ZEE200
Zero Knowledge for Everything and Everyone @ 200 KHz

Sunghyeon Jo* Vladimir Kolesnikov† Yibin Yang‡

August 20, 2026

Abstract

Zero-knowledge execution of high-level programs proceeds by repeatedly evaluating CPU steps. Each such step privately selects and evaluates an instruction (possibly involving memory access) from a rich instruction set. Building on this paradigm, ZEE (Heath et al., S&P'21) realized a full toolchain supporting arbitrary ANSI C programs, demonstrating this capability by proving SIR- and CVE-reported bugs in off-the-shelf Linux programs sed and gzip.

We revamp the state of the art by building a new ZK system ZEE200, which is about 20-40× faster than ZEE. ZEE200 is built on a novel and convenient cryptographic framework for efficiently proving general statements represented as real-world programs. Our framework integrates several crucial recent advances, such as Tight ZK CPU (Yang et al., CCS'24) and fast ZK RAM (Yang and Heath, USENIX Security'24). We develop better encodings for $\mathbb{Z}_{2^{32}}$ arithmetic, and numerous low-level optimizations.

Compared to ZEE's ≈10 KHz CPU speed on a limited ISA, ZEE200 runs at ≈200 KHz (still on a commodity laptop and a LAN!), while supporting a much richer ISA. For example, we rerun a ZEE's benchmark, proving a SIR-reported vulnerability in off-the-shelf Linux utility sed. On a 2021 ThinkPad X1 Carbon Gen 9 under a simulated 1 Gbps LAN (single-threaded), ZEE200 completed the proof in 1.5 seconds, compared to ZEE's 30.1 seconds, a 20× improvement.

*Georgia Institute of Technology, sjo65@gatech.edu.
†Georgia Institute of Technology, kolesnikov@gatech.edu.
‡NTT Research and University of Toronto, yibiny@ece.utoronto.ca.

Contents

1 Introduction **1**
- 1.1 Our Contribution . . . 1
- 1.2 Related Work . . . 2

2 Preliminaries **3**
- 2.1 Security Model . . . 3
- 2.2 ZEE System and ZEE Assembly [HYDK21] . . . 4
- 2.3 Tight ZK CPU [YHH+24]: A Brief Recap . . . 4

3 Technical Overview **7**
- 3.1 Generalized Tight ZK (GTZK) CPU . . . 7
- 3.2 ZEE200: Executing ANSI C over GTZK CPU . . . 9
- 3.3 Security of ZEE200 . . . 11

4 Generalized Tight ZK CPU Functionality **11**

5 ZEE200 **15**
- 5.1 Extended ZEE Assembly IS . . . 15
- 5.2 Public Setup of $\mathcal{F}_{\text{G-ZKCPU}}$ . . . 16
  - 5.2.1 Global Execution Parameters . . . 16
  - 5.2.2 Instruction Specification . . . 18
  - 5.2.3 Initial and Final Register States . . . 19
  - 5.2.4 Universal Unit Profile . . . 19
  - 5.2.5 Representative Assembly Instruction Gadgets . . . 19
- 5.3 Selected Optimizations . . . 22

6 Evaluation **23**
- 6.1 ZEE200 Overall Performance . . . 24
- 6.2 ZEE200 with Varying ZK RAM Sizes . . . 25
- 6.3 Comparison with zkVM . . . 25
- 6.4 ZEE200 with Different Universal Profiles . . . 26

A Deferred Preliminaries **34**
- A.1 Ideal $\mathcal{F}_{\text{CPZK}}$ . . . 34
- A.2 VOLE-Based ZK . . . 34

B Generalized Tight ZK CPU Protocol **36**
- B.1 Additional Notation . . . 37
- B.2 The $\mathcal{F}_{\text{CPZK-UROM-SET-KVS}}$-Hybrid Model . . . 38
- B.3 Generalized Topology Matrices . . . 39
- B.4 Formal $\Pi_{\text{G-ZKCPU}}$ . . . 45

C Assembly Instruction Gadget Constructions **52**
C.1 Auxiliary Gadgets . . . 53
C.2 Gadgets without KVS Access . . . 53
C.3 Gadgets with KVS Access . . . 53

D Gadget Inspection **59**
D.1 Field Encoding and No-Wraparound Facts . . . 59
D.2 Gadgets without KVS Access . . . 60
D.3 Gadgets with KVS Access . . . 65
D.4 Composition of Gadget Inspections . . . 72

1 Introduction

Zero-knowledge proofs (ZKPs) [GMR85] allow a prover $\mathcal{P}$ to convince a verifier $\mathcal{V}$ of the validity of a statement without revealing any information beyond that fact. ZKPs enable many applications, including private blockchains [BCG+14], privacy-preserving program analysis [FDNZ21, LAH+22], private bug bounty systems [HYDK21, YHKD22], secure machine learning [LXZ21, WYX+21], and many more. The last decade has seen rapid advances in the design, implementation, and practical deployment of concretely efficient ZKP systems.

In principle, ZKP supports arbitrary computations, but translating high-level programs (written in languages such as `C`) into efficient ZK proofs remains challenging. A physical machine provides a rich instruction set, native memory, and branching semantics that developers can rely on without understanding how each component works internally. In contrast, representing the same program inside a ZK system typically requires manually translating its control flow, memory accesses, and computations into an algebraic form suitable for ZK. This gap between expressive, familiar computation on real machines and the rigid algebraic structure required by ZK systems makes it difficult to build general-purpose applications directly.

ZK CPU Emulation. A promising direction toward bridging this gap is to emulate, inside zero knowledge, the same computational machinery that real programs rely on: a CPU with a rich instruction set, native arithmetic, branching, and a random-access memory. Rather than forcing developers to rewrite their programs in, e.g., circuit form, one can instead execute them on a *zero-knowledge CPU* that mirrors the semantics of a physical machine.

This natural paradigm was first explored in ZK by [BCG+13]. This should be seen as mostly a proof of concept, as their system ran at about 1 Hz. A reasonable practical realization is the ZEE ZK proof system [HYDK21], running at $\approx$ 10 KHz on a very basic instruction set (IS). ZEE is a full compiler toolchain capable of executing arbitrary `ANSI C` programs in ZK, with emulated memory size being a bottleneck, due to the computation and space overhead required for supporting large memory sizes. By modeling computation as repeated private CPU steps over a general-purpose IS, ZEE demonstrated that expressive applications can be built in ZK without requiring developers to reason about low-level cryptographic details. The ZEE system was expressive enough and fast enough to support execution of off-the-shelf Linux utilities such as `sed` and `gzip` inside ZK.

Despite its power, ZEE incurs two fundamental performance bottlenecks. First, for privacy, its each CPU step pays for all instructions in the instruction set, regardless of which one is actually executed. Second, its ZK RAM construction introduces logarithmic overhead for every memory access. These costs dominate ZEE's concrete performance and ultimately limit its execution rate to $\approx$ 10 KHz or less on realistic workloads.

Recent years have seen major algorithmic advances in CPU-emulation ZK. Most relevant to us are concretely efficient constant-overhead ZK RAM [YH24], and Tight ZK CPU [YHH+24], an elegant approach to CPU emulation ZK that allows cost proportional to the program execution path. In ZEE`200`, we integrate these advances and introduce a number of additional optimizations to build a toolchain supporting ZK of `ANSI C` programs that runs at 200 KHz!

1.1 Our Contribution

We propose a new fast ZKP toolchain ZEE`200` that compiles and executes between $\mathcal{P}$ and $\mathcal{V}$ general `ANSI C` programs describing proof statements. On a commodity hardware (e.g., a 2021 ThinkPad X1 Carbon Gen 9, over 1 Gbps LAN, single-threaded), it emulates a CPU with a rich instruction set at $\approx$ 200 KHz, improving state-of-the-art ZEE [HYDK21] by $\approx$ 20-40$\times$.

Beyond the end-to-end system, our contribution comprises intertwined architectural, crypto- and systems-technical advances:

1. **A modular and extensible ZK CPU architecture.** A central design feature of our work is generalized tight ZK (GTZK) CPU framework. Tight ZK CPU technique [YHH+24] is crucial for CPU-emulation ZK: viewing CPU instructions as arithmetic circuits, it enables ZKP cost proportional to the execution path. We generalize tight ZK CPU by allowing the constraint systems representing the CPU instructions to seamlessly include arbitrary constraints, most critically, referencing RAM. We formalize GTZK CPU functionality in the UC framework. Further, we formalize a general and convenient framework for developing and proving correctness of CPU instruction implementations.
   We exploit the GTZK's efficient support of variable-length instructions. We achieve significant savings by executing large groups of assembly instructions as a single GTZK instruction.
2. **Numerous lower-level crypto optimizations**, including much richer IS than ZEE (enabled by our GTZK framework), fast native field arithmetic and optimized handling of $\mathbb{Z}_{2^{32}}$, etc.
3. **Efficient integration of state-of-the-art primitives** (e.g., ZK RAM, VOLEs) is a significant system-technical undertaking.
4. **ZEE legacy.** ZEE200 (again, with significant technical effort expended) inherits ZEE benefits, such as extensive compiler and library support (ANSI C and libc), high usability by adhering to standard C (programmer solely needs to introduce the special QED instruction in the code), and real validation by supporting off-the-shelf Linux programs.
5. **Scale.** ZEE200 runs far more programs than ZEE due to much better computation, communication, and crucially, RAM scaling.
6. **Artifacts.** We provide an open-source artifact repository at https://github.com/ainta/ZEE200.

1.2 Related Work

ZEE200 is a fast general ANSI C compilation toolchain, directly improving over ZEE [HYDK21] by 20-40× thanks to our new (GTZK) CPU architecture, careful integration of improved ZK RAM [YH24], and numerous low-level optimizations. ZEE is previous state of the art, running at about 5-10 KHz on a limited IS. We review ZEE in Section 2.2, and provide extensive benchmark comparisons in Section 6.

