Fall 10-25-2024

# Prototyping Interactive Tactile Digital Logic Simulations: A Hybrid Approach

Logan Bateman
lbateman@southern.edu

# PROTOTYPING INTERACTIVE TACTILE DIGITAL LOGIC SIMULATIONS: A HYBRID APPROACH

Approved by:

[illegible]

Professor Richard Halterman, Ph.D., Adviser

[illegible]

Professor Tyson Hall, Ph.D., Adviser

[illegible]

Professor Robert Ordóñez

Date Approved 2024-10-14

# PROTOTYPING INTERACTIVE TACTILE DIGITAL LOGIC SIMULATIONS: A HYBRID APPROACH

by

Logan Bateman

A PROJECT DEFENSE

Presented to the Faculty of

The School of Computing at the Southern Adventist University

In Partial Fulfillment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Professor Halterman and Professor Hall

Collegedale, Tennessee

October, 2024

# PROTOTYPING INTERACTIVE TACTILE DIGITAL LOGIC SIMULATIONS: A HYBRID APPROACH

Logan Bateman, M.S.
Southern Adventist University, 2024

Advisers: Richard Halterman, Ph.D. and Tyson Hall, Ph.D.

Tactile exhibits are common in museums and on the walls of university halls. However, few (if any) tools exist for creating tactile exhibits for teaching digital logic or computing concepts. This project implemented a framework for creating tactile digital logic simulation exhibits, with a focus on rapid prototyping and distributed architecture. Prototyping allows for fast iteration, with the ability to simulate unlimited hardware components such as buttons, light emitting diodes (LEDs), and other input or output devices. Through the abstraction of implementations and a distributed communication protocol, switching to real hardware is seamless and works in tandem with simulated hardware. The framework is demonstrated with a central processing unit (CPU) exhibit.

# Contents

# List of Figures

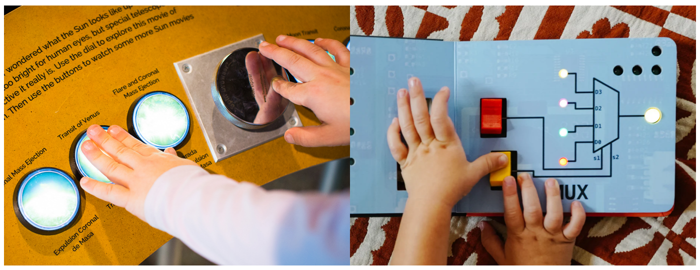
# List of Tables

# Chapter 1

# Introduction

Digital logic simulators which visualize and allow modification of the state of a circuit are prevalent. Their interfaces often use a mouse and keyboard to interact with the simulation. However, this approach may not always be the desired interaction, especially in a museum or other exhibit setting, which may benefit from a tactile interface with dedicated hardware input and output devices, as demonstrated in Figure 1.1.

The design and production of hardware-backed interactive interfaces can be time-consuming and costly, especially when a design requires many iterations to achieve the desired aesthetics and functionality. Iterations of the design can be slow when adding, removing, or modifying hardware.



(a) (b)

Figure 1.1: A tactile museum exhibit with a buttons and a rotating knob on the left (a). A Computer Engineering for Big Babies book showing a multiplexer on the right (b).

We discuss development of a framework and workflow for creating interactive digital logic simulations using a distributed architecture and pre-existing computer-aided design (CAD) tools. Rapid iteration was accomplished using simulated hardware. The underlying software for simulated hardware interaction was written in Rust, aiming for cross-platform support (with an emphasis on Linux). We extended an existing digital logic simulator to enable communication with hardware.

The project used a modular approach to software development, where each module only handled its own concerns and could be implemented with different languages or underlying design patterns. The modular approach resulted in standalone tools (such as the extended digital logic simulator and the hardware simulator) that can be repurposed elsewhere.

Interaction between digital logic and outside hardware (e.g., sensors and actuators) was accomplished using distributed communication concepts such as those employed by software that runs on robots using ROS (Robot Operating System) [1].

As a proof of concept, a tactile and interactive visualization of a CPU was simulated, with the ability for a user to manipulate aspects of the simulation via buttons and knobs. Wires were illuminated with LEDs to indicate their values, with seven-segment liquid-crystal display (LCD) and character displays used to show values of more than one bit. Tables on simulated displays showed the state of memory and register values were shown with LCD character displays. This visualization could eventually be deployed with real hardware components but the scope of this project was limited to simulated hardware. A small demonstration of real hardware interaction was implemented to demonstrate the feasibility of hardware-based deployments.

We explore background material in Chapter 2, explain the project implementation in Chapter 3, describe testing and evaluation methodology with results in Chapter 4, and summarize the project in Chapter 5.

# Chapter 2

# Background

The development of a workflow and accompanying software for tactile interaction with digital logic simulations required a review of several domains. The software had to simulate digital logic and meet reasonable performance, usability, and extensibility requirements. The simulator had to be able to communicate with simulated or physical tactile interface hardware. The simulated hardware had to respond to the user's input with acceptable latency. To demonstrate the completed project, a practical and educational design for a CPU was chosen from among several options. In this section, we review digital logic simulation software, distributed communication protocols, system feedback latency expectations, CPU instruction set architectures (ISAs), and competing solutions which accomplish a subset of the project goals.

## 2.1 Digital Logic Simulation

Broadly, there are two types of digital logic simulations: functional and timing. Functional simulations do not include the timing information necessary to detect problems such as races, hazards, and spikes, while timing simulations do [2]. There are two main methods of implementing digital logic simulations, via code-generation (compiling a circuit to machine instructions), or by interpreted execution (interpreting a circuit and simulating its behavior) which is often referred to as an event-driven simulator [3]. An example of a simulator that uses compilation is G Hardware Design Language (GHDL), which takes VHSIC Hardware Description Language (VHDL) code and generates code, allowing for native execution. This enables fast simulation but requires a compilation step [4]. With an interpreter-based simulator like Logisim-Evolution [5], the simulation can be modified at runtime without a compilation step.

Graphical editing, headless operation, and open-source licensing are important features of a digital logic simulator to use for this project. Headless operation means that the software can be invoked through a command-line interface and run without displaying a graphical user interface (GUI). Several digital logic simulators are compared in Table 2.1.

Table 2.1: Digital Logic Simulators

| **Simulator** | **License** | **GUI Editing** | **Headless** |
|---|---|---|---|
| CircuitVerse [6] | Open | Yes | No |
| Logisim-Evolution [5] | Open | Yes | Yes |
| GHDL [4] | Open | No | Yes |
| ModelSim [7] | Proprietary | No | Yes[1] |
| Turing Complete [8] | Proprietary | Yes | No |
| Digital [9] | Open | Yes | Yes[2] |

It should be noted that simulators such as GHDL and ModelSim are focused on circuit verification while the others have a more pedagogical focus. Likely for this reason, the former two do not have a graphical user interface (GUI) for editing circuits. While not as critical for this project, performance is also a consideration. Between Logisim-Evolution and Digital, the latter performs better, according Tom Niget, who built an ARM CPU in Digital and found that in Logisim he could only simulate his circuit at up to 60 Hz while

[1]ModelSim instructions to perform a functional simulation with command-line commands

[2]Headless operation implemented recently as part of issue 1246.

Digital could reach 20 kHz [10].

## 2.2 Distributed Communication Protocols

Using a distributed system enables the use of multiple programming languages and inherent flexibility through the abstraction of peer implementation, where each peer is like a black box that can be swapped out for another that abides by the same contract. This ability is important to be able to swap simulated hardware for real hardware. Multiple protocols exist for distributed robotics, IoT, or data processing that could be used for this project. Peer discovery, low-latency, ease-of-use, and microcontroller support were important considerations. There are different abstraction levels for solutions providing communication. For example, tools like Robot Operating System 2 (ROS2) [11] provide a high level of abstraction, and tools like ZeroMQ [12] provide a lower level of abstraction. The framework implemented did not need the extra abstractions and robotics specific libraries provided by higher level frameworks like ROS2, so lower-level tools were explored, including ZeroMQ, Data Distribution Service (DDS), Zenoh, and MQ Telemetry Transport (MQTT). A summary of these protocols is shown in Table 2.2.

Table 2.2: Distributed Communication Protocols

| **Protocol** | **Auto-Discovery** | **Latency** | **MCU Support** |
|---|---|---|---|
| ZeroMQ [12, 13, 14, 15] | Manual | 60 $\mu s$ | Limited official support |
| DDS [16, 17, 18] | Yes | 38 $\mu s$ | Yes, DDS-XRCE |
| Zenoh [19, 20, 18] | Yes | 41 $\mu s$ | Yes, Zenoh-pico |
| MQTT [21, 22, 23, 18] | Manual | 45 $\mu s$ | Yes |

ZeroMQ [12] is a mature networking library with extensive documentation and language support. It allows multiple communication patterns such as publish/ subscribe and request/reply. Peer discovery is not built in, but a user-implemented approach is explained in their documentation [13]. In a latency test with ZeroMQ v4.3.2 and a 100 Gbps fiber connection, the latency has been measured to be below 60 $\mu s$ for messages under 1 KB [14]. ZeroMQ does not appear to have a specialized implementation for microcontrollers; however, there is a minimal C library implementation that was last updated in 2016 [15].

DDS is a “middleware protocol and API standard” that is peer-to-peer and low latency, handling QoS (Quality of Service), reliability, and discovery [16]. It runs on top of multiple underlying transport protocols, with multiple implementations including Fast-DDS [24], Connext [25], OpenDDS [26], and CycloneDDS [27]. DDS is the standard used in the default middlewares of ROS2 [28]. For constrained devices such as microcontrollers, a specification called DDS-XRCE (DDS for eXtremely Resource Constrained Environments) exists, which allows clients to talk in a DDS network through an XRCE Agent [17]. A performance study from Liang et al. comparing multiple protocols found CycloneDDS performed with a latency of 38 $\mu s$ over a 100 Gbps link [18].

Zenoh is a modern communication protocol and library designed to overcome shortcomings of prior protocols. Its documentation is not as extensive as that for ZeroMQ, but there is adequate API documentation for the supported programming languages (Rust, C, C++, Python, Kotlin, and Java) [19]. The protocol works in both a brokered or peer-to-peer setting, with auto-discovery supported. Zenoh has an officially-supported implementation for microcontrollers called Zenoh-Pico [20], which includes support for Arduino, FreeRTOS (Free Real-Time Operating System), and Zephyr. The previously referenced performance study from Liang et al. found Zenoh performed with a latency of 41 $\mu s$ for brokered communication and 16 $\mu s$ for peer-to-peer communication over a 100 Gbps link [18].

MQTT is a publish/subscribe IoT messaging protocol using a client/broker architecture (as opposed to peer-to-peer) [21]. It has many community developed client and server implementations, with client libraries for Arduino and Mbed OS [22]. Automatic discovery by clients of the broker (Server) seems possible, but not always supported or recommended [23]. Again, the performance study from Liang et al. found MQTT performed with a latency of

45 $\mu s$ over a 100 Gbps link [18].

All the options reviewed boast low latency and are fairly easy to use. However, MQTT and Zenoh seem to have the best microcontroller support, and Zenoh and DDS seem to have the best built-in discovery support.

## 2.3 System Feedback Latency

If perceived system latency is too high, it can negatively impact user experience. Christiane et al. found that the perceived latency of feedback from an input depends largely on the modality of the feedback (whether it is visual, audio, or tactile) [29]. Also, whether more or less delay is acceptable depends on the complexity of the task being performed [29]. Kaaresoj et al. found that users can detect fairly small amounts of feedback latency and suggest specific guidelines per feedback modality: 5 to 50 ms for tactile feedback, 20 to 70 ms for audio feedback, and 30 to 85 ms for visual feedback [30].

