S.P.U.D.
Single-core Processing Unit and Display

Frank Laterza, Valentina Terry, Gabriel Saborido, Mohamed Moussa

Dept. of Electrical Engineering and Computer Science, University of Central Florida, Orlando, Florida, 32816-2450

***Abstract* — Single-core Processing Unit and Display (S.P.U.D.) is a CPU and SoC implementation that utilizes the open source RISC-V architecture and executes rv32im instructions to control custom peripherals including a custom controller and an LED matrix display as the main interfaces. This paper presents the design methodology of the main processing core, the implementation of key peripherals, and the software structures that integrate these components into a functional unit.**

Index Terms — CPU, Field programmable gate arrays, Light emitting diodes, RAM, Reduced instruction set computing.

I. Introduction

RISC-V (reduced instruction set computing) is an open source instruction set architecture (ISA) that follows the principles of streamlined and simplified instructions for a cheaper and more efficient implementation for processor instructions. It was developed in 2010 at the University of California, Berkeley and was then transferred to RISC-V International (a non-profit, Switzerland based organization) and is currently being maintained and developed further as of 2025. RISC-V being open source means that it is open for anyone and any entity to modify and implement the RISC-V principles as they choose, allowing for a better experience for smaller-scale independent projects, which made it perfect for our project. S.P.U.D. (Single-core Processing Unit and Display), an independently designed unit that employs a RISC-V based processor to run user designed high level programs and drive our peripherals in order to provide a fully functional, fully customizable machine that operates on a level of personal embedded boards/ single purpose machines. The motivation behind S.P.U.D was mainly to design a processor that would be the culmination of the team's years of experience as computer engineers, However, the S.P.U.D. does have some advantages of its own that make it a strong choice for practical usage, its simplistic design using RISC-V means that it does not suffer from the bloat most modern computers have, which is further compounded by the fact that the design is minimalistic, only utilizing functionalities that are necessary; S.P.U.D. also operates on both a lower level hardware implementation using FPGAs, and a higher level implementation using C programs, meaning that it is possible to further improve processing speeds by implementing some critical functionality on the hardware level, as opposed to implementations exclusively in higher level, embedded system based platforms. The processor utilizes open-source hardware designs to streamline the implementation process of the main SoC (system on a chip), while the main focus of the team is on implementing the interfaces between the SoC, the peripherals, and setting up the environment needed to compile and run the higher level code, user input will mainly be through the an arcade controller, and the system's main output will be through an LED matrix display. The final goal for the S.P.U.D. is a full RGB display that can display thousands of colors, a library of programs including games and practical applications that can utilize the arcade controller and the display, and a fully streamlined implementation of both the hardware and the software in order to fully utilize the processing capabilities of the S.P.U.D. system.

II. System Components

S.P.U.D. is best understood through its key system components. Each is carefully chosen to complement and enhance one another, working together to bring the final product to life. Below is an overview of each of these components and why they are crucial to the development of the system.

A. FPGA

The Xilinx Arty S7-50 Spartan-7 is the bread and butter of S.P.U.D., delivering the affordability and versatility that is desired. It is equipped with 52,160 logic cells, 65,200 flip-flops, and 2,700 Kb of BRAM, all crucial to the implementation of our custom CPU architecture. Additionally, the Spartan-7 features external clock speeds of 100MHz, 256MB of DDR3L RAM, and USB-UART bridge connectivity, ensuring smooth integration with the rest of our components.

B. LED Matrix

The Adafruit 64x64 RGB LED Matrix represents the visual output of S.P.U.D., providing a high-resolution grid containing 4096 RGB LEDs and a 3mm pitch at a refresh frequency capable of over 400Hz. Each LED is

individually programmable, making it ideal for any dynamic visuals. The display comes with two 32x64 panels, each with 5 addressable pins and requires around 1.6KB of RAM to buffer the 12 bit color image and a 5V power supply to operate.