In the remainder of this section, we briefly review non-interactive proof systems, highlight their differing goals and much slower performance, but do not compare in detail with ZEE200. We also briefly discuss ZEE200's central underlying primitives and building blocks: ZK disjunctions, ZK RAM, and VOLE-based ZK.

zkVMs (SNARKs for RAM Programs). Motivated by blockchain applications, recent years have seen rapid progress on *zkVM*s, particularly in industry. Examples include RISC0 [BG+23], SP1 [Suc24] and Jolt [AST24]. Despite "zk" in their name, zkVMs are typically designed as *succinct non-interactive arguments* (SNARKs) for RAM programs (studied in, e.g., [KS22, ZGK+18, KST22, KS24, CGG+24, DXNT24, LZZ+25]), which *do not* require zero knowledge. Indeed, existing zkVMs reveal partial information about the program's execution path (beyond length) to $\mathcal{V}$.

ZEE200 (and ZEE) achieve ZK, targeting different applications.

zkVMs can recover ZK using the elegant *zk wrapper* technique. ZK wrappers use non-black-box cryptography and incur substantial overhead (see, e.g., discussions in [GHAK24, LXZ+24]). Our experiments show ZEE200 is 2-3 orders of magnitude faster than zk-wrapped and unwrapped SP1 and 4-10× faster than unwrapped Jolt (Section 6.3).

ZK Disjunctions. ZK disjunctions occur in every CPU step and are particularly relevant to us. They

were considered in [CDS94], and later revisited by [HK20b], who introduced *stacking*, a technique reducing communication (but not computation) to that of the largest branch. Follow-up work [BMRS21, GGHAK22, GHAK24, GHAKS23, HHK+24, YHH+23, YHH+24, Yan25] explores stacking-style ZKPs across ZK techniques. The most relevant prior work, tight ZK CPU [YHH+24] (cf. Section 2.3) additionally hides the boundaries between CPU step, achieving communication and computation proportional to executed program path even if branches (CPU instructions) are of varying sizes. We generalize [YHH+24] by allowing much more powerful gates (e.g., ZK RAM lookup) and by designing a convenient framework. Our GTZK CPU is at the core of ZEE200.

ZK RAM. ZK RAM is fundamental for CPU-emulation ZK. Early constructions [BCG+13, BCTV14, WSR+15, HYDK21, HK20a] relied on oblivious routing or permutation networks, incurring polylogarithmic overhead per access. Recent work achieved constant or near-constant overhead by batch verifying RAM accesses [FKL+21, DOTV22]. [YH24]'s ZK RAM further improved efficiency; it is used in our GTZK framework as a black box.

VOLE-based ZK. A rich line of VOLE-based ZKP protocols [BBMH+21, BBMHS22, BCC+25, BMRS21, DEGL+23, DIO21, DILO22, HY24, LHLA25, LXY24a, LXY24b, WYKW21, YSWW21] builds on efficient VOLEs introduced by [BCGI18]. See [BDSW23] for a survey. VOLE-based ZK constructions are simple, highly efficient, and (usually) information-theoretic in the VOLE-hybrid model. Its linear scalability makes it especially well-suited for applications involving large statements. We use VOLE-based ZK (in particular, [DIO21, YSWW21]) to instantiate the commit-and-prove ZK functionality underlying our protocol.

2 Preliminaries

Notation. The prover is $\mathcal{P}$. The verifier is $\mathcal{V}$. $x := y$ denotes that x is *defined* as y. We denote vectors by bold lower-case letters (e.g., $\boldsymbol{a}$), where a_i (or $a[i]$) denotes the i-th component of $\boldsymbol{a}$ (starting from 1). $|\boldsymbol{a}|$ denotes the length of $\boldsymbol{a}$. We denote $\{1, \ldots, n\}$ by $[n]$, $\{a, \ldots, b-1\}$ by $[a, b)$. We denote a finite field of size p by $\mathbb{F}_p$ where $p \geq 2$ is a prime or a power of a prime. We use $\mathbb{F}$ to denote a sufficiently large finite field whose size is implicit from the context. We denote the set of natural numbers by $\mathbb{N}$, the set of integers by $\mathbb{Z}$, the set of positive integers by $\mathbb{Z}^+$, and the ring of integers modulo n by $\mathbb{Z}_n$. We refer to an element of $\mathbb{Z}_{2^{32}}$ as a (32-bit) machine word.

2.1 Security Model

The cryptographic protocols presented in this work are formalized using the standard universally composable (UC) framework [Can01]. We consider static corruptions. For simplicity, we omit standard UC technicalities such as (sub-)session IDs and the explicit scheduling of outputs by the simulator.

As in the tight ZK CPU protocol [YHH+24], our protocol is defined in the standard *commit-and-prove* hybrid model [CLOS02], formalized as $\mathcal{F}_{\mathsf{CPZK}}$ in Figure 5 (Appendix A.1). In the $\mathcal{F}_{\mathsf{CPZK}}$-hybrid model, the tight ZK CPU protocol (and our generalized variant) relies only on the well-known *Schwartz-Zippel lemma*, uses constant rounds and public coins, and does not require any computational assumptions.

There are several known ways to instantiate the $\mathcal{F}_{\mathsf{CPZK}}$ functionality (e.g., [AHIV17, BMRS21, BBB+18, CHM+20, DOT21, DIO21, IKOS07, MBKM19, YSWW21]). In this work, ZEE200 instantiates $\mathcal{F}_{\mathsf{CPZK}}$ using VOLE-based ZK proofs (e.g., [BMRS21, DIO21, YSWW21]), which are known for their efficient end-to-end performance. A brief discussion of this instantiation is deferred to Appendix A.2.

2.2 ZEE System and ZEE Assembly [HYDK21]

We review the ZEE system [HYDK21], which enables zero-knowledge proofs for programs written in ANSI C. ZEE provides a frontend compilation toolchain that lowers C code to a flat assembly program for a simple 32-bit register machine, together with a ZK-execution backend that proves correct execution of this assembly program.

ZK Execution Model in ZEE. At the proof level, ZEE enforces program execution using a *monolithic CPU circuit* that encodes the semantics of the ZEE assembly instruction set (IS) (cf. Figure 1). Each execution step evaluates the same fixed circuit, which takes as input the current machine state (program counter, registers, and memory interface) and outputs the next-state values. Consequently, the IS is intentionally compact: adding a new assembly operation typically enlarges this universal (for IS) circuit and therefore increases the cost of *every* executed step, even on paths that never use the new operation. As a result, many standard machine operations are lowered by the frontend compiler into longer sequences of supported ZEE instructions.

ZEE Abstract Machine. The ZEE abstract machine (a conceptual model of their computer) maintains a fixed register file $\mathcal{R}$, whose entries represent 32-bit words, including a distinguished register that stores the program counter %pc, together with a byte-addressed memory space $\mathcal{M}$. Execution proceeds by fetching the assembly instruction addressed at %pc, updating $(\mathcal{R}, \mathcal{M})$ according to the instruction semantics, and updating %pc to point to the next instruction unless control flow is explicitly overridden. An execution is deemed accepting if it reaches the distinguished instruction QED (and, depending on the statement being proved, HALT may also be treated as accepting). Figure 1 summarizes the ZEE instruction set and our notation.

Oracle Calls and Inputs. To cleanly support I/O and "compute-then-prove" patterns, ZEE includes an ORACLE instruction that invokes a designated host-side procedure that helps $\mathcal{P}$ generate extended witness by executing the program in plaintext, and an INPUT instruction that loads $\mathcal{P}$-provided 32-bit words into registers.

How ZEE200 Uses ZEE. In this work, we use ZEE assembly, as well as compilation components of ZEE. Namely, we reuse ZEE's frontend to compile ANSI C into ZEE assembly and build on the core of ZEE's runtime, including its handling of oracle calls and inputs (see above paragraph).

We fully replace the ZK protocol core of the system: Rather than relying on ZEE's monolithic ZK execution engine, ZEE200 executes our generalized tight ZK CPU, resulting in 20-40$\times$ improvement.

Our Instruction Set Terminology. One of our technical contributions is the introduction and use of composite instructions consisting of entire straightline program segments (see Section 3.2 for explanation and details, such as performance benefits). Because of this, we must use clear and consistent notation:

Throughout this work, we refer to the basic operations in ZEE (Figure 1) and those in the extended ZEE IS (see Section 5.1) as ZEE *assembly instructions* or just ZEE instructions. We will refer to the composite instructions consisting of *sequences* of ZEE instructions as *GTZK instructions* (GTZK omitted when clear from context).

2.3 Tight ZK CPU [YHH+24]: A Brief Recap

We briefly review the key components of the tight ZK CPU framework [YHH+24]. The tight ZK CPU models program execution as a sequence of arithmetic circuits defined over a finite field $\mathbb{F}$.

Each circuit represents a single *instruction* and maps $n \geq m$ field elements to $m+1$ outputs using fan-in-2 additions and multiplications over $\mathbb{F}$. (Here, n may vary across instructions.) The first m outputs correspond

	Syntax	Semantics
Algebra	`MOV` dst {src}	$\mathcal{R}[\text{dst}] := val(\text{src})$
	`CMOV` dst src_1 {src_2}	$\mathcal{R}[\text{dst}] := \begin{cases} val(\text{src}_2), & \text{if } \mathcal{R}[\text{src}_1] \neq 0 \\ \mathcal{R}[\text{dst}], & \text{otherwise} \end{cases}$
	`ADD` dst src_1 {src_2}	$\mathcal{R}[\text{dst}] := \mathcal{R}[\text{src}_1] + val(\text{src}_2)$
	`SUB` dst src_1 {src_2}	$\mathcal{R}[\text{dst}] := \mathcal{R}[\text{src}_1] - val(\text{src}_2)$
	`MUL` dst src_1 {src_2}	$\mathcal{R}[\text{dst}] := \mathcal{R}[\text{src}_1] \cdot val(\text{src}_2)$
	`XOR` dst src_1 {src_2}	$\mathcal{R}[\text{dst}] := \mathcal{R}[\text{src}_1] \oplus val(\text{src}_2)$
	`AND` dst src_1 {src_2}	$\mathcal{R}[\text{dst}] := \mathcal{R}[\text{src}_1] \wedge val(\text{src}_2)$
	`OR` dst src_1 {src_2}	$\mathcal{R}[\text{dst}] := \mathcal{R}[\text{src}_1] \vee val(\text{src}_2)$
	`EQZ` dst src	$\mathcal{R}[\text{dst}] := \begin{cases} 1, & \text{if } \mathcal{R}[\text{src}] = 0 \\ 0, & \text{otherwise} \end{cases}$
	`MSB` dst src	$\mathcal{R}[\text{dst}] := \begin{cases} 1, & \text{if } \mathcal{R}[\text{src}] \geq 2^{31} \\ 0, & \text{otherwise} \end{cases}$
	`POW2` dst src	$\mathcal{R}[\text{dst}] := 2^{\mathcal{R}[\text{src}]}$ with $\mathcal{R}[\text{src}] < 32$
Control	`JMP` {dst}	%pc $:= val(\text{dst})$
	`BNZ` src {dst}	%pc $:= \begin{cases} val(\text{dst}), & \text{if } \mathcal{R}[\text{src}] \neq 0 \\ \text{next instruction}, & \text{otherwise} \end{cases}$
	`PC` dst {src}	$\mathcal{R}[\text{dst}] :=$ %pc $+ val(\text{src})$
	`HALT`	– no effect, %pc unchanged –
	`QED`	– no effect, %pc unchanged –
Memory	`LOAD` dst a_0 {a_1}	$\mathcal{R}[\text{dst}] := \mathcal{M}[\mathcal{R}[a_0] + val(a_1)]$
	`STORE` src a_0 {a_1}	$\mathcal{M}[\mathcal{R}[a_0] + val(a_1)] := \mathcal{R}(\text{src})$
$\mathcal{P}$ Input	`INPUT` dst	$\mathcal{R}[\text{dst}] := x$ where $x \in [0, 2^{32})$ from $\mathcal{P}$
	`ORACLE` {id}	$\mathcal{P}$ calls procedure $val(\text{id})$

