recyclable materials contributes to the project's sustainability goals. Efforts to minimize waste during the manufacturing process, and to optimize the overall lifespan of components, will also be considered to reduce the environmental footprint of the project. These constraints ensure that SPUD adheres to both economic feasibility and sustainable engineering practices.

4.2.4 Ethical and Security Constraints

SPUD is more than just a technical challenge—it's a reflection of who we are as students and future engineers. While our main goal is to implement a functioning CPU on an FPGA and run a game on it, we must also consider the broader impact of the work we're doing. Ethical and security constraints shape how our work can be used, understood, and trusted.

From an ethical standpoint, we're mindful of where our tools and components come from. Even though SPUD is an academic project and we're working on a

small scale, that doesn't mean we get to ignore things like supply chain ethics or environmental sustainability. We do our best to use open-source tools and platforms, not just because they're accessible and cost-effective, but because they demonstrate transparency. Relying open-sourced resources allows us to avoid being locked into proprietary solutions that might raise ethical concerns later down the line.

We also recognize that just because something is open-source doesn't mean it's automatically free to use however we want. Licensing and copyright still matter. As we integrate existing cores or use reference designs, we take care to respect the terms of those licenses, give proper credit, and avoid any misuse of copyrighted material. We've learned how important it is to double-check licenses before using code, especially when publishing our own work or sharing files publicly. It's easy to overlook small things like including a license in our repository or citing where a snippet of code came from, but those small details are part of responsible development. We want to deliver a quality product after all.

Environmental impact is another area we must look into. While we're not mass-producing hardware, we still want to try to reduce as much waste as possible. Whether that's reusing boards, avoiding unnecessary prints, or planning our builds so we don't burn through any components. Choosing energy-efficient designs when possible and being conscious of waste reflect our commitment to sustainability.

On the security side, we acknowledge that even small projects can be vulnerable if one's not careful. SPUD may not necessarily be handling sensitive data or connecting to networks, but regarding our CPU we must pay attention to things like memory access, privilege levels, and I/O control. We want to avoid security pitfalls like improper instruction decoding or accidentally exposing a bus interface to unexpected inputs. These are subtle issues that could be overlooked if we only focused on making things "work," but our goal is to make things work right.

We also know that just because SPUD is meant for educational purposes now doesn't mean someone won't try to repurpose it later. We want to make sure that our design cannot be easily turned into something harmful—like a system used to invade privacy or exploit users. We're not building anything with those goals in mind, and we want that to be clear in our documentation and in how we talk about the project. Ethical design is about setting intentions and being aware of downstream effects, even if we can't control everything.

Lastly, we recognize that ethical behavior starts with us! We've worked hard to split responsibilities fairly, support each other when we hit roadblocks, and give credit where it's due. That means citing sources, collaborating honestly, and being open about what we understand (and what we don't).

4.2.5 Safety Constraints

Safety is a key consideration in the design of SPUD's hardware. Given that the project involves the development of a custom PCB and uses an FPGA, ensuring the safe operation of the system under varying conditions is essential. This includes electrical safety, such as ensuring proper voltage regulation, current protection, and short-circuit protection to prevent component failure or overheating. The PCB layout and the placement of components will also follow safety guidelines to reduce the risk of electrical hazards. Moreover, ensuring that the system functions reliably in real-world conditions without causing harm to users or operators is essential. Components like the FPGA, which could potentially generate excess heat, will be evaluated for thermal management solutions to prevent burns or damage to surrounding components.

Similarly, when handling equipment for soldering, we must meet any safety guidelines presented to us in the lab to ensure that we protect ourselves from harm. Wearing protective eyewear and potentially, gloves, will allow us to solder without worry of burning ourselves or others around us. Seeing as the soldering iron will be close to 400 degrees celsius, reducing the risk of injury is crucial. When utilizing the reflow oven in the university laboratory, we must do so with authorized personnel in the room as the risk of injury or equipment malfunction is much greater when compared to a soldering iron. With soldering in general, we need to take our time and triple check our joints. Not being cautious when soldering could lead to potential short circuits in our design, thus leading to a potentially dangerous electrical shock. All aspects of the design must meet standard safety regulations to minimize the risk of injury or damage to users and to comply with relevant safety certifications.

4.2.6 Manufacturability Constraints

The manufacturability of the SPUD project must align with practical constraints, ensuring that the design is feasible for production within the given budget and timeline. This includes choosing components that are readily available and cost-effective, while adhering to the limitations of the chosen PCB manufacturer. For example, the PCB design must accommodate the minimum trace width and via size specifications of the manufacturer, as well as other factors like component size restrictions (e.g., components no smaller than 0802) as

indicated by Dr. Weeks. Additionally, ensuring that the assembly process is straightforward and does not require specialized tools or equipment will reduce manufacturing costs and increase the likelihood of a smooth production process. The design will also consider the ease of component placement and soldering during assembly to ensure high yield rates and minimize defects. In terms of testing and quality control, the design must include provisions for easy testing of functionality to identify and address issues quickly during the manufacturing process. These manufacturability constraints ensure that the SPUD hardware can be produced efficiently and at a reasonable cost while maintaining high quality standards.

Chapter 5: AI Resources

In today's world, the use of AI resources is far more popular than it was years ago. In recent years, it's become normal for students to utilize AI when learning different material or studying because of how simple it may seem. This chapter explores the AI resources utilized by SPUD.

Unfortunately, students tend to overlook the limitations of AI and assume that every bit of information being given to them is correct. In this chapter, we'll talk about some popular AI platforms and how each one may or may not be used to our advantage/disadvantage. To do this, we'll utilize case studies, where we'll ask the AI platforms the same questions about our project and compare the answers given to our own research to see if what AI is saying would actually benefit our project or not.

At the end of this chapter, we'll have demonstrated our full understanding of each AI's platform pros and cons and we'll be able to determine whether or not we could use AI to assist us in a proper completion of SPUD. Because of the nature of Senior Design, we believe that regardless of AI's functionality, we will learn best through hard work and our failures/successes when completing our project.

5.1.1 ChatGPT

ChatGPT is the most popular of the AI platforms and is being improved every single day. That being said, it is still far from perfect and issues can arise when trying to get factual information from ChatGPT. A problem with most AI platforms is that they are programmed to recognize patterns rather than facts, in turn, this causes ChatGPT to spit out information it believes is most likely to be correct rather than something that is 100% accurate. This is why it can potentially feed you outdated information or do math partially incorrectly. Because of this, we must be cautious when considering using an answer from ChatGPT for our project.

5.1.1.1 Case Study

Question #1:

"I'm designing a simple RISC-V 32bit CPU using an FPGA and system verilog. What ISA should I start researching with compiler support?"

Answer:

“For designing a simple 32-bit RISC-V CPU using an FPGA and SystemVerilog, you should start by researching the RV32I base instruction set architecture (ISA). RV32I is the fundamental 32-bit integer instruction set and serves as the essential foundation for all 32-bit RISC-V implementations.

Since you're looking for compiler support, the GNU Compiler Collection (GCC) provides robust support for RV32I. The GNU toolchain, including GCC, supports RV32I and various extensions, making it straightforward to compile software for your design. Specifically, the GNU toolchain supports compiling code targeting RV32I and its various extensions, giving you a reliable software toolchain for your FPGA-based CPU.

After mastering RV32I, you can consider adding optional standard extensions such as:

- *M (Integer Multiply/Divide)*
- *A (Atomic instructions)*
- *C (Compressed instructions)*

These extensions are well-supported by existing compilers (like GCC), and they will significantly enhance your CPU's capabilities without overly complicating your initial design.

In summary, begin with the RV32I base ISA due to its simplicity, excellent compiler support through GCC, and abundant documentation. Subsequently, add extensions incrementally as needed.”

Question #2:

“I'm designing a simple RISCV 32bit CPU using an FPGA and system verilog. We are adding a separate hardware driver for an 64x64 LED Matrix to work alongside the CPU? What are some methods to allow the CPU to control the RGB values?”

Answer:

“To enable a RISC-V CPU to control the RGB values of a 64x64 LED matrix via a hardware driver, consider these methods:

1. Memory-Mapped I/O:

- *Allocate a section of the CPU's address space for the LED matrix driver.*
- *Use a framebuffer in memory where each pixel is represented by RGB values (e.g., 24 bits per pixel). The CPU writes RGB data to this framebuffer, and the FPGA reads it to update the matrix.*

2. Command-Based Protocol:

- *Define a set of commands that the CPU sends to the FPGA via a bus (e.g., AXI or SPI). Commands can include setting individual pixel colors, updating regions, or refreshing the display. This reduces the CPU's workload for large updates.*

3. Shared Framebuffer with DMA:

- *Use Direct Memory Access (DMA) to transfer data from the CPU's framebuffer to the FPGA. This approach minimizes CPU intervention and allows high-speed updates.*

4. PWM Control in FPGA:

- *Implement PWM logic in the FPGA for color intensity control since most LED panels lack built-in PWM. The CPU sends raw RGB data, and the FPGA handles brightness modulation.*

These methods balance simplicity, performance, and resource usage depending on your system's requirements.”

5.1.1.2 Limitations

The above case studies were done using GPT-4 Omni, their latest model. ChatGPT gave us a good suggestion as this is actually the ISA that we are using for SPUD and had planned to use originally. While it is missing some minor details such as the instruction count, it points us in the right direction in terms of ISA and could've been a useful answer if we had not already decided our ISA for SPUD.

5.1.1.3 Pros/Cons

Pros:

- *Fast response*
- *Suggested good ISA, pointing us in the right direction*

Cons:

- Didn't give certain important details unless asked to

5.1.2 Claude

5.1.2.1 Case Study

Question #1:

"I'm designing a simple RISC-V 32bit CPU using an FPGA and system verilog. What ISA should I start researching with compiler support?"

Answer:

*"For designing a simple 32-bit RISC-V CPU using an FPGA and SystemVerilog, you should start with the **RV32I** base instruction set architecture. This is the fundamental 32-bit integer instruction set that serves as the foundation for all 32-bit RISC-V implementations.*

RV32I is an excellent starting point because:

- *It's the simplest complete RISC-V base ISA with only 37 base instructions*
- *It has full compiler support through LLVM and GCC toolchains*
- *It's well-documented and widely implemented in educational and small FPGA projects*

For compiler support, both LLVM and GCC fully support RV32I. Many Linux distributions now include RISC-V tools, making it easy to set up your development environment.

After implementing the base RV32I ISA, you can consider adding standard extensions incrementally:

- *M: Integer Multiply/Divide*
- *A: Atomic instructions*
- *C: Compressed instructions (reduces code size)*

These extensions are commonly implemented together, as seen in projects like PicoRV32, which can be configured as RV32E, RV32I, RV32IC, RV32IM, or RV32IMC.

For practical implementation examples, you might want to examine open-source projects like:

- *PicoRV32, a size-optimized RISC-V CPU that supports various configurations*
- *The 32-bit RISC-V implementation for FPGAs on GitHub by ShaheerSajid, which features a 4-stage pipeline and supports integer, multiply, and floating-point instructions*
- *The "Pequeno" RISC-V CPU tutorial series, which walks through building a 32-bit RV32I CPU from scratch*

These resources will provide valuable insights into implementing a RISC-V CPU on an FPGA using SystemVerilog."

Question #2

"I'm designing a simple RISC-V 32bit CPU using an FPGA and system verilog. We are adding a separate hardware driver for an 64x64 LED Matrix to work alongside the CPU? What are some methods to allow the CPU to control the RGB values?"

Answer:

"To allow your RISC-V CPU to control the RGB values of a 64x64 LED matrix, consider the following approaches:

1. Memory-Mapped Framebuffer:

- *Implement a framebuffer in memory where each pixel is represented by its RGB values (e.g., 24 bits per pixel).*
- *The CPU writes pixel data directly to this framebuffer, and the FPGA hardware driver reads it to update the LED matrix. This method is simple and efficient for static or moderately dynamic displays.*

2. Command-Based Interface:

- *Create a command protocol where the CPU sends instructions (e.g., "set pixel", "update row") to the FPGA driver via a bus like AXI or UART.*
- *This reduces memory usage but may increase latency for complex updates.*

3. DMA-Driven Updates:

- *Use Direct Memory Access (DMA) to transfer data from the CPU's framebuffer to the FPGA autonomously.*
- *This minimizes CPU involvement and supports high-speed updates for dynamic content.*

4. PWM and Color Modulation in FPGA:

- *Offload brightness control and color modulation to the FPGA using PWM logic. The CPU sends raw RGB data, while the FPGA handles timing and intensity adjustments.*

These methods balance simplicity, performance, and resource usage based on your design constraints."

5.1.2.2 Limitations

The above case studies were done using Clade 3.7 Sonnet. Question 1 perfectly demonstrates the good and bad with using Claude. The good thing is that it immediately suggests starting with the RV32I base instruction set. The problem lies when Claude begins to talk about the reasons to use RV32I, it states that RV32I is a 37 base instruction ISA, but it's actually a 47 base instruction ISA as stated by Digikey. This is a prime example as to why researching with Claude can be an issue as we were fed false information and had no reason to believe otherwise unless we checked.

5.1.2.3 Pros/Cons

Pros:

- *Fast response*
- *Gave a good general answer*

Cons:

- *Answer had some unfactual information*

5.1.3 Deepseek

5.1.3.1 Case Study

Question #1:

“I'm designing a simple RISC-V 32bit CPU using an FPGA and system verilog. What ISA should I start researching with compiler support?”

Answer:

“For designing a simple RISC-V 32-bit CPU with SystemVerilog and FPGA implementation, start with the RV32I base instruction set architecture due to its simplicity, compiler support, and foundational role in the RISC-V ecosystem. Here's a structured approach:

The RV32I ISA (32-bit integer base set) provides:

- *47 core instructions across six formats (R/I/S/B/U/J)*
- *32 general-purpose registers with a load-store architecture*
- *Fixed-length 32-bit instructions for simplified decoding*
- *Full compiler support via GCC and LLVM toolchains*

Key advantages for FPGA implementation:

- *Minimal logic footprint (ideal for resource-constrained FPGAs)*
- *Proven 5-stage pipeline implementations in SystemVerilog/Verilog*
- *Direct compatibility with open-source toolchains like riscv-gnu-toolchain”*

Question #2:

“I'm designing a simple RISC-V 32bit CPU using an FPGA and system verilog. We are adding a separate hardware driver for an 64x64 LED Matrix to work alongside the CPU? What are some methods to allow the CPU to control the RGB values?”