C. Level Shifter

The TXS0108EPWR is an 8-bit bidirectional level shifter with 1-4ns propagation delay and a voltage range of 1.2V to 5V. Each shifter houses 8 data channels routed to the HUB75 header of the matrix. This allows the system to safely drive the LED matrix by shifting the 3.3V from the Spartan-7 FPGA to 5V for the matrix. Without these shifters, S.P.U.D. risks data corruption during communication.

D. Peripheral Hat

The peripheral hat is a custom PCB designed to route communication between the Spartan-7 FPGA, 64x64 LED matrix, and an arcade controller. This PCB sits directly on top of the FPGA and houses the two TXS01018EPWR level shifters for safely transitioning between 3.3V from the FPGA to 5V for the LED matrix, as well as a connector to drive an arcade controller for user input.

E. Controller

The arcade controller is a custom built PCB representing the main input interface for S.P.U.D. It was designed to connect directly to the peripheral hat PCB and house 10 arcade buttons, each programmable to allow for user interactivity to S.P.U.D. Users will be able to control any program running on S.P.U.D. and have full-view of their output via the LED matrix.

III. Hardware Design

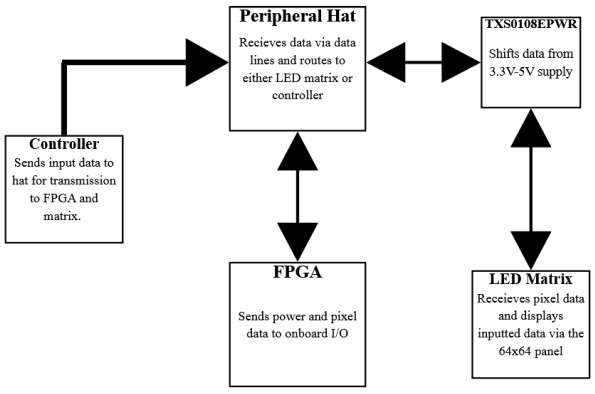


Fig. 1. Hardware flowchart overview

The overall system functions as follows: the user provides input through the arcade controller, which is then processed into the FPGA, where finally, any output will be displayed on the LED matrix; these components are then connected together through the hat, ensuring a secure connection between the components.

A. Peripheral Hat Design Specifications

The peripheral hat PCB serves as the road between the FPGA, LED matrix, and arcade controller. While the PCB is not heavily crowded, the small number of components present have their own place in the success of S.P.U.D. The goal of the design was for the PCB to sit on top of the FPGA without needing a separate enclosure. Carefully modeling the FPGA in KiCad in order to line the headers of the PCB with the Spartan-7 FPGA. Spartan-7 is equipped with 42 I/O pins and utilizes a majority of these pins for the other components. The arcade controller alone takes up 10 of these pins, and their placement on the bottom right headers ensures a short clean routing to the arcade controller header seated on the bottom of the hat. The LED uses 16 I/O pins as data lines to communicate with its HUB75 header. These 16 data lines pass through the two TXS0108EPWR level shifters. Since the shifters are utilizing pins across the top right and bottom headers, they are placed in the center of the peripheral hat to route easily to the FPGA and matrix connector pins. There are 10k ohm resistors on each enable pin to ensure that they are disabled on startup. On each voltage transmission between the FPGA and matrix sits a 0.1uF capacitor to filter out noise and keep power stable between shifts.

B. Arcade Controller Design Specifications

The arcade PCB serves as the road between the user and S.P.U.D. Unlike the peripheral hat, this PCB has a single-purpose of inputting button presses to be processed by the CPU. It is a 2-layer board designed with simplicity in mind. The board contains 10 Adafruit 30mm arcade buttons housed on the board placed to resemble a leverless fighting game controller. 4 buttons for left-right-up-down and 4 action buttons. The last two buttons sit on the bottom to be used for any extra programmable action desired. Each button connects to a thermal pad with the footprint of a keyboard switch. One pad is 3.3V and the other connects to ground. Once a button is pressed it bridges the connection between the pads and the button reads as active low. On top sits a 2x6 header connecting to the peripheral hat and 10 I/O pins of the FPGA. The controller operates on 3.3V provided by the Spartan-7 and each button is paired with a 10k pull-up resistor for stable digital signals on press. In order to account for the lack of a debouncing capacitor, there is a 30ms delay implemented in software per button press.