## 2.4 CPU Instruction Set Architectures (ISAs)

Several different CPU ISAs exist that were options for this project's demo, each with their own use cases and complexities. Several concerns taken into consideration when choosing an ISA to implement and visualize were: licensing, documentation, and complexity. Generally, ISA's can be split into 2 major groups, RISC and CISC. RISC (Reduced Instruction Set Computer) ISAs are generally simpler, with smaller instruction sets, and CISC (Complex Instruction Set Computer) ISAs are generally more complex, with larger instruction sets. The potential ISAs reviewed were RISC-V, ARM, MIPS, and x86. All of these ISAs are common in computer architecture literature (e.g., Patterson and Hennessy books [31, 32]), with x86 processors also reviewed because of their popularity in consumer products.

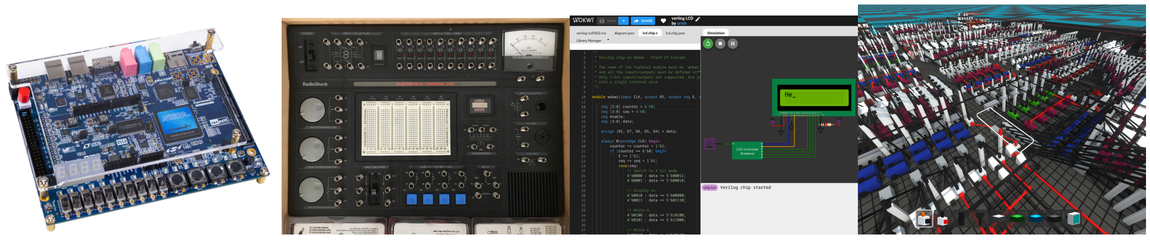
### 2.4.1 Licensing

The only open source ISA that was considered was RISC-V, with the others being proprietary. ARM proprietary ISA and intellectual property (IP) is licensed requiring a membership or subscription and royalties paid per unit [33]. Using the x86 ISA will likely require licensing or royalties to Intel or AMD [34]. MIPS licensing stance is currently unclear, as there is an article from 2018 saying that MIPS ISA was going to be available royalty-free [35], but official documentation on this is difficult to procure. While use of ARM, MIPS, or x86 for educational purposes may be allowed, the legal implications are unclear, especially if patented techniques are required to implement specific instructions. Building a compatible processor without licensing may require using a "clean room" approach such as what was employed by Compaq to reverse engineer the IBM PC [36].

### 2.4.2 Complexity

Determining the number of instructions in an ISA is not a simple task for many ISAs, especially the ones which are not open. Patterson and Hennessy in the ARM and MIPS editions of their book "Computer Organization and Design" describe core instruction subsets in their reference "green sheets" that have 36 (ARM "LEGv8") and 31 (MIPS) instructions respectively [31, 32]. Any arbitrary subset can be implemented; however, compiler support may be limited for a subset of an ISA.

Each ISA generally starts with a base set of instructions and then adds extensions to the base ISA to add more functionality (e.g., floating point arithmetic). The RISC-V approach to ISA design is to provide minimal base ISAs with many optional extensions. The standard minimal ISA is the integer base set for 32 bits, RV32I, which only contains 40 instructions [37].



(a) (b) (c) (d)

Figure 2.1: Competing solutions that satisfy a subset of the project's goals are FPGA development boards (a), electronics learning labs (b), and software such as Wokwi (c) and LogicWorld (d).

## 2.5 Competitive Analysis

Several products and solutions exist that exhibit some of the same goals as this project. There does not seem to be anything that combines all the features. Categorically, there are several types of competing solutions including FPGA (Field Programmable Gate Array) development boards, electronics learning labs, and 2D digital logic circuit/hardware simulators.

### 2.5.1 Field Programmable Gate Array (FPGA) Development Boards

FPGA development boards are used to prototype digital logic circuits, ranging from simple experiments and finite state machines to full CPU's. FPGA development boards often have buttons, switches, LEDs, seven-segment displays and peripheral ports. One such example is the Cyclone V GX Starter Kit [38], which has 18 LEDs, 10 slide switches, four seven-segment displays built-in, and GPIO pins.

Users can design digital logic circuits in a hardware description language (HDL) such as VHDL or Verilog, and then program the FPGA with the design. The design can then be tested and debugged on the FPGA development board, with users immediately able to use the built-in peripherals to interact with the design.

### 2.5.2 Electronics Learning Labs

Electronics learning labs such as those from Elenco and legacy Radioshack provide a way to learn about electronics and digital logic. They usually include a variety of fixed input and output components such as buttons, switches, LEDs, and seven-segment displays. Some also include a breadboard instead of only fixed components. The user can wire up the components to create circuits and immediately test them. However, the user is limited to creating circuits with the components they have available and the space on the breadboard. Thus creating larger circuits, such as a CPU, is not feasible.

### 2.5.3 Software

Several software solutions such as Wokwi and LogicWorld exhibit features that satisfy some of the goals of this project (e.g., 3D interface, simulated hardware, digital logic simulation, GUI based digital logic editing).

Wokwi [39] simulates various microcontrollers and attachable 2D peripherals. It also has experimental support for using a hardware description language to define chip behavior. An example project of controlling an LCD with Verilog demonstrates this ability [40]. However, it does not include a way to graphically edit a circuit. Also, there does not seem to be a way to hide the implementation and only show the user the interface (e.g., buttons, LEDs, LCD character displays).

LogicWorld [41] is a 3D digital logic simulator that allows creation of digital logic circuits and simulation of their behavior. It operates in a first-person view, where components are placed and wired together. However, it does not have a way to encapsulate a circuit into a single component for reuse, instead opting for a copy and paste method where "sub-circuits" remain full-size and are not abstracted. This makes the construction of large circuits challenging. Additionally, because both the circuit design and input/output are in the same visual scene, separating the interface from the logic is not straightforward.

# Chapter 3

# Implementation

The project's goals were accomplished by extending Digital [9] to add Zenoh communication and creating a hardware simulator with the Bevy [42] game engine. The hardware simulator reads a 3D glTF scene produced with Blender [43] or other modeling software, annotated with extra fields to define behavior and Zenoh topic keys. To test the software, we designed a RISC-V CPU in Digital, along with a 3D scene in Blender, for simulated hardware. To test hardware interoperability, a simple hardware interaction example was created with a Raspberry Pi Pico and an STM32 Nucleo F466RE. We will discuss the requirements, approach, tasks, and final deliverables in the subsequent sections.

## 3.1 Summary of Requirements

The project implemented a prototyping workflow comprised of multiple components running together (digital logic simulator, simulated hardware, and firmware on physical hardware). There are several classes of "users" including: simulation designers, operators, and exhibit users.

Simulation designers are able to design a circuit using Digital [9]. After or during design, Zenoh publishers, Zenoh subscribers, Zenoh registers, and Zenoh memories can be added, allowing external interface with hardware or other peers (entities which want to interact with the simulation).

Simulation designers are able to design simulated hardware as a 3D scene in Blender (or other modeling software that can export glTF), with extra annotations on objects to define behavior. A hardware simulator loads the 3D scene and behavior annotations, which is used to set up communication with the digital logic simulation. The interaction setup consists of allowing user interaction on annotated objects and external mutation (e.g., light on/off, movement) of annotated objects. Interactions (sensors) that are supported are buttons (momentary and toggle) and radial knobs. Supported actuators are seven-segment displays, LEDs, LCD character displays, tables, memory mapped displays, and speakers.

The operator is able to run the simulation by starting Digital from the command-line with a specific project file, and subsequently run the hardware simulator, specifying a 3D scene file. The operator is able to run the digital logic simulation and hardware simulator on the same computer or on separate computers, as long as they are able to communicate over a network.

The exhibit user is able to interact with sensors in the hardware simulator to influence the simulation. The supported interactions are defined by the logic simulation and simulated hardware designers.

The exhibit demonstrating the capability of the project is a simulation and interactive visualization of a RISC-V CPU. The hardware interactive visualization shows a diagram of the processor architecture, with lights and seven-segment/character displays showing the value of wires/busses. A knob allows for specifying the clock rate, and a button allows for manually advancing the clock. A reset button restarts execution. Character displays and a table display the values of registers and memory.

One of the project's goals was to allow hardware interactivity, so in place of implementing the entire CPU exhibit interface with hardware, a small demonstration of hardware interactivity was implemented, showing the feasibility of a hybrid or fully hardware design. This was accomplished using a Raspberry Pi Pico and STM32 Nucleo F466RE, which each communicate to a Zenoh router over serial ports.

## 3.2 Approach

The hardware simulator for the project was written in Rust to take advantage of its powerful type system, performance, and safety. Digital was extended to support Zenoh communication using the available Java library for Zenoh. In the hardware simulator, the Entity-Component System [44] design pattern was employed using the Bevy [42] game engine. For communication and abstraction of connected software components, the publish/subscribe pattern was used with the Zenoh protocol. Beyond publishing and subscribing, we also used Zenoh's capability for querying [45] in this project.

Design of the RISC-V processor was accomplished in Digital, referencing other RISC-V CPU designs and the RISC-V specification [37]. The RV32I instruction set was used, since it only contains about 40 instructions, and is targetable by compilers. Also, supervisor mode, ECALL, EBREAK, and Control Status Registers (CSRs) were not implemented.

In order to design a 3D model for visualizing and interacting with the simulated RISC-V CPU, Blender was used. Royalty free models of the needed components were used when available, modified to fit the project's needs.

In the logic simulator and hardware simulator, Zenoh messages are sent on change of sensors (from the hardware simulator side) and on change of publisher input pin (from the logic simulation side). In the logic simulation, Zenoh memories transmit changes to blocks of memory, and also allow setting regions of memory from received messages. On the simulated hardware side, sensors, actuators, and other simulated hardware are annotated

Table 3.1: Tools and Libraries

| Tool | Purpose |
|---|---|
| Bevy [42] | Hardware simulator for interaction with simulated sensors and actuators |
| bevy_mod_picking [46] | Picking for 3D objects to allow clicking on buttons and dragging knobs. |
| bevy_pancam [47] | Bevy plugin for pan, zoom, and rotation using cursor or pointer. |
| Blender [43] | 3D modeling of the visualization |
| Inkscape [48] | Creating a 2D diagram of the CPU architecture as an SVG |
| Digital [9] | Designing the CPU and other circuits |
| Zenoh [45] | Distributed communication between digital logic and hardware |
| glTF [49] | Simulated hardware 3D model and behavior storage |

(via "Custom Properties" in Blender) with fields for the name of a Zenoh topic to use for communication, the type of simulated hardware, and any configuration parameters needed for that hardware.

A visual diagram of how project components interact at runtime with an exhibit user can be seen in Figure 3.1. All project components communicate with each other using Zenoh.

Several open source tools and libraries were employed to accomplish the goals of this project in a reasonable time. Some of the tools, libraries, or formats which were used are listed in Table 3.1.

## 3.3 Task Delineation

This project was divided into several tasks, corresponding to features explained at a high level in Section 3.2. The tasks along with their estimated completion times and estimated times spent are found in Table 3.2, and are described in more detail in the following sections.