Answer:

“To enable a RISC-V CPU to control a 64x64 RGB LED matrix via a separate hardware driver on an FPGA, consider these methods:

- 1. **Framebuffer Approach:** Implement a memory-mapped framebuffer where the CPU writes RGB data to a specific memory region. The FPGA hardware driver reads from this buffer and handles the display refresh timing independently.*
- 2. **Row-Based Data Organization:** Structure the framebuffer to match the row-by-row transmission format of the LED matrix, making it easier for the hardware driver to fetch and transmit data efficiently.*
- 3. **State Machine Controller:** Implement a simple state machine in the FPGA that handles the timing-critical aspects of the display (clocking data, latching, blanking) while accepting commands or data from the CPU.*
- 4. **Dual-Port RAM Interface:** Use dual-port RAM to allow the CPU and display controller to operate at different clock frequencies, avoiding clock domain crossing issues¹.*
- 5. **Split RGB Control:** Since RGB matrices typically display two rows simultaneously, organize your control logic to handle parallel data paths for multiple rows.*
- 6. **PWM Implementation:** Include PWM logic in the FPGA for color depth control, allowing the CPU to specify simple RGB values while the hardware handles brightness modulation.*

This approach offloads the timing-critical display refresh to dedicated hardware, leaving the CPU free to generate content.”

5.1.3.2 Limitations

The above case study was done using Deepseek R1. The Deepseek response was a combination of both GPT and Claude. It suggests the RV32I ISA once again and it actually states the correct facts that Claude had incorrectly stated.

The limitation is when Deepseek talks about the 6 formats of instructions available to us. Quick research suggests that RV32I has only (R/I/S/U), but it requires a more thorough search to discover that RV32I has (R/I/S/B/U/J) as Deepseek suggests. While Deepseek's answer isn't technically wrong, it shows how further research must be done to confirm anything that AI answers as factual. This can become an issue for us if we used Deepseek and believed every single answer automatically because research could prove it or suggest otherwise.

5.1.3.3 Pros/Cons

Pros:

- *Fast response*
- *Factual information in the context of this answer*
- *Suggests good ISA*

Cons:

- *Has potential to include unfactual information based on obscure sources*

Chapter 6: Hardware System Design

Project SPUD is a soft core processor that is designed to run RISC-V instruction. While the logic is in software the project is heavily reliant on hardware. Specifically we have an PGA unit to host the CPU logic and IO, an LED matrix display to show functionality, a PCB to level shift matrix signals, and also RAM, and Flash memory used directly in the pipeline of our processor. All of these components will work together to make our project come to life.

Each hardware component in our design was carefully researched and chosen to give us the best possible results given our contents. Our main goal on part choice relies on a balance of efficiency, ease of development, and reliability.

6.1 FPGA Design

The Arty S7-50 Spartan-7 FPGA serves as the foundation for the SPUD project, leveraging Xilinx's Spartan-7 XC7S50-CSGA324 chip with 52,160 logic cells, 65,200 flip-flops, and 256 MB DDR3L RAM. Its architecture balances density and power efficiency, offering a 450 MHz maximum clock frequency alongside 120 DSP slices for arithmetic acceleration. The board integrates 256 MB of DDR3L memory operating at 650 MT/s for high-bandwidth data access and 128 Mbit Quad-SPI Flash for non-volatile program storage. Hardware interfaces include USB-UART/JTAG for debugging, four Pmod ports for peripheral expansion, and Arduino-compatible headers for shield compatibility, making it adaptable for embedded CPU development.

The Arty S7-50 was chosen for its cost-performance equilibrium, priced at \$150–\$200 while providing resources sufficient for a custom RISC-V CPU implementation. Its 52k logic cells accommodate complex pipelined architectures, while the DDR3L memory subsystem addresses the game engine's demand for low-latency data transfers. The inclusion of Quad-SPI Flash enables persistent storage of compiled programs without external memory chips, critical for standalone operation. Vivado WebPACK's zero-cost licensing and MicroBlaze-compatible toolchain streamline soft-core development, offering synthesis, place-and-route, and in-system debugging capabilities essential for iterative CPU optimization.

Implementing the CPU involves partitioning the FPGA into compute, memory, and IO subsystems. The Spartan-7's block RAM is mapped to cache-like structures for register files and prefetch buffers, while DDR3L handles bulk data via AXI4 interfaces. Clock domain synchronization manages the 100 MHz board

oscillator and derived CPU clock, with phase-locked loops (PLLs) reducing jitter for timing-critical paths. Vivado's IP Integrator accelerates integration by instantiating verified peripherals like UART controllers and SPI interfaces, reducing validation overhead.

6.2 Lower Level Logic Components

Our S.P.U.D CPU design, as with all other logic designs, can be broken down into simpler hardware logic components, when combined, these components can simulate all possible digital circuits depending on their combination, order, amount, and connection logic. In this section, multiple vital components will be introduced: their design and purpose in our circuit will be discussed. These are the components that are just above logic gates in complexity, logic gates will not be covered as they are readily available both as basic syntax in the language and as readily available functions, therefore, there is no need to either design or implement them for our project, same goes for registers/flip flops/latches.

6.2.1 MUX

A multiplexer takes in multiple input signals, a control signal, and a single output signal, based on the control signal, one of the many input signals will be processed into the output. This can be used for many things, line selection for things like register read selections, operand selections, etc. A simple MUX design is done by defining N number of input signals, and K input selection signals which is just $\log_2(N)$, we can then use a (case) statement and map an input out of any N inputs to be selected for an output based on the K signal. We can create modules for MUXs (and DEMUXs) and to ease reusability in our project

6.2.2 DEMUX

A demux is the inverse of a multiplexer, it has a single input, a control signal, and multiple outputs, based on the select input signal, the input will be routed to the appropriate output signal. They are typically used for similar routing purposes as the MUX. The design of the DEMUX is similar to the MUX, an input signal line/register is define, a control signal with a bus size based on the N of output signal is also defined, and a (case) statement is used to route the input to an appropriate output based on the control signal.

6.2.3 Adders/subtractors

Adders and subtractors serve the purposes of adding / subtracting two (and possibly more) operands; this can be used for many crucial operations: arithmetic operations, PC incrementing and decrementing, address calculations, etc. A basic design of an adder simply ORs the operands, and uses an AND gate to check for overflows i.e. $1+1$ should produce 2 but just ORing 1 and 1 produces 1, therefore we do an overflow check, this circuit is sufficient for one bit and is called a full adder (a half adder does not take overflow into account), we then chain multiple of these full adders to create a multi-bit adder. for subtractors, we do a similar process, however, we invert the bits of negative numbers, and based on the subtraction process (1's or 2's complement) we further process the final product (for 2's complement we add one after adding) and the final product is ready.

6.2.4 Multipliers/dividers

Multipliers and divider are utilized in many operations, with some overlap with adders and subtractors, they perform multiplication and division operations on input operands and provide the appropriate output. Both circuits can be used for arithmetic operations, address manipulation, and matrix mathematics, which have many critical uses for high performance processing applications. Unlike addition and subtraction, multiplication and division brings up many complexities in the design. A common multiplier design is shifting the bits then adding to account for odd multiplicands, you can multiply any binary number easily by a factor of 2 just by shifting the bits x bits to the left depending on how much you are multiplying it with; however, the problem arises when you need to multiply by something that is not a factor of 2, i.e. 3 or 5: In this case we account for the difference by adding adding the difference between te nearest factor of 2 i.e. multiplying 3 by 3, we shift 3 to the left one time (thus multiplying it by 2) then use an adder to add the remaining three: this design is satisfactory for lower bit count numbers and simpler operation, but it might get costly performance wise the higher the bit count is for our number; another option in such a case would be to have a fixed array with a fixed set of values for each input combination, this is very fast and can perform the calculation in a cycle, but might be costly hardware and memory wise; both options are being considered for different cases. Dividers on the other hand are more complex to design, one option is to use a restoring divider, it simply subtracts the divisor from the current remainder (initially set to the dividend), if the result is positive, the quotient bit gets incremented by 1, and the process continues, if it is negative, the quotient is not incremented, the remainder gets the divisor added to it again to revert it from a negative (thus restoring it), and the result is saved;

the logic behind this design is simple and easy to implement, however, it might cost more operations and thus wasted processing power and time. For our project, we considered adding in a floating point operation to our system, but decided to not implement such a feature, therefore, division in our project is going to be very limited and it will only be integer division.

6.2.5 Decoders/Encoders

Decoders are another critical component to the operations of our CPU, they are used for many vital operations like instruction decode, enabling register file writes, and mapping outputs/inputs. Decoders work by having an N bit line input and 2^N outputs, for each combination for the N Input, there will be a mapped output that has one enabled bit, for example, 00 as an input would map to 0001 and conversely, 11 would map to 1000; this is useful for acting as a “flag” that signals the enablement of certain operations. Decoders are easy to implement using case statements or basic boolean algebra depending on the input/output bit size. Encoders act the opposite of decoders, taking in a larger 2^N input and encoding it into a N bit output; not many uses are available for encoders, but they can be useful for certain functions like setting a priority between a selection of signals or for simulating interrupts.

6.2.6 Program Counter

The program counter (PC) is a fundamental component in our project that holds the address of the next instruction to execute. In RISC-V, the PC is typically a 32-bit register that updates every clock cycle. By default, it increments by 4 to move to the next 32-bit instruction, but it can also be modified directly during control flow changes such as branch instructions (BEQ, BNE), or jump instructions (JAL, JALR). The PC is used to fetch instructions from memory by supplying the current address to the instruction memory block. When branching or jumping, the new PC value is calculated using the ALU or an address adder (for example, PC plus immediate), and a multiplexer selects whether to increment normally or load a new target address. Overall, the PC is critical for maintaining instruction flow, enabling sequential execution, and supporting conditional and unconditional jumps in program logic. The PC is implemented by defining a register that holds the PC value, and manipulating the register at every clock edge by utilizing a sensitivity list, we then account for constant address increments and add input logic for control signals to account for cases where the PC will jump or branch instead of just incrementing the address, an enable signal may be included to allow for further control by stalling the PC.

6.2.6 Control Unit

The control unit is responsible for interpreting each instruction and generating the appropriate control signals to guide the operation of the datapath components such as the ALU, register file, memory, and program counter. It decodes fields from the fetched instruction—like the opcode, funct3, and funct7—to determine the instruction type (e.g., arithmetic, branch, load/store, jump), and activates the correct execution units accordingly. In a typical implementation, the control unit uses combinational logic to match instruction bit patterns and produce enable signals, register indices, immediate values, and pipeline stall signals. It also handles register forwarding and hazard detection to ensure smooth instruction flow without conflicts. Thus the main purpose behind the control unit is general control of the overall unit and co-ordination between all other components and subsystems. The control unit uses multiple subsystems and has the following design flow: we design and implement a list containing instruction formats and their appropriate routings, we then decode the instructions that are received in the input, generate the appropriate control signals and send them to all the other subsystems. We can ease this process by granulating multiple components into sub-modules and reusing codes from other components (MUXs, decoders, etc.), we can then use case statements and sensitivity lists to update and route the appropriate subsystems based on the input.

6.2 ALU

ALU (Arithmetic Logic Unit) is a crucial component of our CPU design that performs additions, subtractions, increments, decrements, and even basic logic operations like ANDing, ORing, etc. The most common input and output signals in an ALU can be classified as follows: basic input signals, like operands, for arithmetic operations, addresses to be manipulated, input values to check for equality, etc. we then have control input signals, these could help specify where the output gets sent, which operation to use on the received operands, and whether to completely gate a section of the ALU (or the ALU as a whole) for clock gating purposes. Finally, we have the outputs, these could simply be the outputs from our operations, or additional flags that signal a condition has been met (equality, non equality, sign, etc.). Our ALU uses a 32 input register and a 32 output register, and follows the RISC-V standard ISA; the ALU processes the following instruction types: R-type instructions, I-Type instructions, branch instructions, and Load/Store instructions. All these instructions have an OP-code, which can be processed by a case statement in the form of a MUX/DEMUX or decoded using a decoder structure. After deriving the function

of the input instruction, the ALU further processes the input based on the opcode and supplies the intended output.

6.3 ART s7 Ram design

The Arty S7 50 features RAM capabilities, making it suitable for demanding embedded applications. At its core, the Arty S7 is equipped with 256MB of DDR3L RAM, connected via a 16-bit wide interface and operating at up to 325MHz (650 MT/s), which is ideal for high-speed data processing and large program storage. This memory is managed through a dedicated memory controller and physical layer interface, typically implemented using the Xilinx Memory Interface Generator (MIG) Wizard. The MIG core can be configured to provide either a native FIFO-style or AXI4 interface, allowing seamless integration with user logic and offering flexibility for various application requirements.

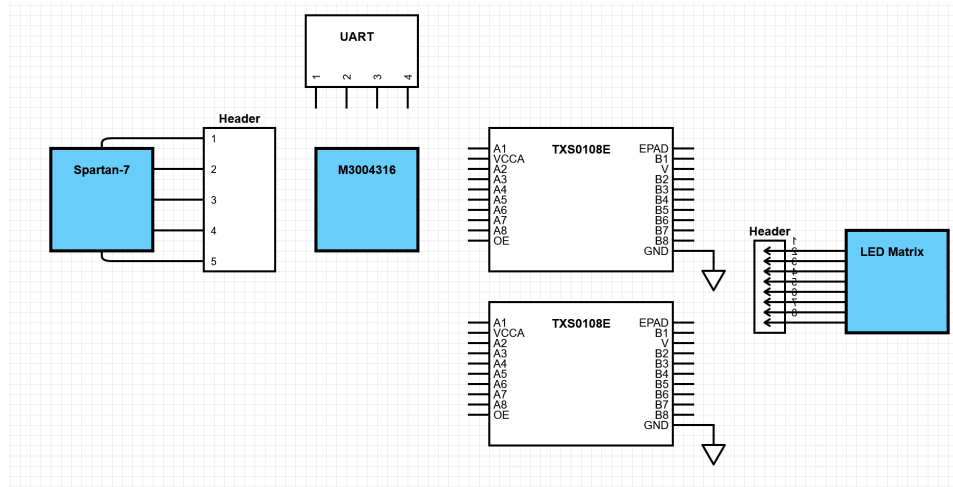
For custom logic and soft processor implementations, such as our RISC-V core, the Arty S7 also leverages the Spartan-7 FPGA's internal block RAM (BRAM). These BRAMs can be instantiated as true dual-port RAMs, supporting simultaneous read and write operations on separate clocks, which is especially useful for multi-core or real-time applications. The combination of high-capacity external DDR3L RAM and flexible internal BRAM enables the Arty S7 to efficiently handle complex memory architectures, supporting everything from instruction and data storage to high-speed video or signal processing tasks

6.4 PCB Design

When designing our PCB, we wanted to ensure that we had a clean layout for our components for not only routing, but for functionality. We started by creating a draft of the final PCB on paper to match our idea of the final design for our project. SPUDs final form is meant to be a stack of the PCB and FPGA with the LED matrix on top of it all. The PCB serves as a hat, it will plug into the top of the FPGA.