IV. Processor Architecture

A. Overview

At the highest level, CPUs are extremely complex and require an entire team of engineers to design and implement the architecture. While it is definitely possible for a team of four computer engineering students to design a CPU in Verilog, there would be very little to demonstrate besides code, since the majority of the focus would be there. To combat this, SPUD focuses on the *customization and extension* of an existing CPU, designed by UltraEmbedded. The goal is to understand the existing architecture enough to be able to make modifications and integrate it into a working, physical system.

The CPU architecture is composed of three parts: the processor core, the SoC, and the FPGA. A useful analogy could be that the core is considered to be SPUD's "brain," the SoC is SPUD's "nervous system," and the FPGA is SPUD's "body." The majority of the focus for this project is on the SoC and FPGA side. The SoC integrates the core, on-chip RAM, memory-mapped peripherals, and a bus interconnect. The FPGA acts as the physical medium where the entire SoC is instantiated. It contains the configurable logic blocks, routing fabric, and I/O interfaces that bring SPUD to life as real, operating hardware. In the sections below, the peripherals within the modified SoC are explored further, program loading within the FPGA using the debug bridge, and testbench verification using Verilator.

B. Memory-Mapped Peripheral Support

SPUD has an assortment of peripherals integrated into its SoC. This allows it to go beyond the operations of the core CPU. Peripherals are additional circuits built into the processor that expand its capabilities and make it more versatile for interacting with other devices and external hardware. There is an assortment of peripherals implemented in SPUD. The standard given with UltraEmbedded included SPI, UART, GPIO, TIMERS, and IRQ. Many of the programs in SPUD utilizes UART to print outputs and also take inputs through the builtin USB serial bridge. For reading the inputs to the arcade controller, the GPIO peripheral is used to read voltages.

Considering SPUD is custom designed, it seemed appropriate to utilize this advantage to add extra peripherals catered to specific needs. The first peripheral expansion was an LED controller. Its sole purpose was to control the FPGA's LEDs and help develop a strong base before moving onto the matrix controller peripheral. To control a peripheral through high level software like C code, SPUD dedicates a special place in memory that can be accessed with SPUD software. When a program wants to turn on an LED, it writes to what appears to be a normal memory address. For the led controller specifically its 0x95000000 as shown below.

Peripheral	Base Address	Info
IRQ	0x90000000	Interrupt requests
Timer	0x91000000	Task scheduling, timeouts, clock cycle counter
UART	0x92000000	Serial communication between CPU and other devices
SPI	0x93000000	High-speed serial communication
GPIO	0x94000000	Input/output configuration
LED Controller	0x95000000	Controls FPGA LEDs
Matrix Controller	0x96000000	Provides CPU register access to update pixel data on display

Fig. 2. Memory-Mapped Registers for Peripherals

The figure above lists all of the memory-mapped registers in the SoC, and the highlighted registers represent the custom registers designed specifically for the SPUD system. The software creates a pointer to the LED controller address, for example, using the volatile keyword (which prevents the compiler from optimizing away the memory access), reads the current value, modifies the specific bit corresponding to the LED it wants to control, and writes the new value back. When the CPU executes this store instruction, it doesn't actually write to RAM. Instead, the address decoder in the SoC recognizes that 0x95000000 falls within the LED controller's address range and routes the write transaction over the AXI4-Lite bus to the LED peripheral module. The LED controller hardware receives this write request through its AXI interface signals. Specifically cfg_awvalid_i (address valid), cfg_wvalid_i (data valid), and cfg_wdata_i (the actual data). When both valid signals are high and the peripheral is ready (cfg_awready_o), a write enable pulse (write_en_w) is generated. This pulse is registered on the

next clock cycle into led_data_wr_q, which then enables the LED data register to capture the bottom two bits of the write data into led_data_q on the following clock cycle.