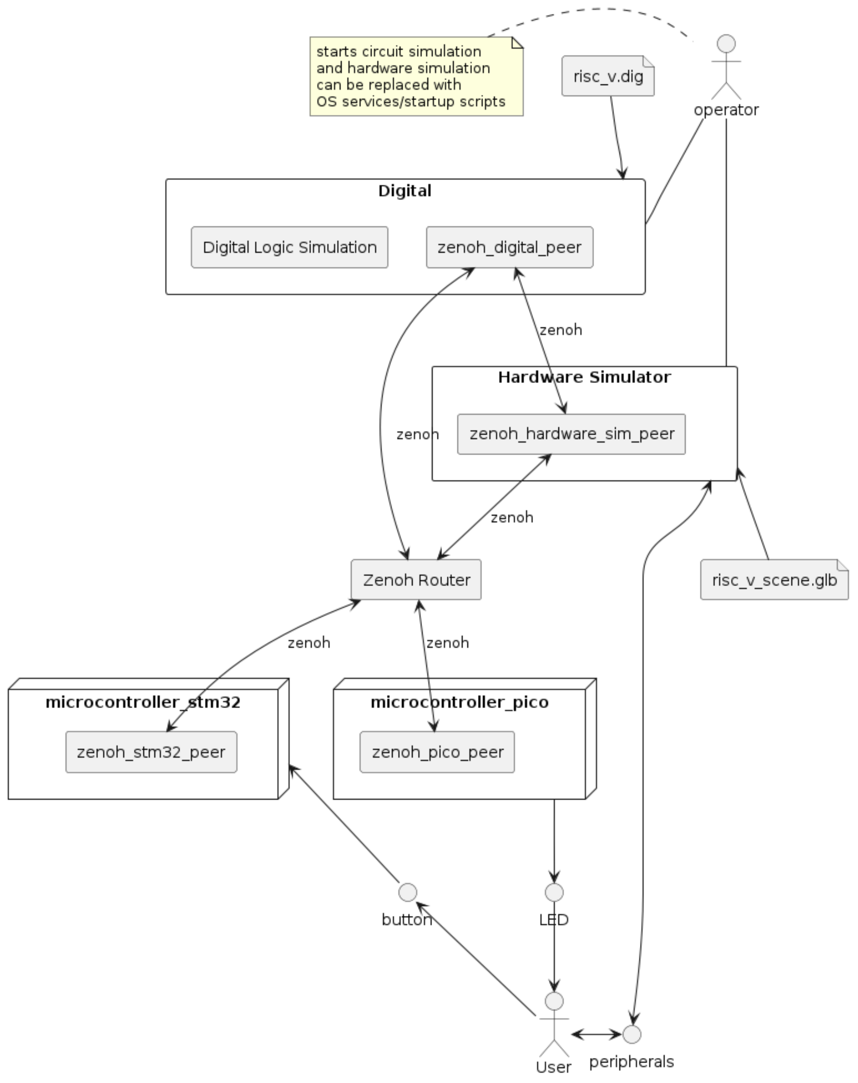


Figure 3.1: During runtime, the exhibit user interacts with either the hardware simulator via computer peripherals (mouse and keyboard) or the microcontrollers via an attached sensor or actuator (button or LED). Each of these components communicates with the extended Digital software via Zenoh to interact with the digital logic simulation. In the diagram, Digital and the hardware simulator consume project artifacts created by the exhibit designer. The `risc_v.dig` file contains the top-level circuit design, to be simulated by Digital. The `risc_v_scene.glb` file contains the 3D model of the hardware simulation, including object annotations, to be consumed by the hardware simulator. An operator starts the Digital simulation and hardware simulator as a setup procedure.

Table 3.2: Task Delineation

| **Task** | **Pre-Estimated Hours Total** | **Estimated Hours Spent** |
|---|---|---|
| Digital Zenoh Extensions (3.3.1) | 65 | 43 |
| RISC-V Processor Design (3.3.2) | 105 | 74 |
| Hardware Simulator (3.3.3) | 133 | 161 |
| CPU Diagram Visualization 3D Scene (3.3.4) | 65 | 38 |
| Hardware Example (3.3.5) | 24 | 49.5 |
| Documentation, Testing, And Bug Fixes (3.3.6) | 24 | 66 |
| **Total** | 416 | 431.5 |

### 3.3.1 Digital Zenoh Extensions

Digital was used for circuit design and simulation, but it did not initially support Zenoh communication. To allow for communication with Zenoh, new elements were added to the component library. These included Zenoh publishers, Zenoh subscribers, Zenoh memories, and Zenoh registers. The `APPLICATION_OCTET_STREAM` encoding type was used for all Zenoh message payloads.

The Zenoh publisher component publishes a value to a Zenoh topic whenever the value changes. It has a single input pin of a specified bit width, which provides the value to publish. The Zenoh subscriber component subscribes via Zenoh and sets its single output pin to the last value received. Both publishers and subscribers allow for configuration of the number of bits in the value, as well as the topic key to use for communication. Values are published as a 64-bit integers, big-endian encoded, regardless of the data bits setting of the publisher. Values received by subscribers are allowed to be 32 or 64 bits, big-endian encoded.

Zenoh memories work like other memories in Digital, except that they publish changes to blocks of memory, and also allow setting regions of memory from received messages. The memory is also queryable for first-time clients to receive the current state of the memory. The topics and formats of messages used for the Queryables, Publishers, and Subscribers of Zenoh memories are shown in Table 3.3.

Table 3.3: Zenoh Memory Topics

| Topic | Type | Description | Format |
|---|---|---|---|
| `{baseKey}/info` | Queryable | Returns the word width and size in words. | `Size:  4 bytes`<br>`Bits:  4 bytes` |
| `{baseKey}/changes` | Publisher | Publishes any change to the RAM, as a range. | `Address:  4 bytes`<br>`Length:  4 bytes`<br>`Data:  length * bytesPerWord` |
| `{baseKey}/set` | Subscriber | Subscribes to memory change messages, and mutates the memory. | `Size:  4 bytes`<br>`Bits:  4 bytes` |
| `{baseKey}/get` | Queryable | Returns a requested region of memory. | `Query`<br>`Size:  4 bytes`<br>`Bits:  4 bytes`<br>`Response`<br>`Address:  4 bytes`<br>`Length:  4 bytes`<br>`Data:  length * bytesPerWord` |

Zenoh registers work like other registers in Digital, except that they publish value changes (at `{baseKey}/changes`, where `{baseKey}` is the Key Expression configured by the user), and allow setting the register from received messages (at `{baseKey}/set`).

### 3.3.2 RISC-V Processor Design

The task of designing and simulating a RISC-V processor involved studying RISC-V ISAs, building out components in Digital, and testing the CPU against simple programs. After basic functionally was completed, further testing with RISC-V Architectural Testing Framework [50] was performed, further described in Section 4.4 of Chapter 4. Once the CPU worked satisfactorily and the simulated hardware design was underway, additional publishers were added and annotated as necessary, and registers and RAM were replaced with Zenoh memories. The finished circuit for the CPU can be seen in Figure 3.2.

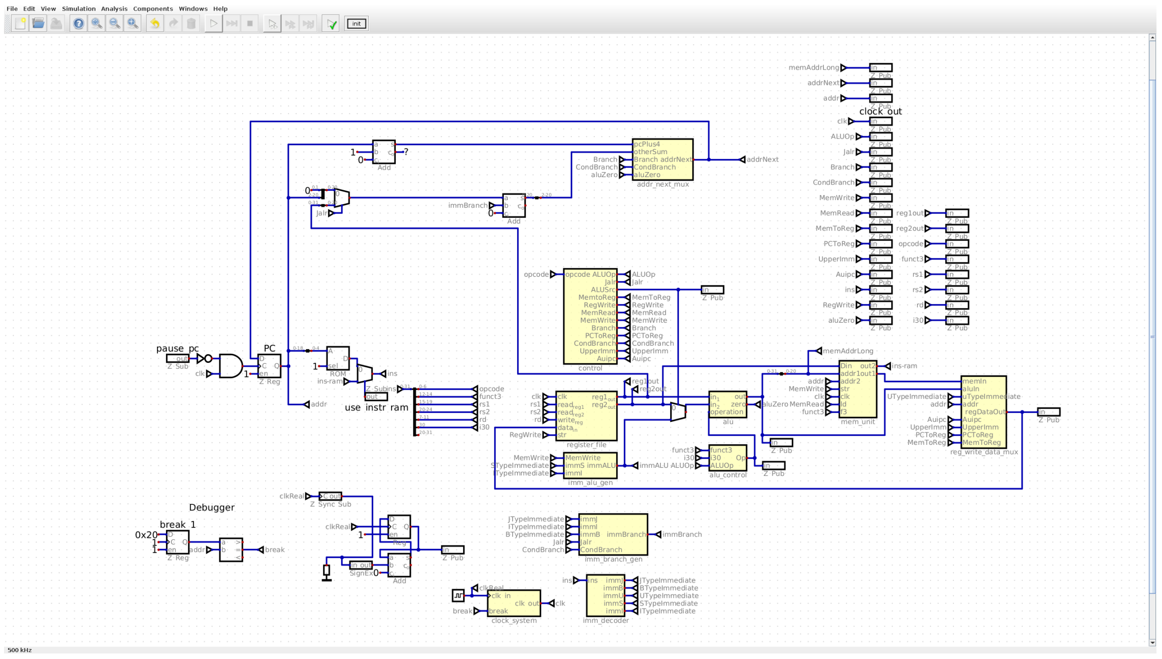


Figure 3.2: The RISC-V design, made up of several components, connected by wire and tunnels. For communication with the human interface, Zenoh publishers are prevalent in the top right, as well as other publishers and subscribers spread throughout, and even inside of some subcircuits.

## 3.3.3 Hardware Simulator

Implementing a hardware simulator was accomplished using the Bevy game engine to show a 3D scene. It consumes a scene in glTF format, with objects annotated for behavior and communication configuration. Behaviors for simulated hardware were implemented for the various supported input and output devices (e.g., LED's, buttons, dials). Basic communication over Zenoh was tested as well as integration testing of a simple circuit with Digital.

Implementation of simulated hardware that can be triggered by a pointer (mouse or touch) was accomplished using a state machine that changes based on the state of the pointer. Hardware entities were composed of ECS components with systems to handle their behavior. In the ECS paradigm, entities are just identifiers that have components (data attributes) attached. Systems read and mutate the components of entities. Besides implementing the behavior of input and output devices, panning and zooming was added to allow navigation of the 3D scene on any size screen.

Zenoh communication was implemented as a separate Bevy plugin used by the hardware simulator. The basic components to configure and use the Zenoh plugin were the `ZenohPublisherKey`, `ZenohPublisherBits`, `ZenohPublisher`, `ZenohSubscriberKey`, `ZenohSubscriberBits`, `ZenohSubscriber`, `ZenohMemoryKey`, `ZenohMemorySettings`, and `ZenohMemoryData` components. Once the right components existed on an entity to configure a publisher or subscriber, systems would query for the components and propogate changes. The Zenoh plugin was used to send and receive messages to and from the digital logic simulation.

### 3.3.4 CPU Diagram Visualization 3D Model

Creating a 3D model for the simulated hardware that interacts with the CPU is not directly a software engineering task, though important for the visualization and behavior design of the test CPU exhibit. Inkscape [48] was used to create/improve a flat diagram of the CPU architecture. Blender [43] was used to create a 3D scene that includes the scalable vector graphic (SVG) CPU diagram, and adds 3D models of sensors and actuators. Each component was annotated in Blender, to be exported glTF as “extras” as defined in Section 5.16 of the glTF 2.0 specification [49]. A screenshot of the finished scene can be seen in Figure 3.3.

### 3.3.5 Hardware Example

A hardware example was created using Raspberry Pi Pico and an STM32 Nucleo F466RE development boards. The microcontrollers run firmware that uses Zenoh-pico [20], a Zenoh

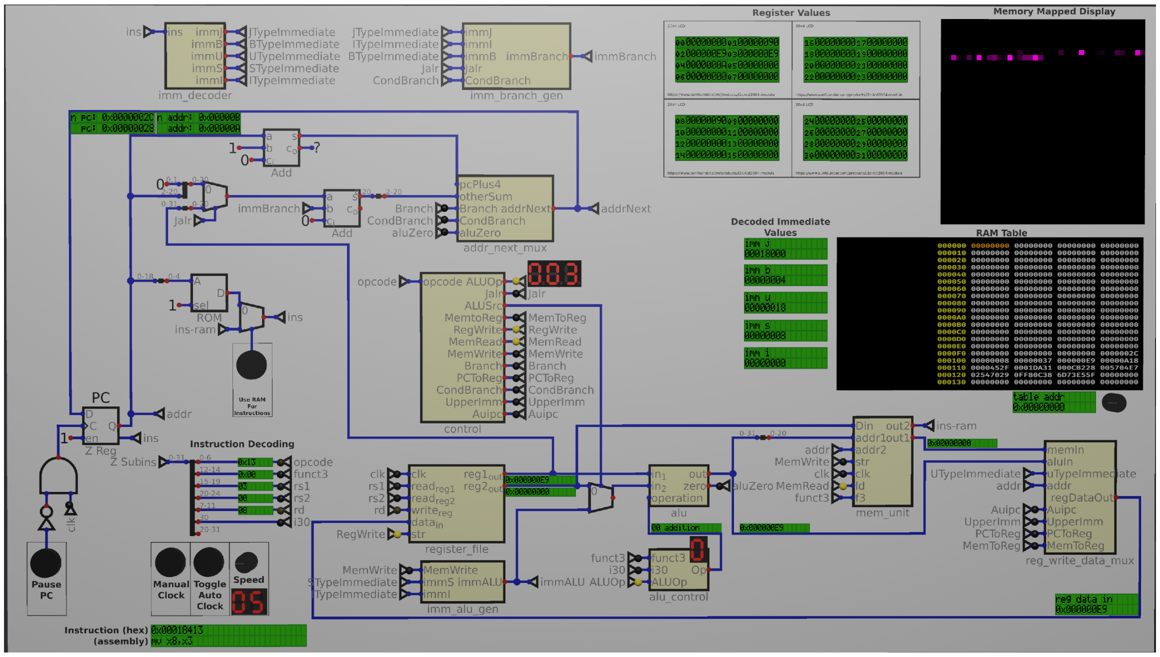


Figure 3.3: The simulated hardware scene for the RISC-V CPU, currently about to execute `mv x8, x3`, which is (`0x00018413`). The opcode is `0x13` with a funct3 of zero, meaning that the instruction is an ADDI (add immediate). The immediate value is zero. `RegWrite` is high, and the ALU operation is set to addition. The current PC is `0x00000028` and the next PC is `0x0000002c`, one instruction (4 bytes) ahead.

implementation for microcontrollers. The example included a button which triggers Zenoh communication, and an LED, which is controlled by observing and waiting for specific Zenoh communication.