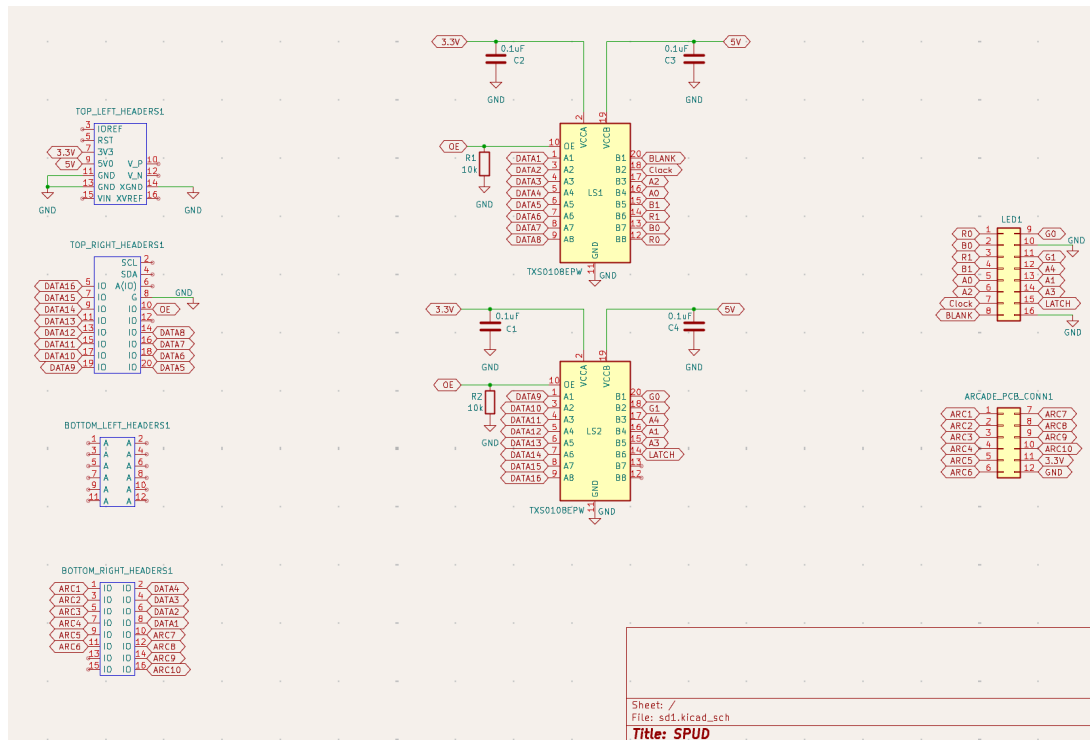
After sketching out a PCB, we created a rough draft schematic drawing using Digikey's free software, Scheme-it. Because of the simplicity of our PCB, the schematic wasn't difficult to create and it left a lot of room to add to later if needed.

Figure 6.1. High-level PCB Schematic. Created by authors.



Our PCB schematic is a simple one consisting of two TXS0108EPW level shifters, our Spartan-7 FPGA, the HUB75E interface for the LED matrix, and a connector for a second PCB. Most of the I/O pins on the Spartan-7 will be utilized and nearly every pin of the level shifters is crucial.

Figure 6.2. Detailed PCB Schematic. Created by authors.

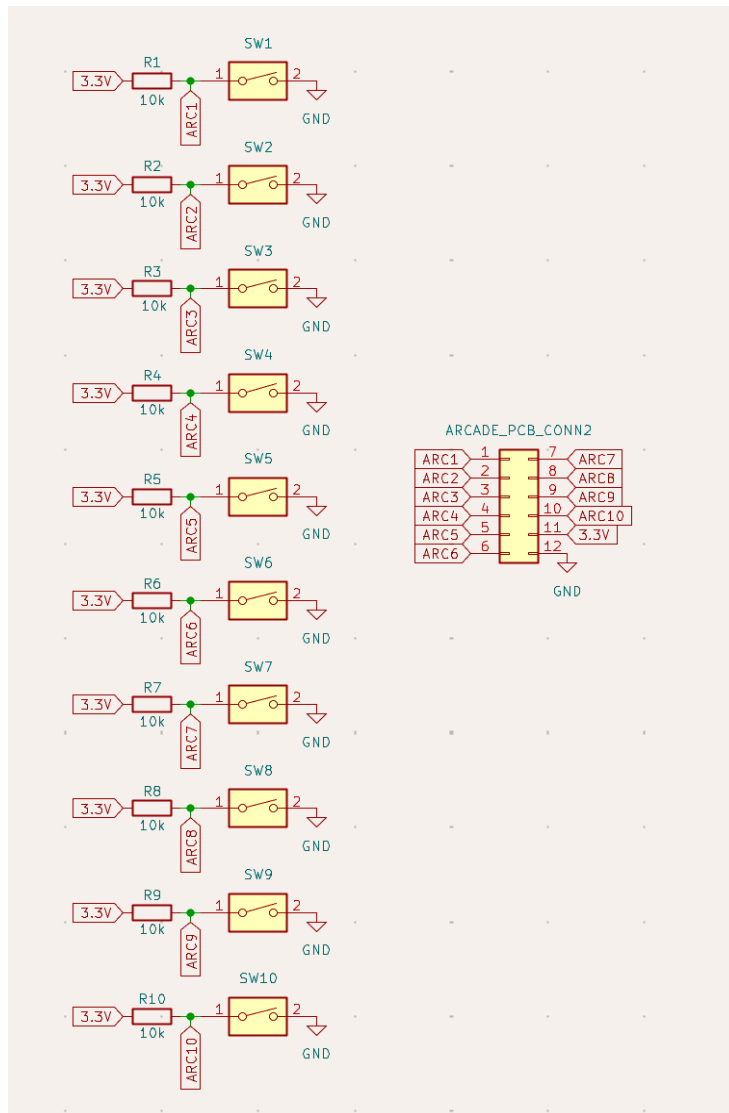


Spartan-7. The most crucial component of SPUD will be housed on our PCB. Nearly every pin on the headers on the FPGA will be utilized in some way for SPUD. There are 16 I/O pins transmitting data bidirectionally between the two TXS0108EPW level shifters through the A# pins. The Spartan-7 will supply 3.3V into each level shifter. The level shifters will then shift the 3.3V into 5V for the LED matrix. The enable (OE) pins of the shifters are both controlled by a single I/O pin on the Spartan-7. We found it unnecessary to utilize one pin for each shifter since both level shifters will be enabled at the same time. Attached to each enable pin is a 10k ohm pull-down resistor just to make sure that the pins start as disabled rather than on. We also have a separate connector for the second PCB containing simple arcade buttons. This PCB will act as a game controller to play games on SPUD.

TXS0108EPW. The level shifters are responsible for shifting voltage so that data can be passed between the Spartan-7 and our LED matrix. As mentioned, the shifters will take in data into the A# pins from the FPGA, convert the logic voltage to 5V, and pass the data into the HUB75E interface for the LED matrix via the B# pins. A 0.1uF is attached to both VCCA and VCCB of the level shifters to filter out noise and keep power stable between shifts. Without these level shifters, the communication between the FPGA and LED matrix would fail, so we took extra caution to ensure the wiring was correct.

A few design ideas needed to be addressed when we were creating the PCB layout. Of course we want our headers to be in easy to access locations without worry of a compact space. Our biggest consideration was deciding what kind of level shifters we wanted and how many. Originally we thought just one would be enough since we only wanted a simple 3.3V-5V conversion, but as we progressed, we realized that we might need a second level shifter. We took our time when considering how the level shifter datasheet wanted the wiring completed. Specifically, the capacitor values on the VCCA/B pins and the pull-down resistor value on the enable pin.

Figure 6.3. Arcade PCB Controller. Created by authors.



The arcade controller PCB is simply applying the 3.3V from the connector to 10 switches. There is a 10k ohm resistor on each switch that acts as a pull up for the signal.

6.5 LED Matrix

The LED matrix comprises two 32x64 panels put together. Driving a 32x64 RGB LED matrix requires precise control of row addressing, data shifting, and timing signals to ensure proper operation while preventing display artifacts such as flickering or ghosting. The matrix is arranged in a grid where rows are selected using 5 address lines (A, B, C, D, E), allowing selection of one out of 32 rows at a

time. These rows are divided into two interleaved banks: the upper bank (rows 0–31) and the lower bank (rows 32–63). Since only one row is actively driven at a time, the display is constantly refreshed through a multiplexing process to create a stable image.

To illuminate the correct pixels in the selected row, data for the 64 columns must be shifted into the matrix's internal shift registers. Each pixel is represented by three color channels: red, green, and blue. Since the matrix is split into two banks, data must be provided separately for the upper and lower banks using the signals R0, G0, B0 for the upper bank and R1, G1, B1 for the lower bank. The shifting process begins by setting the appropriate row address, followed by sending 64 clock pulses to shift pixel data into the shift registers. Once the data is in place, the Latch signal is toggled to transfer the shift register contents into the output buffers, making the pixel data visible on the matrix.

To prevent unwanted visual artifacts, the Blank signal is managed carefully during updates. Briefly disabling the output while updating the display helps avoid ghosting effects, ensuring that old and new data do not overlap incorrectly. Once the new row's data is latched, the Blank signal is re-enabled, allowing the row to be displayed correctly. This process is repeated rapidly for each of the 32 rows, refreshing the display at a high rate, typically 400 Hz or higher, to maintain a stable and flicker-free image.

The entire operation relies on three critical control signals: Clock, Latch, and Blank. The Clock signal shifts RGB data into the registers, ensuring that each column receives the correct pixel data. The Latch signal moves the shifted data from the registers to the active output buffers, making it visible on the matrix. Finally, the Blank signal ensures that visual artifacts are minimized during updates. Proper synchronization of these signals is essential for smooth operation, making LED matrix driving a balance of timing precision and data management.

Driving the 32×64 RGB LED matrix requires multiple critical design constraints, with timing being the first one. The display must refresh at a minimum of 400 Hz, meaning all 32 rows must be updated every 2.5 milliseconds, limiting row update time to $\leq 50 \mu\text{s}$. Since each row consists of 64 pixels, with three color channels (RGB) per pixel, 192 bits must be shifted per row. At a 60 Hz frame rate, this results in 368,640 bits per second, requiring a clock frequency of at least 12.8 MHz to meet timing constraints.

Memory management is also crucial, with the matrix requiring $64 \times 64 \times 3$ bytes = 12.2kBytes per frame. Double buffering is recommended to prevent visual

artifacts, ensuring smooth transitions by updating one buffer while displaying the other. Double buffering is the process of having a “back” frame ready while the current “front” frame is being displayed, once the current frame is done displaying, the back frame is swapped out immediately instead of waiting for another frame to be created; this process minimizes visual artifacts.

For color depth, PWM (Pulse Width Modulation) is required, as LEDs can only be fully on or off. To achieve 8-bit color depth, each pixel must undergo 256 brightness levels per frame, requiring 15,360 refresh cycles per second at 60 Hz. Efficient PWM control and synchronization are essential to maintaining accurate color representation while adhering to the matrix's high-speed refresh demands.

6.6 Flash Memory

Our CPU design will employ Quad-SPI Flash memory integrated into the Arty S7 FPGA board, providing 16Mb of non-volatile storage through a Micron MT25QL128ABA1EW9-0SIT chip. This serial NOR flash operates at 1.8V and supports quad IO modes, enabling four-bit parallel data transfers at speeds up to 108 MHz. Its primary role is dual-purpose: storing the FPGA configuration bitstream and housing the CPU's executable programs, leveraging the shared physical interface for cost efficiency.

During power-up, the Quad SPI Flash serves as the boot medium, delivering the FPGA configuration file to initialize the hardware fabric. Post configuration, the same memory stores compiled RISC-V programs accessible via memory-mapped addressing. Unlike von Neumann architectures, the FPGA's Harvard-style separation allows simultaneous instruction fetches and data access through dedicated SPI controllers. Execution occurs directly from flash via prefetch buffers, avoiding full RAM shadowing while maintaining non-volatility for embedded applications.

Communication uses a Quad SPI protocol with four bidirectional data lines, clock, and chip select. The FPGA implements an AXI Quad SPI IP core to manage command sequencing, addressing, and data transfers through a 32-bit memory-mapped interface. Critical timing parameters include 8 dummy cycles for read operations and 512-byte page program granularity. The interface supports execute in place mode for zero copy instruction fetching, though throughput limitations necessitate cache-like prefetch buffers to mitigate latency

Integrating Quad-SPI Flash introduces several critical constraints and drawbacks. The memory's serial access protocol creates inherent latency, as each instruction fetch requires multi-cycle transfers across the SPI bus. This

bottleneck has a direct impact on our CPU speed. Voltage compatibility further complicates the design, as the flash operates at 1.8V logic levels while the FPGA uses 3.3V, requiring level-shifting circuits to prevent signal integrity issues but is already handled for us as the FPGA is already designed to use this chip. These trade offs underscore the balance between development convenience and performance optimization.

6.7 Power Management

Power design can get complicated in the analog electronics field, which is why we're aiming to keep things as simple as possible. For our FPGA, we're using a 12V barrel jack supply, while our LED display will be powered by a 5V 4A supply. On our PCB, we use 5V to power the level-shifting ICs, which convert digital signals from 3.3V to the 5V required by the LED display.

Fortunately, our power design takes advantage of the onboard regulators built into each component. This means we only need a standard 120V AC supply from a wall outlet to power everything.

A potential improvement for our setup would be to use a single power supply capable of powering all the systems from one source. This would simplify power management even further. However, since we currently only require two power sources, we've decided not to invest in a more streamlined solution at this time.