Finally, these register bits are continuously assigned to the physical output pins (led_o[1:0]), which connect directly to the FPGA's LED pins (LD2 and LD3), causing them to turn on or off based on the bit values. This entire process comes from a simple C statement like *led_ctrl = 0x02 to physically illuminating an LED and takes just a few clock cycles to demonstrate how memory-mapped I/O creates a seamless bridge between high-level software and low-level hardware control, all without requiring any special I/O instructions beyond normal load and store operations.

C. Matrix Core

The current matrix architecture has 2 major components, the matrix core and the matrix controller. The matrix core is the architecture responsible for receiving raw pixel data and processing it into an image, our LED matrix is based on the HUB75 interface, which has 16 pins and 2 panels per display.

R1	**G1**
B1	**GND**
R2	**G2**
B2	**E**
A	**B**
C	**D**
CLK	**LAT**
OE	**GND**

Fig. 3. HUB75 header

The main focus of the architecture are the 5 addressing pins (A,B,C,D,E), the 6 RGB pins (R1,R2,G1,G2,B1,B2), and the LAT pin. The process of displaying an image involves the on-board shift register for the display. Even though there are 64 rows on the display, a 5 bit address only corresponds to 32 unique addresses; this is because each unique address corresponds to 2 rows, one on each panel of the display, this means the 0th address corresponds to both the 0th row and the 31st row. An image is displayed by loading up 64 pixels worth of data per row, the process begins by fetching the correct address, for example, 00000, which corresponds to 0th and 31th rows, will have the pixel data loaded into each of them, The pixel data is mainly set by driving the RGB pins.

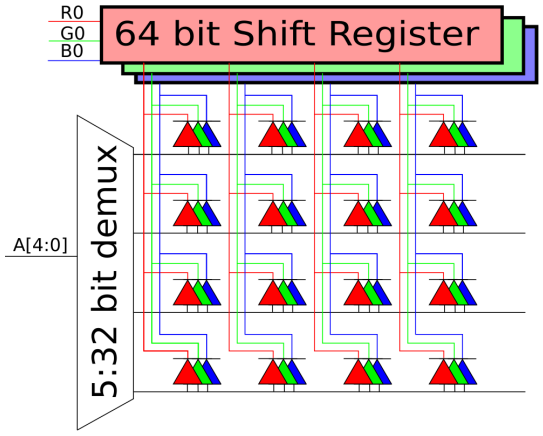


Fig. 4. HUB75 shift register

With R1,G1,B1 for the top panel, and R2,G2,B2 for the lower panel, RGB pins are used to set the color of each pixel, meaning you set the RGB data for an individual pixel in a row before moving onto the next pixel in the row, each pixel is iterated through using the shift register, which shifts onto the next pixel during each clock edge, so for each row, a total of 64 clock edges are needed in order to iterate by each pixel and update their RGB values accordingly; After fully loading a row with pixel data, it is then possible to latch the row data to have the LEDs turn-on in that row, this process is repeated 32 times in order to cycle through all 64 rows of the display, and doing this cycle fast enough will display a flicker-free image on the LED matrix. The current implementation of the matrix core is a state machine, with one state for loading the pixel data into the rows, one for latching said pixel data, and one for finally turning on and maintaining the LED; the pixel data is stored in an array of 4096 RGB registers and is controlled but the matrix controller register.

