RISC-V INSTRUCTION SET EXTENSION FOR BLOCKCHAIN TOKEN TRANSACTION

By

Ivan Ng Yong Zhe

A REPORT

SUBMITTED TO

Universiti Tunku Abdul Rahman

in partial fulfillment of the requirements

for the degree of

BACHELOR OF COMPUTER SCIENCE (HONOURS)

Faculty of Information and Communication Technology

(Kampar Campus)

FEBRUARY 2026

COPYRIGHT STATEMENT

This Final Year Project proposal is submitted in partial fulfillment of the requirements for the degree of **Bachelor of Computer Science (Honours)** at Universiti Tunku Abdul Rahman (UTAR). This Final Year Project proposal represents the work of the author, except where due acknowledgment has been made in the text. No part of this Final Year Project proposal may be reproduced, stored, or transmitted in any form or by any means, whether electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the author or UTAR, in accordance with UTAR's Intellectual Property Policy.

ACKNOWLEDGEMENTS

I would like to express my sincere thanks and appreciation to my supervisor, Ts Dr Ooi Joo On and moderator, Ts Wong Chee Siang, for the constructive comments during the evaluation process. This project involved the study and modification of SP1 zkVM developed by Succinct Labs, whose open-source contribution made this research possible.

Finally, I must say thanks to my parents and my family for their love, support, and continuous encouragement throughout the course.

ABSTRACT

This project investigates the implementation of RISC-V Instruction Set Architecture (ISA) extensions within SP1 zkVM to accelerate blockchain token transaction workloads. SP1 zkVM executes programs compiled to the RISC-V RV32IM instruction set and generates zero-knowledge proofs of their correct execution. However, software-implemented cryptographic operations such as SHA-256, Keccak-256, and Ed25519 generate long instruction traces inside the virtual machine, directly increasing the cost of proof generation. This project designs and implements three ISA extensions — SHA256_FOLD, SPONGE_F1600, and TED255_ADD — within SP1's dedicated 0x01_xx_xx_xx syscall namespace, wiring each extension through all four layers of SP1's syscall architecture without modifying the underlying proof constraint system. A benchmark suite is developed to evaluate instruction count, instruction speedup, transactions per second, and execution speedup across 1, 10, and 100 iteration scales for SHA-256, Keccak-256, Ed25519, and a composite token transaction workload. Results show instruction speedup of 6.0 times for SHA-256, 21.0 times for Keccak-256, and 559.6 times for Ed25519 at 100 iterations. The composite token transaction benchmark achieves 64.6 times instruction speedup and 3,030 transactions per second compared to 157.7 for the baseline. These results demonstrate that direct precompile syscall invocation substantially reduces instruction trace length for blockchain token transaction workloads in SP1 zkVM.

Area of Study: Computer architecture, Embedded system

Keywords: RISC-V, Zero-Knowledge Virtual Machine (zkVM), Instruction Set Extension, SP1 zkVM, Cryptographic acceleration, Blockchain, Token Transaction.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYSMBOLS

Symbol	Meaning
[u64; 8]	Rust array type – 8 elements of unsigned 64-bit integer
[u8; 32]	Rust array type – 32 elements of unsigned 8-bit integer
0x01_xx_xx_xx	Hexadecimal syscall code namespace for ISA extension
0x00_xx_xx_xx	Hexadecimal syscall code namespace for canonical SP1 syscall

LIST OF ABBREVIATIONS

ABI	Application Binary Interface
AIR	Algebraic Intermediate Representation
CLI	Command Line Interface
CPU	Central Processing Unit
ELF	Executable and Linkable Format
ISA	Instruction Set Architecture
ISE	Instruction Set Extension
JIT Executor	Just-In-Time Executor
RAM	Random Access Memory
RISC-V	Reduced Instruction Set Computer Version 5
SHA-256	Secure Hash Algorithm 256-bit
SP1	Succinct Processor 1
SSD	Solid State Drive
STARK	Scalable Transparent Argument of Knowledge
SNARK	Succinct Non-interactive Argument of Knowledge
TED	Twisted Edwards
TPS	Transactions Per Second
VM	Virtual Machine
ZK	Zero-knowledge
ZKP	Zero-knowledge Proof
zkVM	Zero-knowledge Virtual Machine

CHAPTER 1

Introduction

1.1 Problem Statement and Motivation

The RISC-V architecture is an open-source instruction set architecture (ISA) that has sparked a new era of processor innovation due to its modularity and royalty-free nature [1]. At the same time, blockchain technology has evolved into a complex architecture requiring robust security and efficient processing for decentralized applications [2]. Blockchain token transactions involve intensive cryptographic operations, including hashing, digital signature verification and other more. Zero-knowledge proof systems have emerged as a foundational technology for blockchain scalability and privacy.

SP1 zkVM (Succinct Processor 1 Zero-Knowledge Virtual Machine), which is developed by Succinct Labs, can execute programs that compiled to RISC-V instruction set and generates cryptographic proofs of their correct execution [3]. However, SP1 zkVM baseline RV32IM ISA lacks native instructions for cryptographic operations like SHA-256, Keccak-256 and Ed25519 which are part of blockchain token transactions [3]. The software implemented cryptographic operations will cost a lot of instructions per operations in SP1 zkVM because every single instruction needs to be recorded in the execution witness and encoded as algebraic constraints [3]. Besides that, there is still lack of systematic evaluation of the instruction count difference between baseline execution and direct ISA-level precompile for blockchain token transaction workload.

The motivation of this project is to show the implementation of ISA extension in SP1 zkVM. Achieve reduction in the instruction count by using extension which replace thousands of instructions with single operations and enable realistic blockchain workloads. By delivering a measurable performance gains with the validation of the benchmarks, this project positions ISA extensions as a solution for scaling for SP1 zkVM in blockchain.

1.2 Project Objectives

The objective of this project is **to design and implement ISA extensions**, also known as precompiles that accelerate the execution of transaction logic within SP1 zkVM environment.

These extensions are specifically engineered to accelerate the transaction logic. Besides that, this project will **demonstrate the instruction count and execution time reductions** which achieved by extensions compared to baseline software implementation using blockchain token transaction benchmarks.

Lastly, this project will **conduct a comparative performance analysis**. The analysis will focus on SHA-256, Keccak-256, Ed25519 and token transaction, measuring the instruction trace length differences between baseline software execution and extension software execution across 1, 10 and 100 iteration scales.

1.3 Project Scope and Direction

This project operates in SP1 zkVM by Succinct Labs which target the RISC-V RV32IM instruction set. The scope covers three cryptography primitives which are SHA-256 compression, Keccak-f [1600] permutation and Ed25519 point addition which representing the dominant computational components of blockchain token transaction as data hashing, Ethereum compatible address derivation and digital signature verification.

The ISA extensions are implemented as aliased syscall code which route to the existing SP1 precompile chips. This approach will bypass the need of designing or modifying AIR (Algebraic Intermediate Representation) constraint circuits which need advanced expertise in finite field arithmetic and algebraic proof systems. ZK proof generation benchmarking will not be tested as it requires hardware that beyond what is available and also the security analysis of zkVM proof system.

This project aims to demonstrate the extensibility of the SP1 syscall architecture and to measure the practical performance gains which result from direct precompile invocation over software computation in a blockchain.

1.4 Contributions

This project contributes in the field of blockchain infrastructure by demonstrating a software method for accelerating transaction logic. A working implementation method of three ISA

extensions in SP1 zkVM. The implementation involves in modifying eight source files in four layers of SP1's syscall architecture. The extensions are assigned to dedicated namespace and wired through the guest ABI, JIT executor, tracing VM and AIR chip routing without requiring any modification to SP1's proof constraint system.

Besides that, this project contributes in blockchain token transaction benchmark suite with a structured benchmark library and host evaluation harness by combing SHA-256, Keccak-256 and Ed25519 operations into a realistic token transaction workload. A performance analysis is also a contribution by demonstrating instruction count reductions for the targeted three cryptography primitives which are SHA-256 compression, Keccak-256 permutation and Ed25519 point addition.

CHAPTER 2
Literature Review

2.1 Review of Technology

2.1.1 Evaluation and Categorization of Hashing Algorithms Based on Their Applications

In this research a detailed assessment of current cryptographic hash functions is presented in [4], focusing on their overall efficiency, security strengths, and practical usability. The authors measured several key performance metrics, including execution speed, output rates, memory footprint, and the avalanche effect [4]. Furthermore, the study categorizes the application of these algorithms into five primary areas: data forensics, distributed networks like blockchains, system authentication, general data integrity, and core cryptographic security. The varying capabilities and suitability of these functions depending on their applied domain are illustrated in Figure 2.1.1 [4].

Application Domain	Hash Algorithms	Suitability for High-Security Applications	Speed	Resistance to Quantum Attacks
Cryptographic Security	SHA224, SHA256, SHA384, SHA512/224, SHA512/256, SHA512, SHA3-224, SHA3-256, SHA3-384, SHA3-512, Whirlpool, Tiger series, Gost, Gost-Crypto, Haval series, BLAKE variants, Chaos based hash functions	Very Suitable	Moderate to High	Widely used in encryption, digital signatures, secure data transmission
Data Management and Integrity	MD2, MD4, MD5, SHA1, RIPEMD series, Adler32, CRC32, CRC32B, CRC32C, FNV series, CityHash, FarmHash, XXH32, XXH64, XXH3, XXH128	Less Suitable	High	Data storage, checksums, error-checking in networks and storage devices
Authentication and Verification	SHA224, SHA256, SHA384, SHA512/224, SHA512/256, SHA512, SHA3 series, Bcrypt, Scrypt, SipHash, RIPEMD160 (notably used in Bitcoin)	Suitable	Moderate to High	User authentication, data authenticity verification
Blockchain and Distributed Systems	SHA256 (widely used in blockchain technologies), RIPEMD160 (as in Bitcoin), BLAKE variants, Chaos based hash functions (potential application due to their unique properties)	Suitable	Moderate to High	Cryptocurrency mining, transaction validation, distributed ledgers
Forensics and Data Analysis	SHA3 series (due to their resistance to quantum attacks), Whirlpool, Haval series, Gost, Gost-Crypto	Suitable	Moderate to High	Digital forensics, data integrity verification, malware detection

Figure 2.1.1 Suitability and Performance Characteristics of Hash Algorithms across different Application Domains

2.1.2 RISC-V Instruction Set Architecture Extensions: A Survey

This paper provides a comprehensive review of the modular and extensible nature of the RISC-V Instruction Set Architecture (ISA). The authors highlight that while the base RISC-V set is highly efficient for general tasks, there remains a significant performance gap when addressing emerging high-demand computing scenarios such as cloud computing and cryptography. The survey emphasizes that RISC-V's primary strength lies in its explicit support for domain-specific custom extensions, allowing developers to implement specialized instructions to accelerate specific workloads [5]. For this project, this research justifies the necessity of extending the standard RISC-V set to handle the intensive computational requirements of blockchain token transactions, which standard general-purpose instructions cannot process efficiently. Figure 2.1.2 is the diagram of RISC-V Instruction Set Architecture (ISA) [5].

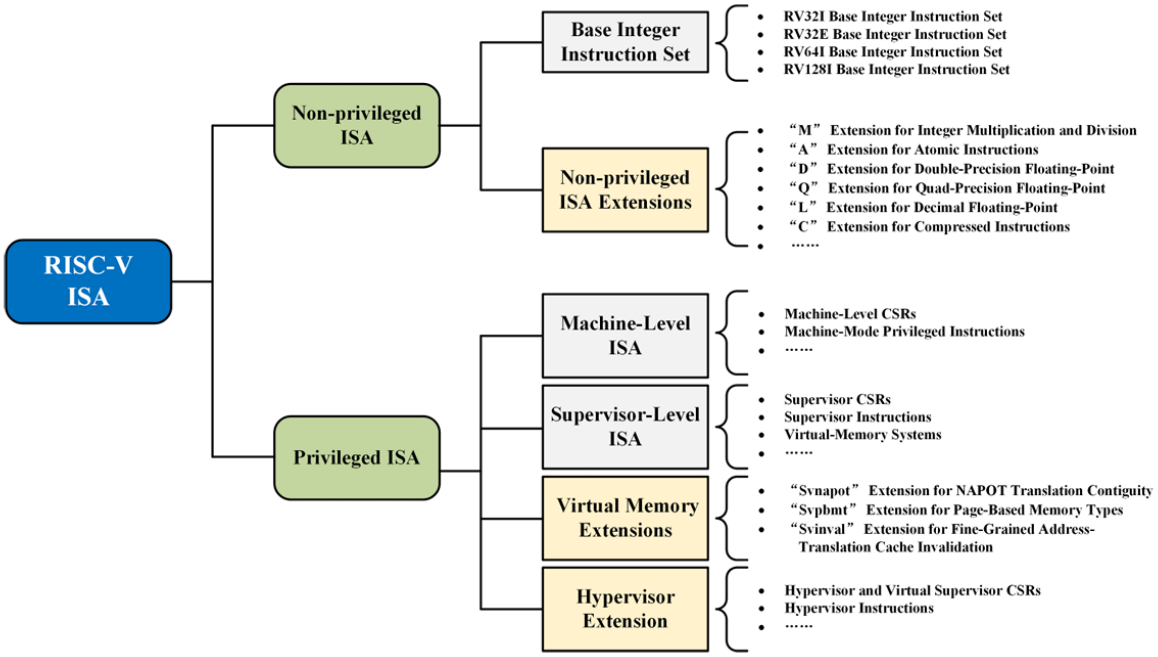


Figure 2.1.2 The RISC-V ISA

2.1.3 The Rust reference

Rust is the primary programming language used in this project for both the host evaluation harness and the guest benchmark programs. Rust is the systems programming language developed by Mozilla Research, first released in 2010, that emphasizes memory safety, concurrency and performance without a garbage collector [25].

Rust was chosen for this project for several reasons that are directly relevant to zkVM development. First, SP1 zkVM requires guest programs to be compiled to RISC-V RV32IM binaries, and Rust's compiler (rustc) supports RISC-V targets including the riscv32im-succinct-zkvm-elf target used by SP1. Second, Rust's no_std mode allows guest programs to run without the standard library, which is required in SP1's zero-overhead execution environment [25]. Third, SP1's SDK is written in Rust and exposes a Rust API for host program development, making Rust the natural and only practical choice for interacting with SP1's execution and proving infrastructure. The project uses Rust edition 2021, which introduces improvements to the borrow checker and module system. The Cargo build system manages all dependencies through a workspace configuration that compiles the host and guest crates separately with their respective targets and feature flags.

2.2 Review of Existing System

2.2.1 Effectively hiding sensitive data with RISC-V Zk and custom instructions

To improve SHA-512 execution, Shchekin (2024) introduced customized instructions utilizing the RISC-V Zk cryptographic subset. These modifications successfully doubled the processing speed while decreasing the overall code size by a tenth [6]. The results emphasize the value of highly targeted optimization strategies for operations like blockchain token transactions, which demand constant cryptographic hashing and signature validation. The exact performance advantages of these customized instructions over standard baseline configurations are detailed in Figure 2.2.1 [6]. Consequently, the study illustrates the inherent performance limitations of relying solely on standard RISC-V architectures for advanced security requirements.

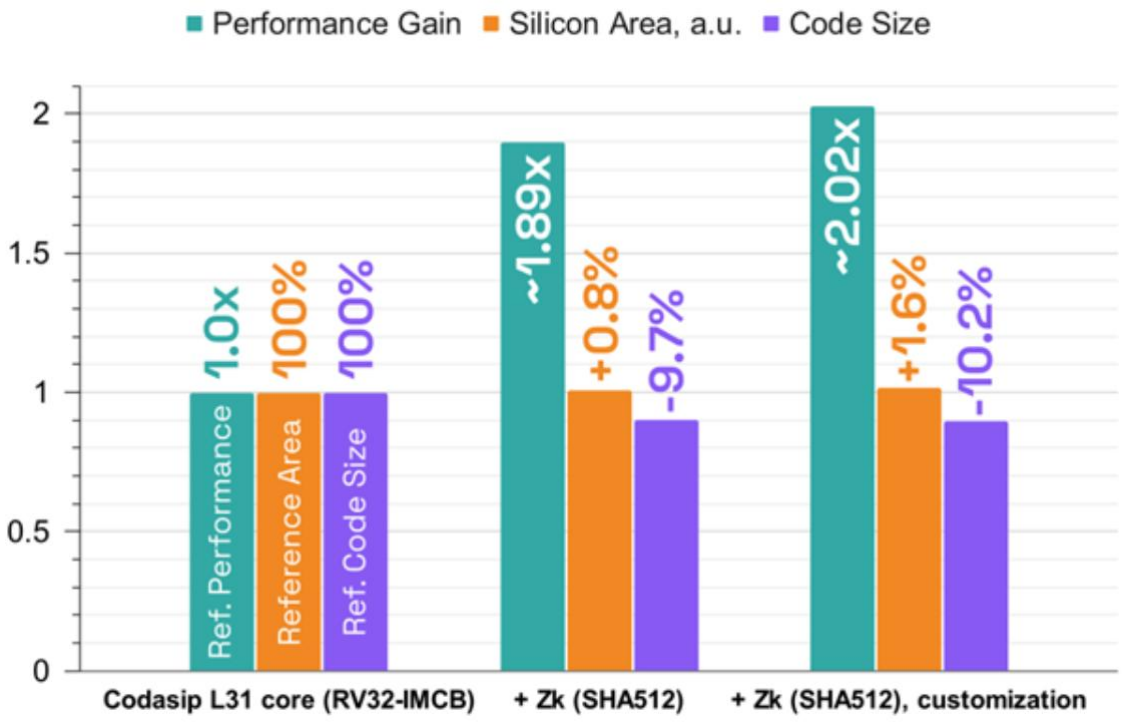


Figure 2.2.1 Performance gain from Zk instruction

2.2.2 Symmetric Cryptography on RISC-V: Performance Evaluation of Standardized Algorithms

The authors conducted a study of the efficient implementation of cryptography algorithms on RISC-V ISA. They are comparing the software implementations using the RV32I instruction set and extended version with custom hardware supported instruction [7]. They focus on integrating cryptography-specific operations such as grev[i], shlf[i] and unshlf[i] which are mentioned in the article into the processor's instruction set to enhance execution efficiency [7]. They also show that compared to implementations with only the base RV32I instruction set, implementations with cryptography set extension provide 1.5x to 8.6x faster execution speed at additional hardware cost of less than 9% which shown in Figure 2.2.2 [7].