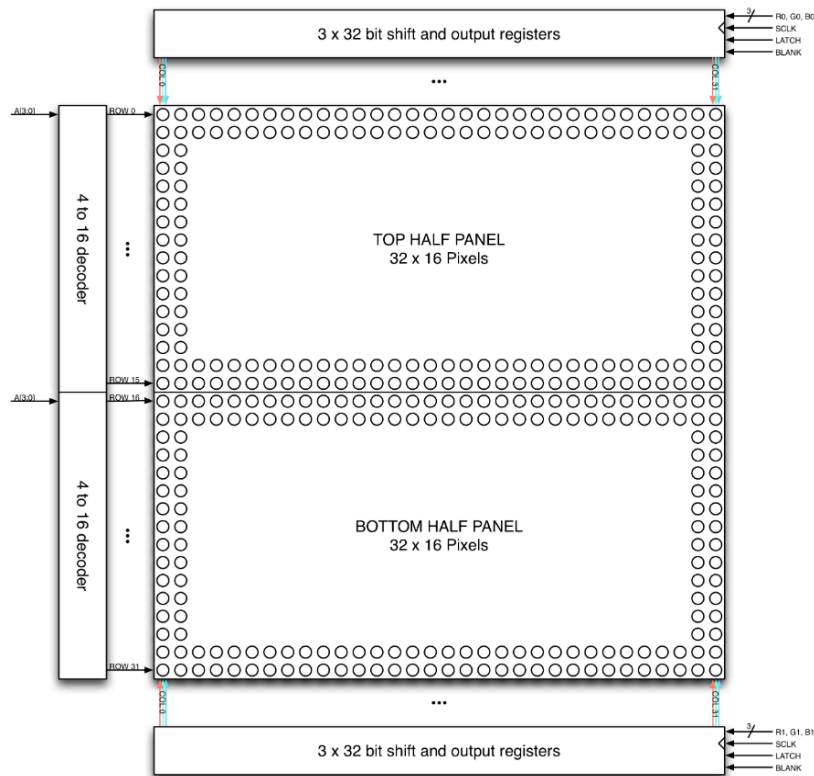
6.8 Conclusion

The hardware system design for Project SPUD encompasses a carefully selected array of components, each playing a crucial role in bringing the soft-core RISC-V processor to life. At the heart of the system lies the Arty S7-50 Spartan-7 FPGA, chosen for its balance of cost-effectiveness and performance capabilities. This FPGA provides the necessary logic cells, memory, and interfaces to implement the custom CPU architecture. Complementing the FPGA is a custom-designed PCB that serves as an interface between the FPGA and other components, particularly the LED matrix display.

The LED matrix, consisting of two 32x64 panels, serves as the primary output device for the project, requiring intricate timing and control mechanisms to ensure smooth, flicker-free operation. The system's memory architecture is bolstered by the inclusion of Quad-SPI Flash memory, which not only stores the FPGA configuration but also houses executable programs for the CPU. Power management is handled through a straightforward design utilizing separate

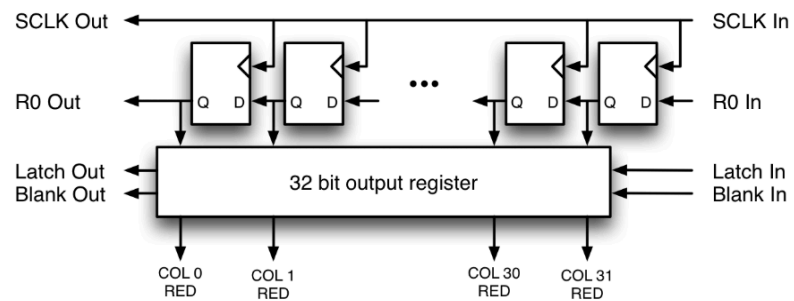
power supplies for the FPGA and LED display, with the PCB facilitating necessary voltage level conversions.

Figure 6.4. LED Matrix Schematic. Retrieved from bikerglen.com (n.d.).



The LED matrix has 16 pins in the back of the board, a ribbon connector will be used to route all 16 pins to the appropriate pins on the FPGA. A 4 pin connector will also be used to connect the board with an external power supply. The LED matrix design is showcased above.

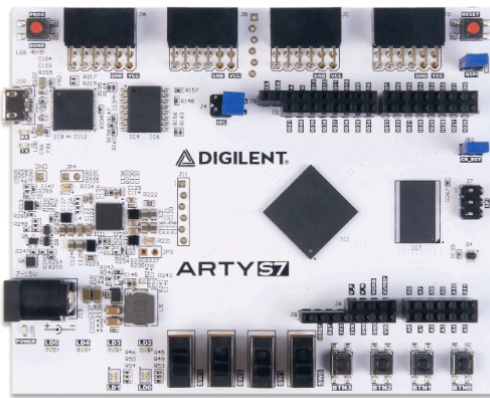
Figure 6.5. RGB Shift Buffers. Retrieved from bikerglen.com (n.d.).



We are also designing and implementing a shift register (diagram showcased above) to help facilitate loading the correct bits onto the LED matrix and buffering upcoming pictures while current ones are being displayed. It mainly uses a 32 Bit register and chained flip-flops to ensure the correct loading and timing of each image.

This hardware design enables the goals of the SPUD project by providing a flexible and capable platform for implementing and testing a custom RISC-V processor. The FPGA's reconfigurability allows for iterative development and optimization of the CPU architecture, while the LED matrix offers a visually engaging way to demonstrate the processor's functionality. The integration of Flash memory supports standalone operation, crucial for embedded applications, and the custom PCB ties these elements together into a cohesive system.

Figure 6.6. Arty S7-50: Spartan-7 FPGA. Retrieved from Digilent (n.d.).



This is our FPGA of choice. A Xilinx development board with the appropriate amount of LUTs/logic cells, we will mainly use the IO pins in the top row in the image above; we will use most of the pins to route to the 64x64 LED display and use the ground pins to ground both the signal pins going into the LED display and the ground of the PCB.

Looking to the future, several improvements and considerations could enhance the hardware development. A unified power supply system could streamline the power management, reducing complexity and potentially improving efficiency. Further optimization of the LED matrix driving circuitry could lead to enhanced display performance or reduced power consumption. Additionally, exploring more advanced FPGA models or incorporating dedicated hardware accelerators could open up possibilities for increased processing power or specialized

functionalities, pushing the boundaries of what the SPUD project can achieve in subsequent iterations.

Chapter 7: Software System Design

SPUD is a RISC-V processor entirely created in Verilog making it a soft core processor. The advantage to this is its versatility, customisation, and portability. Allowing us to create our own custom hardware interfaces, pipeline structure, custom instructions, optimisations, and unique features. In the industry this is advantageous as we can design before we get the silicon manufacturer saving money and time.

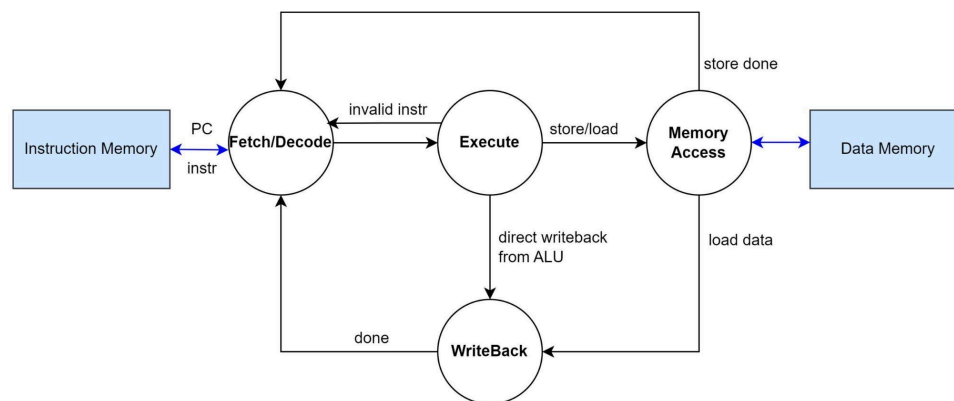
We are taking full advantage of the customization of this as we are adding our own hardware to support an LED matrix screen. By doing this we can make our processor as efficient as possible. While other display solutions need the CPU to constantly update the screen, while we have custom hardware to update the screen for us while our CPU can be doing other things.

Since we are designing our own CPU based on the RISC-V ISA we can also use the RISC-V compiler to write C code to generate the machine code needed to run custom programs. As long as we adhere to RV32IM we can easily generate software for verified hardware.

7.1 Core Processing Blocks

In our RISC-V processors design we leverage a modular design philosophy, enabling us to customize implementations through the RV32 ISA and optionally extend to other versions. Our implementation uses a 5-stage fetch, decode, execute, memory access, writeback pipeline.

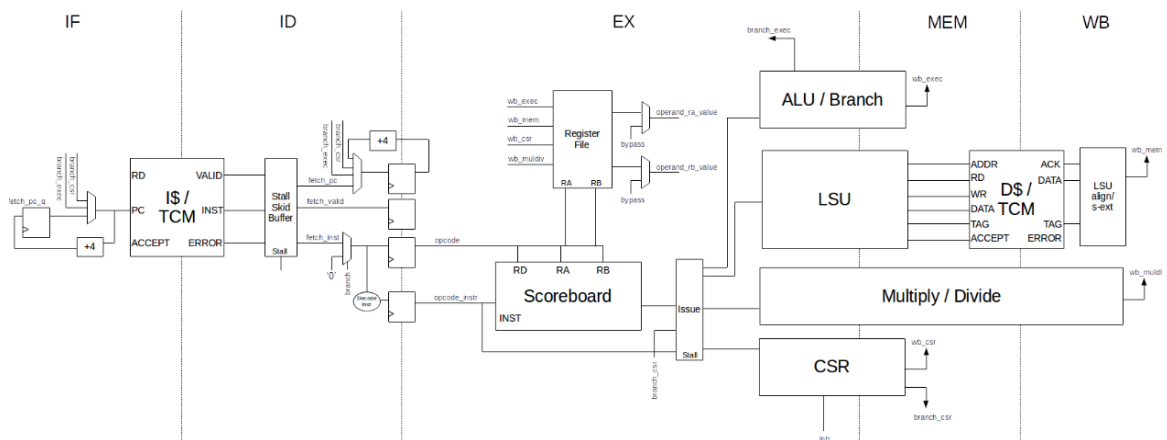
Figure 7.1 High-Level 5-Stage Structure. Retrieved from Chipmunk Logic (n.d.).



The figure above represents a high-level representation of that 5-stage CPU.

It contains Instruction Memory (IM) and Data Memory (DM), and the following stages: Fetch/Decode, Execute, Memory Access, and Write Back. This image is provided to explain at a high-level how each stage communicates with one another. The image below takes a deeper dive into a lower-level representation of our pipeline:

Figure 7.2 *Pipelined RISC-V Instruction Cycle. Retrieved from UltraEmbedded (n.d.)*



Each stage contains several modules that serve unique, important purposes. The architecture includes modules for fetching (IF), decoding (ID), and executing (EX) instructions, along with arithmetic and logic (ALU), multiplication/division (Multiply / Divide), and register storage (Register File). Memory operations are handled by the load/store unit (LSU). The control and status register module (CSR) handles system-level control.

7.2.1 Instruction Fetch Unit

Program Counter manages address sequencing through incremental updates and branch redirection. In RV32IM implementations it typically uses a 32-bit register that increments by 4 for sequential execution or jumps to computed addresses for branches. Branch prediction logic, when present, preemptively adjusts the PC for reduced pipeline stalls.

Instruction Memory stores encoded RISC-V instructions as a ROM or RAM block initialized via .hex files. FPGA implementations often map this to block RAM resources for single-cycle access latency. The Harvard architecture separation from data memory prevents structural hazards which is the approach we will be leveraging.

Branch Prediction reduces pipeline bubbles by guessing branch outcomes. Static predictors assume backward branches as taken, while dynamic variants use history buffers. RV32IM's fixed 32-bit instructions simplify target address calculation.

7.2.2 Decode Stage

Control Unit decodes opcodes into 10+ control signals using finite-state machines. For RV32IM, it interprets 7-bit opcode fields to manage ALU operations, memory access types, and register write enable. Combinational logic ensures single-cycle decode latency.

Register File provides 32x32-bit storage with dual read ports for simultaneous rs1/rs2 access and one write port for rd updates. FPGA implementations use flip-flop arrays or distributed RAM, with bypass logic resolving read-after-write hazards within the same cycle.

7.2.3 Execution Unit

ALU Control translates instruction field signals into 4-bit operation codes. For RV32IM, this handles up to 40 plus operations including signed and unsigned variants. Lookup tables map instruction subsets to ALU functions like ADD, SUB, and SLT.

Branch Comparator evaluates conditional jumps by comparing register values via XOR gates and zero detection. For BEQ, BNE, and BLT instructions, it feeds results to the PC's branch logic while adhering to RISC-V's non-flag-based design.

7.2.4 Memory Access

Data Memory Interface handles load and store operations with byte and halfword masking using write enable signals. FPGA block RAMs implement this with byte-addressable ports, supporting RISC-V's memory model through aligned accesses by default.

Memory Alignment Unit optionally handles unaligned accesses via multi-cycle operations or exception generation. While RV32IM mandates alignment, some implementations add this for software compatibility using shift-and-merge logic.

7.2.5 Writeback Stage

Result Multiplexer routes data from ALU outputs, memory read ports, or PC+4 to the register file. For RV32IM load instructions (LW), it delays writeback by one cycle to align with memory fetch latency in pipelined designs.

Hazard Detection stalls pipelines when data dependencies occur between instructions. It monitors register read/write addresses and injects NOPs during load-use scenarios, critical for maintaining RV32IM's in-order execution correctness.

7.3 Matrix Display Driver

The driver for a 64×64 RGB LED matrix on an Artix-7 FPGA is built around several critical software components that handle the timing, data shifting, and pixel management required to display the image. One of the most important parts of the driver is the memory buffering system, which stores the pixel data that will be sent to the LEDs. The driver typically uses double buffering to ensure smooth transitions between frames. In this system, two memory buffers are maintained: one holds the current frame being displayed, while the other stores the new frame to be shown. This allows the FPGA to update the pixel data in the back buffer while the front buffer is being shown on the matrix, avoiding flickering or tearing when switching frames.

The next key component is the row selection mechanism. Since the matrix is split into two interleaved banks (Bank 0 and Bank 1), the driver must handle the interleaving of these rows to select which bank to update. Each of the 64 rows in the matrix corresponds to a set of RGB data, which is loaded into the shift registers. The FPGA uses the address lines (A–E) to select the active row, and once a row is selected, the driver shifts the RGB data into the shift registers using the clock signal. This shifting process is crucial for ensuring the correct pixel data is sent to the LEDs in sequence.

Another essential software component is the timing controller, which ensures the driver meets the required refresh rate and row update times. The FPGA must cycle through the 64 rows quickly to maintain a stable display, with the row update time needing to be less than 50 microseconds for a 60 Hz frame rate. To meet these timing constraints, the driver must generate accurate clock pulses and manage the latch signal to transfer the shifted data to the output buffers. The latch is pulsed to update the active row of LEDs, and the blank signal is used to momentarily disable the LEDs during the transition to prevent ghosting.