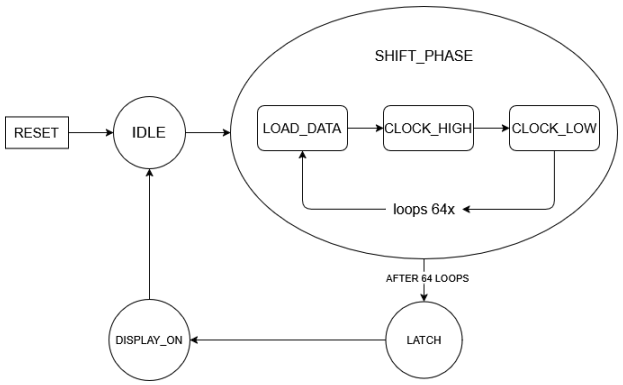


Fig. 5. State machine flowchart

As previously mentioned, the matrix controller is a custom peripheral that manages the display to update

pixels. It takes 36 bits to properly update a pixel; 24 bits for the RGB values (since each color requires a byte of data), and 12 bits for the pixel address (64x64 = 4096 = 2^12 possibilities). Due to the AXI bus payload being 32 bits wide, it is not possible to fit all the pixel data in one payload. Thus, two payloads must be sent sequentially to the matrix controller to write to a pixel.

The first payload contains the RGB data, and the next payload contains the address. The matrix controller, therefore, is split up into two registers: MATRIX_CTRL_DATA (0x96000000) and MATRIX_CTRL_ADDR (0x96000004). Depending on the offset of the address of the payload, the data gets written into its corresponding register. Once the controller receives both the data and the address of the desired pixel, it sends it to the matrix core, which will physically map it and update the matrix display.

D. Program Loading Via Debug Bridge

The debug process in the SPUD RISC-V SoC uses a UART-based protocol to load programs and control the CPU without requiring the CPU itself to be functional, creating a powerful out-of-band debugging mechanism.

When you run the Python script load.py to upload an ELF binary to the FPGA, the process begins by opening a serial connection to /dev/ttyUSB1 at 1 Mbaud and parsing the ELF file to extract loadable program segments containing code and data. The Python script then sends carefully formatted UART packets over the serial connection, where each packet starts with a command byte (0x10 for write or 0x11 for read), followed by a length byte indicating how many bytes to transfer, then a 4-byte address in big-endian format specifying where in the SoC's memory map to write, and finally the actual data payload (up to 128 bytes per packet for efficiency). These UART bytes arrive at the FPGA's debug UART port (separate from the application UART) and enter the dbg_bridge module, which implements a state machine that receives the incoming serial bytes, buffers them in an 8-deep FIFO, and decodes them by transitioning through states: IDLE (waiting for command), LEN (capturing transfer length), ADDR0-3 (capturing the 32-bit address byte by byte), and either WRITE (for memory writes) or READ (for memory reads).

Once the debug bridge has accumulated a complete address and data, it initiates an AXI4-Lite bus transaction by asserting the appropriate valid signals (mem_awvalid_o and mem_wvalid_o for writes, or mem_arvalid_o for reads) with the decoded address on mem_awaddr_o and data on mem_wdata_o, which feeds into an AXI interconnect that routes transactions to the correct destination based on address decoding—RAM accesses go directly to the DDR controller, while peripheral accesses go to the peripheral bus. Critically, the debug bridge can write to a special GPIO control register at address 0xF0000000 where bit 0 controls the CPU reset signal, allowing the Python script to hold the CPU in reset (by writing 0), load the entire program into RAM starting at 0x80000000, and then release the CPU (by writing 1) so it begins executing from the reset vector with the freshly loaded code. This means the CPU is completely halted during program loading, eliminating any race conditions or memory corruption from a running processor, and the debug bridge operates independently with its own clock domain crossing logic to interface between the UART's slower clock and the system's 25 MHz clock.

For memory reads, the process reverses: the debug bridge issues an AXI read transaction, waits for mem_rvalid_i to assert with mem_rdata_i containing the read data, then transmits this data back over UART to the Python script byte-by-byte through states DATA0-3, allowing verification that the program loaded correctly. The entire debug architecture is completely transparent to the running application—once the CPU is released from reset, it has no knowledge the debug bridge exists, and the application UART (used for printf debugging) operates on a completely separate UART port, though both can be multiplexed onto a single USB-serial connection on the host PC by using different device paths. This debug-over-UART approach provides the essential capabilities of JTAG debugging (memory access, CPU control, program loading) without requiring dedicated JTAG hardware, making it perfect for quick iteration during development where you can simply run ./run/load.py -f program.elf, watch it upload in seconds, and immediately see your code execute on the FPGA.