$$val(x) := \begin{cases} x, & \text{if } x \text{ is an immediate} \\ \mathcal{R}[x], & \text{if } x \text{ is a register id} \end{cases}$$

Figure 1: ZEE instruction set [HYDK21]. Each instruction type handles between zero and three arguments. In general, arguments refer to *registers*, but some arguments, denoted $\{\cdot\}$, can also optionally be *immediates* (i.e., compile-time constants). *val* is a helper function that resolves an argument that can be either a register or an immediate. Unless the semantics otherwise mention an effect on the %pc, each instruction also increments the %pc (to fetch next instruction).

Functionality $\mathcal{F}_{\mathsf{ZKCPU}}$

$\mathcal{F}_{\mathsf{ZKCPU}}$ interacts with a prover $\mathcal{P}$, a verifier $\mathcal{V}$, and a simulator $\mathcal{S}$. It is parameterized by a field $\mathbb{F}$, a register dimension $m \in \mathbb{N}$, a positive integer B, an instruction set $\{\mathcal{C}_1, \ldots, \mathcal{C}_B\}$ where each instruction is a standard arithmetic circuit over $\mathbb{F}$ of the form $\mathbb{F}^{m+n_{\mathsf{in}}} \mapsto \mathbb{F}^{m+1}$. For each instruction $\mathcal{C}_{i\in[B]}$ with $n_{\mathsf{in}}^{(i)} + m$ inputs and $n_{\times}^{(i)}$ multiplication, define the instruction size $n^{(i)}$ as $\max\{n_{\mathsf{in}}^{(i)}, n_{\times}^{(i)} + m + 2\}$. The functionality further takes as input an initial register state $\mathbf{st}^{(0)} \in \mathbb{F}^m$, a final register state $\mathbf{st}^{(\mathsf{final})} \in \mathbb{F}^m$.

Upon receiving a message

$$(\texttt{Prove}, \tau, i_1, \ldots, i_\tau, \mathbf{in}_1, \ldots, \mathbf{in}_\tau)$$

from $\mathcal{P}$, where $\tau \in \mathbb{Z}^+$, $i_j \in [B]$ for all $j \in [\tau]$, and each $\mathbf{in}_j \in \mathbb{F}^{n_{\mathsf{in}}^{(i_j)}}$, the functionality proceeds as follows:

1. Initialize $\mathbf{st} := \mathbf{st}^{(0)}$ and a flag $f := \texttt{true}$.
2. For each $j = 1, \ldots, \tau$, in order:
   (a) Evaluate $\mathbf{st}' \| f' := \mathcal{C}_{i_j}(\mathbf{st} \| \mathbf{in}_j)$.
   (b) If $f' \neq 0$ (i.e., invalid checking), set $f := \texttt{false}$.
   (c) Update $\mathbf{st} := \mathbf{st}'$.
3. If $\mathbf{st} \neq \mathbf{st}^{(\mathsf{final})}$, set $f := \texttt{false}$.
4. Let $n := \sum_{j=1}^{\tau} n^{(i_j)}$. If $\mathcal{P}$ is corrupted, $\mathcal{S}$ may send $(\texttt{Cheat}, n')$ with $n' \in \mathbb{Z}^+$, in which case set $f := \texttt{false}$ and $n := n'$.
5. Send $(\texttt{prove}, f, n)$ to $\mathcal{V}$ and $\mathcal{S}$.

Figure 2: Ideal functionality for a tight ZK CPU.

to the machine state (*registers*) and are carried across consecutive instructions. The final output encodes a *zero constraint*: in every valid execution this value must equal 0.

Prover $\mathcal{P}$ and verifier $\mathcal{V}$ agree on a fixed collection of B such circuits, the *instruction set* (IS). Each instruction implements a distinct state-transition function with a common output interface but potentially different input arity. Program execution proceeds by repeatedly selecting and executing instructions from this IS.

$\mathcal{P}$'s goal is to convince $\mathcal{V}$, in ZK, that there exists a sequence of instructions that transforms a given initial machine state to an accepting final state. Execution carries the program state by chaining the register outputs of one instruction to the register inputs of the next. At each execution step, $\mathcal{P}$ additionally supplies the (auxiliary) inputs required by the selected instruction, in ZK, and proves that its associated zero constraint evaluates to 0.

Note that the zero constraint can be used to enforce control flow. In particular, by assigning each instruction a distinct index and reserving one register as a *program counter*, the zero constraint can encode the difference between the program counter and the instruction's index. This forces the prover to execute the instruction specified by the current program counter.

The central feature of tight ZK CPU is that the cost of executing an instruction is proportional to its arithmetic circuit (in prior works, such as [HK20b, HYDK21], parties paid the maximal or even the sum cost of all instructions in the IS). This is possible with tight ZK CPU because it hides the instruction boundaries (and enforces correct instruction implementation by computing and verifying instruction hash on the fly). This gives us the flexibility to design large IS with instructions wildly differing in size (see Section 3.2 for how we take advantage of this). This also means that $\mathcal{V}$ does not learn the total instruction count, but rather the length of execution in the total circuit size required to implement the executed instructions.

Figure 2 depicts the ideal tight ZK CPU functionality from [YHH+24].

3 Technical Overview

We provide a high-level technical overview of our construction. We highlight the core ideas and intuition, deferring formal definitions, constructions, and detailed analyses to the subsequent sections.

3.1 Generalized Tight ZK (GTZK) CPU

There are two significant efficiency issues that affect direct integration of the tight ZK CPU framework and constructions in ZEE200:

First, tight ZK CPU (Section 2.3) is deliberately field-arithmetic-centric. This design choice is well-suited for modeling computation as field arithmetic, but it is a poor fit when the goal is to encode real-world programs (e.g., assembly).

Namely, in tight ZK CPU, each instruction is expressed as an arithmetic circuit over a finite field $\mathbb{F}$, and its cost is measured in terms of field additions and multiplications. However, semantics of assembly instructions are defined over machine words and bit-level state, and are often control-heavy, involving conditional updates, masked writes, and word-level operations.

While assembly programs do perform arithmetic, this arithmetic is typically over rings such as $\mathbb{Z}_{2^{32}}$, with well-defined wraparound and carry semantics. Simulating such operations within a large field $\mathbb{F}$ requires explicit bit-decomposition and carry-handling logic, which introduces substantial overhead unrelated to the instruction's high-level behavior. As a result, even simple assembly instructions may incur significant costs due to encoding artifacts rather than to inherent computational complexity.

Second, tight ZK CPU framework supports random-access memory via *non-black-box* use of the ZK RAM [YH24], in which each load and store is compiled into a sequence of arithmetic operations over $\mathbb{F}$. While this integration is conceptually straightforward and preserves tightness, it substantially inflates cost: a single assembly memory operation expands into many low-level field additions and multiplications. Moreover, this overhead is *multiplicative*, since each field multiplication induced by the compilation must itself be handled by tight ZK CPU, incurring an additional constant factor.

Additionally, we wish to use memory operations far more generally than just handling explicit load and store instructions. They can be used to implement higher-level primitives such as table lookups, which are useful for simulating assembly-level operations within an arithmetic circuit. This further motivates introducing a very fast ZK RAM into our novel GTZK framework.

Both issues stem from the limited ability to introduce non-linearity in the tight ZK CPU model. In the framework of [YHH+24](see Figure 2), the only sources of non-linearity available to an instruction are the introduction of fresh auxiliary inputs and multiplication over an arithmetic field $\mathbb{F}$. All other non-linear operations, such as memory accesses and table lookups, must ultimately be implemented using these limited non-linear resources. Meanwhile, this restriction is a key structural reason why the tight ZK CPU framework is able to achieve *tightness*, i.e. cost proportional to executed instruction/path.

Achieving Tightness. It is instructive to briefly examine tight ZK CPU's mechanism of achieving tightness.

At a high level, tight ZK CPU separates the basic "units" of non-linear computation and batch-processes them outside of instructions. That is, $\mathcal{P}$ and $\mathcal{V}$ build (many) instances of what we call *universal non-linear units* (in, ℓ, r, o). Here, $\mathcal{P}$ chooses and holds these values, and $\mathcal{V}$ holds their commitments. The non-linear relation $o = \ell \cdot r$ is enforced by the method of their construction; in allows to input extended witness into the computation, and can be arbitrarily chosen by $\mathcal{P}$. Crucially, the units are independent of the instruction set. Syntactically, they can be used by any of the instructions, but of course, because $\mathcal{P}$ commits to the specific values, they can be correctly used only in the intended part of the instruction/computation.

Each executed instruction consumes a fixed number of such universal units, determined by the circuit representing the instruction. The instruction *embeds* these units by imposing *linear constraints* that define each ℓ and r as linear combinations of instruction's circuit's other wires, including other in and o it uses (and, if needed, the constant 1). In other words, an instruction, using linear constraints, specifies how the committed (in, ℓ, r, o) tuples must be wired into the circuit so that it exactly computes the function of the instruction.