Color control is another critical aspect of the software driver. To achieve smooth color transitions and brightness levels, the driver implements PWM (Pulse Width Modulation). The FPGA generates precise PWM signals for each color channel (Red, Green, and Blue) of every pixel, adjusting the duty cycle to simulate varying brightness levels. This is especially important for controlling the depth of color (e.g., 8-bit color) and ensuring that the LEDs display a wide range of colors. PWM also ensures that the brightness of each LED can be adjusted dynamically, providing smoother gradients and transitions in the image.

Finally, the driver manages the overall flow of data to the display. This includes ensuring that the data for each row is shifted into the registers in a timely manner and that the buffers are swapped at the right time to maintain a continuous flow of images on the matrix. The FPGA's software must synchronize all of these components to ensure that the display updates properly at the required refresh rate, and that the user can interact with the system in real time without experiencing lag or visual artifacts. Through careful management of timing, memory, and control signals, the driver enables the FPGA to operate the LED matrix efficiently and provide a stable, high-quality display.

In the context of the driver for the 64×64 RGB LED matrix on an Artix-7 FPGA, the system requires several specific hardware components, registers, buses, and memory for efficient operation. The key registers involved include the address registers (5 bits wide for selecting rows), RGB data registers (1536 bits per row, or 192 bytes, for storing pixel colors), and PWM control registers (24 bits per pixel, or 3 bytes per pixel, to control brightness and color depth). These registers are essential for controlling the flow of data and ensuring the correct color is displayed at each pixel.

Regarding buses, the system uses a data bus of 1536 bits (192 bytes) to transfer the pixel data for each row of the matrix, as well as a PWM control bus of the same size to manage brightness and color control for each pixel. The address bus is 5 bits wide to select the row for data shifting, and the shift register bus also needs to accommodate 1536 bits per row to handle the data being shifted into the registers. Additionally, the clock signal and control signal lines (such as Blank and Latch) are used for synchronizing the display, with these typically being 1-bit lines.

7.4 RV32IM Toolchain

The RV32IM toolchain provides a critical bridge between software development and SPUD processor hardware, enabling the translation of high-level programming languages into machine-executable instructions tailored for the

RV32IM base integer instruction set. This ecosystem comprises a compiler, assembler, linker, and runtime libraries specifically configured for 32-bit RISC-V architectures, aligning with the processor's memory model, register operations, and fixed-length instruction format. Developers work with standard C/C++ code, which the toolchain first compiles into RV32IM assembly, a process that maps variables to the processor's 32 general-purpose registers and adheres to its load-store architecture.

Configuration flags during toolchain installation determine ISA compatibility, ensuring generated binaries exclude unsupported extensions like floating-point operations or compressed instructions. The linker then combines object files with minimal runtime libraries optimized for embedded environments, resolving memory addresses according to the processor's Harvard or Von Neumann memory model. For FPGA implementations, this toolchain integration allows direct compatibility with soft-core processors like PicoRV32 or NEORV32, as the generated machine code matches the exact timing and resource constraints of the synthesized hardware. Debugging and optimization leverage the toolchain's awareness of RV32IM's 32-entry register file and 32-bit instruction boundaries, enabling cycle-accurate simulations before deploying to physical hardware.

Figure 7.3 RISC-V Instruction Set, Retrieved by RISC-V (n.d.)

31	27	26	25	24	20	19	15	14	12	11	7	6	0			
funct7					rs2		rs1		funct3		rd		opcode		R-type	
imm[11:0]							rs1		funct3		rd		opcode		I-type	
imm[11:5]					rs2		rs1		funct3		imm[4:0]		opcode		S-type	
imm[12:10:5]					rs2		rs1		funct3		imm[4:1:11]		opcode		B-type	
					imm[31:12]							rd		opcode		U-type
					imm[20:10:11:19:12]							rd		opcode		J-type

RV32I Base Instruction Set

imm[31:12]										rd		0110111		LUI	
imm[31:12]										rd		0010111		AUIPC	
imm[20:10:11:19:12]										rd		1101111		JAL	
imm[11:0]					rs1		000		rd		1100111		JALR		
imm[12:10:5]					rs2		rs1		000		imm[4:1:11]		1100011		BEQ
imm[12:10:5]					rs2		rs1		001		imm[4:1:11]		1100011		BNE
imm[12:10:5]					rs2		rs1		100		imm[4:1:11]		1100011		BLT
imm[12:10:5]					rs2		rs1		101		imm[4:1:11]		1100011		BGE
imm[12:10:5]					rs2		rs1		110		imm[4:1:11]		1100011		BLTU
imm[12:10:5]					rs2		rs1		111		imm[4:1:11]		1100011		BGEU
imm[11:0]					rs1		000		rd		0000011		LB		
imm[11:0]					rs1		001		rd		0000011		LH		
imm[11:0]					rs1		010		rd		0000011		LW		
imm[11:0]					rs1		100		rd		0000011		LBU		
imm[11:0]					rs1		101		rd		0000011		LHU		
imm[11:5]					rs2		rs1		000		imm[4:0]		0100011		SB
imm[11:5]					rs2		rs1		001		imm[4:0]		0100011		SH
imm[11:5]					rs2		rs1		010		imm[4:0]		0100011		SW
imm[11:0]					rs1		000		rd		0010011		ADDI		
imm[11:0]					rs1		010		rd		0010011		SLTI		
imm[11:0]					rs1		011		rd		0010011		SLTIU		
imm[11:0]					rs1		100		rd		0010011		XORI		
imm[11:0]					rs1		110		rd		0010011		ORI		
imm[11:0]					rs1		111		rd		0010011		ANDI		
0000000					shamt		rs1		001		rd		0010011		SLLI
0000000					shamt		rs1		101		rd		0010011		SRLI
0100000					shamt		rs1		101		rd		0010011		SRAI
0000000					rs2		rs1		000		rd		0110011		ADD
0100000					rs2		rs1		000		rd		0110011		SUB
0000000					rs2		rs1		001		rd		0110011		SLL
0000000					rs2		rs1		010		rd		0110011		SLT
0000000					rs2		rs1		011		rd		0110011		SLTU
0000000					rs2		rs1		100		rd		0110011		XOR
0000000					rs2		rs1		101		rd		0110011		SRL
0100000					rs2		rs1		101		rd		0110011		SRA
0000000					rs2		rs1		110		rd		0110011		OR
0000000					rs2		rs1		111		rd		0110011		AND
fm		pred		succ		rs1		000		rd		0001111		FENCE	
1000		0011		0011		00000		000		00000		0001111		FENCE.TSO	
0000		0001		0000		00000		000		00000		0001111		PAUSE	

Above is the **RV32IM** 32 bit RISC-V instruction set that our CPU will support. The toolchain can also be configured to reserve RAM space through linker script modifications by shifting the .data section base address to avoid overlapping with the out RGB matrix. This technique enables our program to write to this reserved place in RAM which our screen driver will read from and update its image independently of the CPU.

7.5 Verilog Design

Verilog is a hardware description language which was standardized as an IEEE 1364, used to model and design digital circuits at varying levels of abstraction, from behavioural description to gate level implementation. Unlike traditional programming languages, verilog focuses on parallel execution and hardware signal propagation that work with timing, concurrency, and signal strength modeling.

Its syntax resembles C, featuring modules that encapsulate functionality through input/output ports and internal logic, making it ideal for FPGA and ASIC development. Verilog supports register-transfer level (RTL) design, enabling precise control over hardware behavior while remaining synthesizable for physical implementation. While also giving engineers a way to review the synthesis of their design.

Synthesis transforms Verilog RTL into a gate-level netlist by mapping HDL constructs to FPGA primitives such as lookup tables (LUTs), flip-flops, and block RAMs. This stage optimizes for performance (clock frequency), area (resource utilization), and power while resolving inferred structures like state machines or memory arrays. Modern synthesis tools perform technology-dependent optimizations, such as merging logic operations into DSP blocks or leveraging dedicated carry chains. The output netlist serves as input for implementation, where physical placement and routing occur.

Implementation consists of place-and-route (P&R), where the synthesized netlist is mapped to specific FPGA resources and interconnected via programmable routing. This stage ensures timing closure by meeting setup/hold constraints through iterative adjustments to logic placement and signal paths. Static timing analysis (STA) verifies critical path delays, with tools reporting slack values to guide optimization. For complex designs, implementation may involve floorplanning to manually allocate high-speed or noise-sensitive circuits to specific FPGA regions.

Bitstream generation converts the placed and routed design into a binary configuration file that programs the FPGA's configuration memory. This file specifies LUT truth tables, switch matrix connections, and I/O bank settings using vendor-specific formats like Xilinx's .bit or Intel's .sof. For our project specifically we will begin using the .bit format. The bitstream is typically loaded via JTAG for prototyping or stored in non-volatile memory for production devices. Post-configuration, in-circuit debugging tools like ChipScope or SignalTap enable real-time signal monitoring.

The complete FPGA development flow progresses through design entry, simulation, synthesis, implementation, and bitstream generation, often iterating between stages to resolve issues. Modern toolchains like Vivado or Quartus integrate these steps, automating constraint management, incremental compilation, and multi-processor optimization. While the process shares similarities with software compilation, the hardware-centric nature demands attention to parallel execution, timing margins, and physical resource limitations. Successful designs balance abstract RTL modeling with low-level awareness of FPGA architecture to achieve performance targets within resource budgets.

7.6 System Inputs

To bring our project to the next level we want to include interfaces to allow our core to update with real time input from a user. This is crucial to reaching our main objective to create a game running on our core. Our methodology includes using the RAM as an intermediate medium to save state encoded into bits. Our approach is to create a reserved space in the RAM memory that works to mirror the state of the inputs.

Specifically we are using buttons as inputs. This can be easily encoded into an array of bits representing the state of the buttons. Using a system where a one represents on and a zero represents an off state. We can pack many of these byte by byte. Since we will be using 10 bits for the button states we can pack them successfully into 4 bytes. This will stay with the 32 bit word size of our processor.

This protected memory will serve as read only so as to not cause collisions and race conditions. Our plan is to manage this memory is to utilizing a custom designed IO controller that will automatically update the RAM memory when updates are detected.

The technique for the output for controlling the LED matrix and similar outputs are similar to the input control. This will be the most frequently used driver. That will work alongside the CPU core.

On the software level we plan to drive the screen by writing to this reserved memory that directly maps to the pixels being displayed on the matrix.

7.7 Programming

So far we've discussed many sections about software that all lead up to this section. Putting together all the layers from hardware logic, computer architecture, instruction set architecture, compilers, and more build up to the developer experience. Essentially the reason humans have structured these logic gates is to allow developers to communicate with computers in a language they can both speak. Leading to an explosion in technology that drives the modern information age. From medical equipment that saves lives, to rocket ships that go beyond, to thin slaps in our pockets that keep us connected. Computers make our lives better in virtually every way. One day we hope to make an impact with our labor of love, but for now that means creating and implementing our own computer that does nothing but play games.

As the developers, we have full control of our computer. The last layer between the human language and computer language is the compiler. This important and complex process involves taking syntax text from a human and converting it into organized ones and zeros to be crunched by our processor. The abstraction has taken software engineering to a level that people make it their whole careers without ever understanding what stack or heap actually is. The advancements of software development have helped us take complexity for granted, which sounds negative, but was ultimately the whole point. To give the power measured in millions of instructions per second in our full control.

For project SPUD, our language of choice is C. Developed in the early 1970s by Dennis Ritchie, C has become one of the most influential programming languages, forming the basis for many modern languages and operating systems. The process of creating machine code from C involves a few steps. At a high level we take the syntax then optimize and generate assembly code, which is a slightly less readable, but a completely valid programming language. At a computer's core we can only do 4 things: read, write, and manipulate memory. Engineers have found ways to add structure to this chaos, called data structures. Which enables a programmer to use queues, stacks, lookup tables, and many more. Understanding these is crucial to successful development.

Pairing this with solid planning and architecture will make reliable, readable, and successful software.

Using the C language is a happy medium between low level and high level programming. Allowing us to easily manipulate memory without guard rails while also giving us full control. We plan to implement techniques previously discussed, to aid us in creating our program. Our goal is to create a simple render engine, to allow easy control of our output hardware. Specifically updating the screen and other lights. This will assist us in making simulations, interactive demos, and games.

Chapter 8: System Fabrication

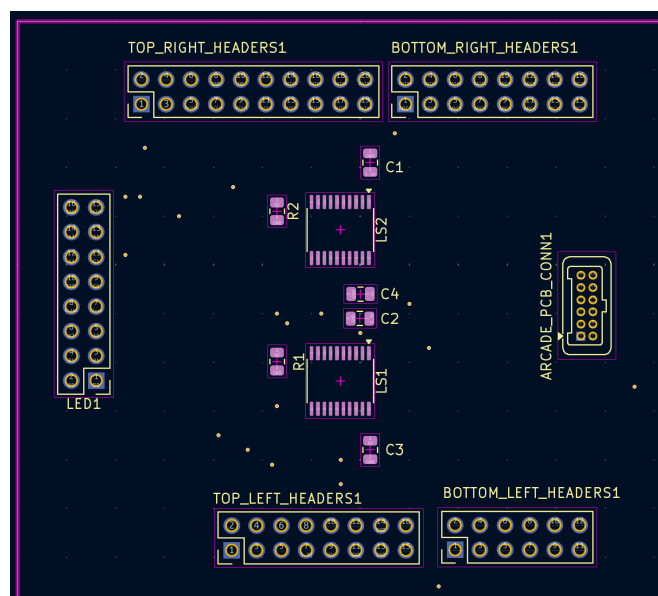
Designing and fabricating our own solutions gives us full customization and control over our project. The Arty S7 offers IO expansion capability that we are taking full advantage of to expand our control interfaces. Like previously disgusted we have a matrix breakout and an input control interface that both utilize IO. To bridge these to the FPGA we are creating our own PCB to route complex connections, save space, and prevent unnecessary debugging

8.1 PCB

Both PCBs were designed and routed in KiCad PCB design software. We created Net Classes for different trace widths for both our power traces and signal traces. Since we would have a maximum of 1A current, we used a wider trace width of 0.38mm for the power traces and kept the signal traces at a thinner 0.25mm.