### 3.3.6 Documentation, Testing, and Bug Fixes

Documentation, testing, and bug fixes were ongoing tasks throughout the project, with a thorough review and cleanup at the end. Documentation was written in Markdown, and exists as multiple markdown files that explain how to build, run, and use the project components. More specific documentation was included in `README.md` files per Rust crate or source repository. Outside testing for user experience and correctness (acceptance) was completed by volunteers. Bugs found as a result of testing were fixed afterwards.

## 3.4 Required Software and Hardware

### 3.4.1 Software

The hardware simulator and digital logic simulator were required to be executable on Linux. Compilation of the project and various components required the following tools:

- Rust compiler
- Java Development Kit (for compiling Digital)
- Blender [43] for creating 3D scenes for the hardware simulator
- Inkscape [48] for creating and modifying 2D diagrams of the CPU architecture to be used in the 3D scenes

### 3.4.2 Hardware

A Linux-capable computer (single board computer (SBC) or small desktop) was required to run the logic simulation and hardware simulation software. Also, a Raspberry Pi Pico and STM32 Nucleo F466RE were required for the hardware interaction example, running firmware using Zenoh-Pico [20].

## 3.5 Deliverables

- Project repository(s) with source code for all components
- Digital circuit files (.dig) for working implementation of RISC-V CPU
- 3D Scene file (.glb) for RISC-V CPU visualization to be consumed by hardware simulator

- Relevant assets (such as 3D models, persisted digital-logic simulation projects)
- Documentation for building and running the software components
- Final Project Report

# Chapter 4

# Testing & Evaluation Results

## 4.1 Acceptance Testing

To verify that project requirements have been met, a user acceptance test was performed. Test participants were given a premade 3D scene (.glb file) to use with the hardware simulator (see in Figure 4.1). They were asked to complete a set of objectives which required modifing a circuit in Digital using the new Zenoh communication components. During the test they were able to ask the test administrator any questions about how to use Digital, and how each of the newly added Zenoh components work.

Test participants were given step-by-step instructions for completing the test, attached in Appendix B.1. A survey was completed by the participants after the test, to gather feedback on the usability of the software. The survey consisted mainly of Yes/No questions, with a ratio of 90% or higher "Yes" responses being the goal. Also included were optional text feedback fields. A fillable form was created for the survey, to allow for easy data collection. The survey is included in Appendix B.2. The two participants who completed the test were able to complete the objectives successfully, and the survey results were positive, with a 100% "Yes" response rate to all Yes/No questions.

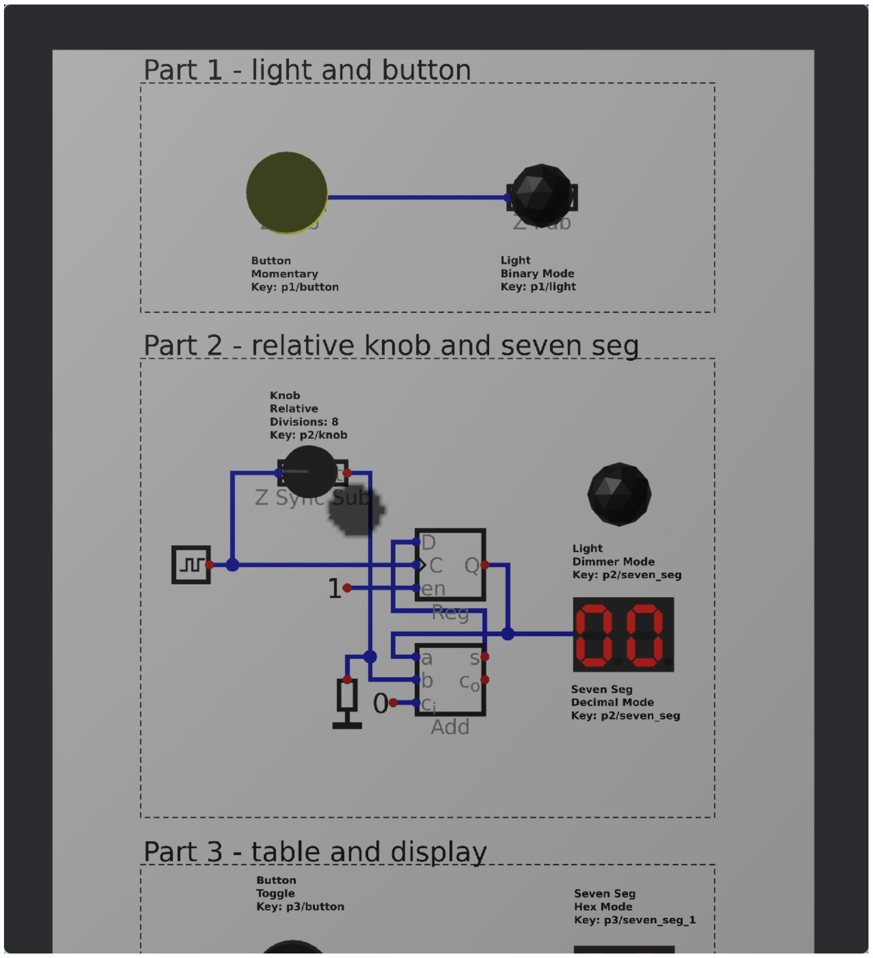
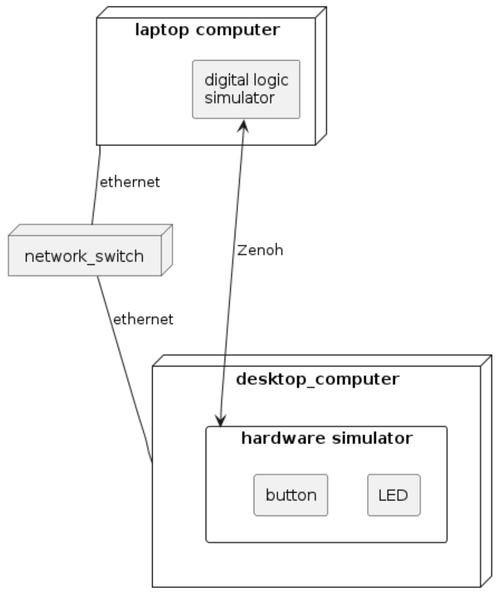


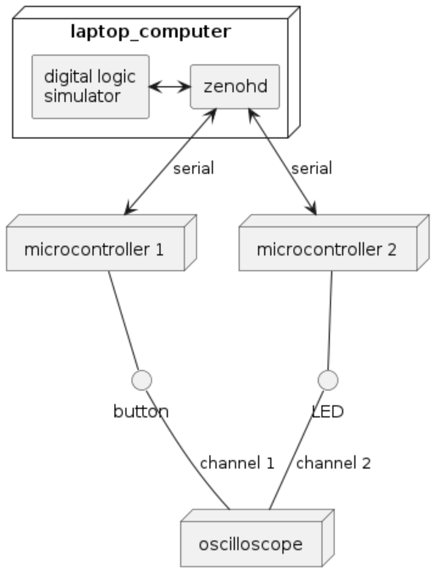
Figure 4.1: A portion of the simulated hardware for the acceptance test, showing in Part 1 a simple exhibit with a button and LED. The user was asked to modify the corresponding circuit in Digital allow the LED to turn on when the button is pressed.

## 4.2 System Testing

Minimizing feedback latency is critical for user interfaces, so that the user does not feel that the input mechanism or system is sluggish. Visual feedback and auditory feedback were implemented (audio only in simulated hardware). The expected latency between user interaction and actuation of the hardware or simulated hardware should be under 70 ms for audio feedback and 85 ms for visual feedback. This was tested using a simple round-trip latency test, using an oscilloscope for hardware, and logging in the simulated hardware, as shown in Figure 4.2.



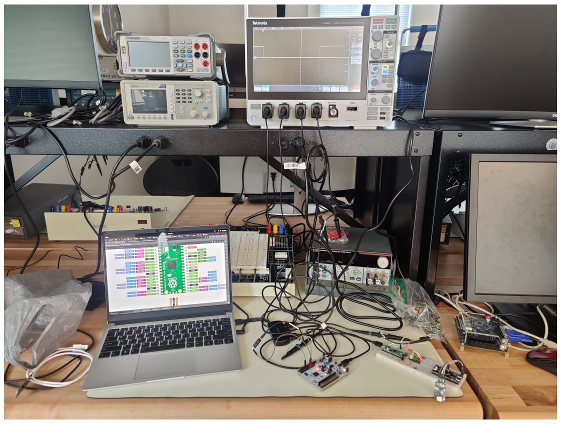
(a) Simulated Hardware Latency Test

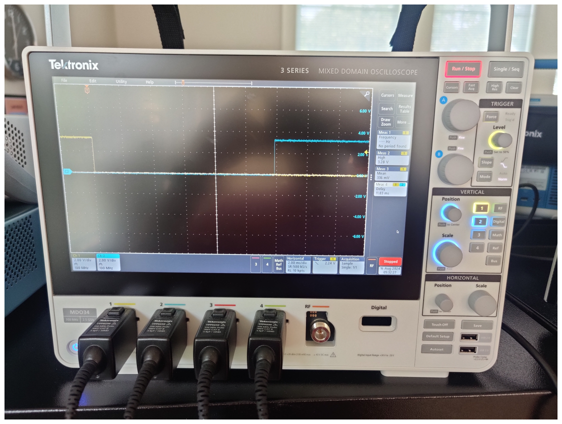


(b) Hardware Latency Test

Figure 4.2: Figures (a) and (b) show the experiment setup for testing round-trip latency in a simulated (a) and hardware (b) setting. In (a), there are 2 computers connected over a network with a switch. Both the digital logic simulator and the hardware simulator are Zenoh peers, which communicate directly through the network. In (b), the digital logic simulator is running on the single computer, which connects over Zenoh to a Zenoh router (zenohd). This Zenoh router is used to allow the microcontrollers using Zenoh-pico to connect over serial, so that the digital logic simulation and microcontroller can communicate. The oscilloscope is connected using 2 channels to an output pin on one microcontroller and the user button GPIO pin on another microcontroller, to be able to measure the time from button press, to light on.

Each test was performed 25 times, and the average was used to determine if the system met the performance expectations. The tests were considered successful since the average latencies were under the desired values. The average round trip latency for the hardware based test was 12.36 ms, and the average round trip latency for the simulated hardware test was 3.82 ms. Therefore, the platform meets feedback performance expectations. The full results are shown in Table C.1 of Appendix C. The experimental setup for the hardware test is shown in Figure 4.3.