We are ready to present the intuition for achieving tightness: (i) $\mathcal{P}$ commits, via universal units, to the non-linear constraints of the entire execution; (ii) these units are partitioned (obliviously to $\mathcal{V}$) according to the executed instructions, and $\mathcal{V}$ must check that they each satisfy linear constraints imposed by a corresponding instruction in a valid execution; and (iii) the above partition is part of $\mathcal{P}$'s extended witness, and the check can be done cheaply.

Together, this implies that $\mathcal{P}$ can assemble instructions in any order (with the boundaries between them hidden and, of course, $\mathcal{P}$ including proofs of correct assembly), and parties only pay for the non-linear operations needed to evaluate the actual execution path.

Accordingly, the entire program execution can be seen as a linear combination of universal non-linear units, whose total number determines the execution cost. This also suggests that the interface of the universal unit itself can be generalized to support richer forms of constraints – see next:

Generalized Tight ZK CPU (GTZK CPU or just GTZK). Motivated by the preceding discussion, we show that the tight ZK CPU framework can be extended to allow *general* non-linear operations, while preserving tightness. Going beyond [YHH+24]'s limited collection of non-linear functions (auxiliary inputs and field multiplications), our generalized tight ZK CPU (GTZK) allows instructions to embed key-value store (KVS) operations (thus implementing RAM) and inputs from restricted sets (useful in enforcing range checks), as well as designate a collection of wires that must be zero. This allows to efficiently implement common assembly behaviors — e.g., loads, stores, and table look-ups — directly, rather than with (expensive!) non-black-box evaluation via field operations.

That is, instead of integrating memory accesses and set constraints in a non-black-box manner as in [YHH+24], we invoke the underlying ZK RAM and ZK SET machinery as black-box primitives to enforce the correctness of key-value store operations and membership in a fixed public set.[1]

All non-linear operations are batch-processed and packaged into (now richer) universal non-linear units. Each such unit will now contain several heterogeneous non-linear constraints: inputs, multiplications, KVS, and set-restricted inputs. As in the original tight ZK CPU framework, $\mathcal{V}$ learns only the total execution length (i.e., the number of universal units used). In our setting, this length induces separate upper bounds for each non-linear constraint class, including key-value store accesses and set-restricted inputs, and — as in the original tight ZK CPU — unrestricted inputs and multiplication gates. Importantly, it does *not* reveal access patterns, indices, values, the executed instruction sequence, or intermediate states. As is standard (see, e.g., discussion in [YHH+24, GKK+12]), if hiding the exact number of such operations is desirable, $\mathcal{P}$ can pad the execution with dummy universal units.

Note, the universal unit allows parties to agree on arbitrary number of operations of each type. This does not affect security (beyond implying corresponding upper bounds), and offers significant opportunity for performance tuning. Similar to building roads and cities in a popular board game "The Settlers of Catan," evaluation of each instruction will require a corresponding minimum number of resources of certain type (multiplications, KVS ops, etc). In GTZK, these resources are dispensed in chunks defined by the profile of the universal unit.

[1] ZK RAM of course implies ZK SET; however, realizing set-restricted inputs directly is cheaper. Since our construction uses ZK SET extensively (e.g., for batched bit decomposition), we treat it as a separate primitive.

Crucially, the GTZK generalization does *not* change the overall execution model: a program execution is still a sequence of instructions, and tightness is achieved. What changes is that individual instructions can now embed richer constraints directly, reducing compilation overhead. Beyond the conceptual idea, defining and implementing our GTZK CPU framework requires careful formalization and refining of tight ZK CPU's core building blocks, such as *topology matrices*. See Section 4 and Appendix B for details.

3.2 ZEE200: Executing ANSI C over GTZK CPU

In this section, we provide a high-level overview of ZEE200, a compiler toolchain that compiles general C programs into (extended) ZEE assembly (Figure 1 and Figure 4), and executes them on the virtual machine (VM), run by $\mathcal{P}$ and $\mathcal{V}$, that implements GTZK CPU. In the remainder, we will explicitly distinguish ZEE and extended ZEE IS where relevant (e.g., in performance evaluation), and for brevity may omit "extended" otherwise.

At this level of abstraction, there are two main components of the system: GTZK CPU handles idealized execution mechanisms — e.g., registers, zero constraints, arithmetic operations, and a key-value store — while the ZEE200's compilation/runtime component is concerned mainly with compiling C into GTZK instructions, and with the systems aspects of the VM.

In the remainder of this section, we discuss several core aspects of our designs. Section 5 covers these issues in detail.

Induced GTZK CPU Instructions. A central feature of the GTZK framework (inherited from tight ZK [YHH+24], but not explored in the original work [YHH+24]) is its ability to cheaply support large instruction sets with arbitrarily varying instruction sizes, with cost proportional to the circuit complexity of the execution path. This allows us to combine entire instruction sequences into a single GTZK instruction. At the high level, this facilitates saving on: (i) instruction prologue and epilogue, which each includes routing of the register file and other operations; and on (ii) circuitry/constraints needed for instruction decoding.

The assembly program consists of machine-level instructions such as ADD and LOAD. A basic design is to treat each *assembly instruction type* (e.g., ADD, LOAD) as a single GTZK instruction. This will incur the per-instruction costs described above (register file routing and instruction decoding).

Instead, we proceed as follows. We build a control flow graph (CFG) of our program and consider the sequences of instructions forming each *basic block* (a straightline program segment). Each *basic block* — a maximal straight-line sequence of assembly instructions with a single entry point and a *single* control-flow exit — is compiled into a single GTZK CPU instruction. Within a basic block, register values are propagated directly along wires, and the fixed per-instruction cost is amortized across all assembly instructions in the block. Control flow is enforced only at block boundaries by appropriately checking and updating the program counter register.

In this way, basic blocks amortize the fixed per-instruction cost of GTZK CPU, while preserving tightness. This approach yields execution costs that scale with the circuit complexity of the program execution path, rather than with executed ZEE instruction count.

Note, the idea of basic blocks as units of execution has appeared in the context of executing real programs in cryptography. Examples include MPC [YPHK23] and outsourced computation [WSR+15] systems.

Encoding Machine Words. Assembly programs typically operate over fixed-size machine words. In ZEE [HYDK21], each register and memory location stores a 32-bit integer, i.e., a value in $\mathbb{Z}_{2^{32}}$. In contrast, the (generalized) tight ZK CPU operates over wires and registers defined over a prime field. Accordingly, in

ZEE200, we choose a sufficiently large field — e.g., $\mathbb{F}_{2^{61}-1}$ — to embed 32-bit integers directly. We note that choosing a sufficiently large field is also important for maintaining soundness in (generalized) tight ZK CPU and ZEE200.

Restricted Sets and Range Checks. In ZEE200, we instantiate a restricted set from which GTZK CPU may draw values. The set is programmed to contain all values in the range $[0, 2^{16})$. This allows $\mathcal{P}$ to supply inputs drawn from this set, while assuring $\mathcal{V}$ that the corresponding wires encode valid 16-bit values embedded in the underlying (sufficiently large prime) field $\mathbb{F}$.

To illustrate the usefulness of this mechanism, consider the assembly instruction ADD, whose semantics are defined over $\mathbb{Z}_{2^{32}}$. Suppose that two field wires encode values x and y, interpreted as elements of $\mathbb{Z}_{2^{32}}$. Our goal is to compute their sum in $\mathbb{Z}_{2^{32}}$ and embed the result back into $\mathbb{F}$.

Since the field is chosen to be sufficiently large, we can first compute the field addition $z = x + y$, which may be a 33-bit value. To enforce correct modular reduction, we ask $\mathcal{P}$ to provide one unrestricted input $in \in \{0, 1\}$ (enforced via the constraint $in(in-1) = 0$), together with two restricted inputs $hi, lo \in [0, 2^{16})$. We then enforce, via a zero constraint, that

$$z = in \cdot 2^{32} + hi \cdot 2^{16} + lo.$$

The correct result of the ADD instruction is obtained as $hi \cdot 2^{16} + lo$.

Choosing 2^{16} as the size of the restricted set represents a trade off. Indeed, the cost of the ZK SET primitive [YH24] includes an additive term proportional to the size of the set. This makes larger sets, such of size 2^{32} correspondingly more expensive. More broadly, decomposing machine words into 16-bit chunks allows us to work over a moderately sized field, such as $\mathbb{F}_{2^{61}-1}$, while still correctly implementing 32-bit machine-word arithmetic. This design avoids the need to choose a much larger field solely to accommodate wide intermediate values (e.g., from multiplication; see Section 5) and leads to better efficiency.

Key-Value Store Structure. We now outline how ZEE200 leverages the key-value store (KVS) abstraction exposed by the generalized tight ZK CPU framework.

KVS maintains a global key-value map that persists across the entire program execution. It is accessed via a dedicated kvs gate, which takes as input a key and a new value, outputs the previously stored value associated with that key, and updates the map accordingly. That is, this gate behaves as an atomic read-write operation. A GTZK instruction may invoke multiple such kvs gates, whose effects are applied in a fixed and well-defined order. In the context of ZEE200, this KVS maintains key-value pairs over $\mathbb{F} \times \mathbb{F}$.

The direct use of the KVS in ZEE200 is to support memory access operations in the ZEE assembly IS. For example, a LOAD instruction can be realized by invoking the memory access primitive at the desired address. Naïvely, a read-only memory access could be implemented using two access gates — first reading the value and then immediately writing it back unchanged. However, because $\mathcal{P}$ knows the fetched value, $\mathcal{P}$ can supply it directly as an unrestricted input, use it as the new value in the access gate, and enforce correctness via a zero constraint requiring that the supplied value equals the value read from memory. (This can be further optimized; see Section 5.3.)

Beyond basic memory operations, ZEE200 leverages KVS in a more general way. Our key observation is that when working over a sufficiently large field $\mathbb{F}$, the key space of the KVS is far larger than what is required to represent program memory. As a result, a large fraction of the key space remains unused when considering only memory accesses. We exploit these unused key ranges to embed lookup tables that assist in executing certain ZEE instructions.