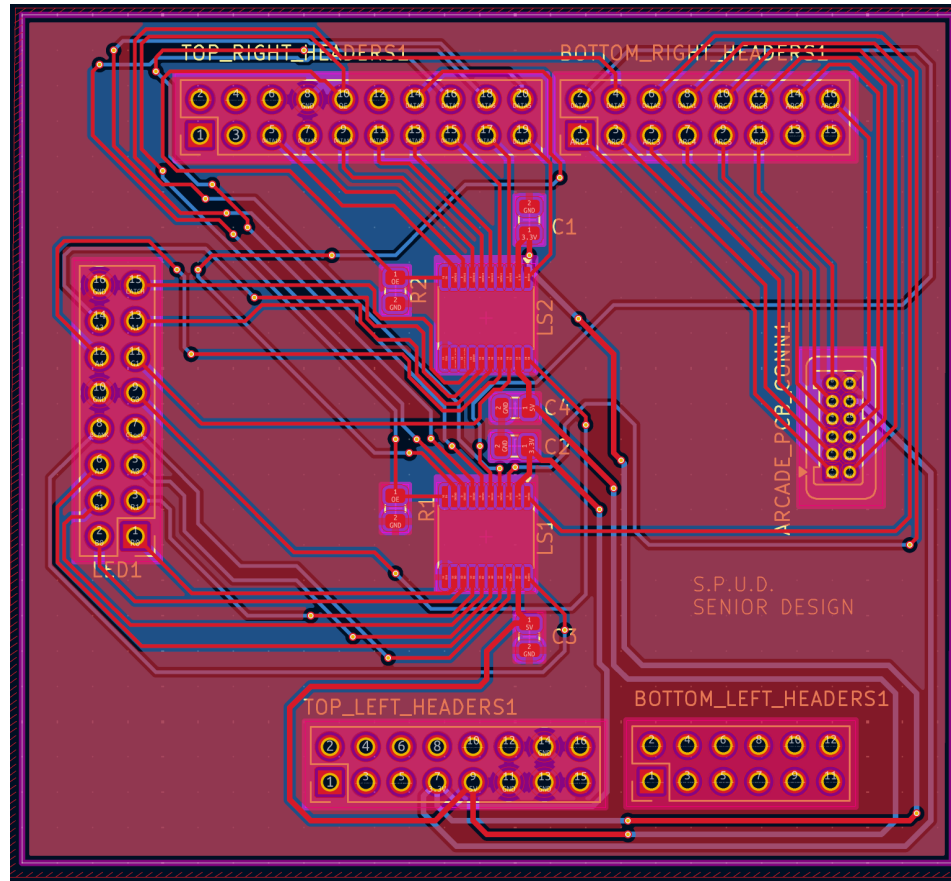
Our first PCB will be utilized as a “hat” for the Spartan-7, allowing for a compact design where our level shifters sit in the middle of the board, in between the FPGA headers. Originally, SPUD was meant to have a single PCB containing both the contents of the peripherals board and the arcade controller board, but we adjusted the peripherals board to feature a controller connector to our second PCB. This was done out of convenience for the user, as holding a long PCB from the base would be uncomfortable because of weight distribution.

Figure 8.1 S.P.U.D. Peripherals PCB. Created by authors.



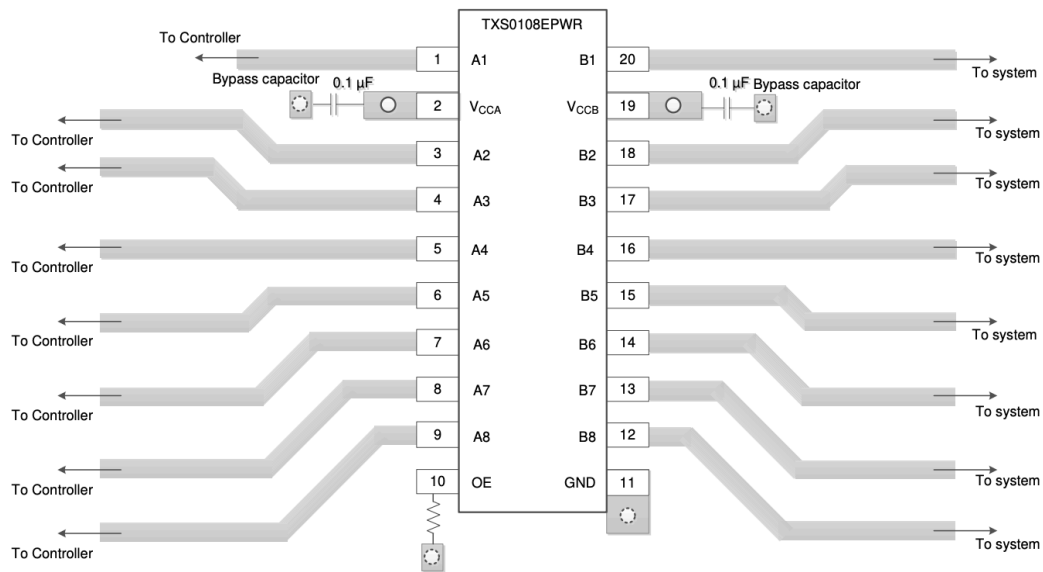
The routing for the boards was made a smooth process with the use of vias. Nearly every I/O pin on the Spartan-7 is utilized, so without vias, the routing would've become convoluted.

Figure 8.2 S.P.U.D. Peripherals PCB. Created by authors.



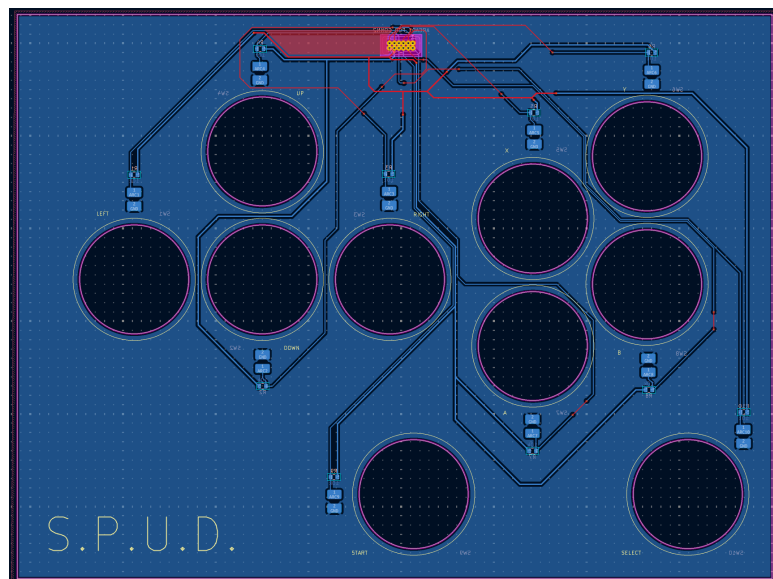
This is the fully routed peripheral board. Utilizing KiCads measuring tools and built-in 3D modeling software, we were able to accurately measure the distances between the FPGA headers to be sure that the hat would fit. The arcade PCB connector will take in 3.3V from the Spartan-7 and pass it onto the arcade controller PCB to power the buttons.

Figure 8.3 TXS0108EPWR Layout, Retrieved from Texas Instruments (n.d.).



The layout for the level shifters is a simple one. Requiring only a decoupling capacitor for each voltage conversion and a pull-down resistor on the enable pin. There are decoupling capacitors for both 3.3V and 5V on each level shifter, these are placed very close to their respective pins to reduce noise as SPUD operates. These shifters are placed close to the LED Matrix header to keep the routes relatively short, as we didn't want the 5V signal to degrade as it traveled to the matrix.

Figure 8.4 S.P.U.D. Controller PCB



The second PCB mimics an old school gaming console controller with a simple 10 button layout of 4 directions, 4 action buttons, and 2 buttons (start/select). The start and select buttons will be multi-faceted. Some features such as pausing a simulation or changing CPU speeds could be programmed into the start/select buttons. This controller is used for controlling the games within SPUD and is programmed directly through I/O pins on the Spartan-7. Attached to our peripheral PCB is a 2x6 connector that attaches directly to the arcade PCB, allowing for easy access to SPUDs controls.

Chapter 9: Prototype and System Testing

We take a layered approach to testing SPUD; beginning at the software level and eventually progressing towards a full system validation of hardware. The first section of this chapter involves the simulation of RISC-V programs using python scripts to verify that our assembly instructions perform the intended operations. We will then discuss our use of Verilog testbenches, where we test the modules individually and as a complete CPU to confirm functionality. We include our plans for hardware synthesis, resource usage examination, and PCB evaluation. Together, these testing stages give us confidence that SPUD is functionally sound and built to perform reliably in real-world use.

9.1 SD1 Preparation and Testing

The following tests have been conducted in order to stay on track and make sure that our workload is manageable by the fall semester. These early efforts focus on verifying core functionality, building confidence in our development flow, and identifying any bottlenecks or technical gaps ahead of time.

9.1.1 Program Simulation

To verify the behavior of RISC-V programs before turning them to hardware, we use Python-based program simulation. The `ultraembedded/riscv_soc` repository includes Python scripts that emulate the processor's core behavior. These scripts read a compiled RISC-V binary file and execute it instruction by instruction. They replicate how the CPU would interpret and execute each command in a real environment. This allows us to simulate full program execution—including arithmetic, memory access, and control flow—without needing an FPGA or any physical hardware yet.

This form of simulation is extremely useful for development and debugging. We get to run programs in a controlled, software-only environment, allowing us to quickly identify any issues with our assembly code, logic, or expected output. It also provides insight into how our CPU should behave when interpreting specific instructions. The Python simulator acts as a first checkpoint in our testing process, ensuring that our programs are functionally correct before moving on to more involved hardware-level simulations or synthesis. This verification in the early stages saves us time, reduces risk, and increases our confidence in the overall correctness of our software and processor interaction.

9.1.2 Testbench Simulation

Once a RISC-V program passes initial verification in software simulation, we move on to Verilog testbench simulation to evaluate how the processor design behaves at the signal level. The riscv_soc repository provides C++ testbenches that utilize Verilator, which is an open-source tool that converts Verilog into cycle-accurate C++ models. These testbenches simulate our processor's RTL design and allow us to run RISC-V binaries through the actual hardware logic, clock-by-clock, while observing waveforms, memory activity, and I/O behavior.

Many of these testbenches are also compatible with SystemC, a C++ library designed for system-level modeling and simulation. SystemC is especially useful for testing how the processor interacts with peripherals like memory-mapped I/O or timing-sensitive modules such as our LED matrix controller.

By combining Verilator with SystemC in C++ testbenches, we simulate our full system on chip in a modular way. This lets us verify individual components (like the instruction decode, ALU, or control unit) while also validating how the full CPU system operates as a whole. These testbenches are crucial for making sure that the source code functions properly and that there are no bugs that might not appear in higher-level software simulations. They allow us to detect timing issues or misaligned control signals.

Figure 9.1 Simulation Waveform

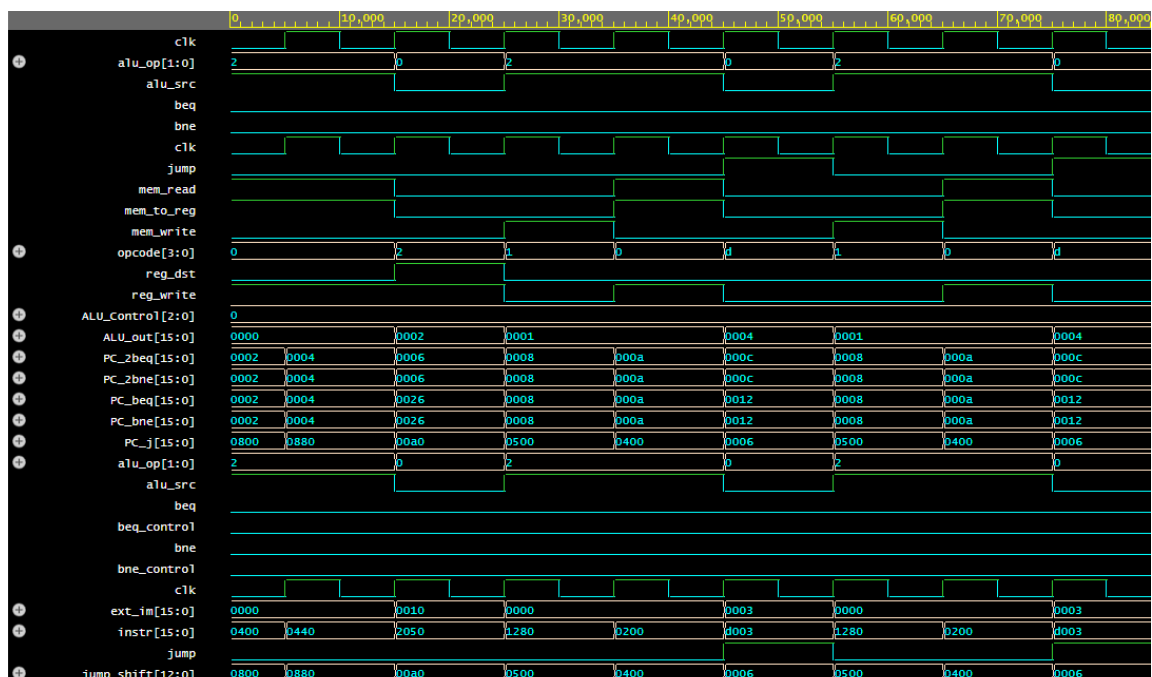


Figure 9.1 depicts a testbench simulation waveform that we ran on a RV32IM processor in EDA playground using Synopsys VCS. Note all the unique signals, including `clk` (clock pulse), `mem_read` (memory read), `mem_write` (memory write), `opcode`, `PC` (program counter), etc. When the clock starts, the processor reads in data from a file and stores that data into memory. The control unit breaks down the opcode and identifies the instruction type. It identifies whether the instruction pointed by the PC is `jump`, `beq`, `mem_read`, `mem_write`, `alu_src`, `reg_dst`, `mem_to_reg`, `reg_write`, and `bne` (as indicated in the image above).

9.1.3 Matrix Tests

To validate the functionality of the 64x64 RGB LED Matrix and its compatibility with the Arty S7-50 FPGA board, I followed a systematic prototyping and testing process. The goal was to ensure that the hardware connections were correct, the logic signals were properly interfaced, and the matrix could successfully display colors driven by the FPGA logic.

The 64x64 LED matrix panel required 5V logic levels for its control signals, whereas the Arty S7-50 operates at 3.3V. To bridge this voltage gap, we used two 8-channel bidirectional level shifters. The control and color signal lines—such as `R1`, `G1`, `B1`, `R2`, `G2`, `B2`, `A`, `B`, `C`, `D`, `E`, `CLK`, `LAT`, and `OE`—were routed from the FPGA I/O pins to the level shifters. The low-voltage side (`VA`) of the level shifters was connected to the 3.3V output of the FPGA board, and the high-voltage side (`VB`) was powered by a 5V line sourced from the same external supply that powered the LED matrix. The LED matrix was powered by a regulated 5V 4A power supply connected through the standard 4-pin connector. A total of 14 control and data lines were assigned from the FPGA to the matrix. These were mapped according to the pinout documentation provided in the Arty S7 master XDC file. Each signal was declared in the Vivado constraints file with both a physical pin location (`PACKAGE_PIN`) and an I/O standard (`IOSTANDARD LVCMOS33`). Special attention was given to ensure the clock input signal (`clk_100mhz`) was properly constrained, as missing constraints can prevent the bitstream from being generated.