(a) Hardware Wiring and Setup



(b) Oscilloscope Output

Figure 4.3: Figures (a) and (b) show the setup of the hardware latency test. In (a), the oscilloscope is connected using 2 channels, one to the input pin for the user button on a microcontroller, and the other to a output pin being set at the same time as the built-in LED. In (b), the oscilloscope is shown after the trigger has been set, showing the round-trip latency between the user button press and the LED turning on, by the time elapsed between the falling edge of channel 1, and the rising edge of channel 2.

## 4.3 Integration Testing

Since the project is made up of multiple components, integration testing was performed to ensure that the components work together as expected. To test the integration of components, a test project was used, consisting of circuit in Digital, a 3D scene in Blender, and a specialized test entry point for the hardware simulator. The hardware simulator entry point (test program) automated mutation of the simulated input hardware. The hardware simulator checked that the behavior of the hardware was as expected, given the simple test circuit being used.

The goal was to run automated tests 10 times to ensure consistent ability for the software to communicate properly. After preventing the tests from running in parallel, tests were able to be repeated 10 times without failure.

For the hardware example, integration testing was performed manually, visually observing the behavior of the hardware and the digital logic simulation and comparing the expected behavior to the observed behavior.

## 4.4 Module Testing

To test each component added to Digital individually, mock Zenoh peers were created to test communication and functionality. Peers sent Zenoh messages to the application and verified the behavior through messages received. The test goal was that repeated tests would work 9 out of 10 times. The tests were able to run 10 times without failure. For the hardware simulator, unit tests ran at various levels, effectively testing the modules.

Testing of the RISC-V CPU design was accomplished using Architecture Compatibility Tests (ACTs) with the RISC-V Compatibility Framework (RISCOF) [50]. It made use of the RISC-V CTG (Compliance Test Generator) [51] to generate tests for selected ISA subsets, RV32I for this project. These tests determined correctness by checking the signature of memory dumped after each test compared with the expected signature generated from the reference processor emulator. A test harness was created using Python to run the test programs and get the required output from the DUT (device-under-test).

This process using Zenoh consisted of publishing to various subscribers in the circuit that controlled the clock, the program counter, and source for instructions. Also, the test programs were set in the CPU's memory via Zenoh publications to the Zenoh memory component in the CPU. Figure 4.4 shows an example of part of the test harness, with a function to send a program to the CPU's memory. A 100% pass rate was achieved for the subset of RISC-V ISA tested, verifying adherence.

```python
def send_binary_to_simulator(binary: bytes):
  # reverse every 4 bytes
  binary = b''.join([binary[i:i+4][::-1] for i in range(0, len(binary), 4)])

  # Send BIN file via Zenoh to the simulator
  session.put("run/set", struct.pack(">Q", 0)) # stop simulator if running
  session.put("use_instr_ram", struct.pack(">Q", 1)) # use instruction RAM

  addr = 0
  data = binary
  bytes_per_word = 4
  buf = struct.pack('>i', addr) + struct.pack('>i', int(len(data) / bytes_per_word)) + data

  session.put("ram/set", buf)

  session.put("pc_reg/set", struct.pack(">Q", 0)) # set PC to 0
  session.put("speed_reg/set", struct.pack(">Q", 20)) # set speed to 20 (max)
  session.put("run/set", struct.pack(">Q", 1)) # start simulator
```

Figure 4.4: In this example code the simulated CPU is configured to use RAM for instructions, and the binary is sent to the RAM. The program counter is set to 0, the speed is set to 20 (fastest), and the simulator is started. The session object is a Zenoh session object.

## 4.5 Unit Testing

Unit testing of the hardware simulator was performed using the testing features built into the Rust programming language. These tests covered over 80% of the produced Rust code. The official Rust compiler documentattion describes two ways to generate code coverage reports, using `gcov` or LLVM's coverage instrumentation [52]. LLVM's coverage instrumentation was used (via Tarpaulin [53]) to determine the code coverage of the unit tests. Unit testing with Bevy ensured correctness of ECS systems by using applicable practices from Bevy documentation on how to test systems [54].

# Chapter 5

# Conclusion

Prototyping and construction of interactive digital logic circuit exhibits can be a challenging task, requiring a combination of software and hardware skills. This paper showed the implementation of a flexible and extensible workflow for prototyping exhibits, first with simulated hardware prior to physical hardware deployment. The solution shown is realized through the extension of Digital (digital logic simulator) to support Zenoh communication along with the implementation of a compatible hardware simulator. These allow the user to design a circuit, simulate the input/output hardware, and then incrementally swap out simulated hardware for physical hardware. To demonstrate the approach, a RISC-V CPU and corresponding simulated hardware were created, with a small physical hardware example.

Future improvements to the project may include performance and usability enhancements. One approach to improve performance may be to use an FPGA to implement the digital logic simulation, with a coprocessor for Zenoh communication. This would likely increase the performance of the simulation with some flexibility loss. For pedagogical visualizations of digital logic circuits, the performance afforded may not be as important. A usuability enhancement may be to combine the hardware visualization and digital logic simulation into a single application, where both can be designed in the same cohesive environment. Also, for portability, the ability to run the simulation and visualization in a web browser, using WebAssembly, would be a significant improvement.

# Appendix A

# Requirements Specification

## A.1 Hardware Requirements

- The digital logic simulation shall run on x86_64 processors, and optionally ARM64 processors.
- The hardware simulator shall run on x86_64 processors, and optionally ARM64 processors.
- The digital logic simulation and hardware simulator shall run on Linux.
- The example hardware firmware shall run on a Raspberry Pi Pico

## A.2 Application Requirements

### A.2.1 Digital Zenoh Extensions

1. Digital shall include a Zenoh publisher component, which publishes changes to the input pin to the configured topic.

2. Digital shall include a Zenoh subscriber component, which modifies its output pin based on received messages to the configured topic.

3. Digital shall include a synchronous Zenoh aware memory component, which allows observing and mutating memory through Zenoh.

4. Digital shall include a synchronous Zenoh aware register component, which allows observing and mutating the value through Zenoh.

5. Digital should optionally include a synchronous Zenoh publisher component, which publishes changes to the input pin to the configured topic, when the clock input is on the rising edge.

6. Digital should optionally include a synchronous Zenoh subscriber component, which buffers received messages to the configured topic, outputing the last value on the buffer on clock rising edge.

7. Digital Zenoh components shall properly unload/clean up when the simulation stops.

### A.2.2 RISC-V Processor Design

1. The processor design shall have an ALU.

2. The processor design shall have a register file.

3. The processor design shall have a control block.

4. The processor design shall implement RV32I.

5. The processor design shall have Zenoh publishers, subscribers, and memories inserted, for user interaction.

6. The processor shall have a clock with a variable speed.

7. The processor shall undergo RISC-V Architecture Compatibility Tests to ensure compliance to the RV32I specification.

### A.2.3 Hardware Simulator

1. The hardware simulator shall read glTF extra fields to specify communication and interaction behavior.

2. The hardware simulator shall include support for momentary buttons.

3. The hardware simulator shall include support for toggle buttons.

4. The hardware simulator shall include support for radial knobs.

5. The hardware simulator shall include support for seven-segment displays.

6. The hardware simulator shall include support for character displays.

7. The hardware simulator shall include support for a tables that show address of rows and values in decimal or hex, with the ability to highlight and scroll to a specific address.

8. The hardware simulator should optionally allow the user to zoom and pan in the 3D environment.

9. The hardware simulator should optionally include support for a “speaker” that can make sound at a specified frequency.

10. For testing, one simple example of each allowed sensor and actuator will be created with a separate glTF scene.

### A.2.4 CPU Diagram Visualization 3D Scene

1. The 3D scene of the CPU visualization shall have a CPU diagram created with Inkscape.
2. The 3D scene shall be laid out in Blender, with created 3D components and free assets.
3. The 3D scene shall have “extra” annotations on interactive components, to be consumable by the simulator.

### A.2.5 Zenoh Web Bridge (Optional)

1. The web bridge should produce a URL every 5 minutes with a code to allow authentication; this code should be communicated over Zenoh.
2. The web application should show the current state of registers.
3. The web application should show the current state of RAM.
4. The web application should show program memory, and where the PC is.
5. The web application should allow generic publishing to all available subscriber topics.

### A.2.6 Hardware Example

1. The hardware example shall use Zenoh-Pico for communication.
2. The hardware example shall include at least one button.
3. The hardware example shall include at least one led.

### A.2.7 Documentation, Testing, And Bug Fixes

1. A suite of projects shall be created to test integration between the logic simulation and the hardware simulation.

2. Documentation shall be produced for the project explaining how to build and use the created components. A `README.md` should be produced per new crate/repository.

3. Bugs found as a result of user testing shall be fixed.

# Appendix B

# Acceptance Test Materials

## B.1 Acceptance Test Instructions

# Acceptance Test Instructions

## Goals

1. Familiarize the user with the added components of the Zenoh enabled Digital (digital logic simulator).
2. Familiarize the user with the hardware simulator.
3. Gauge usability of Zenoh enabled Digital and the hardware simulator.

## Prerequisites

- Linux or Windows
- Java Runtime Environment

## Setup

1. Download Zenoh enabled Digital.jar
2. Start `Digital.jar` using `java -jar Digital.jar`
3. Open the acceptance test circuit `acceptance-test-dist.dig` in Digital
4. Download the hardware simulator `hardware_sim` (Linux) or `hardware_sim.exe` (Windows)`

   > Note: the hardware simulator may be detected as a virus by some antivirus software on Windows. This is a false positive. You can safely ignore this warning (and you may need to tell your antivirus to un-quarantine/keep it).

5. Start the hardware simulator using `./hardware_sim acceptance_test.glb` (Linux) or `hardware_sim.exe acceptance_test.glb` (Windows)

## Digital Crash Course

> Skip if you are already familiar with Digital

1. Digital is a digital circuit simulator that allows you to design and simulate digital circuits.
2. The play button starts the simulation, the stop button stops the simulation.
   1. No circuit changes can be made while the simulation is running.
   2. The simulation will fail to start if an input is not connected, or if there is a bit width mismatch.

3. Primary click and drag to select components and wires.
4. Primary click and release to select a component.
5. Primary click and release on a pin or empty space to start a wire
6. Primary click and release on another pin or wire to end a wire, or on empty space to add a corner.
7. When item(s) are selected, Ctrl-C to copy, Ctrl-V to paste, and Delete to delete.
8. Secondary click on a component to open the component menu (settings of a component).

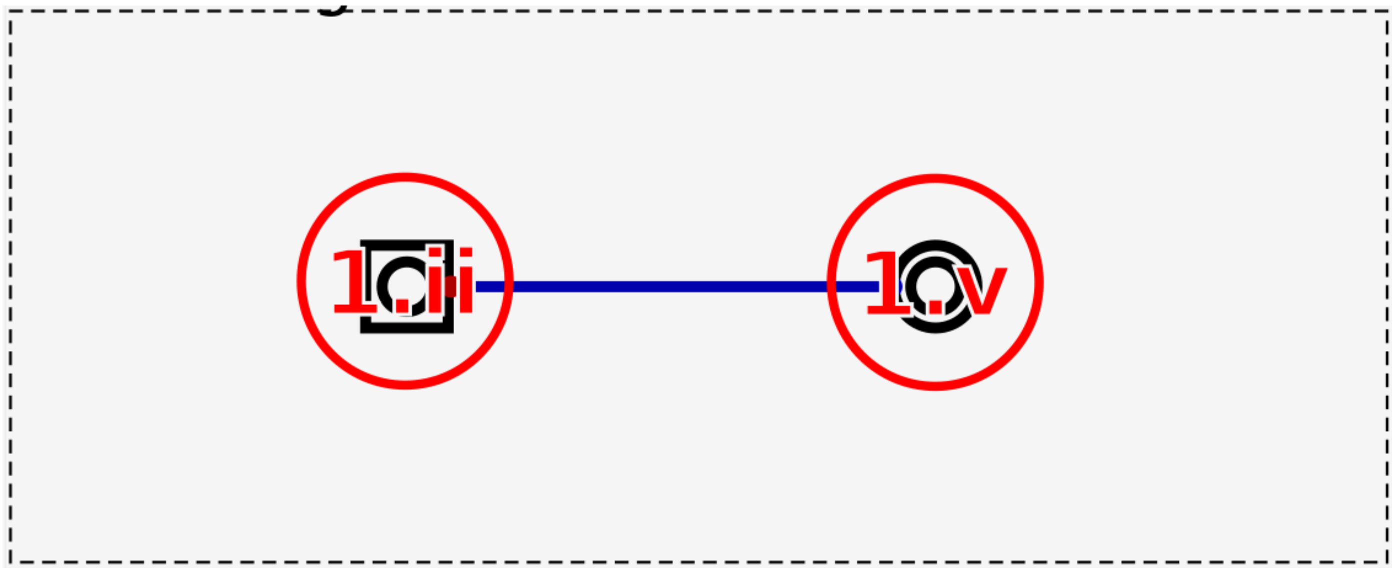
## Hardware Simulator Crash Course

1. The hardware simulator is a visualizer for various input and output devices.
2. The hardware simulator has one required positional argument, the path to glTF scene ( `.glb` or `.gltf` ) file.
3. Pan the camera by middle click and drag, zoom by scrolling, and rotate by right click and drag.
4. Primary click on input devices to interact with them.
   i. Click on buttons to press them.
   ii. Click and drag on knobs to the left and right to rotate.