E. System Verification

In any design process– especially in a complex system such as SPUD– it is important to implement a testing framework that verifies functionality. Luckily, the UltraEmbedded design includes Verilator/SystemC support. Verilator is an open-sourced tool that works with SystemC to turn a Verilog design into an equivalent C++ model for simulation.

While the UltraEmbedded design came with this support, it's important to note that Verilator is notorious for its difficult and tedious setup process. As an open-sourced tool that is constantly updating, varying dependencies and versions add to the complexity of this setup. To keep everything consistent and clear, SPUD introduces new documentation that lists the step-by-step setup process for installing the correct version of Verilator. The goal is to ensure that future users of SPUD, or similar

projects, have a dependable reference for reproducing a working Verilator environment.

When running the simulation, Verilator compiles the Verilog source files into C++ and links them with the SystemC testbench. This produces an executable model of the design which behaves like real hardware but runs entirely in software. It loads the compiled binary file (ELF) into the simulated memory space before runtime. During simulation, the CPU fetches and executes instructions from this memory exactly as it would on the FPGA. Using the SPUD environment, C program demos get converted into ELF files (using a custom 'build' command) and then transferred into the Verilator environment (using a custom 'sim' command), creating a 'dump' file in the form of a .vcd. Using Verilator commands, the .vcd file opens a GUI of the waveform of the program, allowing for detailed observation of internal signals and performance. SPUD utilizes these waveforms to verify that the system exhibits the correct functionality.

The Verilator waveform is a powerful debugging tool for SPUD; when designing the custom peripherals, these waveforms were vital for determining whether custom peripherals were even possible, and if they behaved as intended. Without this level of visibility, verifying the design would have been significantly more time-consuming and would have limited SPUD's overall development progress.

Fig. 6. Sample waveform of LED Controller test

The image above demonstrates the debug process for the LED Controller peripheral, a custom SPUD peripheral designed for controlling the LEDs on the FPGA. The write_port_o signal represents the AXI TAP Controller decoded peripheral number (in this case 110 or decimal 6, which corresponds to the LED controller). The led_out signal represents which LED gets toggled. The waveform verifies that the CPU can directly access the FPGA LEDs. By modifying the SystemC testbenches for any hardware changes, SPUD can accurately verify its custom peripheral designs.

V. Software Architecture

The SPUD software architecture has been carefully designed to abstract the difficult and delicate nature of controlling hardware peripherals with special register access as well as creating a flexible build environment. Several layers have been developed to accomplish this starting with SPUDkit. A library of useful APIs and helper functions designed to aid the developer to quickly make programs for SPUD. Utilizing the gcc-riscv tool chain SPUDkit can generate C programs into rv32im machine code.