To begin testing, I developed a simple Verilog module named `Matrix_Test.v`. This module created a slow, divided-down clock using a counter driven by the 100 MHz onboard clock. The purpose of this slow clock was to pulse the matrix's `CLK` line at a human-observable rate, confirming visual changes on the panel.

I hardcoded the `R1`, `G1`, `B1` signals to display red (`R1 = 1`, `G1 = 0`, `B1 = 0`) on the top half and white on the bottom half (`R2 = 1`, `G2 = 1`, `B2 = 1`). The row address

lines A–E were set to zero to target the first rows of the matrix. LAT and OE were set low to enable output. This setup allowed me to verify that: The LED matrix was receiving data, the level shifters were correctly translating logic levels, the clock signal was active and correctly latched.

The display correctly rendered basic static colors on the expected rows. Upon generating the bitstream and programming the FPGA, the matrix displayed the expected red and white color output, confirming successful operation. The Light was flickering at an expected rate and there was no unexpected behavior. There were slight issues with coloring and connectivity due to the connection between the FPGA, level shifters, and the display being routed through jumper cables and a breadboard, forcing us to physically hold the cables in place in order to ensure the display worked and the colors displayed correctly.

Due to some issues with how loose the wiring is due to the nature of jumper cables, there were slight issues with the displayed lines, however, the display otherwise worked as intended and is more than capable of being demonstrable during the prototype.

9.2 SD2 Testing Plan

The following tests were conducted during Senior Design 2, following the fabrication of our PCB. To maintain an organized approach and allow for sufficient time to add or tweak existing tests to improve the functionality and reliability of our system, we implemented the following tests.

9.2.1 Hardware Synthesis

After testbench verification, we conducted hardware synthesis, which involves translating the CPU source code into a gate-level netlist compatible with the target FPGA. This step maps the logical components, such as adders, multiplexers, and registers, onto the physical logic blocks available on the board. The synthesis process also simultaneously optimizes several factors like area, speed, and power.

Using Vivado we analyzed the synthesized design to confirm that all modules are correctly mapped and that no logic errors were introduced. Key synthesis metrics like logic utilization, timing paths, and critical delays are reviewed to ensure the design can meet real-world constraints like the 25 MHz clock frequency in our specs.

Successful synthesis guarantees that our RTL description is not only functionally correct but also realizable on hardware. This was a huge milestone because it marks the point where the processor design moves from simulation into physical implementation. Once complete, we proceeded to place-and-route and generate a bitstream.

9.2.2 MIPS Processor Performance

Once the design had been successfully synthesized, we evaluated the MIPS performance of the processor. The SPUD processor achieves 18.2 million instructions per second (MIPS) based on direct measurements taken with an oscilloscope. To obtain this value, the design includes a flag signal that the CPU toggles after executing a fixed number of instructions. By selecting different toggle intervals—such as every 1 million, 500,000, or 2 million instructions—we can observe the timing consistency across multiple scales. The oscilloscope measures the precise time between these flag transitions, giving us a reliable hardware-level indication of execution speed.

To validate the measurement, the hardware results were compared against cycle-accurate simulation outputs. In simulation, the instruction count and corresponding cycle timing are known exactly, so the expected execution rate can be computed. Matching the oscilloscope-based timing with the simulated timing confirms that the CPU is running at the intended frequency and behaving deterministically. Using the relationship $\text{MIPS} = (\text{Number of Instructions}) / (\text{Measured Time})$, the calculations consistently converged to roughly 18.2 MIPS, giving confidence in both the testing methodology and the processor's performance. Here is our results table after conducting all ten tests:

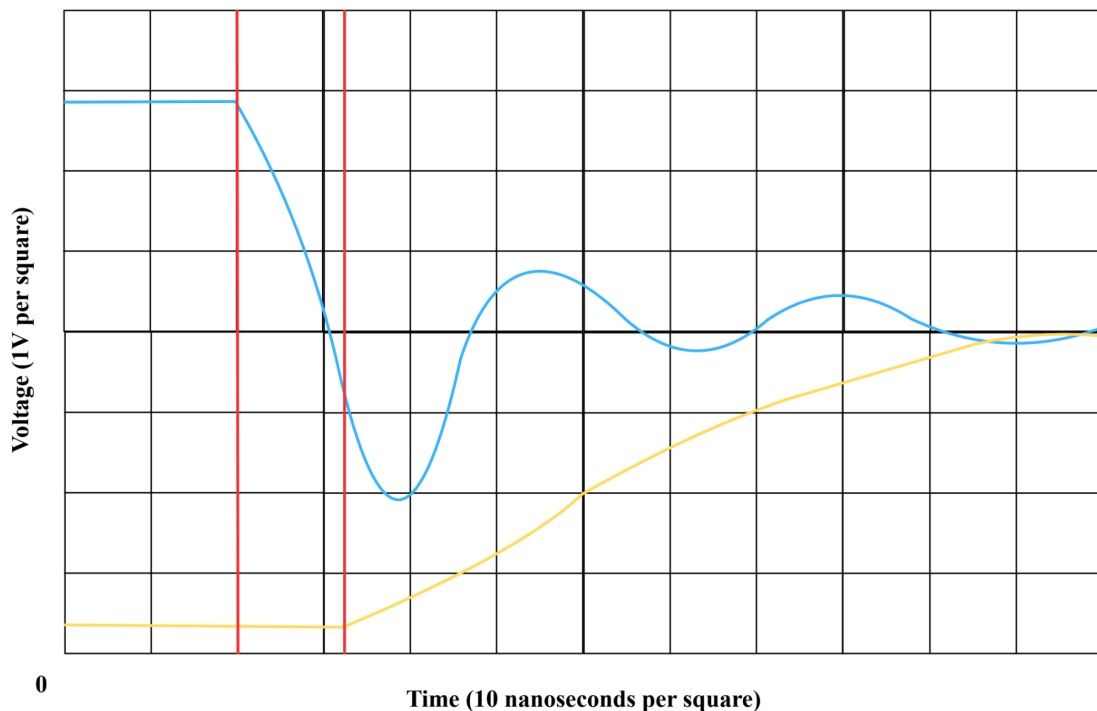
Test #	# of Instructions	Hardware Delay (s)	Hardware MIPS	Simulation Delay (s)	Simulation MIPS	Match
1	1M	0.055	18.2	0.055	18.2	Yes!
2	1M	0.055	18.2	0.055	18.2	Yes!
3	1M	0.055	18.2	0.055	18.2	Yes!
4	1M	0.055	18.2	0.055	18.2	Yes!

5	500k	0.028	18.2	0.028	18.2	Yes!
6	500k	0.028	18.2	0.028	18.2	Yes!
7	500k	0.028	18.2	0.028	18.2	Yes!
8	2M	0.11	18.2	0.11	18.2	Yes!
9	2M	0.11	18.2	0.11	18.2	Yes!
10	2M	0.11	18.2	0.11	18.2	Yes!

9.2.3 Button Input Tests

To support interactive games, our system incorporates button inputs via the custom PCB. These buttons allow users to play the game and interact with the CPU. Thus, we performed tests that verify proper signal routing from the PCB to the FPGA, debounce handling, and responsiveness under different user inputs.

We confirmed that each button correctly triggers the expected control signal and is interpreted reliably by the CPU during execution. We also assessed latency between pressing a button and the corresponding change on the LED matrix using the oscilloscope. These tests help ensure that the gameplay experience is smooth, responsive, and intuitive.



Test #	Latency (ns)	Match
1	10	Yes
2	13	Yes
3	10	Yes
4	11	Yes
5	10	Yes
6	20	Yes
7	13	Yes
8	14	Yes
9	11	Yes
10	12	Yes

9.2.4 Matrix Refresh Rate

We tested the matrix refresh rate via simulation using a C program to toggle a specified pin each full matrix display update (32 addresses cycle) and calculating the time between each toggle. We use the formula of inverse of time between each toggle ($1/t$) to calculate refresh rate. We also conducted a hardware test by programming a similar pin on hardware and connecting an oscilloscope to said pin in order to measure the time between each toggle in real time.

Test #	Full display refresh rate (ms)	Simulation display refresh rate (ms)	Results matching
1	6.57	6.65	Yes
2	6.67	6.65	Yes
3	6.7	6.65	Yes

4	6.66	6.65	Yes
5	6.65	6.65	Yes
6	6.69	6.65	Yes
7	6.69	6.65	Yes
8	6.68	6.65	Yes
9	6.55	6.65	Yes
10	6.71	6.65	Yes

Chapter 10: Administrative Content

This chapter lists all of the administrative aspects of our project. It includes budgeting, Bill of Materials, team responsibilities, and project milestones. In the *Budgeting and Financing* section, we provide a financial overview, discussing each component's quantity and cost. In the *Bill of Materials* section, we list a formal set of necessary materials. We provide a breakdown of individual contributions in the *Distribution of Work* section, and a timeline that tracks key milestones throughout both semesters of senior design in the *Milestones* section.

It is extremely important to keep track of the administrative content because it prepares us for our work in industry. Engineers are expected to not only develop technical solutions, but also to plan, document, and manage resources efficiently. Clear administrative planning helps to avoid overspending and keep the project on track. By practicing these skills now, we're gaining experience with real-world project management processes that will be essential in our professional careers.

10.1 Budget and Financing

The estimated budget for SPUD was determined by several factors. First, we estimated reasonable pricing for each component and created a grand total from that value. We also had to consider that we may need to spend a bit more than we expect in order to maintain a high quality project. With all of this in mind, we aimed for a cap of \$500. The biggest expense is the Arty S7-50 FPGA development board, priced at \$199.00 for one unit. The 64x64 RGB LED matrix, which serves as the main display, costs \$59.95. To power the system, a 5V 4A power supply is included at \$14.95, and a \$2.00 female DC power adapter for connectivity. Two level shifters are also required to handle voltage compatibility between components, each priced at \$8.49. These items make up the core of our project's hardware expenses.

We are grateful for the financial support from Dr. Mike Borowczak, whose contribution played a critical role in bringing SPUD to life. His support allowed us to acquire key components such as the Arty S7-50 FPGA and the LED matrix (two of our most essential components) By alleviating some of the financial burden on us, Dr. Borowczak allowed us to focus more fully on the technical aspects of the project and pursue a more ambitious design. We sincerely appreciate his investment in our success and the opportunities it has created.

Table 10.1 Budget

Component	Quantity	Total	Details
Arty S7-50: Spartan-7 FPGA	1	\$199.00	Programmed CPU
Printed Circuit Board	1	\$100	Integrates interface and CPU
64x64 LED Display	1	\$50	Interface
Controller/Button s/LEDS	MISC	\$100	Simple Momentary Push Buttons
Total:			\$500 max

10.2 Bill of Materials

The bill of materials includes all essential hardware components required for the successful implementation of SPUD. Each item listed serves a specific function in the system architecture, whether for computation, display, or user interaction. The components were selected based on compatibility with our FPGA. The final bill of materials will be refined once the prototype is tested and any additional materials are identified.

Table 10.2 Bill of Materials

Type	Component	Quantity	Total
FPGA	Arty S7-50: Spartan-7 FPGA	1	\$199.00
PCB	Peripheral Board	1	\$2
PCB	Controller Board	1	\$15.80

Type	Component	Quantity	Total
LED Matrix	64x64 LED Display	1	\$37.90
PSU	5V 4A power supply	1	\$14.95
Power Adapter	Female DC Power adapter	1	\$2.00
Level Shifter	HiLetGo 5pcs TXS0108E Conversion Module	1	\$9.75
Button	Arcade Button - 30mm Translucent Red	10	\$53.60
Connector	Harwin Gecko 12POS 1.25mm	2	\$6.15
Total:			\$341.15

10.3 Distribution of Work

Work on SPUD was distributed evenly among the four of us, with tasks assigned based on each of our strengths and interests. Responsibilities include CPU integration, simulation and verification, hardware interfacing, PCB design, and documentation. Our team also collaborated often to review our progress and reassign tasks if needed.

Table 10.3 Workload Distribution

Team Member	Responsibilities
Gabriel Saborido <i>Computer Engineering</i>	ARTY S7 Expansion Board PCB and Power Management

	User Control Interface PCB
	Soldering and Assemble
Mohamed Moussa <i>Computer Engineering</i>	Matrix display interface
	Bit Shift Display Control Logic
	Display Control C library development
Frank Laterza <i>Computer Engineering</i>	Team Lead, overseeing all tasks
	System integration and infrastructure setup
	Demo C library development (Spudkit)
Valentina Terry <i>Computer Engineering</i>	SoC design customization
	GCC RISC-V toolchain setup
	CPU simulation and testing

10.4 Milestones

The SPUD project was structured into key milestones spanning both semesters of senior design. These milestones ensured timely progress and alignment with course deliverables. The first semester focused on project definition, architectural planning, and documentation, while the second semester prioritized implementation, testing, and demo preparation. Each milestone corresponds with a specific deliverable or internal checkpoint to track development and refine the design.

Table 10.4 *Initial Project Milestones*

Senior Design I		
Week	Description	Due Date
1-2	Brainstorm project ideas	-
3	Research and D&C Document	01/17/25
4	Finish D&C Document	01/24/25

5	Meet with SD Coordinators to Discuss Project	01/27/25
6	Begin 60-page Draft	03/24/25
7	Study Possible Architecture and Design Patterns	-
8-10	Brush Up With SystemVerilog and Start Simulations	-
11	Finish 60-page Draft	03/24/25
12-13	Midterm Report Feedback Meeting and Work on Final Report	-
14	Order FPGA and LED screens	-
15	Continue working on Final Report	04/22/25
16	Submit 120-page Final Report	04/22/25

Table 10.5 End Project Milestones

Senior Design II		
Week	Description	Due Date
1	Scheme and plan work breakdown	08/22/25
2	Implement testing software with SystemVerilog	08/29/25
3	Start Design of Display Controller	09/05/25
4-7	Design Verilog For SoC Matrix controller, LED controller	09/12/25
8	Test design with Verilog and Integration Testing Software With New Modules.	10/10/25
9	Finish Controller Logic	10/17/25
10	Test CPU on machine code	10/24/25
11	Start Making Instructions with RISC-V GCC Toolchain	10/31/25

Senior Design II		
Week	Description	Due Date
12	Start making simple programs and confirm Instructions run correctly after the compiler makes machine code.	11/07/25
13	Make CPU machine code loader for instruction memory	11/14/25
16	Complete last TODOs and Finish Strong!	11/19/25

Chapter 11: Conclusion

Project SPUD stands for Single-core Processing Unit and Display. This is a project that brings our best qualities into one project that sparks our passion. Our goal was to build and design our own CPU to challenge ourselves and fully earn the degree of computer engineering. That project for us is a 5-stage pipeline, single-core RISC-V processor that's designed to work alongside our matrix display driver.