## Test Instructions

Start the hardware visualizer by running the following command: `./hardware_vizualizer acceptance_test_hardware.glb` .

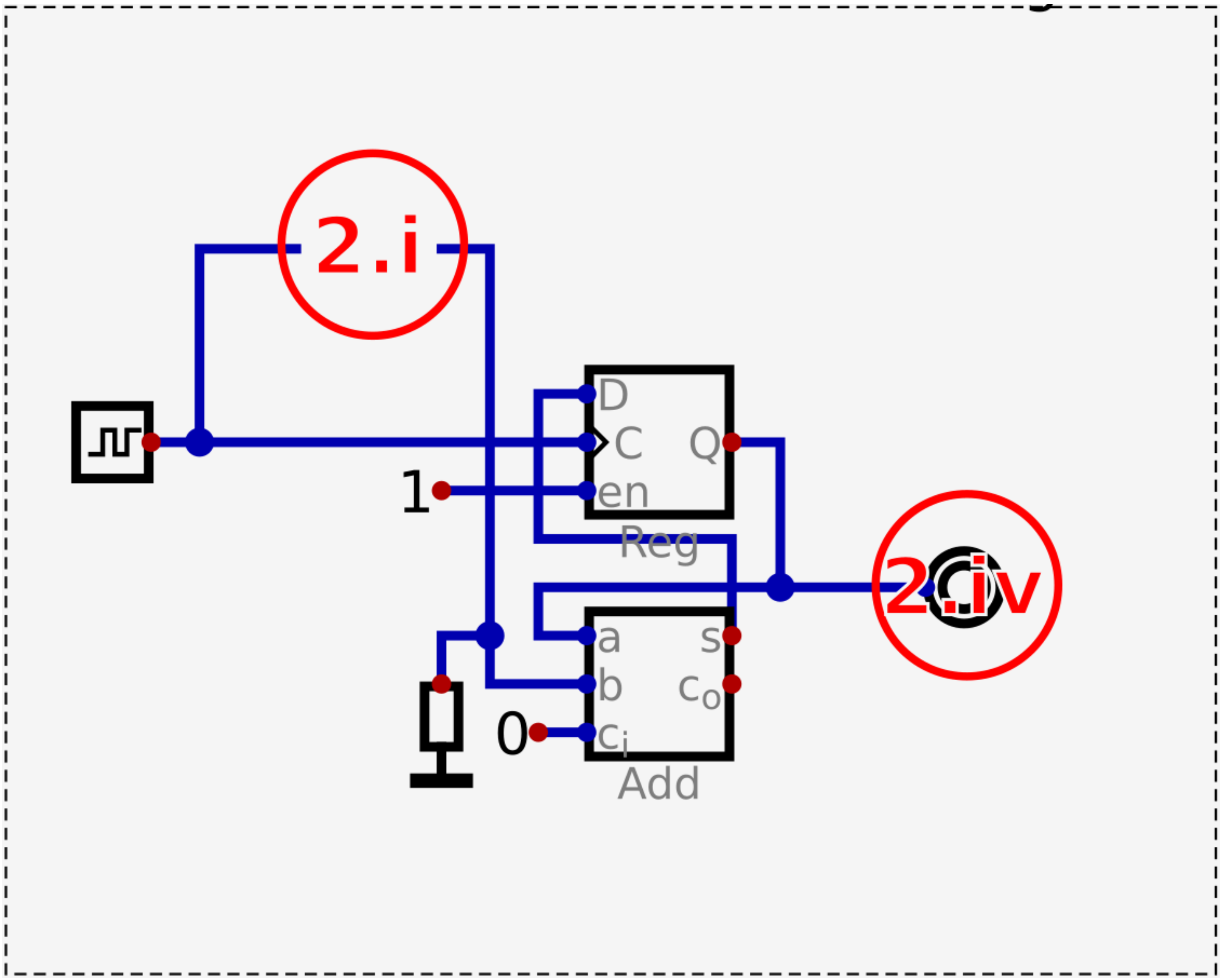
1. Basic Button and Light



   i. Primary click on the existing input and delete it.
   ii. Insert a Zenoh Subscriber in its place (from Components -> IO -> Peripherals -> Zenoh Subscriber).
   iii. Secondary click on the Zenoh Subscriber and make sure the following properties are set:

    a. Data Bits: 1
    b. Zenoh Key Expression: `p1/button`
iv. Primary click on the existing output and delete it.
v. Insert a Zenoh Publisher in its place (from Components -> IO -> Peripherals -> Zenoh Publisher).
vi. Secondary click on the Zenoh Publisher and make sure the following properties are set:
    a. Data Bits: 1
    b. Zenoh Key Expression: `p1/light`
vii. Run the simulation (with the play button).
viii. Click on the button (left side) in the hardware simulator.
ix. Notice that the light (right side) turns on while the momentary button is pressed.
x. Stop the simulation.

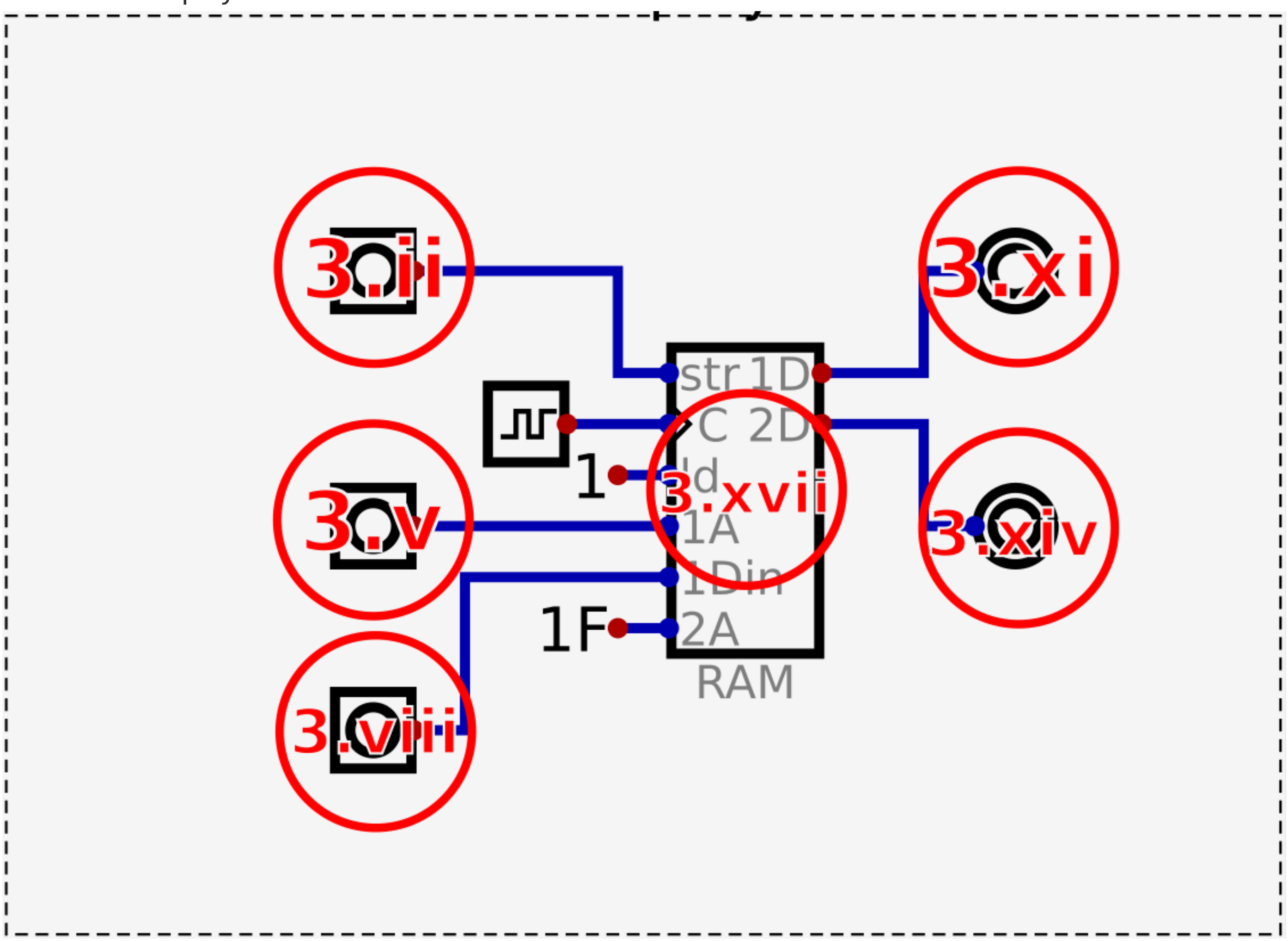
2. Relative Knob and Seven Segment Display



i. Insert a Zenoh Synchronous Subscriber (from Components -> Peripherals -> Zenoh Synchronous Subscriber) connecting its clock input on left, and its data output on right.

ii. Secondary click on the Zenoh Synchronous Subscriber and make sure the following properties are set:
    a. Data Bits: 5
    b. Zenoh Key Expression: `p2/knob`
iii. Primary click on the existing output and delete it.
iv. Insert a Zenoh Publisher in its place (from Components -> IO -> Peripherals -> Zenoh Publisher).
v. Secondary click on the Zenoh Publisher and make sure the following properties are set:
    a. Data Bits: 5
    b. Zenoh Key Expression: `p2/seven_seg`
vi. Run the simulation (with the play button).
vii. Click and drag on the knob (left side) left and right to rotate it.
viii. Notice that the seven segment display (right side) shows the value stored in the register.
ix. Stop the simulation.