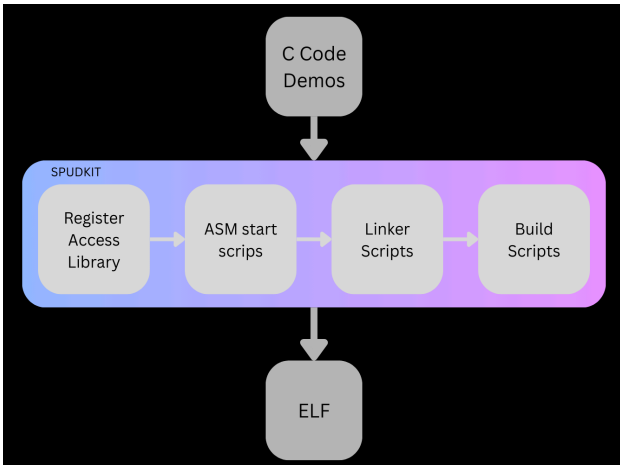


Fig. 7. C code program execution process

The process of building a RISC-V program with SpudKit begins with 'make <project name>', which invokes the GNU RISC-V cross-compiler toolchain (riscv64-unknown-elf-gcc) configured for bare-metal 32-bit RISC-V with the flags -march=rv32im_zicsr (32-bit with multiply/divide extensions), -mabi=ilp32 (32-bit calling convention), and critically -nostdlib -nostartfiles to operate without any operating system or standard library since we're building firmware that runs directly on hardware. The build happens in three phases: first, the compiler translates all SpudKit library C files (display.c, uart.c, gpio.c, timer.c, etc.) and the application's main.c into relocatable object files (.o) containing RISC-V machine code but without final memory addresses assigned yet. Second, the assembler processes start.s, the critical startup code that defines the _start entry point—this assembly code sets up the stack pointer to 0x80100000 (1MB above the program base), walks through the BSS section zeroing out all uninitialized global variables using the __bss_start and __bss_end symbols, and then calls your main() function. Third, the linker reads the rv32i.ld linker script which acts as a blueprint specifying exactly how to arrange everything in memory. First it declares the entry point as _start, defines RAM starting at 0x80000000 with 256MB capacity, and specifies that sections must be laid out as .text (code, with .text.init containing _start first), .rodata (constants), .data (initialized globals), and .bss (uninitialized globals with special boundary symbols for the startup code). The linker takes all the object files, resolves function calls and variable references by calculating final addresses,

combines matching sections from different files, and produces a single ELF file containing the complete program with metadata like section headers, symbol tables, and program headers that tell loaders where to place code in memory. The resulting ELF file is self-contained: when loaded into RAM at 0x80000000 and the CPU jumps to the entry point, it executes _start which initializes the environment, calls spudkit_init() to configure peripherals, enters main(), and your application runs with direct hardware access through SpudKit's memory-mapped I/O drivers—all orchestrated by the linker script ensuring code, data, and the 1MB stack don't collide in the address space.

Now putting everything together the low level components of the SPUD architecture and its peripherals can be seamlessly integrated together. After building the C program, it is loaded through a python based script and starts communicating with the hardware using UART through the debug bridge (as previously discussed), which then establishes communications with the DDR memory and the RISC-V core. The DDR memory is used as the main memory for our C programs and contains the buffer for the LED matrix pixel data, and the processor runs instructions that are needed to further process the program.:

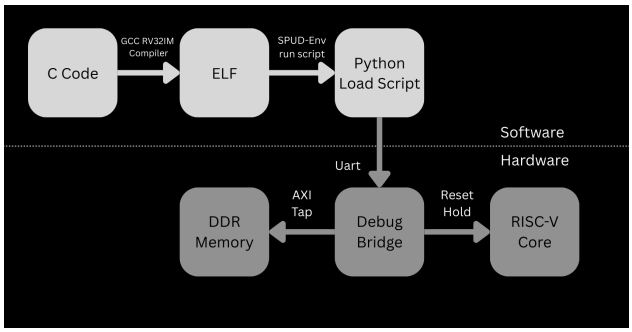


Fig. 8. Software integration into hardware

VI. Results

The demo process for S.P.U.D. serves as a way to showcase the system's full functionality and design integration. Each major component was tested thoroughly to ensure that the systems desired specifications were met.

A. Processor Performance

To evaluate SPUD's processor performance, a simple benchmark was implemented to measure the execution rate in MIPS (Millions of Instructions Per Second). The test program was designed to execute exactly one million RISC-V instructions before toggling an LED connected to the FPGA. By measuring the time between LED toggles, the number of instructions executed per second could be calculated.

To maintain accuracy, loop unrolling was used to reduce the number of additional branch and control instructions that might otherwise skew the instruction count. This approach guarantees that the loop body contains only the operations being measured.

The experiment was conducted both in Verilator simulation and on the physical hardware. In simulation, the total time between LED toggles was measured as approximately 0.055 seconds, corresponding to a performance of 18.2 MIPS. The same test was then performed on the FPGA using an oscilloscope connected to a pin that toggled alongside the LED. The measured time difference was again 0.055 seconds, confirming a consistent 18.2 MIPS result between simulation and hardware execution.