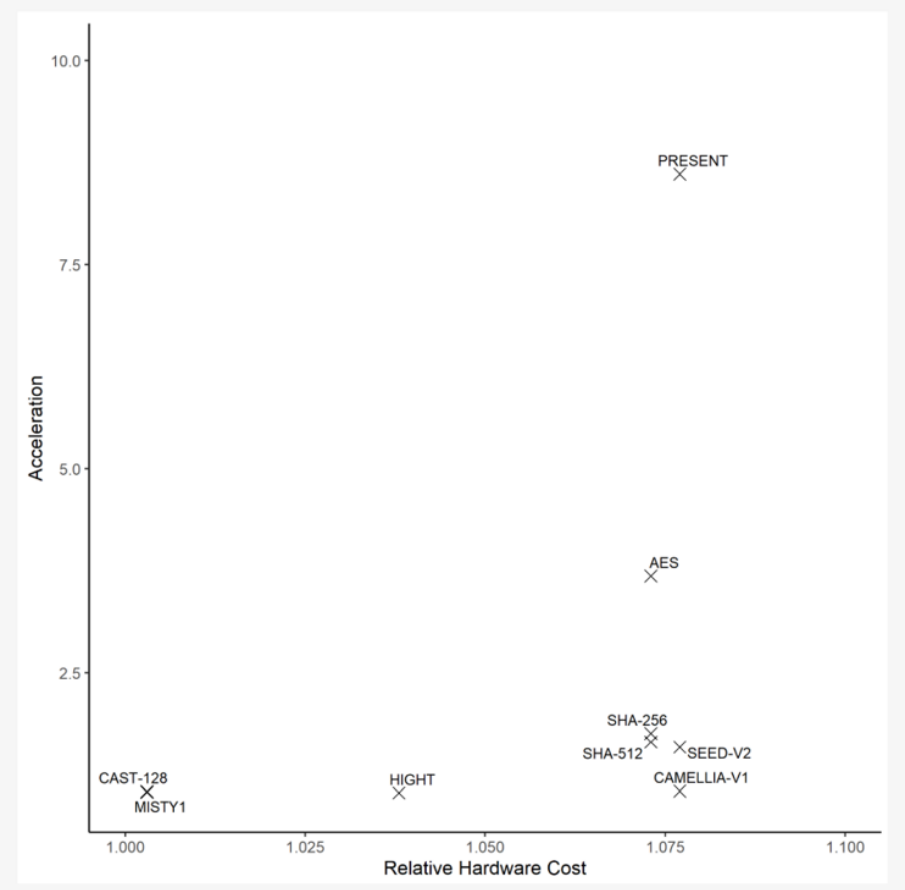


Figure 2.2.2 Acceleration vs hardware cost of implementation of cryptographic implementation

2.2.3 Evaluating the Efficiency of zk-SNARK, zk-STARK, and Bulletproof in Real-World Scenarios: A Benchmark Study

This study provides a real-world benchmark of non-interactive zero-knowledge proof protocols, specifically zk-SNARK, zk-STARK and Bulletproofs. By using dynamic hashing applications as a benchmark, the study measures proof generation and verification efficiency across multiple libraries. The researchers found that while zk-SNARKs produce the smallest proof sizes, zk-STARKs demonstrate the fastest proof generation and verification times compared to others in real-world scenarios [8]. This paper is critical for this project as it provides the mathematical justification for choosing a STARK-based virtual machine for example, SP1. By citing this study, the project establishes that the high-speed execution of blockchain primitives is best achieved through the STARK framework, which prioritizes prover efficiency and scalability over proof compactness. Figure 2.2.3.1 is the Proof time benchmark plot and Figure 2.2.3.2 is the Verification time benchmark plot from paper [8].

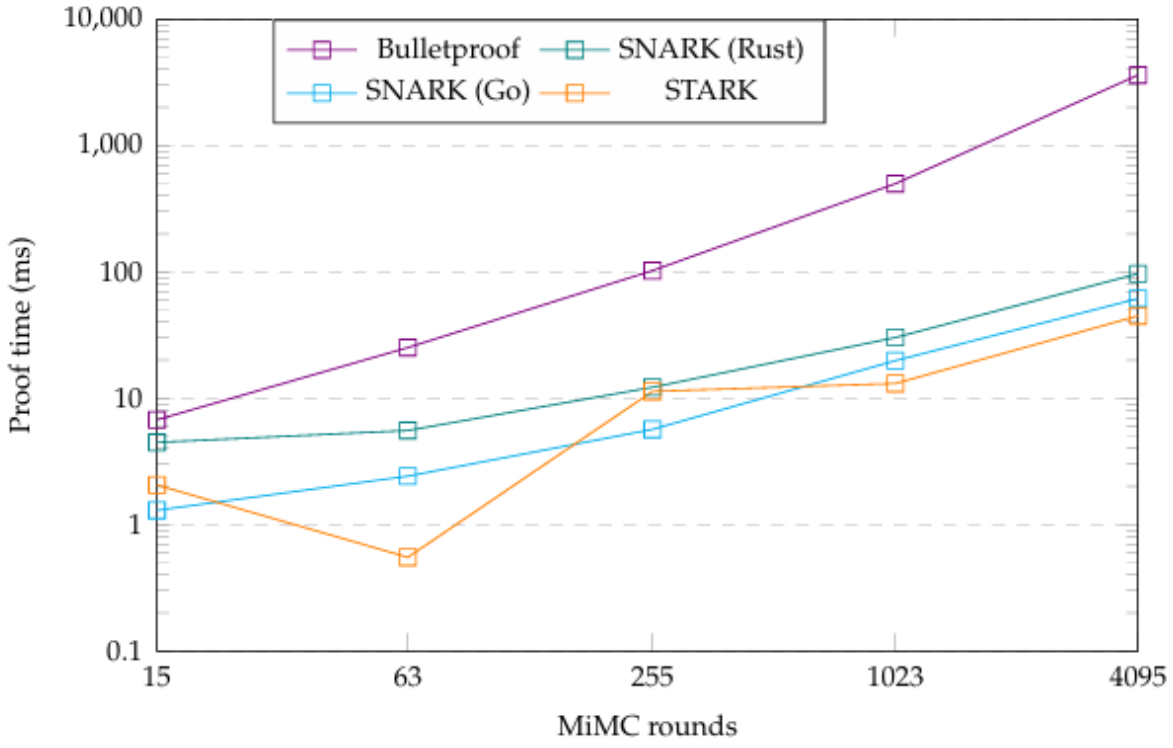


Figure 2.2.3.1 Proof time benchmark plot

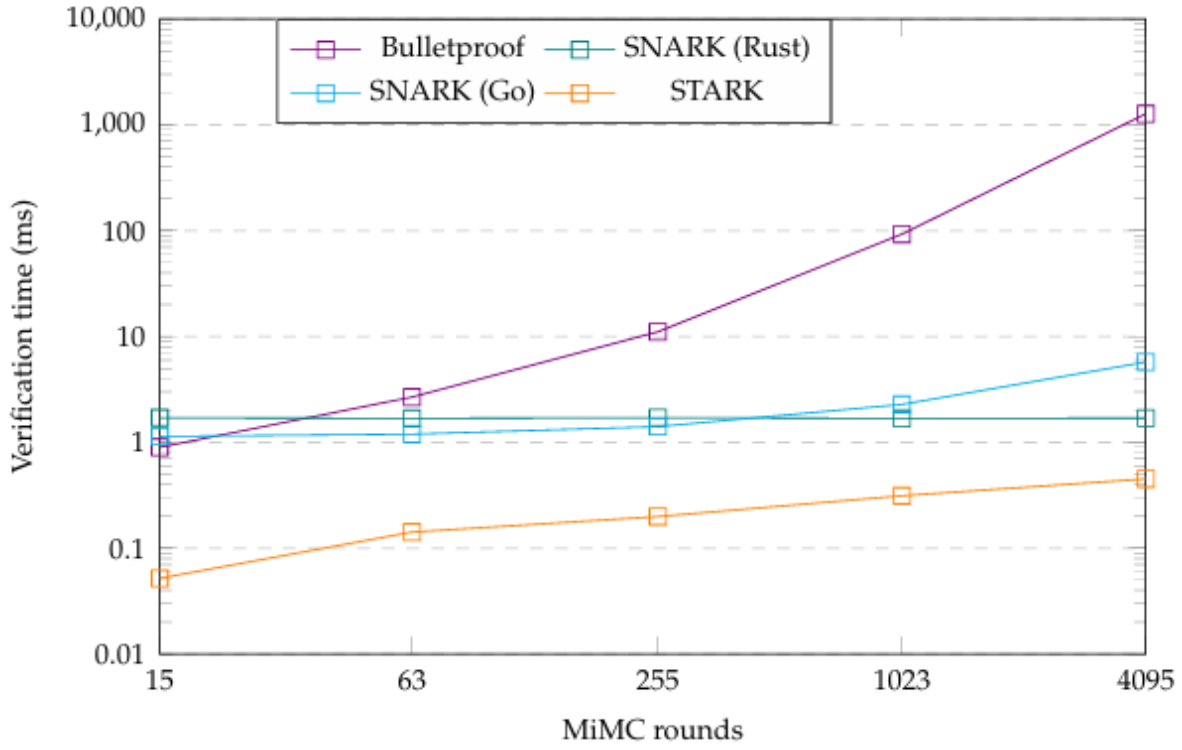


Figure 2.2.3.2 Verification time benchmark plot

2.2.4 Evaluating Compiler Optimization Impacts on zkVM Performance

This research provides the systematic study of compiler optimizations specifically targeting RISC-V based Zero-Knowledge Virtual Machines, including SP1 and RISC Zero. The authors demonstrate that traditional compiler optimizations have a different impact on zkVMs compared to traditional CPUs, as features like instruction-level parallelism are absent in a ZK environment [9]. The study benchmarks critical cryptographic primitives including SHA-2, Keccak-256, and Ed25519 verification. The paper shows that software-level optimizations and the use of specialized precompiles are the modern equivalent of hardware instruction extensions, capable of improving zkVM performance by over 40% [9]. Figure 2.1.6 is the list of library versions that SP1 and RISC Zero used from paper [9].

Program	Library	Version/Commit
PolyBench [22]	–	babdd97
NPB [41]	–	4bd879b
ecdsa-verify	k256	0.13.3
eddsa-verify	ed25519_dalek	2.1.1
merkle	rs_merkle	1.5.0
regex-match	regex	1.11.1
sha2-bench	sha2	0.10.8
sha2-chain	sha2	0.10.8

Program	Library	Version/Commit
sha3-bench	sha3	0.10.8
sha3-chain	sha3	0.10.8
rsp	serde	1.0.204
	rsp-client-executor	249b34e (SP1)
	rsp-client-executor	4ceefdf (RISC Zero)
	c-kzg	1.0.3
	bytemuck_derive	1.8.0
	bincode	1.3.3

Figure 2.2.4 Programs and their associated libraries and versions

CHAPTER 3

System Methodology

3.1 System Design Diagram

3.1.1 System Architecture Diagram

The overall system architecture consists of three components and four layers from SP1 source code. The three components are modified SP1 zkVM core, the guest benchmark library and the host evaluation harness. Then, the four layers are guest ABI, JIT Executor, Tracing VM and AIR Chip Routing. The main architectural insight is that the ISA extensions can be introduced at layer 1 through layer 3 without modifying layer 4.

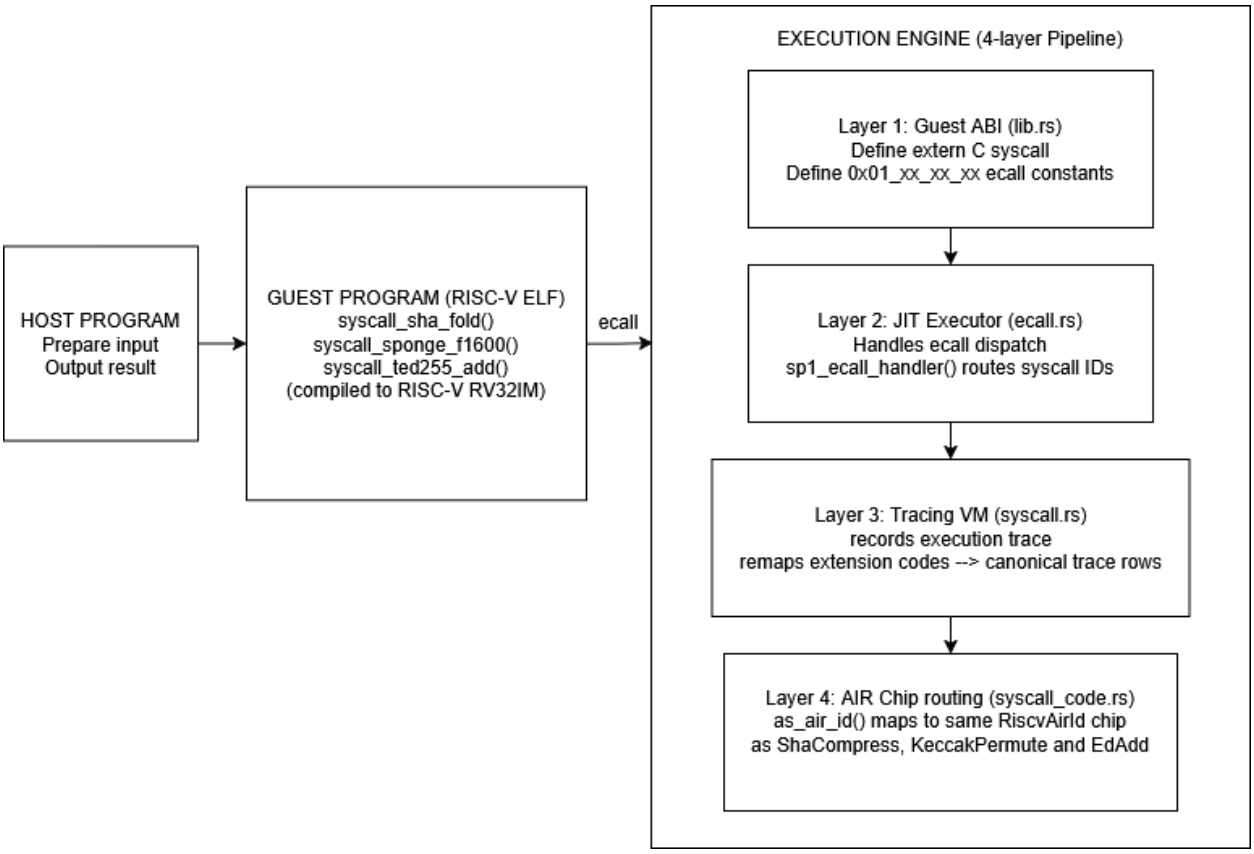


Figure 3.1.1 System Architecture Diagram

3.1.2 Use Case Diagram and Description

The system has two primary actors which are the Developer, who will runs the host harness and the Guest Program, which will executes inside the zkVM. The following use case diagram describes their interactions.

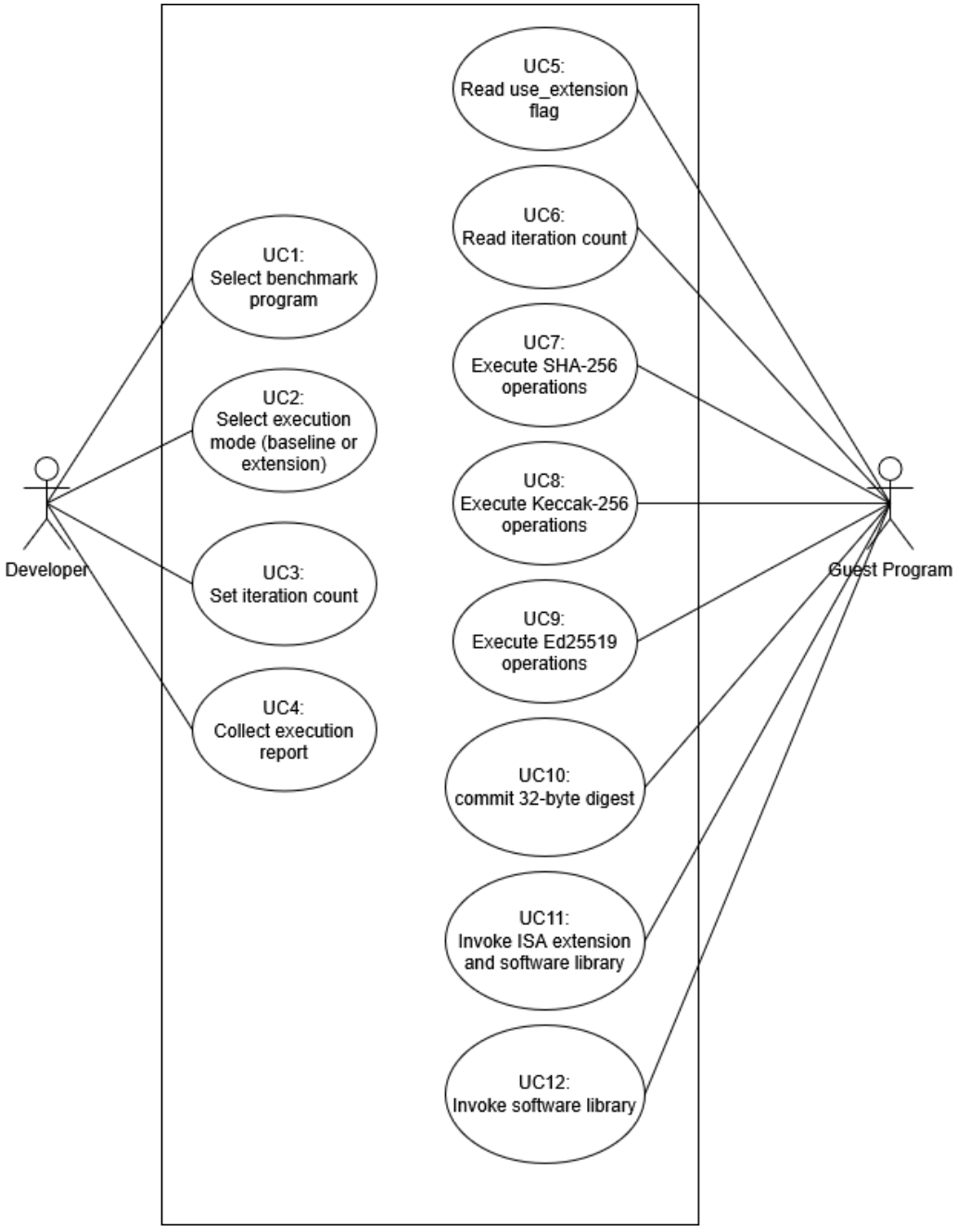


Figure 3.1.2 Use Case diagram

Use Case	Actor	Description	Precondition
UC1	Developer	Select one of four benchmarks via CLI –program flag	Host binary compiled
UC2	Developer	Select baseline or extension mode via CLI –mode flag	UC1 complete
UC3	Developer	Set number of iterations via –iteration flag	UC1 complete
UC4	Developer	Read JSON metrics from stdout after execution	UC1-3 complete
UC5	Guest Program	Read Boolean use_extension from SP1 hint stream	ELF loaded by SP1
UC6	Guest Program	Read u32 iteration count from SP1 hint stream	UC5 complete
UC7-9	Guest Program	Execute selected cryptographic operations for N iterations	UC5-6 complete
UC8	Guest Program	Commit 32-byte accumulator digest as public output	UC7-9 complete
UC9	Guest Program	Invoke extension syscall via ecall with 0x01 namespace code	use_extension = true
UC10	Guest Program	Invoke sha2/sha3/curve25519-dalek library functions	use_extension = false

Table 3.1.2 Use Case Descriptions

3.1.3 Activity Diagram

The following activity diagram will describe the full execution flow from CLI invocation through to metrics output. This will cover both baseline and extension execution paths inside the guest program.