As a team of full computer engineers, our goal is to fully understand what it means to be exposed to professional tools and hardware. Some have expertise in verilog and verification, some are better at software, and some are good at PCB design. This was the perfect project for all of us to learn to excel in our fields. On a small scale this project is representative of what major corporations are capable of. We hope the skills we learn and the impact we make with this project will make waves in our impressions as we transition into our careers.

This project is also quite unique when being compared to others taking senior design one and two. Our team brainstormed many ideas before deciding on this project, but not without careful consideration. We were warned that a standard project should have a microcontroller involved with some moving parts while project SPUD has moving parts in its microcontroller. This choice we made willingly will make our project uniquely different from others. That is why we were lucky enough to get Dr. Mike Borowczak to mentor and guide us through the troubling part of this project. We are thankful for this contribution and expertise.

The core idea of the project is to create a RISC-V processor to run our own custom code, in the aim to control a matrix display to interactive demos like snake or conway's game of life. We think our project is interesting enough for demo day and will require enough labor to not raise any questions about credibility.

In detail, the processor we are designing includes a single cycle with a 5 stage pipeline supporting fetch-decode-execute cycle. The main modules of our CPU are the PC, Decode, Register file, ALU, Data memory, and Control block. Cleverly putting these together and thoroughly testing these against a RV32IM instruction set architecture will ensure our code runs smoothly on our CPU.

Our FPGA of choice, after careful consideration, resulted in us choosing the Arty S7-50. Due to its compatibility with the Vivado software and community support. We think this FPGA is a perfect choice for our project. Another advantage is the

arduino header like pins that we can use to test the I/O. We've also been able to create our own pcb to connect directly to the top of the FGPA dev board. Also on the Arty S7 50 has a RAM that supports DDR3 that we will be utilizing to support data memory and instruction memory. Our CPU will be loaded on the FPGAs fabric and the program instruction will be loaded through an exposed UART channel and written to RAM to be read by the CPU and pointed by the program counter.

Down the line our project will pay off in many ways. Not only is it a fancy feather in our cap, but a useful reference and experience that we can use in our future ventures. By completing this challenging project we are

In summary, SPUD is more than just a senior design project. It is a testament to our growth as engineers and our ability to collaborate, innovate, and solve complex problems. By building a custom RISC-V processor, developing the supporting hardware and software, and successfully interfacing with a dynamic LED matrix, we have demonstrated a comprehensive understanding of computer engineering principles. The skills and insights gained throughout this journey will serve us well as we transition into our professional careers, and we hope that SPUD will inspire future students to pursue ambitious, hands-on projects that push the boundaries of what is possible.

Appendix A - References

- [1] A. Abdelmoaty, M. Al-Shyoukh, and A. Fayed, "A high-voltage level shifter with sub-nano-second propagation delay for switching power converters," 2022 IEEE Applied Power Electronics Conference and Exposition (APEC), pp. 2437–2440, Mar. 2016, doi: <https://doi.org/10.1109/apec.2016.7468207>.
- [2] Adafruit Industries, "5V 4A (4000mA) switching power supply - UL Listed," Adafruit.com, 2025. <https://www.adafruit.com/product/1466>
- [3] Adafruit Industries, "64x64 RGB LED Matrix - 3mm Pitch," Adafruit.com, 2020. <https://www.adafruit.com/product/4732>
- [4] Adafruit Industries, "Arcade Button - 30mm Translucent Red," Adafruit.com, 2025. <https://www.adafruit.com/product/473#technical-details>.
- [5] Adafruit Industries, "Female DC Power adapter - 2.1mm jack to screw terminal block," Adafruit.com, 2025. <https://www.adafruit.com/product/368>.
- [6] "Amazon.com: HiLetgo 5pcs TXS0108E High Speed Full Duplex 8 Way 8 Channel Level Conversion Module 8-Bit 8 CH Logic Level Bi-Directional Converter Module : Electronics," Amazon.com, 2025. <https://www.amazon.com/HiLetgo-TXS0108E-Conversion-Bi-Directional-Converter/dp/B07BNYVJBB/>.
- [7] "AMD Vivado™ Product Details," AMD, Jan. 23, 2025. <https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vivado/vivado-buy.html#tabs-413944f675-item-d41fa24f69-tab>.
- [8] Are, "Are BJTs suitable for level-shifters? It seems FETs are more common,how do they compare?," Electrical Engineering Stack Exchange, Aug. 19, 2017. <https://electronics.stackexchange.com/questions/324795/are-bjts-suitable-for-level-shifters-it-seems-fets-are-more-common-how-do-they>.
- [9] "Arty S7: Spartan-7 FPGA Development Board," Digilent, 2020. <https://digilent.com/shop/arty-s7-spartan-7-fpga-development-board/>.
- [10] "Arty A7-100T: Artix-7 FPGA Development Board," Digilent, 2020. <https://digilent.com/shop/arty-a7-100t-artix-7-fpga-development-board/?srsltid=AfmBOoqv64IP1QeboNJhh8a300sjdpW-gZmvIJAqCJZfWM8abww7wuhC>.

- [11] B. Giovino, "How the RISC-V Multiply Extension Adds an Efficient 32-bit Multiply to the RV32I," DigiKey, Nov. 08, 2021. <https://www.digikey.com/en/blog/how-the-risc-v-multiply-extension-adds-an-efficient-32-bit>
- [12] chipmunk, "Designing RISC-V CPU from scratch – Part 2: Specifications & Architecture – Chipmunk Logic," Chipmunk Logic – Learn. Think. Design your own Chip, Apr. 30, 2023. <https://chipmunklogic.com/digital-logic-design/designing-pequeno-risc-v-cpu-from-scratch-part-2-specifications-and-architecture/>.
- [13] Chipmunk Logic, "Part 2: Specifications & Architecture", 2025. <https://chipmunklogic.com/digital-logic-design/designing-pequeno-risc-v-cpu-from-scratch-part-2-specifications-and-architecture/> (accessed Apr. 22, 2025).
- [14] "Compare Editions Pro Standard Lite (Free) Simulation Tools Additional Development Tools Embedded Tools Questa*-Intel® FPGA Edition Questa*-Intel® FPGA Starter Edition Intel® FPGA Power and Thermal Calculator Intel® FPGA AI Suite Intel® Advanced Link Analyzer Tool Intel® HLS compiler DSP Builder for Intel® FPGAs Intel® SoC FPGA Embedded Development Suite Nios® V P..." Accessed: Apr. 22, 2025. [Online]. Available: <https://www.intel.com/content/dam/www/central-libraries/us/en/documents/2023-11/quartus-prime-comparison-infographic.pdf>
- [15] Digilent Inc., "Arty S7 Reference Manual," Digilent Documentation, [Online]. Available: <https://digilent.com/reference/programmable-logic/arty-s7/reference-manual>.
- [16] Discrete, "KiCad vs. Eagle vs. Altium: (Paid vs. Open Source) A Comprehensive Guide," discrete, 2025. <https://discrete.co.in/blog/kicad-vs-eagle-vs-altium-a-comprehensive-comparison-guide>.
- [17] D. Level, "Digital Level Shifter with Op-Amp," Electrical Engineering Stack Exchange, Oct. 21, 2015. <https://electronics.stackexchange.com/questions/196409/digital-level-shifter-with-op-amp>.
- [18] "EDA Playground," Edaplayground.com, 2017. <https://edaplayground.com/x/yiR>.
- [19] "EDA Playground," Edaplayground.com, 2017. <https://www.edaplayground.com/x/hJv>.

- [20] Five Embeddev, "RISC-V Instruction Set Manual, Volume I: RISC-V User-Level ISA," Five EmbedDev, 2021. <https://five-embeddev.com/riscv-user-isa-manual/Priv-v1.12/instr-table.html>
- [21] "FPGAs 2: Multiplexed RGB LED Panels," *Rhye.org*, Apr. 04, 2020. <https://rhye.org/post/fpgas-2-led-panel-display/> .
- [22] "G125-2041296L0 | DigiKey Electronics," DigiKey Electronics, 2025. <https://www.digikey.com/en/products/detail/harwin-inc/G125-2041296L0/3919278>.
- [23] G. Akins, "RGB LED Panel Driver Tutorial," Bikerglen.com, 2014. [Online]. Available: <https://bikerglen.com/projects/lighting/led-panel-1up/>.
- [24] HarieshAnbalagan, "GitHub - HarieshAnbalagan/RV32I: Minimalistic RV32I RISC-V Processor in System Verilog," GitHub, 2022. <https://github.com/HarieshAnbalagan/RV32I>.
- [25] "How to Choose The Best PCB Design Software," circuitdigest.com. <https://circuitdigest.com/tutorial/best-pcb-design-software>
- [26] "I'm design a simple RISCV 32bit CPU using an FPGA and system verilog. We are...," Perplexity AI, 2025. <https://www.perplexity.ai/search/i-m-design-a-simple-riscv-32bi-e0HxnP49TXSguNuC7Sw1pw>.
- [27] "I'm design a simple RISCV 32bit CPU using an FPGA and system verilog. What ISA...," Perplexity AI, 2025. <https://www.perplexity.ai/search/i-m-design-a-simple-riscv-32bi-YLo2EOhVT1iehP8qM.BuYw#0>.
- [28] Intel.com, 2025. https://cdrdv2-public.intel.com/666995/cv_5v2-683375-666995.pdf.
- [29] Khadija411, "GitHub - Khadija411/RV32I: RISC-V (RV32-I) is single cycle processor in system verilog," GitHub, 2021. <https://github.com/Khadija411/RV32I>.
- [30] "Logic Level Shifters for Driving LED Strips," Electric Fire Design, Mar. 12, 2021. <https://electricfiredesign.com/2021/03/12/logic-level-shifters-for-driving-led-strips/>.

- [31] markos-stefanidis, "GitHub - markos-stefanidis/riscv-rv32im-core: System Verilog Implementation of a basic Risc-V core.," GitHub, 2022. <https://github.com/markos-stefanidis/riscv-rv32im-core>.
- [32] mikeselectricstuff, "Driving LED matrix displays with an FPGA," *YouTube*, May 04, 2014. <https://www.youtube.com/watch?v=Sq8SxVDO5wE> (accessed Apr. 22, 2025).
- [33] "MOSFET level shift - Electronics information from PenguinTutor," PenguinTutor.com, 2023. <https://www.penguintutor.com/electronics/mosfet-levelshift>
- [34] ProtoSupplies, 2025. *TXS0108E 8-Ch logic level converter module*, <https://protosupplies.com/product/txs0108e-8-ch-logic-level-converter-module/>
- [35] P. Burgess, "RGB LED Matrix Basics," *Adafruit Learning System*, Dec. 11, 2012. <https://learn.adafruit.com/32x16-32x32-rgb-led-matrix/how-the-matrix-works>.
- [36] P. Kiaei and P. Schaumont, "SoC Root Canal!: Root Cause Analysis of Power Side-Channel Leakage in System-on-Chip Designs," ResearchGate, Aug. 2022. [Online]. Available: https://www.researchgate.net/figure/Block-diagram-of-RISCV-SoC-and-its-five-stage-RISC-V-processor-Resources-from-different_fig3_363175823.
- [37] "Reddit - The heart of the internet," Reddit.com, 2025. https://www.reddit.com/r/AskElectronics/comments/npwl0t/why_does_this_logic_level_shifter_work/.
- [38] "Reddit - The heart of the internet," Reddit.com, 2022. https://www.reddit.com/r/FPGA/comments/xkbs3j/i_compared_16_different_fpga_boards_and_chose_my/?rdt=62046.
- [39] "Reddit - The heart of the internet," Reddit.com, 2022. https://www.reddit.com/r/WLED/comments/ywdeet/best_logic_level_shifter_to_use/.
- [40] "RISC-V GNU Compiler Toolchain," GitHub, May 13, 2022. <https://github.com/riscv-collab/riscv-gnu-toolchain>
- [41] "RISC-V Ratified Specifications," RISC-V International, 2014. <https://riscv.org/specifications/ratified/>

- [42] “RISC-V Technical Specifications - Home - RISC-V Tech Hub,” Atlassian.net, 2024. <https://lf-riscv.atlassian.net/wiki/spaces/HOME/pages/16154769/RISC-V+Technical+Specifications>
- [43] “Scheme-it | Free Online Schematic and Diagramming Tool | DigiKey,” Digikey.com, 2020. <https://www.digikey.com/en/schemeit/project>
- [44] Tamanna, “What Is a Level Shifter?,” INTLBM, Sep. 23, 2024. <https://intlbm.com/2024/09/23/what-is-a-level-shifter/>.
- [45] Terasic Technologies, “Terasic - DE Boards - Cyclone - Cyclone V GX Starter Kit,” Terasic.com.tw, 2025. <https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=165&No=830&PartNo=4#contents>.
- [46] The Signal Path, “TSP #62 - Tutorial on the Design & Implementation of an FPGA RGB LED Matrix Driver,” *YouTube*, Aug. 31, 2015. https://www.youtube.com/watch?v=h6NH_3KpWe8.
- [47] “TXS0108E,” Ti.com, 2024. <https://www.ti.com/product/TXS0108E>.
- [48] “unpriv-isa-asciidoc_20240411.pdf,” Google Docs. <https://drive.google.com/file/d/1uviu1nH-tScFfgrovvFCrj7Omv8tFtkp/view>
- [49] Youtube, 2025. <https://youtu.be/UwYFnTbY-8k?si=Qbu8JMtydAcZpylm>
- [50] G. Akins, “RGB LED Panel Driver Tutorial,” Bikerglen.com, 2014. [Online]. Available: <https://bikerglen.com/projects/lighting/led-panel-1up/>.

Appendix B - Copyright Permissions

- [1] ultraembedded, “GitHub - ultraembedded/riscv at 264e1aea7b4874746a020d69ca98166a9c0615b4,” GitHub, 2025. <https://github.com/ultraembedded/riscv/tree/264e1aea7b4874746a020d69ca98166a9c0615b4?tab=License-1-ov-file>