3. Table and Display



i. Delete the input node connected to the the `str` pin of the RAM.
ii. Insert a Zenoh Subscriber in its place (from Components -> IO -> Peripherals -> Zenoh Subscriber).

iii. Secondary click on the Zenoh Subscriber and make sure the following properties are set:
   a. Data Bits: 1
   b. Zenoh Key Expression: `p3/button`

iv. Delete the input node connected to the `1A` pin of the RAM.

v. Insert a Zenoh Subscriber in its place (from Components -> IO -> Peripherals -> Zenoh Subscriber).

vi. Secondary click on the Zenoh Subscriber and make sure the following properties are set:
   a. Data Bits: 5
   b. Zenoh Key Expression: `p3/knob_addr`

vii. Delete the input node connected to the `1Din` pin of the RAM.

viii. Insert a Zenoh Subscriber in its place (from Components -> IO -> Peripherals -> Zenoh Subscriber).

ix. Secondary click on the Zenoh Subscriber and make sure the following properties are set:
   a. Data Bits: 8
   b. Zenoh Key Expression: `p3/knob_data`

x. Delete the output node connected to the `1D` pin of the RAM.

xi. Insert a Zenoh Publisher in its place (from Components -> IO -> Peripherals -> Zenoh Publisher).

xii. Secondary click on the Zenoh Publisher and make sure the following properties are set:
   a. Data Bits: 8
   b. Zenoh Key Expression: `p3/seven_seg_1`

xiii. Delete the output node connected to the `2D` pin of the RAM.

xiv. Insert a Zenoh Publisher in its place (from Components -> IO -> Peripherals -> Zenoh Publisher).

xv. Secondary click on the Zenoh Publisher and make sure the following properties are set:
   a. Data Bits: 8
   b. Zenoh Key Expression: `p3/seven_seg_2`

xvi. Delete the RAM.

xvii. Insert a "Zenoh RAM, Dual Port" (from Components -> Memory -> RAM, Zenoh RAM, Dual Port) in its place.

xviii. Secondary click on the Zenoh RAM, Dual Port and make sure the following properties are set:
   a. Data Bits: 8
   b. Address Bits: 5
   c. Zenoh Key Expression: `p3/ram`

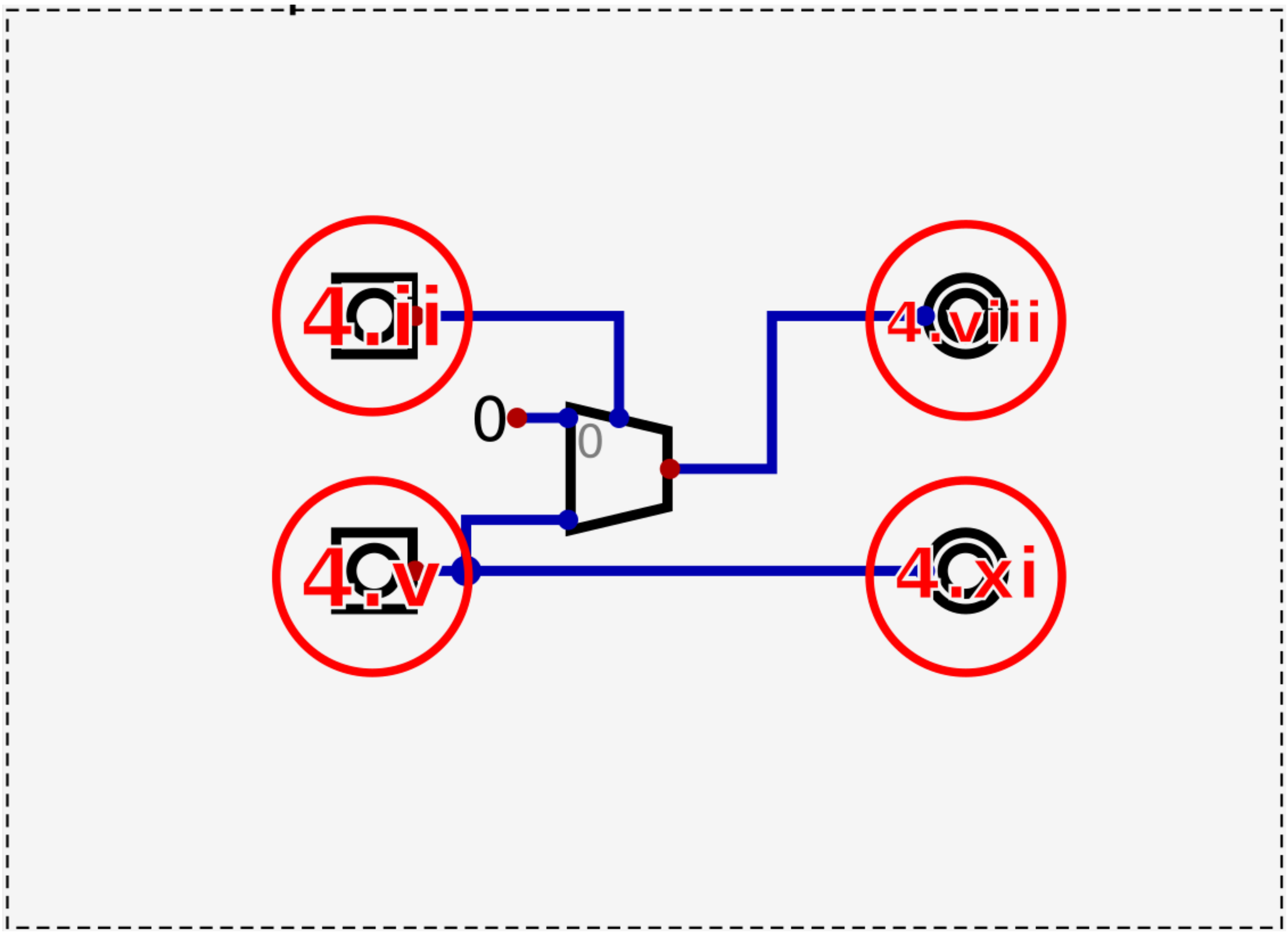
xix. Start the simulation.

xx. Try writing to the RAM by clicking on the button (to enable writing) and rotating the address and data knobs.

a. Notice that the address and data knob values are displayed on seven segment displays.
b. Notice that the tables update to reflect the data in the RAM.
c. Notice that the seven segment displays connected to the RAM (via `1D` and `2D`) update to reflect the data at the addresses `1A` and `2A`.
d. Notice that the display elements' pixels update to reflect the data in the RAM.
   a. The first display shows a shade of color per pixel based on each byte of RAM.
   b. The second display shows a black or colored (Monochrome) pixel based on each byte of RAM.
   c. The third display shows a colored pixel based on each 4 bytes of RAM (red, green, blue, and alpha).

xxi. Stop the simulation.

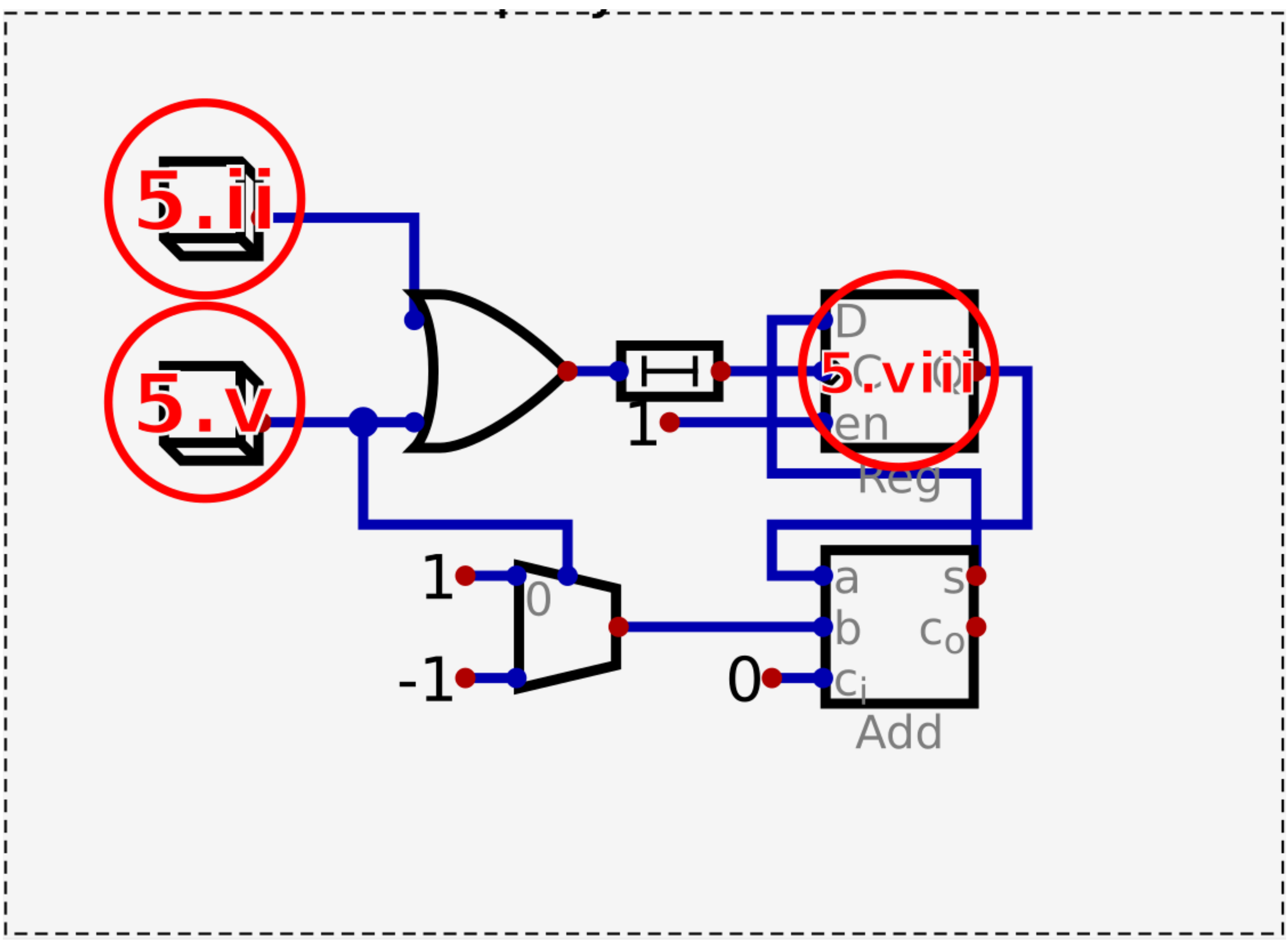
4. Speaker



i. Delete the top left input node connected to the selector bit of the MUX.
ii. Insert a Zenoh Subscriber in its place (from Components -> IO -> Peripherals -> Zenoh Subscriber).
iii. Secondary click on the Zenoh Subscriber and make sure the following properties are set:
   a. Data Bits: 1
   b. Zenoh Key Expression: `p4/button`

iv. Delete the bottom left input node connected to the second input pin of the MUX.
v. Insert a Zenoh Subscriber in its place (from Components -> IO -> Peripherals -> Zenoh Subscriber).
vi. Secondary click on the Zenoh Subscriber and make sure the following properties are set:
   a. Data Bits: 8
   b. Zenoh Key Expression: `p4/knob_note`
vii. Delete the output node connected to the MUX output pin.
viii. Insert a Zenoh Publisher in its place (from Components -> IO -> Peripherals -> Zenoh Publisher).
ix. Secondary click on the Zenoh Publisher and make sure the following properties are set:
   a. Data Bits: 8
   b. Zenoh Key Expression: `p4/speaker`
x. Delete the output node that is connected directly to the `p4/knob_note` subscriber.
xi. Insert a Zenoh Publisher in its place (from Components -> IO -> Peripherals -> Zenoh Publisher).
xii. Secondary click on the Zenoh Publisher and make sure the following properties are set:
   a. Data Bits: 8
   b. Zenoh Key Expression: `p4/seven_seg`
xiii. Start the simulation.
xiv. Click on the button to enable the speaker.
xv. Rotate the knob to change the note played by the speaker.
   a. Notice that the speaker plays a note based on the knob value (interpreted as a MIDI note number, middle C is 60).
   b. Notice that the seven segment display shows the MIDI note played by the speaker.
xvi. Stop the simulation.

5. Character Displays



i. Delete the top left input node connected to the an OR gate.
ii. Insert a Zenoh Subscriber in its place (from Components -> IO -> Peripherals -> Zenoh Subscriber).
iii. Secondary click on the Zenoh Subscriber and make sure the following properties are set:
    a. Data Bits: 1
    b. Zenoh Key Expression: `p5/button_inc`
iv. Delete the bottom left input node connected to the other input of the OR gate, and the selector bit of the MUX.
v. Insert a Zenoh Subscriber in its place (from Components -> IO -> Peripherals -> Zenoh Subscriber).
vi. Secondary click on the Zenoh Subscriber and make sure the following properties are set:
    a. Data Bits: 1
    b. Zenoh Key Expression: `p5/button_dec`
vii. Delete the register (node with D, C, en, and Q pins).
viii. Insert a Zenoh Register in its place (from Components -> Memory -> Zenoh Register).
ix. Secondary click on the Zenoh Register and make sure the following properties are set:
    a. Data Bits: 5
    b. Zenoh Key Expression: `p5/reg`

x. Start the simulation.
xi. Click on the increment button to increment the register value, and the decrement button to decrement the register value.
xii. Notice that the register value is displayed on the character displays (using the Liquid template shown after `Text:` ).
    a. The first display shows an enumeration description of the register value (0: zero, 1: one, 2: two, 3: three), and a padded hexadecimal representation of the register value.
    b. The second display shows a decimal representation of the register value.
xiii. Stop the simulation.