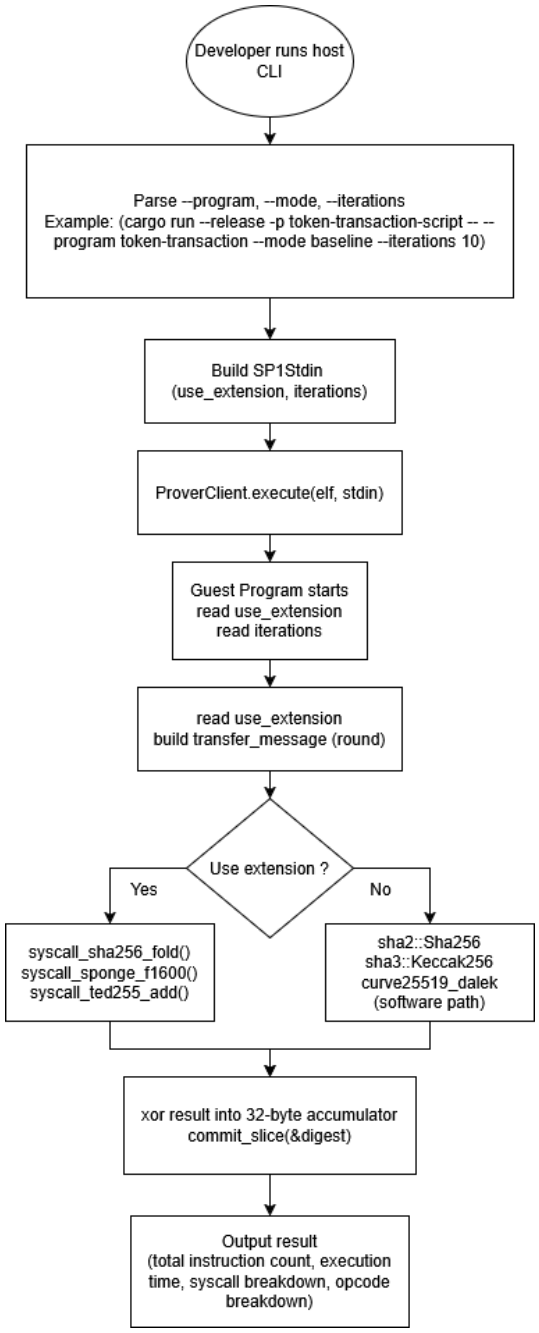


Figure 3.1.3 Activity Diagram

Chapter 4

System Design

4.1 System Block Diagram

The system is divided into two crates in a Cargo workspace. The host crate runs natively on the developer machine and carry out executions. The guest code crate compiles to RISC-V ELF binaries that run inside the SP1 zkVM. The modified SP1 core sits between them, which route syscalls through four layers to either the precompile chips, which is extension path or the software computation path, which is baseline path.

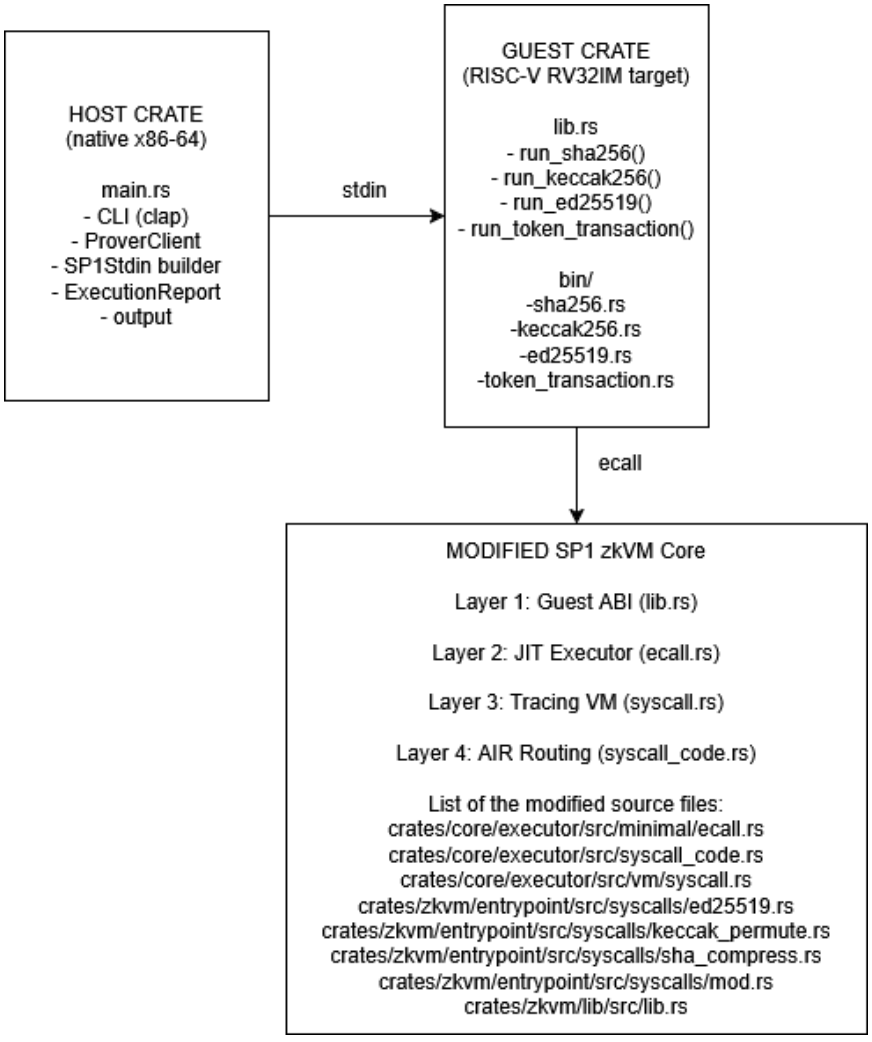


Figure 4.1 System Block Diagram

4.2 System Components Specifications

This will show the core components of SP1 zkVM by listing their functional role, implementation technologies and key dependencies. This enables the systematic comparison between baseline software implementations and extension implementations.

Component	Role	Language / Tech	Key Dependency
Host CLI harness	Drives execution, collects metrics	Rust	sp1-sdk, clap, serde_json
Guest benchmark	Implements baseline and extension paths	Rust (no_std)	sp1-zkvm, sha2, sha3, curve25519-dalek
Guest binary bins	Entry points for each benchmark	Rust (no_std)	sp1-zkvm entrypoint macro
SP1 Guest ABI	Defines syscall constraints in 0x01 namespace	Rust	core::arch::asm
SP1 JIT Executor	Routes extension codes at runtime	Rust	sp1-jit
SP1 Tracing VM	Remaps and dispatches to chip handlers	Rust	sp1-core-executor
SP1 AIR Routing	Maps codes to chip identifiers	Rust	sp1-stark
SHA precompile	Proves SHA-256 compress + extend	Rust /AIR	sp1-curves
Keccak precompile chip	Proves Keccak-f[1600] permutation	Rust /AIR	sp1-curves
Ed25519 precompile chip	Proves twisted Edwards point addition	Rust / AIR	sp1-curves

Table 4.2 System Component Specifications

4.3 Extension Code Design

This section will be explaining the design of the three ISA extension syscall codes and the namespace scheme that is different from standard SP1 syscalls. The purpose of using different namespace for extensions is to isolates them from standard SP1 syscalls. This can prevent SP1 zkVM from breaking and also will still have the full compatibility with SP1's existing ecall dispatch infrastructure. The namespace design will ensure deterministic routing through SP1's four layer execution pipeline without requiring modifications to the core proof constraint system. Each 32-bit syscall code is structured with the high byte as a namespace. The standard syscalls will use 0x00 while the extension syscalls will be introduced using 0x01.

Syscall Name	Standard / Extension	Hex code	Syscall Code Structure (32-bit)			
SHA_COMPRESS	Standard	0x00010106	0x00	0x01	0x01	0x06
SHA256_FOLD	Extension	0x01010106	0x01	0x01	0x01	0x06
Keccak_PERMUTE	Standard	0x00010109	0x00	0x01	0x01	0x09
SPONGE_F1600	Extension	0x01010109	0x01	0x01	0x01	0x09
ED_ADD	Standard	0x00010107	0x00	0x01	0x01	0x07
TED255_ADD	Extension	0x01010107	0x01	0x01	0x01	0x07

Table 4.3 Syscall Code Structure

4.4 System Components Interaction Operations

The three core components are the host evaluation harness, the guest benchmark program and the modified SP1 zkVM core. In the first interaction (host to guest), the host harness constructs will construct an SP1Stdin object containing two values which are boolean of "use_extension" flag and unsigned 32-bit iteration count. Then, these will be written to the hint stream before execution begins. The host then selects the appropriate RISC-V ELF binary based on the chosen benchmark program and invokes "ProverClient.execute()" to begin execution inside the SP1 zkVM.

Next interaction (guest to SP1 core), the guest program reads the two input values from the hint stream using "sp1_zkvm::io::read()". Based on the value of "use_extension", the guest program will choose either one of the two execution paths which are baseline or extension. The

baseline path, the guest invokes standard Rust cryptographic library functions which are compiled to sequences of individual RISC-V arithmetic and memory instructions. The extension path, the guest will call the syscalls which are syscall_sha256_fold(), syscall_sponge_f1600() or syscall_ted255_add(). These syscalls will emit a single RISC-V 'ecall' instruction with the corresponding extension syscall code. When the 'ecall' instruction is encountered during the execution, the SP1 zkVM core intercepts it and begins routing it through the four-layer syscall architecture which are layer 1 the Guest ABI, layer 2 JIT executor, layer 3 Tracing VM and layer 4 AIR routing.

Layer 1-2 recognize and execute the extension syscall codes as equivalent to their standard counterparts. Then at layer 3 these extension codes are remapped to standard codes to ensure consistency in the execution trace. This will prevent the layer 4 from breaking, where layer 4 is for mapping AIR chips which extension will map to the same chip as their counterparts. After all four layers, the execution will remain correct. The next interaction (guest to host), the result of the benchmark will be written back to the guest state and once execution completes the host gets the result and display it out.

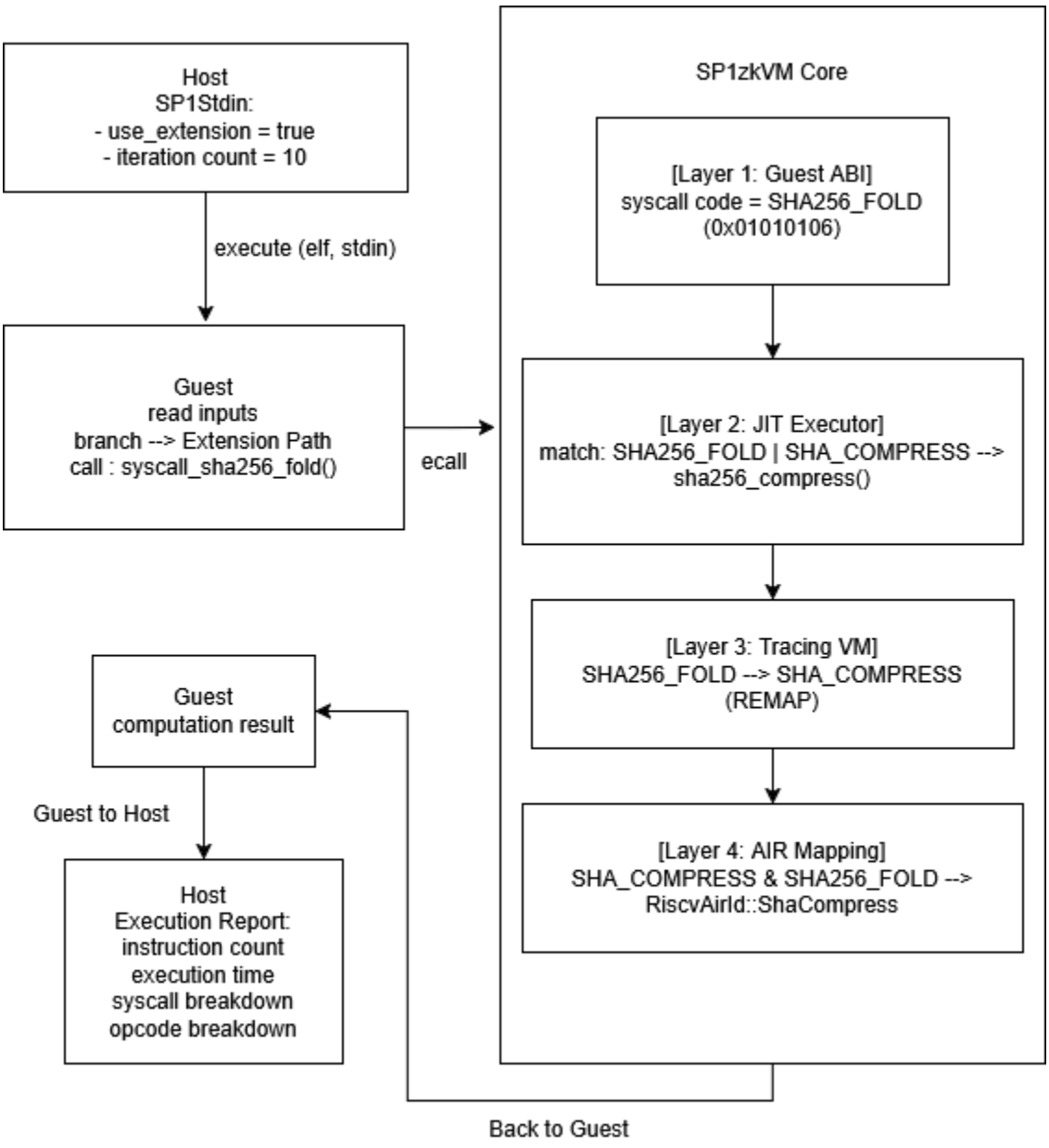


Figure 4.4 Component Interaction Sequence Diagram

Chapter 5

System Implementation

5.1 Hardware Setup

This project is a software-only implementation that runs entirely on a standard development machine. There will be no specialised hardware required. The development was conducted on the following hardware.

Component	Specification
Processor	Intel i5-8350U (x86-64 CPU)
Memory	16GB RAM
Storage	Standard SSD (500GB NVME SSD)
Operating System	Ubuntu 24.04.3 LTS
Network	Required for Cargo dependency resolution during initial build only

Table 5.1 Hardware Setup

5.2 Software Setup

The following are the software components must be installed and configured before building and running the project.

Software	Version/Source	Purpose
Rust	Stable (2021 edition)	Primary programming language for both host and guest crates
SP1 Toolchain (sp1up)	SP1 version 6.1.0	Cross-compiles guest Rust code to RISC-V RV32IM ELF binary
Cargo prove	Installed via sp1up	Builds and embeds guest ELF binaries into host binary
Git	Latest	Version control
SP1 zkVM source code	SP1 version 6.1.0	The core zkVM that was modified to add ISA extensions
Cargo workspace	-	Manages both host and guest crates in a single workspace
Visual Studio Code	Latest	Editing or writing code
Ubuntu (Linux)	24.04.3 LTS	Operating System

Table 5.2 Software Setup

5.3 Setting and Configuration

5.3.1 Workspace Structure

```
SP1_Project/
|---- Cargo.toml (workspace manifest)
|---- script/ (host)
|       |---- Cargo.toml (host crate)
|       |---- src/
|               |---- bin/
|                       |----main.rs (CLI + ProverClient harness)
|---- program/ (guest)
        |---- Cargo.toml (guest crate)
        |---- src/
                |---- lib.rs (shared benchmark logic)
                |---- main.rs
                |---- bin/
                        |---- sha256.rs
                        |---- keccak.rs
                        |---- ed25519.rs
                        |---- token_transaction.rs
```

Figure 5.3.1 Project Directory Structure

5.3.2 Key Cargo Configuration

The following are the setup for each Cargo.toml. The first one will be the root Cargo.toml (workspace manifest).

Figure 5.3.2.1 Cargo.toml (workspace manifest)

In Cargo.toml (workspace manifest), the "[patch.crates-io]" is required to link to modified SP1 source code.