As a representative example, we use such tables to implement bitwise operations, such as the XOR instruction. Suppose that two field wires encode values x and y, interpreted as elements of $\mathbb{Z}_{2^{32}}$, and our goal is to compute their bitwise XOR. In ZEE200, we instantiate a lookup table for 8-bit XOR. Concretely,

we fix a public offset beyond the allowed range of RAM addresses (e.g., 2^{40}) and populate the KVS so that the (*public*) value stored at key $2^{40} + 2^8 a + b$ equals $a \oplus b$, for all $a, b \in [0, 2^8)$.

To compute the bitwise XOR of x and y, we first decompose each value into four 8-bit chunks. We then perform four table lookups, each shifted by the fixed offset, and combine the results to obtain the final 32-bit XOR output.

KVS incurs additive cost proportional to the number of stored items. Consequently, ZEE200 instantiates the XOR lookup table at 8-bit granularity, balancing expressiveness and efficiency.

Summary. Given a C assembly program, ZEE200 compiles it into a sequence of GTZK CPU instructions, each corresponding to a single basic block. Each such instruction is realized as a straight-line arithmetic circuit that performs the computation of the basic block and enforces its control-flow semantics via program-counter constraints, while invoking kvs gates and restricted-set inputs as needed. Importantly, once this structure is fixed, the compilation task largely reduces to specifying how individual ZEE assembly instructions are translated into circuit constraints. This per-instruction compilation constitutes the core technical component of ZEE200 and is the focus of the detailed treatment in Section 5.

3.3 Security of ZEE200

We briefly discuss the security of ZEE200.

Cryptographically, this work defines a GTZK CPU functionality and presents a protocol that UC-realizes it, together with a simulation-based proof. This functionality provides a ZK execution model for programs expressed as sequences of generalized tight ZK CPU instructions. See Section 4.

Given this execution model, ZEE200 acts as a compiler and a runtime executing C programs on GTZK CPU. Thus, the security of ZEE200 follows directly from: (i) the security of GTZK CPU functionality, and (ii) correctness and security of implementation of each ZEE and GTZK instruction (and their sequences) as a constraint system, according to exact assembly semantics; we also provide formal proofs of this property, covering every corner case, such as division by 0, etc. See Section 5.

4 Generalized Tight ZK CPU Functionality

In this section, we formalize a generalized tight ZK CPU functionality that will serve as the ideal execution model for zero-knowledge evaluation of assembly programs. Our formulation generalizes the tight ZK CPU abstraction of [YHH+24] to support richer instruction semantics, including structured auxiliary inputs (drawn from a fixed set), multiple zero constraints, and key-value store operations.

We begin by extending the notion of an instruction. This notion is general and is applicable to both standard assembly instructions, as well as our basic-block GTZK instructions.

Definition 4.1 (Instruction)**.** An *instruction* is specified by a circuit

$$\mathcal{C} : \mathbb{F}^{m+n_{\mathsf{set}}+n_{\mathsf{in}}} \to \mathbb{F}^m,$$

together with (i) a restricted auxiliary input domain $\mathsf{set} \subseteq \mathbb{F}$ and (ii) a set of designated zero wires zero.

The circuit $\mathcal{C}$ is a fan-in-2 arithmetic circuit over $\mathbb{F}$, composed of standard addition and multiplication gates, and may additionally contain a special fan-in-2 *key-value store* gate, denoted kvs. (We note that kvs gates are not standard arithmetic gates: they represent externally enforced key-value store constraints whose semantics are defined with respect to a global map $\mathcal{M}$.) The inputs to $\mathcal{C}$ consist of m register inputs, n_{set} inputs drawn from set, and n_{in} unrestricted auxiliary inputs; the outputs of $\mathcal{C}$ are the updated m register values.

A KVS gate kvs takes as input an index wire idx and a value wire $\mathsf{val}_{\mathsf{new}}$, and outputs a wire $\mathsf{val}_{\mathsf{old}}$. Its semantics are defined with respect to a global key-value map $\mathcal{M}$: the output satisfies $\mathsf{val}_{\mathsf{old}} = \mathcal{M}[\mathsf{idx}]$, and the map is updated so that $\mathcal{M}[\mathsf{idx}] := \mathsf{val}_{\mathsf{new}}$.[2]

Finally, an execution of $\mathcal{C}$ is considered *valid* only if for all $i \in \mathsf{zero}$, the value of wire i equals 0, and every kvs gate is applied only to indices in the domain of $\mathcal{M}$.

We now formalize the notion of a *universal unit*, discussed in Section 3. A universal unit is a collection of constraints, whose type and number is determined by a tuple

$$(u_{\mathsf{in}}, u_{\mathsf{set}}, u_{\mathsf{kvs}}, u_{\times}),$$

referred to as its *profile*. Profile specifies how many non-linear constraints of each type (unrestricted inputs, inputs drawn from the fixed set, key-value store constraints, and multiplication constraints) are included in a single universal unit.

In the GTZK CPU protocol (see Appendix B), a universal-unit profile is fixed once per proof, and each instruction consumes a number of such universal units, enough to meet its needs in each instruction type; we refer to this quantity as the *size* of the instruction, as it reflects the actual cost. Formally, for an instruction with m registers, n_{in} unrestricted auxiliary inputs, n_{set} inputs from set, n_{kvs} kvs gates, and $n_{\times}$ multiplication gates, its size n is defined as

$$n \;=\; \max\left\{ \left\lceil \frac{n_{\mathsf{in}}}{u_{\mathsf{in}}} \right\rceil, \left\lceil \frac{n_{\mathsf{set}}}{u_{\mathsf{set}}} \right\rceil, \left\lceil \frac{n_{\mathsf{kvs}}}{u_{\mathsf{kvs}}} \right\rceil, \left\lceil \frac{n_{\times} + m + 2}{u_{\times}} \right\rceil \right\} \tag{1}$$

We make the following remarks:

1. The size of an instruction exactly captures the number of universal units required to encode all of its non-linear constraints; sufficient units must be allocated to accommodate every (non-linear) constraint type.

2. The instruction size determines both the execution length revealed to $\mathcal{V}$ and the cost incurred by the underlying ZK protocol, which is linear in the instruction size, with the per-unit cost determined by the chosen universal-unit profile.

3. As in the original tight ZK CPU construction [YHH+24], register updates are encoded via multiplication constraints, with two additional multiplications accounting for a constant-1 wire and a global zero-checking wire (obtained via random linear combination of the designated zero wires).

4. A universal unit is fully specified by its profile; any ordering of the non-linear constraints within a unit is acceptable for execution and may be arbitrary (set at the beginning of the proof). The profile determines the concrete cost of handling each unit and thus directly impacts the overall performance.

In a GTZK CPU execution, parties agree on the instruction set (IS), and the initial and final states of the program (note, it is sufficient to limit these states just to the register file, cf. Figure 3). $\mathcal{P}$ shows, in ZK, that the final state is reachable from the initial state via a sequence of (possibly repeated) instructions in IS. We formalize this functionality in Figure 3 and make several remarks:

[2]When multiple kvs gates appear in an instruction, they are applied sequentially in a fixed order specified by the instruction, which determines the resulting key-value store state. As in assembly programs, we assume this order is implicitly defined and do not model it explicitly.

Functionality $\mathcal{F}_{\text{G-ZKCPU}}$

$\mathcal{F}_{\text{G-ZKCPU}}$ interacts with a prover $\mathcal{P}$, a verifier $\mathcal{V}$, and a simulator $\mathcal{S}$. It is parameterized by a field $\mathbb{F}$, a register dimension $m \in \mathbb{N}$, an auxiliary input domain $\mathsf{set} \subseteq \mathbb{F}$, a positive integer B, an instruction set $\{\mathcal{I}_1, \ldots, \mathcal{I}_B\}$ as in Definition 4.1 (where each instruction satisfies $\mathcal{I}_i.\mathsf{set} = \mathsf{set}$), and a universal-unit profile. The profile induces an instruction size $n^{(i)}$ for each instruction $\mathcal{I}_i$, as defined in Equation (1). The functionality further takes as input an initial register state $\mathbf{st}^{(0)} \in \mathbb{F}^m$, a final register state $\mathbf{st}^{(\mathsf{final})} \in \mathbb{F}^m$, and an initial global key-value map $\mathcal{M} : \mathbb{F} \mapsto \mathbb{F}$, whose domain $\mathsf{dom}(\mathcal{M})$ may be a subset of $\mathbb{F}$.

Upon receiving a message

$$(\texttt{Prove}, \tau, i_1, \ldots, i_\tau, \mathbf{in}_1, \ldots, \mathbf{in}_\tau)$$

from $\mathcal{P}$, where $\tau \in \mathbb{Z}^+$, $i_j \in [B]$ for all $j \in [\tau]$, and $\mathbf{in}_j \in \mathbb{F}^{n_{\mathsf{set}}^{(i_j)} + n_{\mathsf{in}}^{(i_j)}}$ with the first $n_{\mathsf{set}}^{(i_j)}$ entries in set, the functionality proceeds as follows.

1. Initialize $\mathbf{st} := \mathbf{st}^{(0)}$ and a flag $f := \texttt{true}$.
2. For each $j = 1, \ldots, \tau$, in order:
   (a) Evaluate $\mathbf{st}' := \mathcal{I}_{i_j}.\mathcal{C}(\mathbf{st} \,\|\, \mathbf{in}_j)$, where any kvs gates are evaluated with respect to the current global map $\mathcal{M}$.
   (b) If any wire indexed by $\mathcal{I}_{i_j}.\mathsf{zero}$ evaluates to a non-zero value, or if any kvs gate is undefined (i.e., $\mathcal{M}[\mathsf{idx}]$ is not defined), set $f := \texttt{false}$.
   (c) Update $\mathbf{st} := \mathbf{st}'$.
3. If $\mathbf{st} \neq \mathbf{st}^{(\mathsf{final})}$, set $f := \texttt{false}$.
4. Let $n := \sum_{j=1}^{\tau} n^{(i_j)}$. If $\mathcal{P}$ is corrupted, $\mathcal{S}$ may send $(\texttt{Cheat}, n')$ with $n' \in \mathbb{Z}^+$, in which case set $f := \texttt{false}$ and $n := n'$.
5. Output $(\texttt{Prove}, f, n)$ to $\mathcal{V}$ and $\mathcal{S}$.

Figure 3: Ideal functionality for a generalized tight ZK CPU.