6. Complete short survey: <redacted survey url>

## B.2 Acceptance Test Survey

# User Acceptance Test Survey

* Indicates required question

1. Were you able to find the Zenoh Publisher? *

*Mark only one oval.*

- Yes
- No

2. Were you able to find the Zenoh Subscriber? *

*Mark only one oval.*

- Yes
- No

3. Were you able to find the Zenoh Synchronous Subscriber? *

*Mark only one oval.*

- Yes
- No

4. Were you able to find the Zenoh Dual Port RAM? *

*Mark only one oval.*

- Yes
- No

5. Were you able to find the Zenoh Register? *

*Mark only one oval.*

- Yes
- No

6. Were you able to operate the simulated buttons? *

*Mark only one oval.*

- Yes
- No

7. Were you able to operate the simulated knobs? *

*Mark only one oval.*

- Yes
- No

8. Were you able to see changes to the simulated lights? *

*Mark only one oval.*

- Yes
- No

9. Were you able to see the seven-segment displays change values? *

*Mark only one oval.*

- Yes
- No

10. Were you able to hear the simulated speaker (assuming your volume was up)? *

*Mark only one oval.*

- Yes
- No

11. Were you able to see the contents of RAM in the simulated tables? *

*Mark only one oval.*

- Yes
- No

12. Were you able to see pixels change on the display element (memory backed graphics output)? *

*Mark only one oval.*

- Yes
- No

13. Did interaction with the simulated hardware feel responsive? *

*Mark only one oval.*

- Yes
- No

14. Comments on the responsiveness:

______________________________

15. Comments on the usability of the added Zenoh components in Digital and the hardware simulator:

______________________________
______________________________
______________________________
______________________________
______________________________

## Appendix C

## System Latency Testing Data

Table C.1: System Latency Testing Collected Data

| **Run #** | **Hardware Tests** | **Software Tests** |
|---|---|---|
| 1 | 11.83 ms | 7.05 ms |
| 2 | 11.04 ms | 4.49 ms |
| 3 | 13.15 ms | 3.80 ms |
| 4 | 16.71 ms | 4.17 ms |
| 5 | 13.72 ms | 3.01 ms |
| 6 | 12.89 ms | 4.19 ms |
| 7 | 11.95 ms | 3.32 ms |
| 8 | 13.65 ms | 2.73 ms |
| 9 | 10.71 ms | 2.74 ms |
| 10 | 11.72 ms | 3.11 ms |
| 11 | 11.91 ms | 4.13 ms |
| 12 | 11.40 ms | 3.29 ms |
| 13 | 12.77 ms | 5.39 ms |
| 14 | 12.82 ms | 3.72 ms |
| 15 | 11.43 ms | 3.94 ms |
| 16 | 11.67 ms | 2.96 ms |
| 17 | 11.84 ms | 3.65 ms |
| 18 | 12.89 ms | 2.94 ms |
| 19 | 11.94 ms | 3.77 ms |
| 20 | 12.41 ms | 3.45 ms |
| 21 | 11.56 ms | 2.97 ms |
| 22 | 13.20 ms | 5.52 ms |
| 23 | 12.13 ms | 3.90 ms |
| 24 | 12.20 ms | 4.29 ms |
| 25 | 11.43 ms | 2.85 ms |
| Average | 12.36 ms | 3.82 ms |
| Max | 16.71 ms | 7.05 ms |
| Min | 10.71 ms | 2.73 ms |
| Median | 11.95 ms | 3.72 ms |
| Range | 6.00 ms | 4.32 ms |
| Std Dev | 1.20 ms | 1.00 ms |

# Bibliography

[1] "ROS/Introduction - ROS Wiki." [Online]. Available: http://wiki.ros.org/ROS/Introduction

[2] M. Gunes, M. Thornton, F. Kocan, and S. Szygenda, "A survey and comparison of digital logic simulators," in *48th Midwest Symposium on Circuits and Systems, 2005.*, Aug. 2005, pp. 744–749 Vol. 1, iSSN: 1558-3899. [Online]. Available: https://ieeexplore.ieee.org/document/1594208

[3] J. Wang and C. Tropper, "Compiled Code in Distributed Logic Simulation," in *Proceedings of the 2006 Winter Simulation Conference*, Dec. 2006, pp. 981–986, iSSN: 1558-4305. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/4117709

[4] T. Gingold, "ghdl/ghdl," Jan. 2024, original-date: 2015-11-18. [Online]. Available: https://github.com/ghdl/ghdl

[5] "Logisim-Evolution," https://github.com/logisim-evolution/logisim-evolution.

[6] "CircuitVerse - Online Digital Logic Circuit Simulator." [Online]. Available: https://circuitverse.org/

[7] "ModelSim." [Online]. Available: https://eda.sw.siemens.com/en-US/ic/modelsim/

[8] "Turing Complete." [Online]. Available: https://turingcomplete.game/

[9] H. Neemann, “Digital,” https://github.com/hneemann/Digital.

[10] T. Niget, “Crabs All the Way Down: Running Rust on Logic Gates,” Aug. 2022. [Online]. Available: https://zdimension.fr/crabs-all-the-way-down/

[11] “ROS.” [Online]. Available: https://www.ros.org/

[12] “ZeroMQ.” [Online]. Available: https://zeromq.org/

[13] “8. A Framework for Distributed Computing.” [Online]. Available: https://zguide.zeromq.org/docs/chapter8/

[14] “100GbE Tests with ZeroMQ v4.3.2 - zeromq.” [Online]. Available: http://wiki.zeromq.org/results:100gbe-tests-v432

[15] “libzmtp.” [Online]. Available: https://github.com/zeromq/libzmtp

[16] “What is DDS?” [Online]. Available: https://www.dds-foundation.org/what-is-dds-3/

[17] “About the DDS For Extremely Resource Constrained Environments Specification Version 1.0.” [Online]. Available: https://www.omg.org/spec/DDS-XRCE/About-DDS-XRCE/

[18] W.-Y. Liang, Y. Yuan, and H.-J. Lin, “A Performance Study on the Throughput and Latency of Zenoh, MQTT, Kafka, and DDS,” Mar. 2023, arXiv:2303.09419 [cs]. [Online]. Available: http://arxiv.org/abs/2303.09419

[19] “Zenoh.” [Online]. Available: https://zenoh.io/

[20] “eclipse-zenoh/zenoh-pico,” Feb. 2024, original-date: 2020-09-18. [Online]. Available: https://github.com/eclipse-zenoh/zenoh-pico

[21] “MQTT.” [Online]. Available: https://mqtt.org/

[22] "MQTT Software." [Online]. Available: https://mqtt.org/software/

[23] "HOWTO : automatic discovery of the MQTT server · arendst/Tasmota · Discussion #11403." [Online]. Available: https://github.com/arendst/Tasmota/discussions/11403

[24] "Fast-DDS," Feb. 2024, original-date: 2014-05-29. [Online]. Available: https://github.com/eProsima/Fast-DDS

[25] "Connext Professional." [Online]. Available: https://www.rti.com/products/connext-professional

[26] "OpenDDS." [Online]. Available: https://opendds.org/

[27] "CycloneDDS," Feb. 2024, original-date: 2018-01-02. [Online]. Available: https://github.com/eclipse-cyclonedds/cyclonedds

[28] "REP 2000 – ROS 2 Releases and Target Platforms (ROS.org)." [Online]. Available: https://www.ros.org/reps/rep-2000.html

[29] C. Attig, N. Rauh, T. Franke, and J. F. Krems, "System Latency Guidelines Then and Now – Is Zero Latency Really Considered Necessary?" in *Engineering Psychology and Cognitive Ergonomics: Cognition and Design*, ser. Lecture Notes in Computer Science, D. Harris, Ed. Cham: Springer International Publishing, 2017, pp. 3–14.

[30] T. Kaaresoja, S. Brewster, and V. Lantz, "Towards the Temporally Perfect Virtual Button: Touch-Feedback Simultaneity and Perceived Quality in Mobile Touchscreen Press Interactions," *ACM Transactions on Applied Perception*, vol. 11, no. 2, pp. 9:1–9:25, Jun. 2014. [Online]. Available: https://dl.acm.org/doi/10.1145/2611387

[31] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design ARM Edition: The Hardware Software Interface*. Morgan Kaufmann, 2016.

[32] ——, *Computer Organization and Design MIPS Edition: The Hardware/Software Interface, 5th Edition*. Morgan Kaufmann, 2013.

[33] A. Ltd, "Licensing Arm Technology." [Online]. Available: https://www.arm.com/products/licensing

[34] G. Tang, "Intel and the x86 Architecture: A Legal Perspective," Jan. 2011. [Online]. Available: https://jolt.law.harvard.edu/digest/intel-and-the-x86-architecture-a-legal-perspective

[35] E. Brown, "MIPS ISA to be available under royalty-free license," Dec. 2018. [Online]. Available: https://linuxgizmos.com/mips-isa-to-be-available-under-royalty-free-license/

[36] R. Mitchell, "Tales from 80s Tech: How Compaq's Clone Computers Skirted IBM's IP and Gave Rise to EISA - News," Aug. 2017. [Online]. Available: https://www.allaboutcircuits.com/news/how-compaqs-clone-computers-skirted-ibms-patents-and-gave-rise-to-eisa/

[37] "Specifications - RISC-V International." [Online]. Available: https://riscv.org/technical/specifications/

[38] Terasic, "Terasic - all FPGA boards - Cyclone V - Cyclone V GX starter kit." [Online]. Available: https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=167&No=830

[39] "Wokwi - Online ESP32, STM32, Arduino Simulator." [Online]. Available: https://wokwi.com/

[40] "Verilog LCD - Wokwi ESP32, STM32, Arduino Simulator." [Online]. Available: https://wokwi.com/projects/347069718502310484

[41] "Logic World." [Online]. Available: https://logicworld.net/

[42] "Bevy Engine." [Online]. Available: https://bevyengine.org//

[43] B. Foundation, "blender.org - Home of the Blender project - Free and Open 3D Creation Software." [Online]. Available: https://www.blender.org/

[44] "Bevy Book: ECS." [Online]. Available: https://bevyengine.org//learn/book/getting-started/ecs/

[45] A. Corsaro, L. Cominardi, O. Hecart, G. Baldoni, J. E. P. Avital, J. Loudet, C. Guimares, M. Ilyin, and D. Bannov, "Zenoh: Unifying communication, storage and computation from the cloud to the microcontroller," in *2023 26th Euromicro Conference on Digital System Design (DSD)*, 2023, pp. 422–428.

[46] Aevyrie, "bevy_mod_picking." [Online]. Available: https://github.com/aevyrie/bevy_mod_picking

[47] J. K. Helsing, "bevy_pancam," Sep. 2024, original-date: 2021-08-01. [Online]. Available: https://github.com/johanhelsing/bevy_pancam

[48] "Inkscape." [Online]. Available: https://inkscape.org/

[49] "glTF™ 2.0 Specification," Oct. 2021.

[50] "1. Introduction — RISCOF 1.24.0 documentation." [Online]. Available: https://riscof.readthedocs.io/en/stable/intro.html

[51] "RISC-V Compliance Test Generator," Mar. 2024, original-date: 2021-02-22. [Online]. Available: https://github.com/riscv-software-src/riscv-ctg

[52] "Instrumentation-based Code Coverage - The rustc book." [Online]. Available: https://doc.rust-lang.org/rustc/instrument-coverage.html

[53] xd009642, “Tarpaulin,” Sep. 2024, original-date: 2017-04-07. [Online]. Available: https://github.com/xd009642/tarpaulin

[54] “bevy/tests/how_to_test_systems.rs.” [Online]. Available: https://github.com/bevyengine/bevy/blob/main/tests/how_to_test_systems.rs