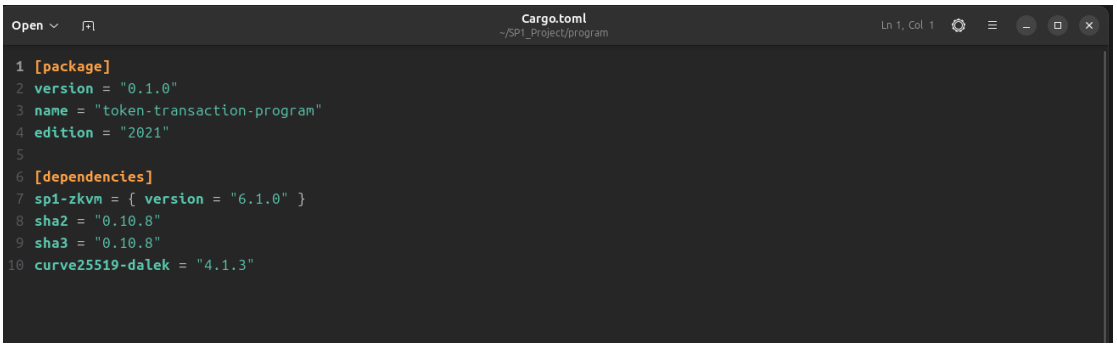


Figure 5.3.2.2 Cargo.toml (guest)

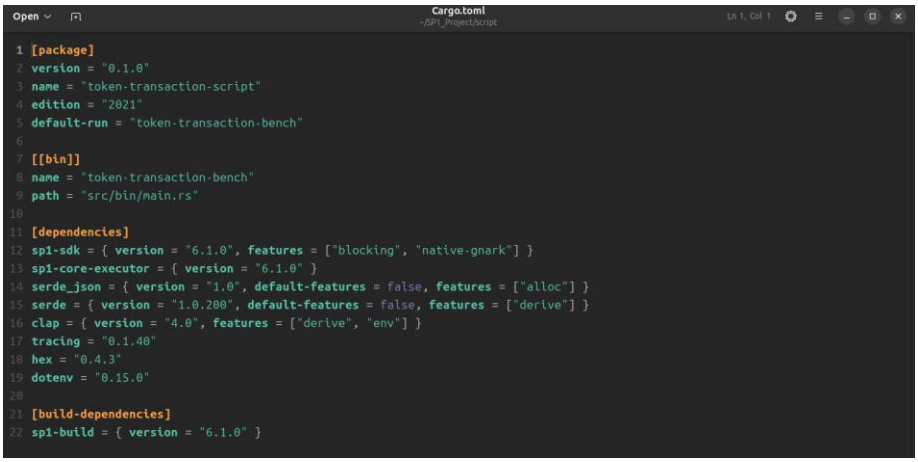


Figure 5.3.2.3 Cargo.toml (host)

5.3.3 SP1 source code modification

To make the extension working, the modification of SP1 source code is necessary. Here are the configurations that are done to make the extensions work. There are eight modified source code files.

sp1/crates/zkvm/entrypoint/src/syscalls/mod.rs

```rust
/// Executes the `POSEIDON2` permutation syscall.
pub const POSEIDON2: u32 = 0x00_00_01_33;

/// Executes the project `sha256.fold` ISA extension.
pub const SHA256_FOLD: u32 = 0x01_01_01_06;

/// Executes the project `sponge.f1600` ISA extension.
pub const SPONGE_F1600: u32 = 0x01_01_01_09;

/// Executes the project `ted255.add` ISA extension.
pub const TED255_ADD: u32 = 0x01_01_01_07;
```

Figure 5.3.3.1 Add extensions in 0x01 namespace

sp1/crates/zkvm/lib/src/lib.rs

```rust
extern "C" {
    /// Executes the project `sha256.fold` ISA extension.
    pub fn syscall_sha256_fold(w: *mut [u64; 64], state: *mut [u64; 8]);

    /// Executes the project `sponge.f1600` ISA extension.
    pub fn syscall_sponge_f1600(state: *mut [u64; 25]);

    /// Executes the project `ted255.add` ISA extension.
    pub fn syscall_ted255_add(p: *mut [u64; 8], q: *const [u64; 8]);
```

Figure 5.3.3.2 Add extern C declaration for three new syscall functions

sp1/crates/zkvm/entrypoint/src/syscalls/sha_compress.rs

```rust
/// Executes the project `sha256.fold` ISA extension.
///
/// This is an SP1 syscall/precompile alias for SHA-256 compression. The executor routes it to the
/// existing SHA compression chip, so the result is still validated by the established cross-table
/// lookup constraints.
#[allow(unused_variables)]
#[no_mangle]
pub extern "C" fn syscall_sha256_fold(w: *mut [u64; 64], state: *mut [u64; 8]) {
    #[cfg(target_os = "zkvm")]
    unsafe {
        asm!(
            "ecall",
            in("t0") crate::syscalls::SHA256_FOLD,
            in("a0") w,
            in("a1") state,
        );
    }
}
```

Figure 5.3.3.3 Implemented syscall_sha256_fold() with ecall asm

sp1/crates/zkvm/entrypoint/src/syscalls/keccak_permute.rs

```rust
/// Executes the project `sponge.f1600` ISA extension.
///
/// This aliases SP1's Keccak-f[1600] permutation syscall and is proven by the existing Keccak
/// precompile and controller chips.
#[allow(unused_variables)]
#[no_mangle]
pub extern "C" fn syscall_sponge_f1600(state: *mut [u64; 25]) {
    #[cfg(target_os = "zkvm")]
    unsafe {
        asm!(
            "ecall",
            in("t0") crate::syscalls::SPONGE_F1600,
            in("a0") state,
            in("a1") 0
        );
    }

    #[cfg(not(target_os = "zkvm"))]
    unreachable!()
}
```

Figure 5.3.3.4 Implemented syscall_sponge_f1600() with ecall asm

sp1/crates/zkvm/entrypoint/src/syscalls/ed25519.rs

```rust
/// Executes the project `ted255.add` ISA extension.
///
/// This aliases SP1's Ed25519 add precompile. The executor writes the resulting affine point into
/// `p`, and the Ed25519 add chip validates the point-addition relation by cross-table lookup.
#[allow(unused_variables)]
#[no_mangle]
pub extern "C" fn syscall_ted255_add(p: *mut [u64; 8], q: *const [u64; 8]) {
    #[cfg(target_os = "zkvm")]
    unsafe {
        asm!(
            "ecall",
            in("t0") crate::syscalls::TED255_ADD,
            in("a0") p,
            in("a1") q
        );
    }

    #[cfg(not(target_os = "zkvm"))]
    unreachable!()
}
```

Figure 5.3.3.5 Implemented syscall_ted255_add() with ecall asm

sp1/crates/core/executor/src/minimal/ecall.rs

```rust
pub fn ecall_handler(ctx: &mut impl SyscallContext, code: SyscallCode) -> u64 {
    let arg1 = ctx.rr(RiscRegister::X10);
    let arg2 = ctx.rr(RiscRegister::X11);

    // Unconstrained mode is not allowed for any syscall other than WRITE and HALT.
    if ctx.is_unconstrained()
        && (code != SyscallCode::WRITE && code != SyscallCode::EXIT_UNCONSTRAINED)
    {
        panic!("Unconstrained mode is not allowed for this syscall: {code:?}");
    }

    match code {
        SyscallCode::SHA_EXTEND => unsafe { sha256_extend(ctx, arg1, arg2) },
        SyscallCode::SHA_COMPRESS | SyscallCode::SHA256_FOLD => unsafe {
            sha256_compress(ctx, arg1, arg2)
        },
        SyscallCode::KECCAK_PERMUTE | SyscallCode::SPONGE_F1600 => unsafe {
            keccak_permute(ctx, arg1, arg2)
        },
        SyscallCode::ED_ADD | SyscallCode::TED255_ADD => unsafe {
            edwards_add::<Ed25519>(ctx, arg1, arg2)
        },
        SyscallCode::SECP256K1_ADD => unsafe {
            weierstrass_add_assign_syscall::<Secp256k1>(ctx, arg1, arg2)
        },
```

Figure 5.3.3.6 Add three extension codes to match arm | group

crates/core/executor/src/vm/syscall.rs

```rust
pub(crate) fn sp1_ecall_handler<'a, RT: SyscallRuntime<'a>>(
    rt: &mut RT,
    code: SyscallCode,
    args1: u64,
    args2: u64,
) -> Option<u64> {
    // Precompiles may directly modify the clock, so we need to save the current clock
    // and reset it after the syscall.
    let clk = rt.core().clk();

    #[allow(clippy::match_same_arms)]
    let ret = match code {
        // Edwards curve operations
        SyscallCode::ED_ADD => {
            precompiles::edwards::edwards_add::<RT, Ed25519>(rt, code, args1, args2)
        }
        SyscallCode::TED255_ADD => precompiles::edwards::edwards_add::<RT, Ed25519>(
            rt,
            SyscallCode::ED_ADD,
            args1,
            args2,
        ),
        SyscallCode::ED_DECOMPRESS => {
            precompiles::edwards::edwards_decompress(rt, code, args1, args2)
        }
        SyscallCode::UINT256_MUL => uint256::uint256_mul(rt, code, args1, args2),
        SyscallCode::UINT256_MUL_CARRY | SyscallCode::UINT256_ADD_CARRY => {
            uint256_ops::uint256_ops(rt, code, args1, args2)
        }
        SyscallCode::SHA_COMPRESS => precompiles::sha256::sha256_compress(rt, code, args1, args2),
        SyscallCode::SHA256_FOLD => precompiles::sha256::sha256_compress(
            rt,
            SyscallCode::SHA_COMPRESS,
            args1,
            args2,
        ),
        SyscallCode::SHA_EXTEND => precompiles::sha256::sha256_extend(rt, code, args1, args2),
        SyscallCode::KECCAK_PERMUTE => {
            precompiles::keccak256::keccak256_permute(rt, code, args1, args2)
        }
        SyscallCode::SPONGE_F1600 => precompiles::keccak256::keccak256_permute(
            rt,
            SyscallCode::KECCAK_PERMUTE,
            args1,
            args2,
        ),
```

Figure 5.3.3.7 Add three extension dispatch cases with canonical remapping

sp1/crates/core/executor/src/syscall_code.rs

```
other codes.........

pub enum SyscallCode {
    ......
    SHA256_FOLD = 0x01_01_01_06,
    SPONGE_F1600 = 0x01_01_01_09,
    TED255_ADD = 0x01_01_01_07,
    ......
}

pub fn from_u32(value: u32) -> Self {
    match value {
        ....
        0x01_01_01_06 => SyscallCode::SHA256_FOLD,
        0x01_01_01_09 => SyscallCode::SPONGE_F1600,
        0x01_01_01_07 => SyscallCode::TED255_ADD,
        ....
    }
}

pub fn as_air_id(self) -> Option<RiscvAirId> {
    Some(match self {
        ....
        SyscallCode::SHA_COMPRESS | SyscallCode::SHA256_FOLD => RiscvAirId::ShaCompress,
        SyscallCode::ED_ADD | SyscallCode::TED255_ADD => RiscvAirId::EdAddAssign,
        SyscallCode::ED_DECOMPRESS => RiscvAirId::EdDecompress,
        SyscallCode::KECCAK_PERMUTE | SyscallCode::SPONGE_F1600 => {
        RiscvAirId::KeccakPermute
        }
        ....
    })
}

pub fn touched_addresses(&self) -> usize {
    match self{
        ....
        SyscallCode::SHA_COMPRESS | SyscallCode::SHA256_FOLD => 72,
        SyscallCode::KECCAK_PERMUTE | SyscallCode::SPONGE_F1600 => 25,
        SyscallCode::ED_ADD | SyscallCode::TED255_ADD => 16,
        ....
    }
}

pub fn touched_pages(&self) -> usize {
    match self{
        ....
        SyscallCode::SHA_COMPRESS | SyscallCode::SHA256_FOLD => 3 * 2,
        SyscallCode::KECCAK_PERMUTE | SyscallCode::SPONGE_F1600 => 2 * 2,
        SyscallCode::ED_ADD | SyscallCode::TED255_ADD => 2 * 2,
        ....
    }
}

other codes.........
```

Figure 5.3.3.8 Add extension variants

File Modified	Layer	Change
mod.rs	Layer 1 – Guest ABI	Add SHA256_FOLD, SPONGE_F1600, TED255_ADD constants in 0x01 namespace
lib.rs	Layer 1 – Guest ABI	Add extern C declarations for syscall functions
sha_compress.rs	Layer 1 – Guest ABI	Implement syscall_sha256_fold() with ecall asm
keccak_permute.rs	Layer 1 – Guest ABI	Implement syscall_sponge_f1600() with ecall asm
ed25519.rs	Layer 1 – Guest ABI	Implement syscall_ted255_add() with ecall asm
ecall.rs	Layer 2 – JIT Executor	Add three extension codes to match arm | groups
syscall.rs	Layer 3 – Tracing VM	Add three extension dispatch cases with canonical remapping
syscall_code.rs	Layer 4 – AIR Routing	Add extension variants to as_air_id() match arms

Table 5.3.3 Summary of SP1 source code modifications

5.4 System Operation

5.4.1 Running the Benchmark

The benchmark is executed from the host crate using Cargo with the following command pattern:

Run sha-256 program,

Baseline with 10 iterations:

cargo run –release -p token-transaction-script -- --program sha256 –mode baseline –iterations 10

Extension with 10 iterations:

cargo run –release -p token-transaction-script -- --program sha256 –mode extension –iterations 10

Run Keccak-256 program,

Baseline with 10 iterations:

cargo run –release -p token-transaction-script -- --program keccak256 –mode baseline –iterations 10

Extension with 10 iterations:

cargo run –release -p token-transaction-script -- --program keccak256 –mode extension –iterations 10

Run ed25519 program,

Baseline with 10 iterations:

cargo run –release -p token-transaction-script -- --program ed25519 –mode baseline –iterations 10

Extension with 10 iterations:

cargo run –release -p token-transaction-script -- --program ed25519 –mode extension –iterations 10

Run token transaction program,

Baseline with 10 iterations:

cargo run –release -p token-transaction-script -- --program token-transaction –mode baseline –iterations 10

Extension with 10 iterations:

cargo run –release -p token-transaction-script -- --program token-transaction –mode extension –iterations 10

The following are one of the performance result:

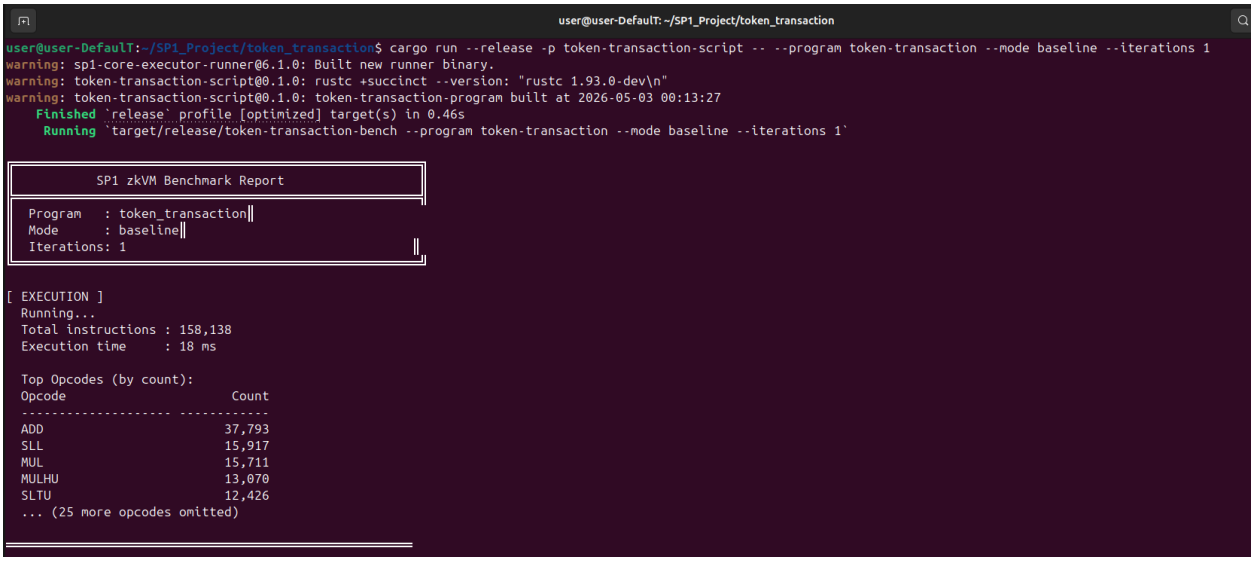


Figure 5.4.1.1 Performance result of token-transaction baseline mode 1 iteration

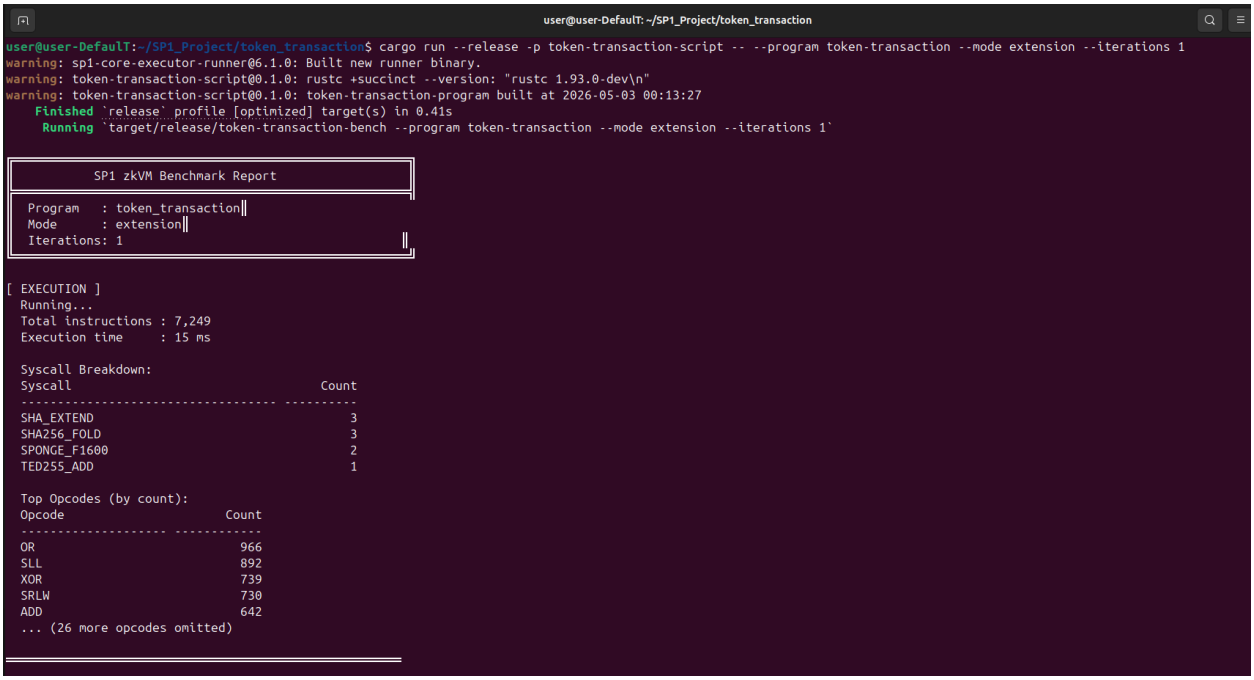


Figure 5.4.1.2 Performance result of token-transaction extension mode 1 iteration

5.4.2 Syscall Routing Verification

To verify that the extension is successfully executed, inspecting the syscall breakdown part of the report. If the extension is successfully executed there should be syscall code while for the baseline there will not be syscall code.