1. $\mathcal{V}$ learns only the total execution length, and does not learn the number of executed instructions τ explicitly.

2. The tight ZK CPU functionality of [YHH+24] can be seen as a special case of our generalized formulation in which (i) there are no inputs drawn from set and no key-value store gates present, (ii) the set zero contains a single wire, and (iii) the universal-unit profile is fixed to $u_{\mathsf{in}} = u_{\times} = 1$ (the parameters u_{set} and u_{kvs} are discarded in this case, or $u_{\mathsf{set}} = u_{\mathsf{kvs}} = 0$).

3. As in the original tight ZK CPU functionality [YHH+24], $\mathcal{P}$ may select instructions adaptively. Looking ahead, we will restrict instruction selection by enforcing a zero constraint on a designated *program counter* register.

4. The universal-unit profile is fixed globally for the entire execution and is independent of the instruction sequence; it determines both the instruction size and the per-unit cost incurred by the underlying ZK protocol (specified next).

Instantiation of $\mathcal{F}_{\text{G-ZKCPU}}$. The generalized tight ZK CPU functionality $\mathcal{F}_{\text{G-ZKCPU}}$ can be instantiated using standard commit-and-prove techniques, by closely following the construction of the original tight ZK CPU protocol Π_{ZKCPU} from [YHH+24].

Recall that Π_{ZKCPU} is realized in the standard commit-and-prove ZK model augmented with a ZK unbalanced read-only memory (ZK UROM) functionality, denoted $\mathcal{F}_{\text{CPZK-UROM}}$, which itself can be implemented

based on the standard commit-and-prove ZK functionality $\mathcal{F}_{\mathsf{CPZK}}$ using the techniques in [YH24, YHH+24]. At a high level, $\mathcal{P}$ commits to a sequence of universal-unit witnesses consisting of unrestricted inputs and multiplication triples, and $\mathcal{V}$ checks that all multiplication relations hold via calls to $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$. Since instructions in [YHH+24] involve only unrestricted inputs and multiplication gates, it suffices to enforce that each left and right wire is a linear combination of inputs and multiplication outputs, which can be verified using a so-called *topology vector* via ZK UROM.

Our key observation is that this idea can be extended to $\mathcal{F}_{\mathsf{G\text{-}ZKCPU}}$. In particular, the $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$ can be augmented with a ZK SET and a ZK key-value store by again using the technique in [YH24], resulting in functionality $\mathcal{F}_{\mathsf{CPZK\text{-}UROM\text{-}SET\text{-}KVS}}$ (Figure 7 in Appendix B.2). This allows $\mathcal{P}$ to additionally commit to inputs drawn from a fixed public set and to key-value store triples, whose correctness relations are enforced by the corresponding ideal procedures.

With this update, all non-linear constraints introduced by our extended definition of instruction (Definition 4.1) are handled by the universal-unit abstraction.[3] The only changes are that (i) the left and right wires of multiplication gates may now be linear combinations of unrestricted inputs, set-constrained inputs, multiplication outputs, and key-value store outputs, and (ii) the inputs to kvs gates must satisfy analogous linear constraints. Equivalently, from the hybrid protocol's perspective, although these commitments may be generated by different hybrid procedures, once they are fixed, restricted inputs (KVS triples, resp.) can be treated identically to unrestricted inputs (standard multiplication triples, resp.).

We defer the (detailed) formalization of the corresponding protocol and proof to Appendix B. Our protocol is defined in the $\mathcal{F}_{\mathsf{CPZK}}$-hybrid model. In turn, we instantiate $\mathcal{F}_{\mathsf{CPZK}}$ with VOLE-based ZK (see Appendix A.2). We conclude by stating our main theorem:

Theorem 4.2. *Protocol $\Pi_{\mathsf{G\text{-}ZKCPU}}$ (Figures 8 to 10 in Appendix B) UC-realizes $\mathcal{F}_{\mathsf{G\text{-}ZKCPU}}$ (Figure 3) in the $\mathcal{F}_{\mathsf{VOLE}}$-hybrid model (Figure 6 in Appendix A.2) with perfect zero-knowledge and soundness error*

$$\frac{n(4u_{\mathsf{in}} + 7u_{\mathsf{set}} + 14u_{\mathsf{kvs}} + 7u_{\times} + 7) + 2|\mathsf{set}| + 2|\mathcal{M}| + \alpha + 8}{|\mathbb{F}|}$$

where $\alpha := \max\{n, n^{(1)}, \ldots, n^{(B)}\}(u_{\mathsf{in}} + u_{\mathsf{set}} + u_{\mathsf{kvs}} + u_{\times}) + \sum_{i=1}^{B} |\mathcal{I}_i.\mathsf{zero}|$, against any static, computationally unbounded adversary. The protocol achieves the following efficiency:

- ***Communication:*** $n(2u_{\mathsf{in}} + 5u_{\mathsf{set}} + 12u_{\mathsf{kvs}} + 4u_{\times} + 8) + 2|\mathsf{set}| + 3|\mathcal{M}| + 2B + \mathcal{O}(1)$ *field elements.*
- ***Computation:*** $\mathcal{O}(n + |\mathsf{set}| + |\mathcal{M}| + \sum_{i\in[B]} |\mathcal{I}_i.\mathcal{C}|)$ *field operations, where each $u_{_}$ is treated as a constant.*
- ***Rounds:*** $\mathcal{O}(1)$.
- *It requires a VOLE correlation of length $\mathcal{O}(n + |\mathsf{set}| + |\mathcal{M}| + B)$, where each $u_{_}$ is treated as a constant.*

Universal Unit and its Profile. Universal unit profile affects performance of GTZK CPU. E.g., an execution known (to both parties) to heavily use LOADs would benefit from universal units with larger u_{kvs}. Because it cleanly interfaces with $\mathcal{F}_{\mathsf{G\text{-}ZKCPU}}$, it is a convenient abstraction for improving efficiency. We leave a full exploration of this direction to future work. All our experiments use the $(1, 1, 1, 1)$ profile; we briefly report on experiments with different profiles in Appendix 6.4, demonstrating promise of a future investigation.

Note that participants of computation must learn some information about its duration. In computing (proving) a function encoded as a fixed circuit, the length of execution is *a priori* known to both parties. In

[3] Multiple zero wires are handled via a random linear combination, as discussed in [YHH+24].

CPU-emulation-based computations, the length of execution *necessarily* does reveal additional information about the computation. This is well-understood. Typically, the length of computation is defined as the number of CPU steps. Tight ZK CPU [YHH+24] pioneered the approach where the length is measured in the number of field multiplications (and inputs). In this work, we generalize the cost to be measured in the *well-defined* number of universal units (which induces upper bounds on the number of inputs, multiplications, KVS, restricted-set lookups used in the computation). This is a natural generalization of prior notions of program length, and is very useful in fine-tuning system performance.

5 ZEE200

We describe the system design of ZEE200, which targets GTZK CPU ideal functionality (Figure 3) as the cryptographic backend for assembly programs. Our focus is on how conventional 32-bit programs (C, compiled to ZEE or other assemblies) are mapped into executions of GTZK CPU, and on the architectural choices that govern this mapping. These choices determine both the expressiveness of the supported ZEE assembly IS and how efficiently its (assembly) instructions are realized as GTZK CPU instructions.

System Pipeline. At a high level, ZEE200 comprises a conventional compilation toolchain, the GTZK CPU execution backend, and a component for mapping (extended) ZEE assembly to GTZK basic-block-based instructions. Programs specifying ZK statements are written as off-the-shelf C code and compiled using the standard ZEE frontend toolchain into a flat assembly program for a 32-bit register-based IS (see Section 2.2). This frontend step is largely reused from prior work [HYDK21]; we do *not* claim it as our contribution.

The core responsibility of ZEE200 is, given a ZEE assembly program, translate and execute it on a GTZK CPU. Concretely, the assembly program is decomposed into basic blocks, and each basic block is compiled into a single GTZK instruction per Definition 4.1. Each such instruction is represented as a constraint system that invokes arithmetic operations, key-value store accesses, accepts unrestricted and restricted-set inputs, and that enforces control flow via explicit program-counter constraints.

Of course, ZEE200 includes an implementation of GTZK CPU protocol described in Section 4, which serves as the cryptographic backend for executing the compiled instruction sequence. It is implemented as a VM run by $\mathcal{P}$ and $\mathcal{V}$. Execution of the compiled program proceeds by simply invoking this backend in a black-box manner. Therefore, throughout this section, we treat GTZK CPU as the ideal functionality $\mathcal{F}_{\text{G-ZKCPU}}$ (see Figure 3).

Thus, the focus of this section is on specifying, given a flat ZEE assembly program (note, this can be naturally extended to assembly programs in other languages), how $\mathcal{P}$ and $\mathcal{V}$ agree on the public parameters required by $\mathcal{F}_{\text{G-ZKCPU}}$, including the choice of field, register layout, instruction set, etc.

5.1 Extended ZEE Assembly IS

Recall that in the original ZEE [HYDK21], every execution step evaluates a monolithic CPU circuit that encodes the entire assembly IS. As a result, the IS is deliberately kept small, and many standard assembly operations must be compiled into sequences of simpler ZEE instructions. Indeed, the ZEE frontend compiler explicitly lowers unsupported instructions into such sequences during compilation.