These results validate that the processor core behaves deterministically and that simulation results accurately reflect real hardware timing. Moving forward, additional performance tests will be conducted with varied instruction counts and more complex instruction mixes to better characterize the CPU's behavior across different workloads.

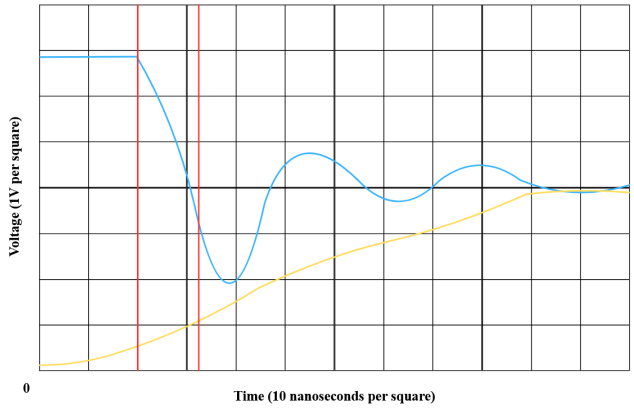
B. LED Matrix Refresh Rate

The refresh rate of the LED matrix is a critical factor in testing SPUD's visual output. A higher refresh rate allows animations and dynamic content to appear consistently to the human eye, while a lower rate can result in noticeable flickering and a less responsive display. Verifying this rate was therefore an essential step in confirming the overall functionality of SPUD. To begin, the current LED matrix core and controller were simulated to estimate the timing behavior of a full screen update. The simulation indicated that a complete cycle—where all 64 rows of the display are updated—took approximately 6.65 ms, corresponding to a refresh frequency of 150 Hz. To validate these results in hardware, a physical setup was constructed. This setup included a flag signal that toggled each time all 64 rows were refreshed, effectively marking the completion of a full screen cycle. The signal was connected to an oscilloscope, which measured the time interval between consecutive toggles. The measured interval matched the simulation results, also coming out to 6.65 ms, confirming the refresh rate of 150 Hz. This verification demonstrates that the LED matrix controller operates reliably and that the display updates at a consistent frequency.

C. Controller Button Latency

Testing the controller's button latency involved measuring the time delay between the button presses on

the arcade controller and the CPU's processing time.



Legend

Blue: Button response

Yellow: FPGA response

Red: Measurement of latency

Fig. 9. Button latency measurement showcase

The results above were measured utilizing an oscilloscope hooked to the voltage pin of the buttons and a flag raised on the FPGA whenever a button was pressed. The blue line demonstrates the button response vs. the yellow line which demonstrates the FPGA reaction. After testing each button for a total of 10 tests, the system performed with an average of 12ns for a full system response. The controller was anticipated to perform at a 30ns response time and this was far exceeded by over half.

VI. Conclusion

S.P.U.D. represents the final product of years of computer engineering education. A complete embedded computing system that brings custom hardware and software components to create a unique experience to the user. This system showcases the power of a single core processor and how its implementation can drive a fun and creative system like S.P.U.D.

Acknowledgement

The authors would like to express their sincere gratitude to Dr. Mike Borowczak for his guidance and support throughout this project. They also wish to thank Dr. Sazadur Rahman for recommending the use of the UltraEmbedded processor core, and Dr. Chung Yong Chan for his valuable feedback and insights.

References

[1] UltraEmbedded, "riscv-soc," GitHub. https://github.com/ultraembedded/riscv_soc (Accessed: Sept. 2025).

[2] F. Laterza, V. Terry, M. Moussa, and G. Saborido, "SPUD Project Repositories: spud_riscv_soc, spud_env, spud_rv32i_tools," GitHub. https://github.com/SD-SPUD (Accessed: Oct. 2025).

[3] R. Schlaikjer, "Multiplexed RGB LED Panels," Rhye.org. https://rhye.org/post/fpgas-2-led-panel-display/ (Accessed: Oct. 2025).