For example:

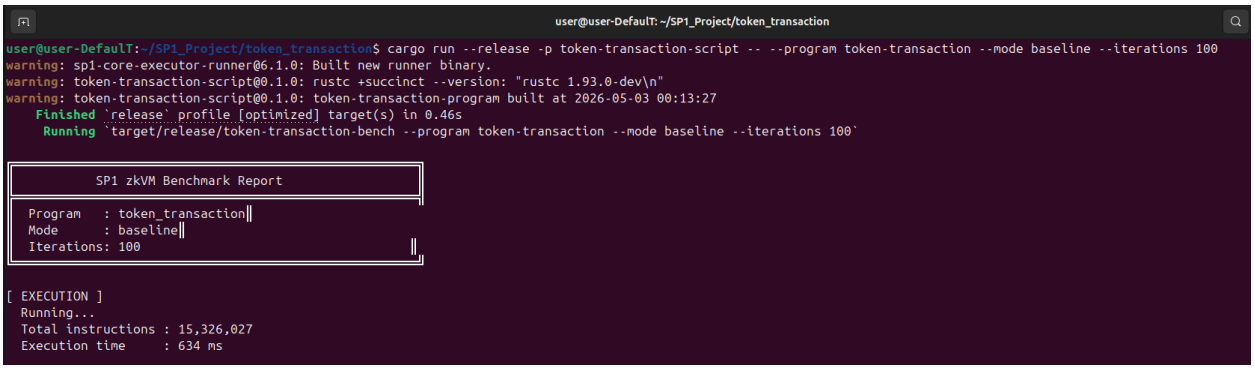


Figure 5.4.2.1 Performance result of baseline mode in token transaction

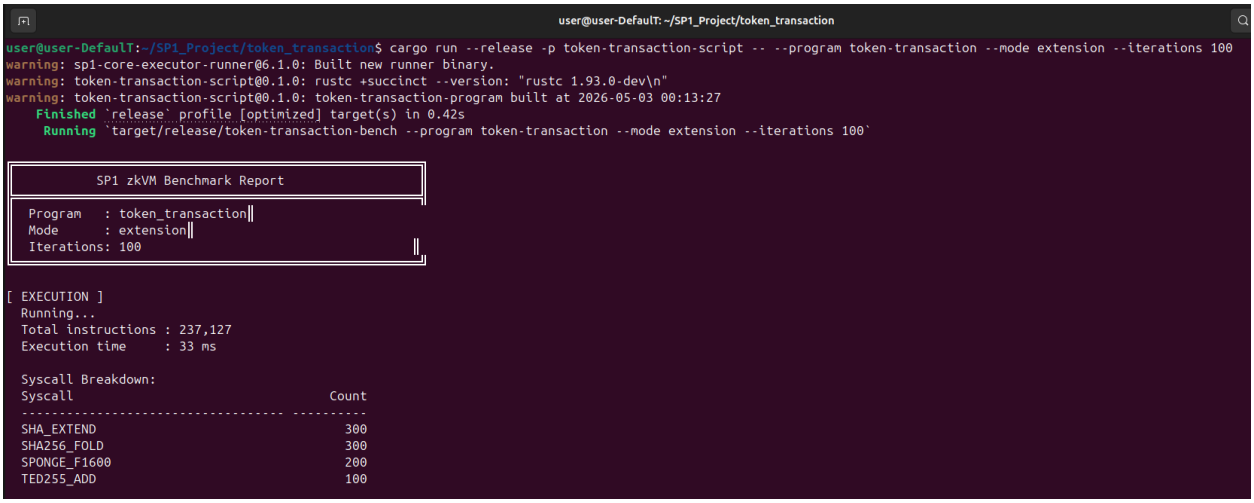


Figure 5.4.2.2 Performance result of extension mode in token transaction

5.5 Implementation Issues and Challenges

In this project, the most time-consuming challenge is that to understand the internal architecture of SP1 zkVM. This is required because in order to implement the extensions correctly having a good understanding of the internal architecture of SP1 zkVM is critical. The core finding is that the Layer 3 in SP1 zkVM core which is Tracing VM must remap the extension codes to canonical codes before chip dispatch. This is because if the extension codes do not remap and pass through unchanged. The execution record would contain events with the extension codes that cannot be match with the prover's chip against its expected event stream, which will cause fail. This is not documented clearly in SP1's public documentation and need to be discovered through code analysis.

Another issue that encounters in this project is that SP1 provides library-level precompile acceleration through the "[patch.crates-io]" mechanism. This sets the question to whether the unpatched software baseline is the most suitable comparison. After analysis, it is determined that the unpatched software baseline can represent the execution cost without the library-level precompile acceleration. This is a useful information for understanding the maximum possible instruction count reduction. To enable extension for Ed25519, the correct input format must be solved as it is required to match the SP1's internal extended affine point representation. The reason is that an Ed25519 point is a location on elliptic curve and every point has two coordinates and each coordinate is a 256-bit number. SP1 expects each Ed25519 point as 64-bit so it will be split into eight 64-bit little-endian numbers "[u64; 8]". This matches the format used by SP1's existing ED_ADD syscall, so that it can prevent wrong input and wrong output [10].

Besides that, creating an extension for SHA-256 will bypass the sha2 library and cause format issue. This is because the sha2 library is responsible for message padding before processing. So manual padding is needed as the solution is to append a 0x80 byte immediately after the last message byte to mark the end and filling the remaining bytes with zeros. Then, encoding the message length in bits as a 64-bit big-endian integer in the final 8 bytes of the padded block. Without this it will produce incorrect output [22]. The full zk proof generation for the extension could not be completed because of the available development hardware is lack of memory (RAM). This is because the syscalls create "lookup tables" that need a lot of memory, limited the evaluation to execute-only mode.

5.6 Concluding Remark

The implementation of extension successfully demonstrates that SP1 zkVM's syscall architecture supports new named ISA entry points through namespace separation without the modification of AIR proof constraint system. All three extensions are correctly wired through all four architectural layers and produce execution results. The benchmark suite provides a complete and reproducible framework for evaluating the performance differences between software computation (baseline) and precompile invocation (extension) for blockchain token transaction workloads.

CHAPTER 6

System Evaluation and Discussion

6.1 System Testing and Performance Metrics

6.1.1 Testing Approach

The system testing was conducted at two levels which are functional correctness testing verified that each ISA extension successfully execute and produce instruction counts and also the expected syscall codes in the execution report. Another level is performance testing which compare the total instruction counts between the baseline and extension across three iteration scales.

Instruction count is selected as the primary performance metric because it directly determines execution witness size in SP1 zkVM. Every RISC-V instruction executed by the virtual machine contributes to the proof witness and a longer witness requires more memory and time to prove. Besides instruction count, instruction speedup, wall-clock execution time, transaction per second and execution speedup ratio will also be recorded as performance metrics. With all of these performance metrics, it provides a comparative performance analysis between baseline and extension.

6.1.2 Performance Metrics Definition

Metric	Formula
Instruction Count	Total_instruction_count()
Instruction speedup ratio	Baseline instruction / extension instruction
Transaction per second (TPS/Throughput)	Transaction / execution time
Wall-clock execution time	Std::time::instant
Execution speedup ratio	Baseline execution time / extension execution time

Table 6.1.2 Performance Metrics Definitions

6.2 Testing Setup and Results

6.2.1 Testing Environment

Parameter	Value
Execution mode	Execute only (no proof generation)
Baseline	Unpatched sha2=0.10.8, sha3=0.10.8, curve25519-dalek=4.1.3
Extension path	Direct syscall_sha256_fold, syscall_sponge_f1600, syscall_ted255_add
Iteration scales	1,10,100 operations per benchmark run (transaction)
Benchmarks	SHA-256, Keccak-256, Ed25519, token transaction (combine)

Table 6.2.1 Testing Environment Configuration

6.2.2 Syscall Routing Correctness Verification

Before analyzing performance, the correctness of extension syscall routing was verified through syscall_counts field of the execution report. The following are the syscall result of baseline and extension of token transaction at 10 iterations.

Path	SHA_EXTEND	SHA256_FOLD	SPONGE_F1600	TED255_ADD	Total
Baseline	0	0	0	0	0
Extension	30	30	20	10	90

Table 6.2.2 Syscall Breakdown verification at 10 iterations

The syscall breakdown results show the difference between baseline and extension. The baseline will have zero syscall while the extension will have total 90 syscall in token transaction program at 10 iterations. For SHA-256 specifically, SHA_EXTEND and SHA256_FOLD appear together because sha-256 compress is a two-stage process. SHA_EXTEND expands message for SHA256_FOLD and must be called so SHA_EXTEND and SHA256_FOLD will appear together.

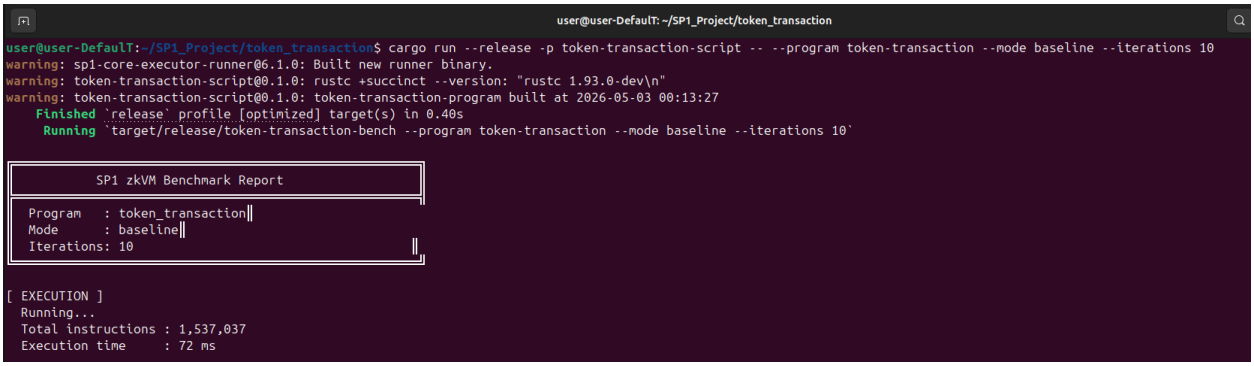


Figure 6.2.2.1 Baseline syscall result in 10 iterations

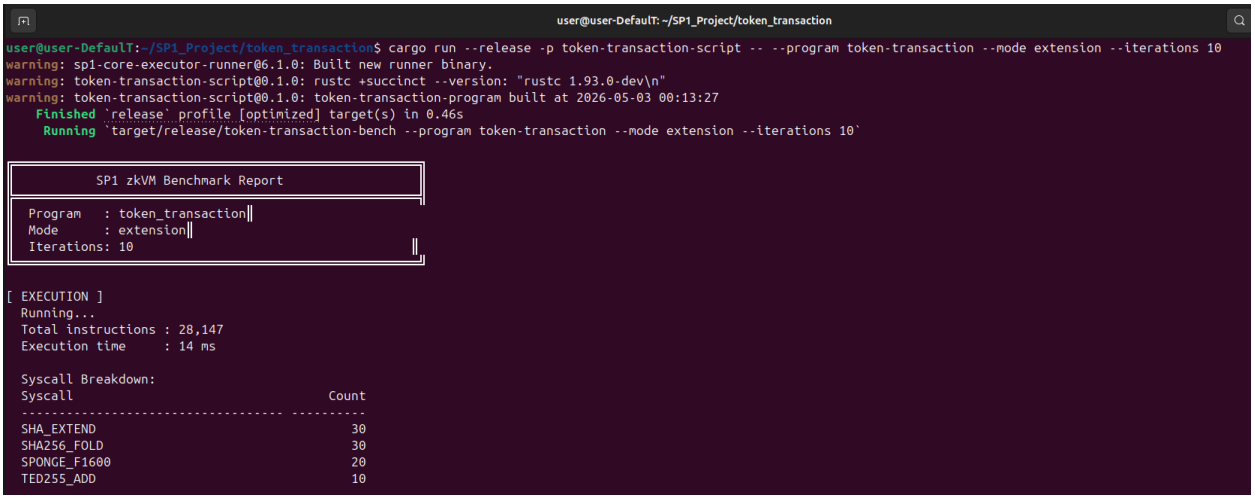


Figure 6.2.2.2 Extension syscall result in 10 iterations

6.2.3 SHA-256 Results

Iteration	Baseline Instruction	Extension Instruction	Instruction Speedup
1	17,345	6,965	2.5x
10	129,134	25,334	5.1x
100	1,247,024	209,024	6.0x

Table 6.2.3.1 SHA-256 Instruction Count Metrics

Iteration	Base execution time (ms)	Ext execution time (ms)	Throughput (base)	Throughput (extension)	Execution time ratio
1	9	9	111.1	111.1	1.0x
10	14	10	714.3	1000	1.4x
100	60	21	1,666.7	4,761.9	2.9x

Table 6.2.3.2 SHA-256 Time and Throughput Metrics

The extension for SHA-256 achieves 6 times instruction speedup at 100 iterations and reducing the baseline instruction count from 1,247,024 to 209,024 instructions. The instruction speedup increases with the increase of iteration. At 1 iteration both baseline and extension had the same execution time which recorded as 9ms. This shows that the execution speedup only visible at higher iteration counts.

6.2.4 Keccak-256 Results

Iteration	Baseline Instruction	Extension Instruction	Instruction Speedup
1	11,693	5,207	2.2x
10	72,542	7,682	9.4x
100	681,032	32,432	21.0x

Table 6.2.4.1 Keccak-256 Instruction Count Metrics

Iteration	Base execution time (ms)	Ext execution time (ms)	Throughput (base)	Throughput (extension)	Execution time ratio
1	9	8	111.1	125	1.1x
10	12	8	833.3	1,250	1.5x
100	37	10	2,702,7	10,000	3.7x

Table 6.2.4.2 Keccak-256 Time and Throughput Metrics

The extension for Keccak-256 achieves 21 times instruction speedup at 100 iterations and reducing the baseline instruction count from 681,032 to 32,432 instructions. The instruction speedup increases with the increase of iteration. The throughput of extension of Keccak-256 at 100 iterations are 10,000 which means it can process 10,000 instruction per second compare to the baseline which is just 2,702.7 instruction per second.

6.2.5 Ed25519 Results

Iteration	Baseline Instruction	Extension Instruction	Instruction Speedup
1	132,710	5,127	25.9x
10	1,283,018	6,738	190.4x
100	12,786,098	22,848	559.6x

Table 6.2.5.1 Ed25519 Instruction Count Metrics

Iteration	Base execution time (ms)	Ext execution time (ms)	Throughput (base)	Throughput (extension)	Execution time ratio
1	15	9	66.7	111.1	1.7x
10	64	11	156.3	909.1	5.8x
100	532	18	188.0	5,555,6	29.6x

Table 6.2.5.2 Ed25519 Time and Throughput Metrics

The extension for Ed25519 achieves 559.6 times instruction speedup at 100 iterations and reducing the baseline instruction count from 12,786,098 to 22,848 instructions. The execution time speedup ratio is 29.6 times improvement. But an important finding is that the SP1's patched curve25519-dalek library does not activate for the baseline of Ed25519. So, the extension of Ed25519 can have a huge improvement.

6.2.6 Token Transaction Results

Iteration	Baseline Instruction	Extension Instruction	Instruction Speedup
1	158,138	7,249	21.8x
10	1,537,037	28,147	54.6x
100	15,326,027	237,127	64.6x

Table 6.2.6.1 Token Transaction Instruction Count Metrics

Iteration	Base execution time (ms)	Ext execution time (ms)	Throughput (base)	Throughput (extension)	Execution time ratio
1	18	15	55.6	66.7	1.2x
10	77	15	129.9	666.7	5.1x
100	634	33	157.7	3,030.3	19.2x

Table 6.2.6.2 Token Transaction Time and Throughput Metrics

The token transaction benchmark demonstrates the performance gains from individual SHA-256, Keccak-256 and Ed25519 extensions compound in realistic blockchain workloads. At 100 iterations, the extension achieves 64.6 times instruction reduction and throughput improve from 157.7 to 3030.3 which means that the extension can process over 3,000 token transactions per second compared to baseline. The execution speedup ratio improved from 634ms to 33ms which is 19.2 times improvement. This show that token transaction can scale with the help of extension. Note that, most of the improvement is come from Ed25519 as stated at 6.2.5.

6.2.7 Summary of all Performance Metrics

Benchmark	Instruction speedup ratio	Throughput(TPS, Ext)	Execution time speed up ratio
SHA-256 (100)	6.0x	4,761.9	2.9x
Keccak-256 (100)	21.0x	10,000	3.7x
Ed25519 (100)	559.6x	5,555.6	29.6x
Token transaction (100)	64.6x	3,030.3	19.2x

Table 6.2.7 Summary of all Performance Metrics at 100 iterations

The summary confirms that all four performance metrics show consistent improvement across all benchmarks. Ed25519 achieves the highest instruction speedup at 559.6 times and the lowest per-transaction instruction cost at 229 instructions. Keccak-256 achieves the highest throughput at 10,000 operations per second at 100 iterations. The token transaction composite demonstrates that these gains translate to a practical throughput of over 3,000 transactions per second for the full workload.

6.3 Project Challenges

One of the biggest challenges for this project is that proof generation could not be completed because of hardware limitations. The current method of extension in SP1 zkVM will require a lot of memory (RAM). This is because the SP1 prover requires substantial memory to construct the proof witness specifically for extension in this project. This limited the project's evaluation to execution only, which measures instruction count but did not have the proof generation time. This is acknowledged as a limitation of the project's evaluation.

The next project challenge is understanding the SP1's multi-layer syscall architecture. SP1's syscall architecture will have four distinct layer which are guest ABI, JIT executor, tracing VM and AIR chip routing each implemented at different crate with different roles. Without clear understanding can cause a lot of breaking in SP1 vkZM source code. Another challenge is when it comes to setting the baseline for this project. This is because SP1 already provides a library-level precompile acceleration with "[patch.crates-io]" mechanism. After careful analysis, the baseline for this project will be unpatched library implementation as it represents the execution cost without any acceleration.

6.4 Objectives Evaluation

The objectives of this project are achieved successfully as the objectives mentioned in section 1.2 within the project scope. Develop and implement the three ISA extensions which are SHA256_FOLD, SPONGE_F1600 and TED255_ADD are correctly implemented. To show the instruction count and execution time reductions are achieved. Conduct a comparative performance analysis is also achieved.

Objective	Status	Evidence
Implement ISA extensions	Achieved	Modified 8 source code files, syscall counts confirm routing, extension code appear in execution report.
Demonstrate instruction count and execution reductions with extension	Achieved	Extension version have lower instruction count and execution compare to baseline version
Conduct a comparative performance analysis	Achieved	All the performance metrics collected at all 3 scales which are 1, 10 and 100 iterations

Table 6.4 Objective Evaluation Summary

6.5 Concluding Remark

The evaluation results confirm that RISC-V ISA extensions implemented as precompile-aliased syscalls in SP1 zkVM produce consistent improvements across all the performance metrics. The evidence of it is that token transaction with the help of extension will have 64.6 times instruction count speedup and 19.2 times execution time speedup. The syscall breakdown analysis is an additional validation that both baseline and extension execute different computational strategies. Besides that, the unpatched Ed25519 shows a big gap in performance as the extension version of Ed25519 has huge improvement.