In ZEE200, this restriction no longer applies. Under the (generalized) tight ZK CPU execution model, cryptographic cost is charged only for the instructions that are actually executed. Consequently, assembly instructions can be supported natively whenever they admit efficient constraint representations, without incurring overhead on execution paths that do not use them. Accordingly, ZEE200 directly operates on the unlowered assembly programs produced by the ZEE frontend compiler. These programs include an

	Syntax	Semantics
Algebra	UDIV dst src_1 $\{\mathsf{src}_2\}$	$\mathcal{R}[\mathsf{dst}] := \lfloor \mathcal{R}[\mathsf{src}_1] / val(\mathsf{src}_2) \rfloor$ (unsigned) with $val(\mathsf{src}_2) \neq 0$
	UREM dst src_1 $\{\mathsf{src}_2\}$	$\mathcal{R}[\mathsf{dst}] := \mathcal{R}[\mathsf{src}_1] \bmod val(\mathsf{src}_2)$ (unsigned) with $val(\mathsf{src}_2) \neq 0$
	SRA dst src $\{\mathsf{sh}\}$	$\mathcal{R}[\mathsf{dst}] := \mathcal{R}[\mathsf{src}] \gg val(\mathsf{sh})$ (signed) with $val(\mathsf{sh}) < 32$
	ULT dst src_1 src_2	$\mathcal{R}[\mathsf{dst}] := 1$ if $\mathcal{R}[\mathsf{src}_1] < \mathcal{R}[\mathsf{src}_2]$, else 0 (unsigned)
	LT dst src_1 src_2	$\mathcal{R}[\mathsf{dst}] := 1$ if $\mathcal{R}[\mathsf{src}_1] < \mathcal{R}[\mathsf{src}_2]$, else 0 (signed)
Memory	LOADB dst a_0 $\{a_1\}$	$\mathcal{R}[\mathsf{dst}] := \mathsf{sign_extend}\big(\mathsf{MEM_BYTE}[\mathcal{R}[a_0] + val(a_1)]\big)$
	ULOADB dst a_0 $\{a_1\}$	$\mathcal{R}[\mathsf{dst}] := \mathsf{zero_extend}\big(\mathsf{MEM_BYTE}[\mathcal{R}[a_0] + val(a_1)]\big)$
	STOREB src a_0 $\{a_1\}$	$\mathsf{MEM_BYTE}[\mathcal{R}[a_0] + val(a_1)] := \big(\mathcal{R}[\mathsf{src}] \bmod 2^8\big)$
	LOADH dst a_0 $\{a_1\}$	$\mathcal{R}[\mathsf{dst}] := \mathsf{sign_extend}\big(\mathsf{MEM_HALF}[\mathcal{R}[a_0] + val(a_1)]\big)$ with an even address
	ULOADH dst a_0 $\{a_1\}$	$\mathcal{R}[\mathsf{dst}] := \mathsf{zero_extend}\big(\mathsf{MEM_HALF}[\mathcal{R}[a_0] + val(a_1)]\big)$ with an even address
	STOREH src a_0 $\{a_1\}$	$\mathsf{MEM_HALF}[\mathcal{R}[a_0] + val(a_1)] := \big(\mathcal{R}[\mathsf{src}] \bmod 2^{16}\big)$ with an even address

Figure 4: Extended assembly instructions supported by ZEE200. They follow the same notation as Figure 1. *All original* ZEE *instructions are also supported in* ZEE200. All operations act on 32-bit words; arithmetic is modulo 2^{32} unless otherwise noted.

extended set of assembly instructions; see Figure 4. We refer to this extended set (combined with ZEE) as *extended* ZEE.

We remark that the modular design of ZEE200 naturally supports further extensions of the assembly IS. Exploring additional instruction support and its interaction with (non-crypto) compiler design is an interesting direction for future work.

5.2 Public Setup of $\mathcal{F}_{\mathsf{G\text{-}ZKCPU}}$

We describe ZEE200's public set up of the GTZK CPU for a given flat ZEE assembly program. This set up determines the global execution parameters shared by all instructions and the public specification of the GTZK instructions induced by the assembly program.

These design choices are inherently interdependent, even though our presentation first fixes the global execution parameters and then specifies the induced instructions under these parameters.

5.2.1 Global Execution Parameters

We start by describing global execution parameters of $\mathcal{F}_{\mathsf{G\text{-}ZKCPU}}$ as fixed by ZEE200 for a given flat ZEE assembly program. These parameters define the execution environment shared by all induced generalized GTZK instructions.

Field. $\mathcal{F}_{\mathsf{G\text{-}ZKCPU}}$ requires fixing a field $\mathbb{F}$. To achieve a sufficiently small statistical soundness error (e.g., 40 bits), the field must have size at least 2^{40} (see Theorem 4.2), which already suffices to embed 32-bit machine words and RAM values accessed via the KVS.

In ZEE200, we instantiate $\mathbb{F}$ as the 61-bit Mersenne prime field $\mathbb{F}_{2^{61}-1}$. This provides sufficient headroom to support lookup-table key spaces (recall, our KVS is addressed by keys from $\mathbb{F}$) and all intermediate values arising from the field-level realization of the extended ZEE assembly IS, while preserving efficient field arithmetic. Larger prime fields are also compatible with our design; we adopt this choice as a balance between security margin, compilation simplicity, and backend efficiency.

Registers. The ZEE assembly program operates over a fixed set of architectural registers, including program counter, each storing a 32-bit machine word; see Section 2.2. In ZEE200, these architectural

registers are mapped directly to the register state of $\mathcal{F}_{\text{G-ZKCPU}}$. Accordingly, we set the register dimension m to be exactly the number of architectural registers defined by the ZEE IS.

We note that the ZEE frontend compiler may introduce additional temporary registers when lowering unsupported instructions. Because ZEE200 natively supports a richer assembly IS, such compiler-introduced temporaries are unnecessary and are therefore not included in the register state tracked by $\mathcal{F}_{\text{G-ZKCPU}}$.

For technical reasons (see the discussion in [YHH+24]), the (generalized) tight ZK CPU execution model assumes the availability of a wire encoding the constant value $1 \in \mathbb{F}$, which is carried across instruction executions in the same manner as the register state. This constant wire is handled at the protocol level and is already accounted for in the instruction-size and leakage definitions; it is therefore not included in the register dimension m. For intuition, it is nevertheless convenient to imagine the presence of a distinguished constant register, denoted %one, whose value is fixed to 1 throughout execution. This register is conceptual only and does not form part of the architectural register state tracked by $\mathcal{F}_{\text{G-ZKCPU}}$.

Finally, we note that the register dimension m directly affects performance, since enforcing register consistency across instructions incurs field multiplication cost. See Section 5.3 for optimizations that reduce this overhead by storing registers in the key-value store.

Restricted Auxiliary Input Domain. The domain set $\subseteq \mathbb{F}$ in $\mathcal{F}_{\text{G-ZKCPU}}$ is fixed globally for all instructions as $[0, 2^{16})$, as motivated and discussed in Section 3.2.

Initial KVS State. $\mathcal{F}_{\text{G-ZKCPU}}$ maintains a global key-value store $\mathcal{M} : \mathbb{F} \mapsto \mathbb{F}$ that persists across the entire execution and is shared by all instructions. As part of the public setup, the KVS $\mathcal{M}^{(0)}$ is set up with a desired domain (a subset of $\mathbb{F}$) and initialized.

The primary use of the KVS is to realize RAM accessed by the assembly program via instructions such as LOAD, STORE, LOADB, and LOADH. The ZEE assembly IS employs a 32-bit byte-addressed memory model. Thus, ZEE200 reserves the first 2^{32} keys of the KVS as the addressable memory space.

As part of the public setup, $\mathcal{P}$ and $\mathcal{V}$ agree on the size of the RAM actually used by the program. Only the corresponding subset of these 2^{32} keys is assumed to be defined in the initial KVS state $\mathcal{M}^{(0)}$, while accesses outside this range are treated as invalid by $\mathcal{F}_{\text{G-ZKCPU}}$.

For efficiency, memory is stored in the KVS at word granularity. Specifically, only addresses aligned with 32-bit machine words — whose two least significant bits are zero — are initialized (to 0 or public raw data specified by the assembly program) in $\mathcal{M}^{(0)}$, with each such key storing a single 32-bit word embedded into $\mathbb{F}$. All other keys in this range are intentionally left undefined. The ZEE frontend compiler ensures that word-level memory accesses occur only at word-aligned addresses.[4] Sub-word memory accesses are supported by first retrieving the containing 32-bit word via a single KVS access and then extracting the requested byte or halfword.

In addition to RAM, ZEE200 initializes lookup tables in the KVS to efficiently realize certain assembly-level operations. Each table occupies a *disjoint, publicly agreed* key range determined by a *fixed* offset, and all table entries are populated in the initial state $\mathcal{M}^{(0)}$.

Lookup tables are intended to be *read-only*. This is enforced by the construction of the generalized tight ZK CPU instructions (see Section 5.2.2 and Appendix 5.2.5): any attempt by a malicious prover to modify a table entry would cause a rejection by $\mathcal{F}_{\text{G-ZKCPU}}$. We formally prove this in Appendix D.

ZEE200 includes the following lookup tables:

- **XOR table.** For a fixed offset $\text{offset}_{\text{XOR}}$ (e.g., 2^{40}),

$$\mathcal{M}\left[\text{offset}_{\text{XOR}} + 2^8 a + b\right] = a \oplus b \quad \text{for all } a, b \in [0, 2^8).$$

[4] Even if this alignment condition were violated, any access to an undefined KVS key would be rejected by $\mathcal{F}_{\text{G-ZKCPU}}$ (see Figure 3), resulting in a rejected proof.

- **AND table.** For a fixed offset $\mathsf{offset}_{\mathsf{AND}}$ (e.g., 2^{41}),

$$\mathcal{M}\big[\mathsf{offset}_{\mathsf{AND}} + 2^8 a + b\big] = a \wedge b \quad \text{for all } a, b \in [0, 2^8).$$

- **OR table.** For a fixed offset $\mathsf{offset}_{\mathsf{OR}}$ (e.g., 2^{42}),

$$\mathcal{M}\big[\mathsf{offset}_{\mathsf{OR}} + 2^8 a + b\big] = a \vee b \quad \text{for all } a, b \in [0, 2^8).$$

- **Power-of-two table.** For a fixed offset $\mathsf{offset}_{\mathsf{POW}}$ (e.g., 2^{43}),

$$\mathcal{M}\big[\mathsf{offset}_{\mathsf{POW}} + a\big] = 2^a \quad \text{for all } a \in [0, 32).$$

- **Low-2-bit table.** For a fixed offset $\mathsf{offset}_{\mathsf{LO2}}$ (e.g., 2^{44}),

$$\mathcal{M}\big[\mathsf{offset}_{\mathsf{LO2}} + a\big] = a \bmod 4 \quad \text{for all } a \in [0, 2^{16}).$$

- **High-8-bit table.** For a fixed offset $\mathsf{offset}_{\mathsf{HIGH8}}$ (e.g., 2^{45}),

$$\mathcal{M}\big[\mathsf{offset}_{\mathsf{HIGH8}} + 2^8 a + b\big] = a \quad \text{for all } a, b \in [0, 2^8).$$

We remark that the offsets of all lookup tables are chosen so that their key ranges are pairwise disjoint and, *importantly*, disjoint from the RAM region. In particular, lookup table keys lie outside the 32-bit address space reserved for program memory. As a result, any memory address arising from a well-formed execution of the ZEE assembly IS cannot collide with lookup table entries.