CHAPTER 7

Conclusion and Recommendations

7.1 Conclusion

This project set out to find out whether RISC-V ISA extensions in SP1 zkVM could reduce the instruction trace cost of blockchain token transaction workloads. Demonstrate that SP1's syscall architecture supports new named entry points without requiring modifications to its proof constraint system and both of this are achieved.

Three ISA extensions were designed and implemented, SHA256_FOLD aliasing SP1's SHA-256 compression precompile. SPONGE_F1600 aliasing the Keccak-f[1600] permutation precompile. Lastly, TED255_ADD aliasing with the Ed25519 twisted Edwards curve addition precompile. Each extensions are assigned to unique syscall code in dedicated 0x01_xx_xx_xx namespace. It is also correctly wired through all four layers of SP1's syscall stack. The implementation of these ISA extensions requires modifications to eight source files.

Performance evaluations confirmed the instruction count and execution time reductions compared to the software baseline. One of the results is that the token transaction achieved 64.6 times instruction count speedup and 19.2 times execution time speedup with the implementation of extension.

An important finding is that SP1's automatic library patch acceleration does not activate the ED_ADD precompile for baseline Ed25519 because the patch targets scalar multiplication operations. So, the TED255_ADD extension is expected to provide huge improvement. In summary, this project successfully demonstrates the extensibility of SP1 zkVM's RISC-V ISA through namespace syscall aliasing. This project produces a reproducible benchmark framework for token transaction workloads and also provides the evidence that precompile invocation (extension) is a effective strategy for reducing Zk proof costs in blockchain.

7.2 Recommendation

The most impactful extension of this work would be a dedicated AIR chip for the complete token transaction operation. For example, a single chip validating SHA-256 compression, Keccak-256 permutation and Ed25519 point addition. This can eliminate the cross-chip lookup overhead and is the natural next step. Future work should measure the actual proof generation time, peak memory consumption and proof size for both baseline and extension. This is because that is the complete practical cost reduction achievable through ISA-level precompile acceleration.

Besides that, the ISA extension namespace can be extended to additional primitives relevant to blockchain applications. For example, Blake2s, Poseidon2, BLS12-381 pairing operations and secp256k1 ECDSA verification. Each of these has an existing precompile chip in SP1 that can be aliased using the same pattern demonstrated in this project. Furthermore, the correctness of the four-layer syscall design was checked by examining the execution reports. However, formal verification would need to be guarantee that extension and canonical syscalls always behave identically in all cases.

REFERENCES

[1] Synopsys, 2024, “What is RISC-V?” [Online]. Available: https://www.synopsys.com/glossary/what-is-risc-v.html

[2] M. N. M. Bhutta et al., "A Survey on Blockchain Technology: Evolution, Architecture and Security," in IEEE Access, vol. 9, pp. 61048-61073, 2021, doi: 10.1109/ACCESS.2021.3072849.

[3] Succinct Labs, “Introduction - Succinct Docs,” Succinct Documentation. [Online]. Available: https://docs.succinct.xyz/docs/sp1/introduction. [Accessed: May 3, 2026].

[4] Patra, S.P. and Rani, M. (2025) ‘Evaluation and Categorization of Hashing Algorithms Based on Their Applications’, IAENG International Journal of Applied Mathematics, 55(3), pp. 540–552.

[5] E. Cui, T. Li, and Q. Wei, "RISC-V Instruction Set Architecture Extensions: A Survey," IEEE Access, vol. 11, pp. 24696-24711, 2023, doi: 10.1109/ACCESS.2023.3246491.

[6] A. Shchekin, “Effectively hiding sensitive data with RISC-V Zk and custom instructions,” Codasip, Jan. 31, 2024. https://codasip.com/2024/01/31/effectively-hiding-sensitive-data-with-risc-v-zk-and-custom-instructions/ (accessed May 01, 2025).

[7] G. Nişancı, P. G. Flikkema, and T. Yalçın, “Symmetric Cryptography on RISC-V: Performance Evaluation of Standardized Algorithms,” Cryptography, vol. 6, no. 3, p. 41, Aug. 2022, doi: https://doi.org/10.3390/cryptography6030041

[8] M. El-Hajj and B. O. Roelink, "Evaluating the Efficiency of zk-SNARK, zk-STARK, and Bulletproof in Real-World Scenarios: A Benchmark Study," Information, vol. 14, no. 7, p. 395, Jul. 2023, doi: 10.3390/info14070395.

[9] T. Gassmann, S. Chaliasos, T. Sotiropoulos, and Z. Su, "Evaluating Compiler Optimization Impacts on zkVM Performance," arXiv preprint arXiv:2508.17518, Aug. 2025. [Online]. Available: https://arxiv.org/abs/2508.17518

[10] Succinct Labs, "SP1 zkVM source code — ed25519 syscall implementation," GitHub, 2024. [Online]. Available: https://github.com/succinctlabs/sp1

[11] Succinct Labs, “Introducing SP1: A performant, 100% open-source, contributor-friendly zkVM,” *Succinct Blog*, Feb. 14, 2024. [Online]. Available: https://blog.succinct.xyz/introducing-sp1/. [Accessed: May 3, 2026].

[12] B. W. Mezger, D. A. Santos, L. Dilillo, C. A. Zeferino and D. R. Melo, "A Survey of the RISC-V Architecture Software Support," in IEEE Access, vol. 10, pp. 51394-51411, 2022, doi: 10.1109/ACCESS.2022.3174125.

[13] E. Cui, T. Li and Q. Wei, "RISC-V Instruction Set Architecture Extensions: A Survey," in IEEE Access, vol. 11, pp. 24696-24711, 2023, doi: 10.1109/ACCESS.2023.3246491.

[14] RISC-V International, “The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA,” 2019. [Online]. Available: https://riscv.org/

[15] A. Waterman and K. Asanović, “The RISC-V Instruction Set Manual, Volume I: User-Level ISA,” EECS Department, University of California, Berkeley, 2014.

[16] S. Nakamoto, “Bitcoin: A Peer-to-Peer Electronic Cash System,” 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[17] G. Wood, “Ethereum: A Secure Decentralised Generalised Transaction Ledger,” Ethereum Project Yellow Paper, 2014.

[18] National Institute of Standards and Technology (NIST), “Secure Hash Standard (SHS),” FIPS PUB 180-4, Aug. 2015.

[19] National Institute of Standards and Technology (NIST), “SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions,” FIPS PUB 202, Aug. 2015.

[20] S. Josefsson and I. Liusvaara, “Edwards-Curve Digital Signature Algorithm (EdDSA),” RFC 8032, Jan. 2017.

[21] Succinct Labs, “SP1: A Zero-Knowledge Virtual Machine,” GitHub Repository, 2025. [Online]. Available: https://github.com/succinctlabs/sp1

[22] RustCrypto Project Developers, “SHA-2 Hash Functions Implementation,” GitHub Repository, 2024. [Online]. Available: https://github.com/RustCrypto/hashes

[23] D. Tolnay, “tiny-keccak: Keccak Hash Function Implementation in Rust,” GitHub Repository, 2024. [Online]. Available: https://github.com/debris/tiny-keccak

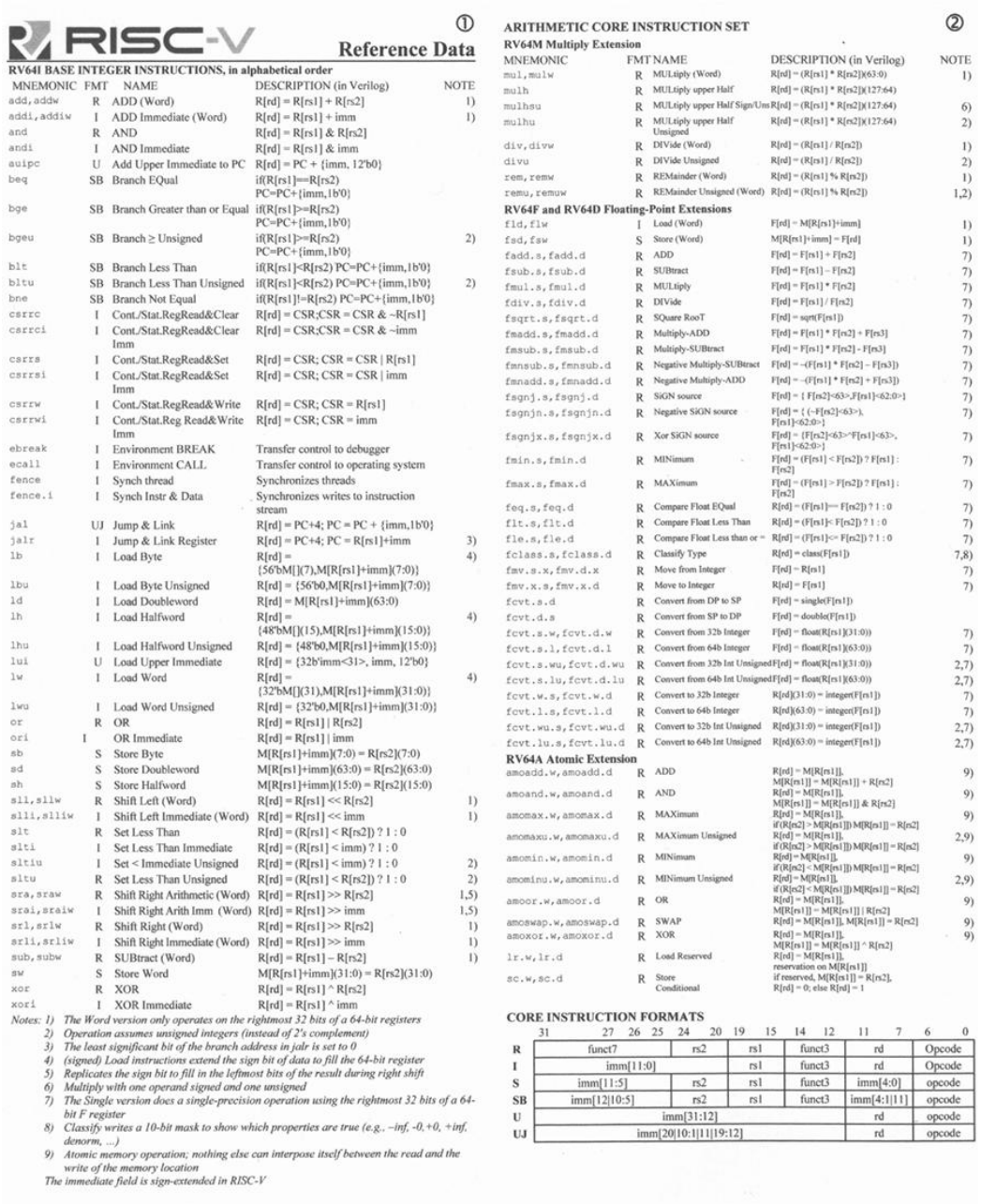
[24] Dalek Cryptography, “ed25519-dalek: Ed25519 Digital Signature Library,” GitHub Repository, 2024. [Online]. Available: https://github.com/dalek-cryptography/ed25519-dalek

[25] The Rust Programming Language, "The Rust reference," 2024. [Online]. Available: https://doc.rust-lang.org/reference/

APPENDIX

Appendix A

RISC-V Architecture



RISC-V Reference Data ①

RV64I BASE INTEGER INSTRUCTIONS, in alphabetical order

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
add,addw	R	ADD (Word)	R[rd] = R[rs1] + R[rs2]	1)
addi,addiw	I	ADD Immediate (Word)	R[rd] = R[rs1] + imm	1)
and	R	AND	R[rd] = R[rs1] & R[rs2]	
andi	I	AND Immediate	R[rd] = R[rs1] & imm	
auipc	U	Add Upper Immediate to PC	R[rd] = PC + {imm, 12'b0}	
beq	SB	Branch EQual	if(R[rs1]==R[rs2) PC=PC+{imm,1b'0}	
bge	SB	Branch Greater than or Equal	if(R[rs1]>=R[rs2) PC=PC+{imm,1b'0}	
bgeu	SB	Branch ≥ Unsigned	if(R[rs1]>=R[rs2) PC=PC+{imm,1b'0}	2)
blt	SB	Branch Less Than	if(R[rs1]<R[rs2) PC=PC+{imm,1b'0}	
bltu	SB	Branch Less Than Unsigned	if(R[rs1]<R[rs2) PC=PC+{imm,1b'0}	2)
bne	SB	Branch Not Equal	if(R[rs1]!=R[rs2) PC=PC+{imm,1b'0}	
csrrc	I	Cont./Stat.RegRead&Clear	R[rd] = CSR;CSR = CSR & ~R[rs1]	
csrrci	I	Cont./Stat.RegRead&Clear Imm	R[rd] = CSR;CSR = CSR & ~imm	
csrrs	I	Cont./Stat.RegRead&Set	R[rd] = CSR; CSR = CSR | R[rs1]	
csrrsi	I	Cont./Stat.RegRead&Set Imm	R[rd] = CSR; CSR = CSR | imm	
csrrw	I	Cont./Stat.RegRead&Write	R[rd] = CSR; CSR = R[rs1]	
csrrwi	I	Cont./Stat.Reg Read&Write Imm	R[rd] = CSR; CSR = imm	
ebreak	I	Environment BREAK	Transfer control to debugger	
ecall	I	Environment CALL	Transfer control to operating system	
fence	I	Synch thread	Synchronizes threads	
fence.i	I	Synch Instr & Data	Synchronizes writes to instruction stream	
jal	UJ	Jump & Link	R[rd] = PC+4; PC = PC + {imm,1b'0}	
jalr	I	Jump & Link Register	R[rd] = PC+4; PC = R[rs1]+imm	3)
lb	I	Load Byte	R[rd] = {56'bM[](7),M[R[rs1]+imm](7:0)}	4)
lbu	I	Load Byte Unsigned	R[rd] = {56'b0,M[R[rs1]+imm](7:0)}	
ld	I	Load Doubleword	R[rd] = M[R[rs1]+imm](63:0)	
lh	I	Load Halfword	R[rd] = {48'bM[](15),M[R[rs1]+imm](15:0)}	4)
lhu	I	Load Halfword Unsigned	R[rd] = {48'b0,M[R[rs1]+imm](15:0)}	
lui	U	Load Upper Immediate	R[rd] = {32b'imm<31>, imm, 12'b0}	
lw	I	Load Word	R[rd] = {32'bM[](31),M[R[rs1]+imm](31:0)}	4)
lwu	I	Load Word Unsigned	R[rd] = {32'b0,M[R[rs1]+imm](31:0)}	
or	R	OR	R[rd] = R[rs1] | R[rs2]	
ori	I	OR Immediate	R[rd] = R[rs1] | imm	
sb	S	Store Byte	M[R[rs1]+imm](7:0) = R[rs2](7:0)	
sd	S	Store Doubleword	M[R[rs1]+imm](63:0) = R[rs2](63:0)	
sh	S	Store Halfword	M[R[rs1]+imm](15:0) = R[rs2](15:0)	
sll,sllw	R	Shift Left (Word)	R[rd] = R[rs1] << R[rs2]	1)
slli,slliw	I	Shift Left Immediate (Word)	R[rd] = R[rs1] << imm	1)
slt	R	Set Less Than	R[rd] = (R[rs1] < R[rs2]) ? 1 : 0	
slti	I	Set Less Than Immediate	R[rd] = (R[rs1] < imm) ? 1 : 0	
sltiu	I	Set < Immediate Unsigned	R[rd] = (R[rs1] < imm) ? 1 : 0	2)
sltu	R	Set Less Than Unsigned	R[rd] = (R[rs1] < R[rs2]) ? 1 : 0	2)
sra,sraw	R	Shift Right Arithmetic (Word)	R[rd] = R[rs1] >> R[rs2]	1,5)
srai,sraiw	I	Shift Right Arith Imm (Word)	R[rd] = R[rs1] >> imm	1,5)
srl,srlw	R	Shift Right (Word)	R[rd] = R[rs1] >> R[rs2]	1)
srli,srliw	I	Shift Right Immediate (Word)	R[rd] = R[rs1] >> imm	1)
sub,subw	R	SUBtract (Word)	R[rd] = R[rs1] – R[rs2]	1)
sw	S	Store Word	M[R[rs1]+imm](31:0) = R[rs2](31:0)	
xor	R	XOR	R[rd] = R[rs1] ^ R[rs2]	
xori	I	XOR Immediate	R[rd] = R[rs1] ^ imm	

Notes: 1) The Word version only operates on the rightmost 32 bits of a 64-bit registers
2) Operation assumes unsigned integers (instead of 2's complement)
3) The least significant bit of the branch address in jalr is set to 0
4) (signed) Load instructions extend the sign bit of data to fill the 64-bit register
5) Replicates the sign bit to fill in the leftmost bits of the result during right shift
6) Multiply with one operand signed and one unsigned
7) The Single version does a single-precision operation using the rightmost 32 bits of a 64-bit F register
8) Classify writes a 10-bit mask to show which properties are true (e.g., –inf, -0, +0, +inf, denorm, ...)
9) Atomic memory operation; nothing else can interpose itself between the read and the write of the memory location
The immediate field is sign-extended in RISC-V

ARITHMETIC CORE INSTRUCTION SET ②

RV64M Multiply Extension

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
mul,mulw	R	MULtiply (Word)	R[rd] = (R[rs1] * R[rs2])(63:0)	1)
mulh	R	MULtiply upper Half	R[rd] = (R[rs1] * R[rs2])(127:64)	
mulhsu	R	MULtiply upper Half Sign/Uns	R[rd] = (R[rs1] * R[rs2])(127:64)	6)
mulhu	R	MULtiply upper Half Unsigned	R[rd] = (R[rs1] * R[rs2])(127:64)	2)
div,divw	R	DIVide (Word)	R[rd] = (R[rs1] / R[rs2])	1)
divu	R	DIVide Unsigned	R[rd] = (R[rs1] / R[rs2])	2)
rem,remw	R	REMainder (Word)	R[rd] = (R[rs1] % R[rs2])	1)
remu,remuw	R	REMainder Unsigned (Word)	R[rd] = (R[rs1] % R[rs2])	1,2)

RV64F and RV64D Floating-Point Extensions

fld,flw	I	Load (Word)	F[rd] = M[R[rs1]+imm]	1)
fsd,fsw	S	Store (Word)	M[R[rs1]+imm] = F[rd]	1)
fadd.s,fadd.d	R	ADD	F[rd] = F[rs1] + F[rs2]	7)
fsub.s,fsub.d	R	SUBtract	F[rd] = F[rs1] – F[rs2]	7)
fmul.s,fmul.d	R	MULtiply	F[rd] = F[rs1] * F[rs2]	7)
fdiv.s,fdiv.d	R	DIVide	F[rd] = F[rs1] / F[rs2]	7)
fsqrt.s,fsqrt.d	R	SQuare RooT	F[rd] = sqrt(F[rs1])	7)
fmadd.s,fmadd.d	R	Multiply-ADD	F[rd] = F[rs1] * F[rs2] + F[rs3]	7)
fmsub.s,fmsub.d	R	Multiply-SUBtract	F[rd] = F[rs1] * F[rs2] - F[rs3]	7)
fmnsub.s,fmnsub.d	R	Negative Multiply-SUBtract	F[rd] = –(F[rs1] * F[rs2] – F[rs3])	7)
fmnadd.s,fmnadd.d	R	Negative Multiply-ADD	F[rd] = –(F[rs1] * F[rs2] + F[rs3])	7)
fsgnj.s,fsgnj.d	R	SiGN source	F[rd] = { F[rs2]<63>,F[rs1]<62:0>}	7)
fsgnjn.s,fsgnjn.d	R	Negative SiGN source	F[rd] = { (–F[rs2]<63>), F[rs1]<62:0>}	7)
fsgnjx.s,fsgnjx.d	R	Xor SiGN source	F[rd] = {F[rs2]<63>^F[rs1]<63>, F[rs1]<62:0>}	7)
fmin.s,fmin.d	R	MINimum	F[rd] = (F[rs1] < F[rs2]) ? F[rs1] : F[rs2]	7)
fmax.s,fmax.d	R	MAXimum	F[rd] = (F[rs1] > F[rs2]) ? F[rs1] : F[rs2]	7)
feq.s,feq.d	R	Compare Float EQual	R[rd] = (F[rs1]== F[rs2]) ? 1 : 0	7)
flt.s,flt.d	R	Compare Float Less Than	R[rd] = (F[rs1]< F[rs2]) ? 1 : 0	7)
fle.s,fle.d	R	Compare Float Less than or =	R[rd] = (F[rs1]<= F[rs2]) ? 1 : 0	7)
fclass.s,fclass.d	R	Classify Type	R[rd] = class(F[rs1])	7,8)
fmv.s.x,fmv.d.x	R	Move from Integer	F[rd] = R[rs1]	7)
fmv.x.s,fmv.x.d	R	Move to Integer	R[rd] = F[rs1]	7)
fcvt.s.d	R	Convert from DP to SP	F[rd] = single(F[rs1])	
fcvt.d.s	R	Convert from SP to DP	F[rd] = double(F[rs1])	
fcvt.s.w,fcvt.d.w	R	Convert from 32b Integer	F[rd] = float(R[rs1](31:0))	7)
fcvt.s.l,fcvt.d.l	R	Convert from 64b Integer	F[rd] = float(R[rs1](63:0))	7)
fcvt.s.wu,fcvt.d.wu	R	Convert from 32b Int Unsigned	F[rd] = float(R[rs1](31:0))	2,7)
fcvt.s.lu,fcvt.d.lu	R	Convert from 64b Int Unsigned	F[rd] = float(R[rs1](63:0))	2,7)
fcvt.w.s,fcvt.w.d	R	Convert to 32b Integer	R[rd](31:0) = integer(F[rs1])	7)
fcvt.l.s,fcvt.l.d	R	Convert to 64b Integer	R[rd](63:0) = integer(F[rs1])	7)
fcvt.wu.s,fcvt.wu.d	R	Convert to 32b Int Unsigned	R[rd](31:0) = integer(F[rs1])	2,7)
fcvt.lu.s,fcvt.lu.d	R	Convert to 64b Int Unsigned	R[rd](63:0) = integer(F[rs1])	2,7)

RV64A Atomic Extension

amoadd.w,amoadd.d	R	ADD	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] + R[rs2]	9)
amoand.w,amoand.d	R	AND	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] & R[rs2]	9)
amomax.w,amomax.d	R	MAXimum	R[rd] = M[R[rs1]], if(R[rs2] > M[R[rs1]]) M[R[rs1]] = R[rs2]	9)
amomaxu.w,amomaxu.d	R	MAXimum Unsigned	R[rd] = M[R[rs1]], if(R[rs2] > M[R[rs1]]) M[R[rs1]] = R[rs2]	2,9)
amomin.w,amomin.d	R	MINimum	R[rd] = M[R[rs1]], if(R[rs2] < M[R[rs1]]) M[R[rs1]] = R[rs2]	9)
amominu.w,amominu.d	R	MINimum Unsigned	R[rd] = M[R[rs1]], if(R[rs2] < M[R[rs1]]) M[R[rs1]] = R[rs2]	2,9)
amoor.w,amoor.d	R	OR	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] | R[rs2]	9)
amoswap.w,amoswap.d	R	SWAP	R[rd] = M[R[rs1]], M[R[rs1]] = R[rs2]	9)
amoxor.w,amoxor.d	R	XOR	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] ^ R[rs2]	9)
lr.w,lr.d	R	Load Reserved	R[rd] = M[R[rs1]], reservation on M[R[rs1]]	
sc.w,sc.d	R	Store Conditional	if reserved, M[R[rs1]] = R[rs2], R[rd] = 0; else R[rd] = 1	

CORE INSTRUCTION FORMATS

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

Figure A-1.1 RISC-V Green Card

PSEUDO INSTRUCTIONS ③

MNEMONIC	NAME	DESCRIPTION	USES
beqz	Branch = zero	if(R[rs1]==0) PC=PC+{imm,1b'0}	beq
bnez	Branch ≠ zero	if(R[rs1]!=0) PC=PC+{imm,1b'0}	bne
fabs.s,fabs.d	Absolute Value	F[rd] = (F[rs1]< 0) ? –F[rs1] : F[rs1]	fsgnx
fmv.s,fmv.d	FP Move	F[rd] = F[rs1]	fsgnj
fneg.s,fneg.d	FP negate	F[rd] = –F[rs1]	fsgnjn
j	Jump	PC = {imm,1b'0}	jal
jr	Jump register	PC = R[rs1]	jalr
la	Load address	R[rd] = address	auipc
li	Load imm	R[rd] = imm	addi
mv	Move	R[rd] = R[rs1]	addi
neg	Negate	R[rd] = –R[rs1]	sub
nop	No operation	R[0] = R[0]	addi
not	Not	R[rd] = –R[rs1]	xori
ret	Return	PC = R[1]	jalr
seqz	Set = zero	R[rd] = (R[rs1]== 0) ? 1 : 0	sltiu
snez	Set ≠ zero	R[rd] = (R[rs1]!= 0) ? 1 : 0	sltu

OPCODES IN NUMERICAL ORDER BY OPCODE

MNEMONIC	FMT	OPCODE	FUNCT3	FUNCT7 OR IMM	HEXADECIMAL
lb	I	0000011	000		03/0
lh	I	0000011	001		03/1
lw	I	0000011	010		03/2
ld	I	0000011	011		03/3
lbu	I	0000011	100		03/4
lhu	I	0000011	101		03/5
lwu	I	0000011	110		03/6
fence	I	0001111	000		0F/0
fence.i	I	0001111	001		0F/1
addi	I	0010011	000		13/0
slli	I	0010011	001	0000000	13/1/00
slti	I	0010011	010		13/2
sltiu	I	0010011	011		13/3
xori	I	0010011	100		13/4
srli	I	0010011	101	0000000	13/5/00
srai	I	0010011	101	0100000	13/5/20
ori	I	0010011	110		13/6
andi	I	0010011	111		13/7
auipc	U	0010111			17
addiw	I	0011011	000		1B/0
slliw	I	0011011	001	0000000	1B/1/00
srliw	I	0011011	101	0000000	1B/5/00
sraiw	I	0011011	101	0100000	1B/5/20
sb	S	0100011	000		23/0
sh	S	0100011	001		23/1
sw	S	0100011	010		23/2
sd	S	0100011	011		23/3
add	R	0110011	000	0000000	33/0/00
sub	R	0110011	000	0100000	33/0/20
sll	R	0110011	001	0000000	33/1/00
slt	R	0110011	010	0000000	33/2/00
sltu	R	0110011	011	0000000	33/3/00
xor	R	0110011	100	0000000	33/4/00
srl	R	0110011	101	0000000	33/5/00
sra	R	0110011	101	0100000	33/5/20
or	R	0110011	110	0000000	33/6/00
and	R	0110011	111	0000000	33/7/00
lui	U	0110111			37
addw	R	0111011	000	0000000	3B/0/00
subw	R	0111011	000	0100000	3B/0/20
sllw	R	0111011	001	0000000	3B/1/00
srlw	R	0111011	101	0000000	3B/5/00
sraw	R	0111011	101	0100000	3B/5/20
beq	SB	1100011	000		63/0
bne	SB	1100011	001		63/1
blt	SB	1100011	100		63/4
bge	SB	1100011	101		63/5
bltu	SB	1100011	110		63/6
bgeu	SB	1100011	111		63/7
jalr	I	1100111	000		67/0
jal	UJ	1101111			6F
ecall	I	1110011	000	000000000000	73/0/000
ebreak	I	1110011	000	000000000001	73/0/001
CSRRW	I	1110011	001		73/1
CSRRS	I	1110011	010		73/2
CSRRC	I	1110011	011		73/3
CSRRWI	I	1110011	101		73/5
CSRRSI	I	1110011	110		73/6
CSRRCI	I	1110011	111		73/7

REGISTER NAME, USE, CALLING CONVENTION ④

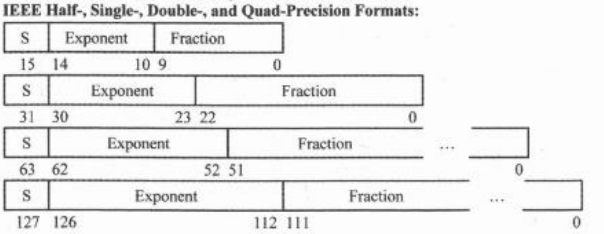
REGISTER	NAME	USE	SAVER
x0	zero	The constant value 0	N.A.
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	–
x4	tp	Thread pointer	–
x5–x7	t0–t2	Temporaries	Caller
x8	s0/fp	Saved register/Frame pointer	Callee
x9	s1	Saved register	Callee
x10–x11	a0–a1	Function arguments/Return values	Caller
x12–x17	a2–a7	Function arguments	Caller
x18–x27	s2–s11	Saved registers	Callee
x28–x31	t3–t6	Temporaries	Caller
f0–f7	ft0–ft7	FP Temporaries	Caller
f8–f9	fs0–fs1	FP Saved registers	Callee
f10–f11	fa0–fa1	FP Function arguments/Return values	Caller
f12–f17	fa2–fa7	FP Function arguments	Caller
f18–f27	fs2–fs11	FP Saved registers	Callee
f28–f31	ft8–ft11	R[rd] = R[rs1] + R[rs2]	Caller

IEEE 754 FLOATING-POINT STANDARD

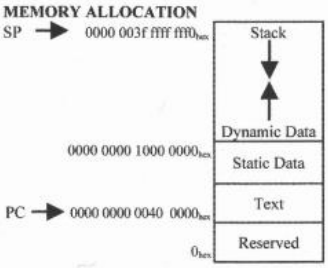
$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$

where Half-Precision Bias = 15, Single-Precision Bias = 127, Double-Precision Bias = 1023, Quad-Precision Bias = 16383

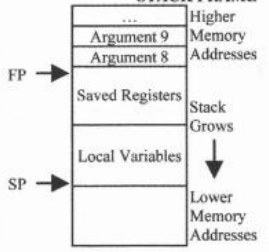
IEEE Half-, Single-, Double-, and Quad-Precision Formats:



MEMORY ALLOCATION



STACK FRAME



SIZE PREFIXES AND SYMBOLS

SIZE	PREFIX	SYMBOL	SIZE	PREFIX	SYMBOL
10^3	Kilo-	K	2^{10}	Kibi-	Ki
10^6	Mega-	M	2^{20}	Mebi-	Mi
10^9	Giga-	G	2^{30}	Gibi-	Gi
10^{12}	Tera-	T	2^{40}	Tebi-	Ti
10^{15}	Peta-	P	2^{50}	Pebi-	Pi
10^{18}	Exa-	E	2^{60}	Exbi-	Ei
10^{21}	Zetta-	Z	2^{70}	Zebi-	Zi
10^{24}	Yotta-	Y	2^{80}	Yobi-	Yi
10^{-3}	milli-	m	10^{-15}	femto-	f
10^{-6}	micro-	μ	10^{-18}	atto-	a
10^{-9}	nano-	n	10^{-21}	zepto-	z
10^{-12}	pico-	p	10^{-24}	yocto-	y

Figure A-1.2 RISC-V Green Card

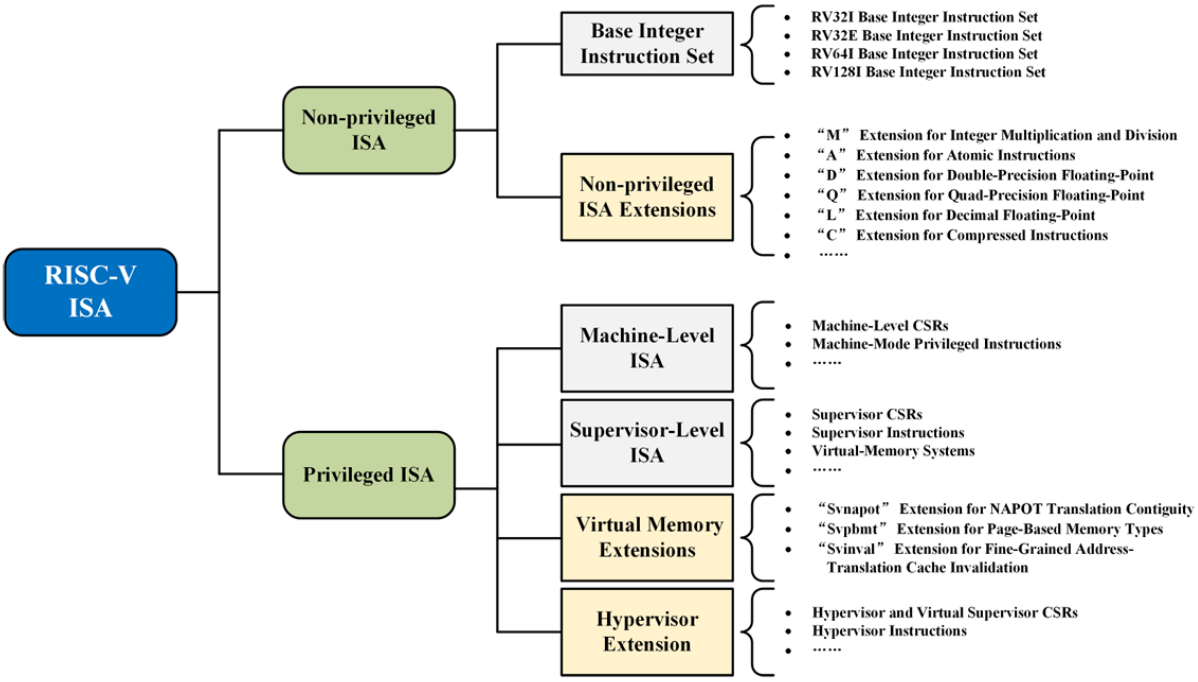


Figure A-3 RISC-V ISA

APPENDIX B

Host code and Guest code

```
use clap::Parser;
use sp1_sdk::{
    blocking::{EnvProver, ProveRequest, Prover, ProverClient},
    include_elf, Elf, ProvingKey, SP1Stdin,
};
use std::time::Instant;


const SHA256_ELF: Elf = include_elf!("sha256");
const KECCAK256_ELF: Elf = include_elf!("keccak256");
const ED25519_ELF: Elf = include_elf!("ed25519");
const TOKEN_TRANSACTION_ELF: Elf = include_elf!("token_transaction");


#[derive(Clone, Copy, Debug, clap::ValueEnum)]
enum GuestProgram {
    Sha256,
    Keccak256,
    Ed25519,
    TokenTransaction,
}


impl std::fmt::Display for GuestProgram {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        match self {
            GuestProgram::Sha256 => write!(f, "sha256"),
            GuestProgram::Keccak256 => write!(f, "keccak256"),
            GuestProgram::Ed25519 => write!(f, "ed25519"),
            GuestProgram::TokenTransaction => write!(f, "token_transaction"),
        }
    }
}
```

```
#[derive(Clone, Copy, Debug, clap::ValueEnum)]
enum Mode {
    Baseline,
    Extension,
}

impl std::fmt::Display for Mode {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        match self {
            Mode::Baseline => write!(f, "baseline"),
            Mode::Extension => write!(f, "extension"),
        }
    }
}

#[derive(Parser, Debug)]
#[command(author, version, about, long_about = None)]
struct Args {
    /// Guest program to run.
    #[arg(long, value_enum)]
    program: GuestProgram,

    /// Run the software baseline or the custom syscall ISA extension path.
    #[arg(long, value_enum)]
    mode: Mode,

    /// Number of repeated operations inside each guest program.
    #[arg(long, default_value = "1")]
    iterations: u32,

    /// Also generate and verify a STARK proof after execution.
    /// WARNING: this is significantly slower than execution only.
```

```
    #[arg(long, default_value = "false")]
    prove: bool,
}

// -- Entry point --

fn main() {
    sp1_sdk::utils::setup_logger();
    dotenv::dotenv().ok();

    let args = Args::parse();
    let client: EnvProver = ProverClient::from_env();
    let use_extension = matches!(args.mode, Mode::Extension);
    let elf = program_elf(args.program);

    println!();
    println!("[ EXECUTION ]");
    println!("  Running...");

    let start = Instant::now();
    let (_public_values, report) = client
        .execute(elf.clone(), stdin(use_extension, args.iterations))
        .run()
        .expect("execution failed");
    let exec_ms = start.elapsed().as_millis();

    let total = report.total_instruction_count();
    println!("  Total instructions : {}", fmt_num(total));
    println!("  Execution time     : {} ms", exec_ms);

    let syscalls: Vec<_> = report
        .syscall_counts
        .iter()
```

```
        .filter(|(_, &c)| c != 0)
        .collect();

    if !syscalls.is_empty() {
        println!();
        println!("  Syscall Breakdown:");
        println!("  {:<35} {:>10}", "Syscall", "Count");
        for (name, count) in &syscalls {
            println!("  {:<35} {:>10}", format!("{name:?}"), count);
        }
    }

    let mut opcodes: Vec<_> = report
        .opcode_counts
        .iter()
        .filter(|(_, &c)| c != 0)
        .collect();
    opcodes.sort_by(|a, b| b.1.cmp(a.1));

    if !opcodes.is_empty() {
        println!();
        println!("  Top Opcodes (by count):");
        println!("  {:<20} {:>12}", "Opcode", "Count");
        for (name, count) in opcodes.iter().take(5) {
            println!("  {:<20} {:>12}", format!("{name:?}"), fmt_num(**count));
        }
        if opcodes.len() > 5 {
            println!("  ... ({} more opcodes omitted)", opcodes.len() - 5);
        }
    }

    if args.prove {
        println!();
```

```
        println!("[ PROOF GENERATION ]");
        let pk = client.setup(elf).expect("key setup failed");

        let prove_start = Instant::now();
        let proof = client
            .prove(&pk, stdin(use_extension, args.iterations))
            .core()
            .run()
            .expect("proof generation failed");
        let prove_ms = prove_start.elapsed().as_millis();

        let vk = pk.verifying_key();
        let verify_start = Instant::now();
        client.verify(&proof, vk, None).expect("proof verification failed");
        let verify_ms = verify_start.elapsed().as_millis();

        println!("  Proving time      : {} ms", prove_ms);
        println!("  Verification time : {} ms", verify_ms);
        println!("  Verified          : YES");
    }
}

// -- Helpers --

fn stdin(use_extension: bool, iterations: u32) -> SP1Stdin {
    let mut stdin = SP1Stdin::new();
    stdin.write(&use_extension);
    stdin.write(&iterations);
    stdin
}

fn fmt_num(n: u64) -> String {
    let s = n.to_string();
```

```
    let mut result = String::new();
    for (i, ch) in s.chars().rev().enumerate() {
        if i > 0 && i % 3 == 0 { result.push(','); }
        result.push(ch);
    }
    result.chars().rev().collect()
}


fn program_elf(program: GuestProgram) -> Elf {
    match program {
        GuestProgram::Sha256 => SHA256_ELF,
        GuestProgram::Keccak256 => KECCAK256_ELF,
        GuestProgram::Ed25519 => ED25519_ELF,
        GuestProgram::TokenTransaction => TOKEN_TRANSACTION_ELF,
    }
}
```

Figure B-1 Host code (rust code)

```
use curve25519_dalek::edwards::CompressedEdwardsY;

use sha2::{Digest as Sha2Digest, Sha256};

use sha3::Keccak256;

use sp1_zkvm::syscalls::{

    syscall_sha256_extend, syscall_sha256_fold, syscall_sponge_f1600,
syscall_ted255_add,

};



const SHA256_INITIAL_STATE: [u32; 8] = [

    0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a, 0x510e527f, 0x9b05688c,
0x1f83d9ab, 0x5be0cd19,

];



const A_AFFINE: [u8; 64] = [

    195, 166, 157, 207, 218, 220, 175, 197, 111, 177, 123, 23, 73, 72, 114, 103,
28, 246, 66, 207,

    66, 146, 187, 234, 136, 238, 133, 145, 47, 196, 216, 199, 79, 31, 224, 30, 179,
122, 51, 84,

    116, 12, 4, 189, 198, 198, 190, 22, 71, 201, 143, 249, 92, 56, 147, 133, 92,
187, 130, 33, 152,

    19, 171, 73,

];



const B_AFFINE: [u8; 64] = [

    197, 189, 200, 77, 201, 212, 57, 105, 191, 133, 123, 170, 167, 50, 114, 38, 37,
102, 188, 29,

    215, 227, 157, 142, 252, 31, 129, 67, 24, 255, 114, 136, 115, 94, 94, 55, 43,
200, 117, 224,

    139, 251, 238, 45, 80, 154, 70, 213, 219, 78, 201, 108, 73, 203, 72, 45, 167,
131, 199, 47, 82,

    134, 53, 62,

];



pub fn run_sha256(use_extension: bool, iterations: u32) -> [u8; 32] {

    let mut acc = [0u8; 32];

    for round in 0..iterations {
```

```
        let msg = transfer_message(round);
        let digest = if use_extension {
            sha256_extension(&msg)
        } else {
            sha256_baseline(&msg)
        };
        xor_into(&mut acc, &digest);
    }
    acc
}

pub fn run_keccak256(use_extension: bool, iterations: u32) -> [u8; 32] {
    let mut acc = [0u8; 32];
    for round in 0..iterations {
        let msg = transfer_message(round);
        let digest = if use_extension {
            keccak256_extension(&msg)
        } else {
            keccak256_baseline(&msg)
        };
        xor_into(&mut acc, &digest);
    }
    acc
}

pub fn run_ed25519(use_extension: bool, iterations: u32) -> [u8; 32] {
    let mut acc = [0u8; 32];
    for round in 0..iterations {
        let mut point = if use_extension {
            ted255_add_extension()
        } else {
            ed25519_add_baseline()
        };
```

```
        point[0] ^= round as u8;
        xor_into(&mut acc, &point);
    }
    acc
}

pub fn run_token_transaction(use_extension: bool, iterations: u32) -> [u8; 32] {
    let mut receipt = [0u8; 32];
    for round in 0..iterations {
        let tx = transfer_message(round);
        let sha = if use_extension {
            sha256_extension(&tx)
        } else {
            sha256_baseline(&tx)
        };
        let keccak = if use_extension {
            keccak256_extension(&sha)
        } else {
            keccak256_baseline(&sha)
        };
        let ed_result = if use_extension {
            ted255_add_extension()
        } else {
            ed25519_add_baseline()
        };

        let mut verification_transcript = [0u8; 96];
        verification_transcript[..32].copy_from_slice(&sha);
        verification_transcript[32..64].copy_from_slice(&keccak);
        verification_transcript[64..].copy_from_slice(&ed_result);

        let digest = if use_extension {
            keccak256_extension(&verification_transcript)
```

```
        } else {
            keccak256_baseline(&verification_transcript)
        };
        xor_into(&mut receipt, &digest);
    }
    receipt
}

fn transfer_message(round: u32) -> [u8; 128] {
    let mut msg = [0u8; 128];
    let prefix = b"RISC-V ISA extension token transfer:";
    msg[..prefix.len()].copy_from_slice(prefix);
    msg[40..72].copy_from_slice(b"alice-token-account-000000000001");
    msg[72..104].copy_from_slice(b"bob-token-account-00000000000003");
    msg[104..112].copy_from_slice(&1_000_000u64.wrapping_add(round as
u64).to_le_bytes());
    msg[112..120].copy_from_slice(&42u64.to_le_bytes());
    msg[120..124].copy_from_slice(&round.to_le_bytes());
    msg[124..128].copy_from_slice(&0x54584e31u32.to_le_bytes());
    msg
}

fn sha256_baseline(input: &[u8]) -> [u8; 32] {
    let mut hasher = Sha256::new();
    hasher.update(input);
    hasher.finalize().into()
}

fn sha256_extension(input: &[u8]) -> [u8; 32] {
    let mut padded = [0u8; 256];
    let len = input.len();
    padded[..len].copy_from_slice(input);
    padded[len] = 0x80;
    let padded_len = ((len + 9 + 63) / 64) * 64;
```

```
    let bit_len = (len as u64) * 8;
    padded[padded_len - 8..padded_len].copy_from_slice(&bit_len.to_be_bytes());

    let mut state = [0u64; 8];
    for (dst, src) in state.iter_mut().zip(SHA256_INITIAL_STATE) {
        *dst = src as u64;
    }

    let mut offset = 0;
    while offset < padded_len {
        let mut w = [0u64; 64];
        for i in 0..16 {
            let j = offset + i * 4;
            w[i] =
                u32::from_be_bytes([padded[j], padded[j + 1], padded[j + 2],
padded[j + 3]]) as u64;
        }
        syscall_sha256_extend(&mut w);
        syscall_sha256_fold(&mut w, &mut state);
        offset += 64;
    }

    let mut out = [0u8; 32];
    for (i, word) in state.iter().enumerate() {
        out[i * 4..i * 4 + 4].copy_from_slice(&(*word as u32).to_be_bytes());
    }
    out
}

fn keccak256_baseline(input: &[u8]) -> [u8; 32] {
    let mut hasher = Keccak256::new();
    hasher.update(input);
    hasher.finalize().into()
```

```
}


fn keccak256_extension(input: &[u8]) -> [u8; 32] {
    const RATE: usize = 136;
    let mut state = [0u64; 25];
    let mut offset = 0;

    while offset + RATE <= input.len() {
        absorb_block(&mut state, &input[offset..offset + RATE]);
        syscall_sponge_f1600(&mut state);
        offset += RATE;
    }

    let mut block = [0u8; RATE];
    let remaining = input.len() - offset;
    block[..remaining].copy_from_slice(&input[offset..]);
    block[remaining] ^= 0x01;
    block[RATE - 1] ^= 0x80;
    absorb_block(&mut state, &block);
    syscall_sponge_f1600(&mut state);

    let mut out = [0u8; 32];
    for i in 0..4 {
        out[i * 8..i * 8 + 8].copy_from_slice(&state[i].to_le_bytes());
    }
    out
}


fn absorb_block(state: &mut [u64; 25], block: &[u8]) {
    for (lane, chunk) in block.chunks_exact(8).enumerate() {
        let mut bytes = [0u8; 8];
        bytes.copy_from_slice(chunk);
        state[lane] ^= u64::from_le_bytes(bytes);
```

```
    }
}

fn ed25519_add_baseline() -> [u8; 32] {
    let a = CompressedEdwardsY(compress_affine(&A_AFFINE))
        .decompress()
        .unwrap();
    let b = CompressedEdwardsY(compress_affine(&B_AFFINE))
        .decompress()
        .unwrap();
    (a + b).compress().to_bytes()
}

fn ted255_add_extension() -> [u8; 32] {
    let mut p = bytes64_to_words(&A_AFFINE);
    let q = bytes64_to_words(&B_AFFINE);
    syscall_ted255_add(&mut p, &q);
    compress_affine(&words_to_bytes64(&p))
}

fn compress_affine(point: &[u8; 64]) -> [u8; 32] {
    let mut compressed = [0u8; 32];
    compressed.copy_from_slice(&point[32..64]);
    compressed[31] |= (point[0] & 1) << 7;
    compressed
}

fn bytes64_to_words(bytes: &[u8; 64]) -> [u64; 8] {
    let mut words = [0u64; 8];
    for (i, word) in words.iter_mut().enumerate() {
        let start = i * 8;
        *word = u64::from_le_bytes(bytes[start..start + 8].try_into().unwrap());
    }
```

```
    words
}


fn words_to_bytes64(words: &[u64; 8]) -> [u8; 64] {
    let mut bytes = [0u8; 64];
    for (i, word) in words.iter().enumerate() {
        bytes[i * 8..i * 8 + 8].copy_from_slice(&word.to_le_bytes());
    }
    bytes
}


fn xor_into(acc: &mut [u8; 32], rhs: &[u8; 32]) {
    for (a, b) in acc.iter_mut().zip(rhs) {
        *a ^= *b;
    }
}
```

Figure B-2 Guest code (rust code)

Appendix C

Project Timeline

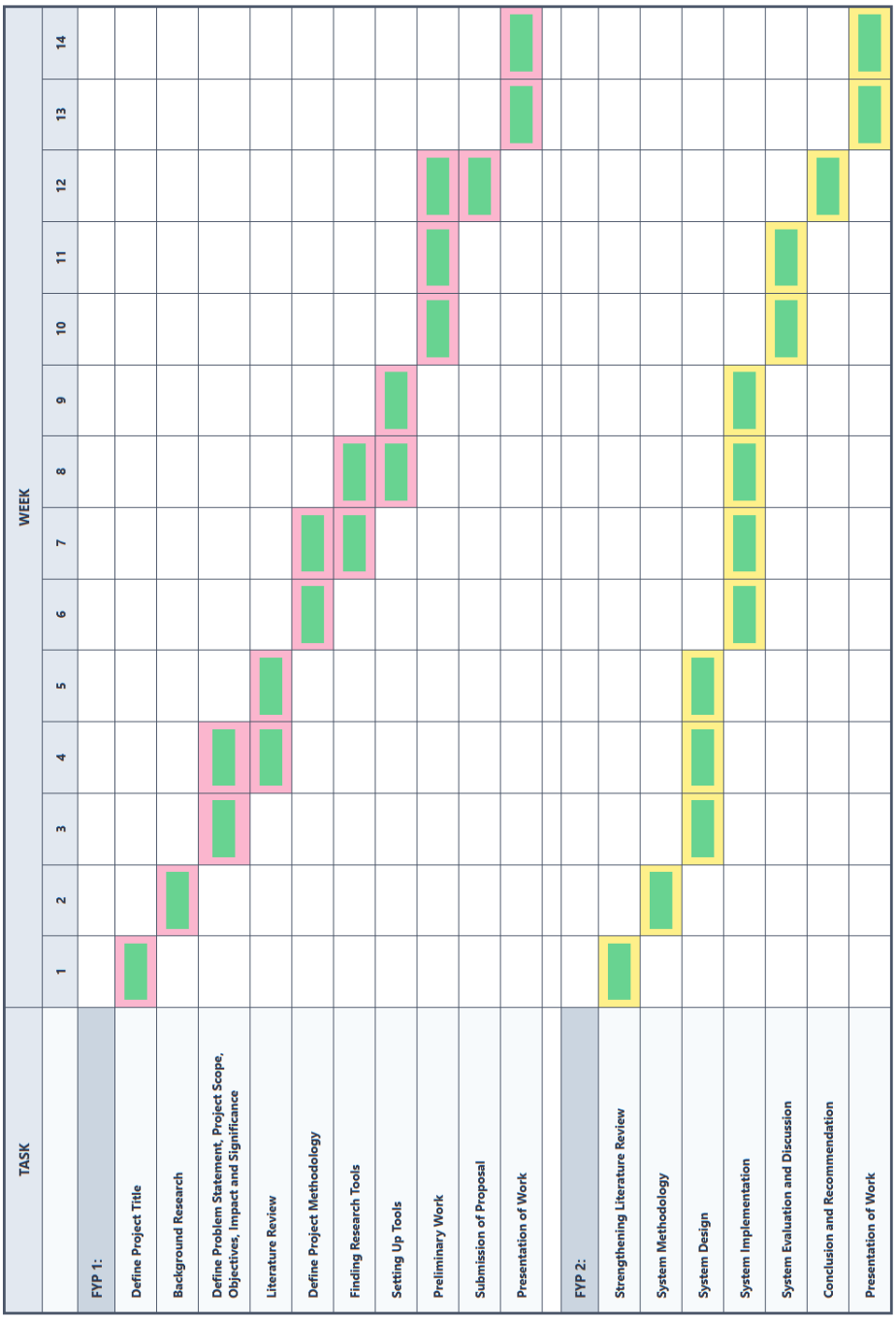


Figure C-1 Project Timeline

POSTER

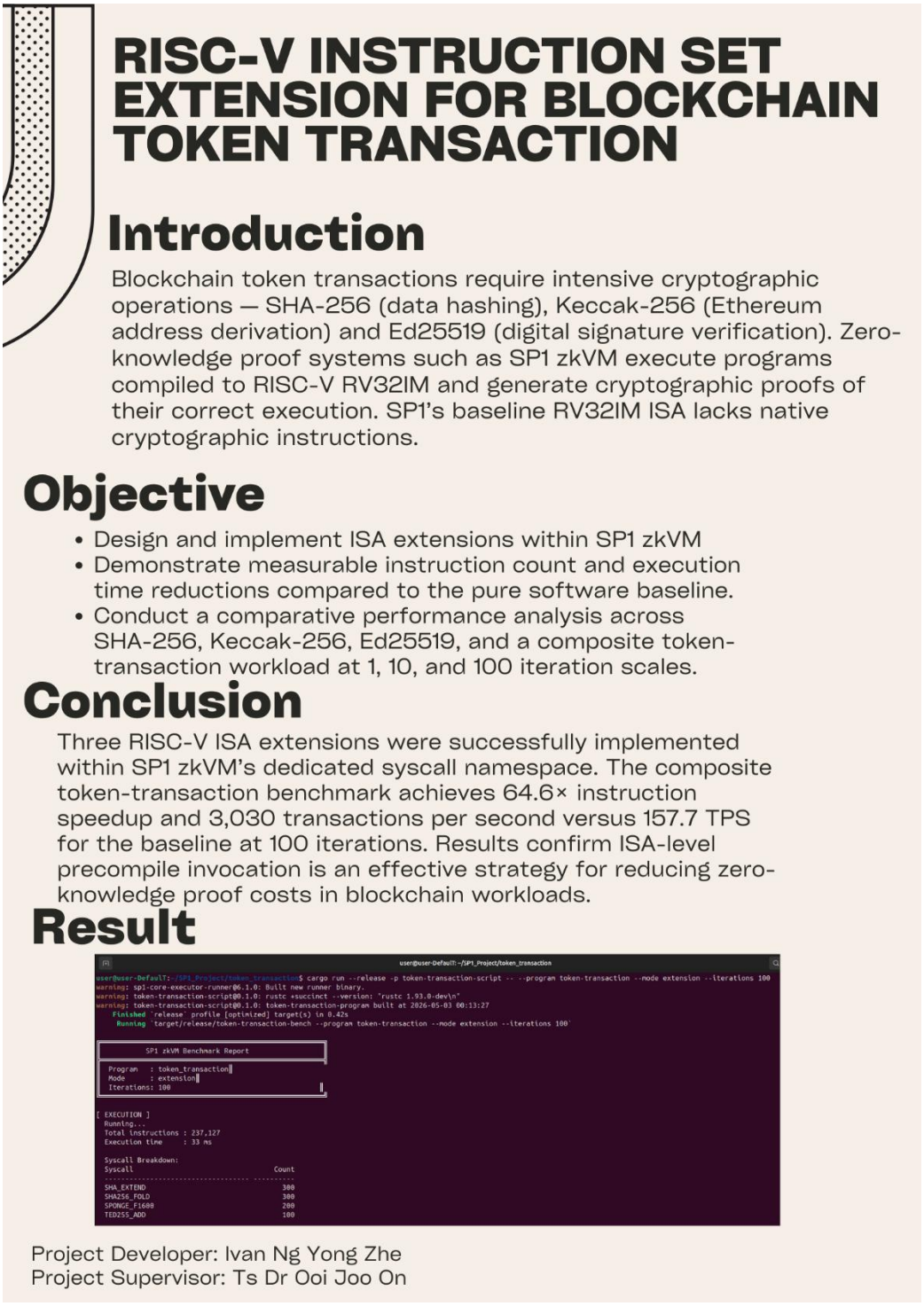


Figure D POSTER