Moreover, each GTZK instruction hard-codes the specific lookup table offset — and hence the key range — it is allowed to access. This enforces isolation between tables: for example, an instruction implementing `XOR` can only issue KVS accesses within the XOR table range, and cannot access the AND or OR table ranges. Whenever a lookup table is accessed, the instruction's constraint system ensures that the computed table index lies in the intended domain, so that all table accesses are confined to keys initialized in $\mathcal{M}^{(0)}$.

$\mathcal{F}_{\mathsf{G\text{-}ZKCPU}}$ incurs an additive cost proportional to the number of initialized key-value pairs, i.e., $|\mathcal{M}|$ (see Theorem 4.2). Our initial KVS state therefore balances expressiveness with efficiency.

5.2.2 Instruction Specification

We now show how ZEE`200` derives the public specification of GTZK CPU instructions from a given flat ZEE assembly program, under the global execution parameters fixed in Section 5.2.1. The resulting instruction specifications are computed at set up time and known to $\mathcal{P}$ and $\mathcal{V}$.

Basic Blocks. ZEE`200` partitions the flat ZEE assembly program into basic blocks in the standard (non-cryptographic) sense: maximal straight-line sequences of instructions with a single entry point and no internal control flow. Control-flow instructions (e.g., conditional branches and jumps) terminate a basic block.

Each basic block forms a *single* GTZK CPU instruction (per Definition 4.1). It is specified by a single constraint system over $\mathbb{F}$, whose arithmetic constraints, restricted-set inputs, designated zero wires, and embedded kvs gates jointly enforce the semantics of the entire sequence of assembly instructions within the block.

Gadget 1 $\mathsf{ADD}(\mathsf{src}_1, \mathsf{src}_2)$:

Require: $\mathsf{src}_1, \mathsf{src}_2 \in \mathbb{F}$ (intended to encode 32-bit words)
Ensure: $\mathsf{dst} \in \mathbb{F}$ (low 32 bits of $\mathsf{src}_1 + \mathsf{src}_2$)
Cost: 1 input, 1 $\mathbb{F}$ mult., 2 set, 0 kvs

1: $c := \mathsf{input}()$, $\mathsf{checkzero}(c(c-1))$ ▷ carry bit $c \in \{0, 1\}$
2: $\mathsf{hi} := \mathsf{set}()$, $\mathsf{lo} := \mathsf{set}()$ ▷ 16-bit limbs
3: $\mathsf{checkzero}(\mathsf{src}_1 + \mathsf{src}_2 - 2^{32}c - 2^{16}\mathsf{hi} - \mathsf{lo})$
4: $\mathsf{dst} := 2^{16}\mathsf{hi} + \mathsf{lo}$
5: **return** dst

This constraint system is obtained by composing, in program order, circuits corresponding to the composed assembly instructions (see Appendix 5.2.5). Register values are represented by circuit wires, naturally connected according to the composition.

Control-Flow Enforcement. Control flow is enforced entirely within the instruction circuit induced by each basic block. The program counter (%pc) is treated as part of the architectural register state, and the circuit includes a designated zero constraint that enforces that the input %pc equals the byte address of the basic block being executed. The circuit then computes and enforces the successor %pc value according to the control-flow instruction terminating the block (e.g., BNZ, JMP). This computation subsumes the IS-defined fall-through behavior (i.e., execute next assembly instruction) and other %pc-update semantics, which are encoded directly in the circuit. In particular, the %pc is updated only once at the end, and no intermediate %pc updates occur within the block.

5.2.3 Initial and Final Register States

An instance of $\mathcal{F}_{\mathsf{G\text{-}ZKCPU}}$ additionally specifies an initial and a final register state, $\mathbf{st}^{(0)}, \mathbf{st}^{(\mathsf{final})} \in \mathbb{F}^m$ (cf. Figure 3), encoding the ZEE architectural registers (including %pc). By default, ZEE200 treats these endpoint states as public instance parameters (e.g., fixed initialization and a public output convention). Optionally, either endpoint state may be kept private by having $\mathcal{P}$ commit to it (and ZEE200 will thus prove that the execution connects the committed endpoints).

5.2.4 Universal Unit Profile

$\mathcal{F}_{\mathsf{G\text{-}ZKCPU}}$ is also parameterized by the universal-unit profile $(u_{\mathsf{in}}, u_{\mathsf{set}}, u_{\mathsf{kvs}}, u_{\times})$, which determines how different non-linear resource types are packed into universal units, impacting performance, VM operation and cost accounting. In ZEE200, each GTZK CPU instruction corresponds to an entire basic block.

By default, ZEE200 uses the uniform profile $(1, 1, 1, 1)$, which yields a simple and robust cost model and is assumed throughout this work. More generally, ZEE200 exposes an interface to fix the profile as a public setup parameter of $\mathcal{F}_{\mathsf{G\text{-}ZKCPU}}$. Since the induced instruction set (i.e., the collection of basic-block circuits) is public, alternative profiles *may* be chosen at setup time to better match the resource mix of the program and reduce padding overhead.

5.2.5 Representative Assembly Instruction Gadgets

We now describe representative gadgets implementing ZEE semantics used in GTZK. Each gadget corresponds to a single assembly instruction and specifies a fixed collection of arithmetic constraints, auxiliary

Gadget 2 $\mathsf{MUL}(\mathsf{src}_1, \mathsf{src}_2)$:

Require: $\mathsf{src}_1, \mathsf{src}_2 \in \mathbb{F}$ (intended to encode 32-bit words)
Ensure: $\mathsf{dst} \in \mathbb{F}$ (low 32 bits of $\mathsf{src}_1 \cdot \mathsf{src}_2$)
Cost: 1 input, 3 $\mathbb{F}$ mult., 7 set, 0 kvs

1: $\mathsf{hi}_1 := \mathsf{set}(),\ \mathsf{lo}_1 := \mathsf{set}()$, $\mathsf{checkzero}(\mathsf{src}_1 - 2^{16}\mathsf{hi}_1 - \mathsf{lo}_1)$
2: $\mathsf{hi}_2 := \mathsf{set}(),\ \mathsf{lo}_2 := \mathsf{set}()$, $\mathsf{checkzero}(\mathsf{src}_2 - 2^{16}\mathsf{hi}_2 - \mathsf{lo}_2)$
3: $t_0 := \mathsf{lo}_1 \cdot \mathsf{src}_2, t_1 := \mathsf{hi}_1 \cdot \mathsf{lo}_2, t := t_0 + 2^{16}t_1$ ▷ $t < 2^{49}$
4: $c := \mathsf{input}()$, $\mathsf{checkzero}(c(c-1))$ ▷ carry bit $c \in \{0, 1\}$
5: $u_2 := \mathsf{set}(),\ u_1 := \mathsf{set}(),\ u_0 := \mathsf{set}()$ ▷ 16-bit limbs
6: $\mathsf{checkzero}(t - 2^{48}c - 2^{32}u_2 - 2^{16}u_1 - u_0)$
7: $\mathsf{dst} := 2^{16}u_1 + u_0$
8: **return** dst

Gadget 3 $\mathsf{EQZ}(\mathsf{src})$:

Require: $\mathsf{src} \in \mathbb{F}$ (intended to encode a 32-bit word)
Ensure: $\mathsf{dst} \in \mathbb{F}$ ($\mathsf{dst} = 1$ iff $\mathsf{src} = 0$, else $\mathsf{dst} = 0$)
Cost: 2 input, 3 $\mathbb{F}$ mult., 0 set, 0 kvs

1: $z := \mathsf{input}()$, $\mathsf{checkzero}(z(z-1))$ ▷ $z \in \{0, 1\}$
2: $w := \mathsf{input}()$ ▷ witness for inverse when $\mathsf{src} \neq 0$
3: $\mathsf{checkzero}((1-z) - \mathsf{src} \cdot w)$, $\mathsf{checkzero}(z \cdot \mathsf{src})$
4: $\mathsf{dst} := z$
5: **return** dst

inputs, designated zero wires, and kvs gates. These gadgets serve as building blocks for constructing the constraint systems defined in Section 5.2.2.

In this section, we present only a representative subset of (simpler and further simplified) gadgets; a complete specification of all (other) gadgets appears in Appendix C.

We provide a formal compositional framework and prove correctness (and security) of our gadget constructions in Appendix D. Our formalism allows easy additions of new assembly instructions with corresponding gadgets. Our proofs cover all corner cases.

Immediate and Constant Operands in Gadgets. The (extended) ZEE assembly IS allows instructions to take immediate (constant) operands. In ZEE200, such immediates can be represented using the conceptual constant register %one described above: a public constant $c \in \mathbb{Z}_{2^{32}}$ is embedded as a fixed field element and introduced into the instruction gadget via wiring from the constant-1 wire (for free). For clarity of presentation, we therefore do not distinguish immediate operands from register operands at the level of the gadget descriptions below. Because immediates are publicly known at setup time, ZEE200 exploits this to apply various constant-specific optimizations for certain instructions; see Section 5.3.

Gadget Description and Cost Model. We specify each gadget using pseudocode over the GTZK CPU, explicitly indicating the arithmetic constraints, unrestricted auxiliary inputs, restricted-set inputs, designated zero checks, and kvs gates it introduces. In this model, $\mathcal{P}$ supplies values for both unrestricted auxiliary inputs (e.g., carry bits) and restricted-set inputs (e.g., bounded limbs), while correctness is enforced by the accompanying zero constraints.

For each gadget, we annotate the resources it uses, which determines the instruction size via the universal-unit cost model (cf. Equation (1)). Accordingly, gadget design is guided not by minimizing any

Gadget 4 LOAD(src):

Require: src $\in \mathbb{F}$
Ensure: dst $\in \mathbb{F}$ (the 32-bit word stored at address src)
Cost: 1 input, 0 $\mathbb{F}$ mult., 0 set, 1 kvs

1: new := input() ▷ proposed new value
2: dst := kvs(src, new) ▷ returns old value
3: checkzero(new − dst) ▷ enforce read-only access
4: **return** dst

Gadget 5 STORE($\mathsf{src}_1, \mathsf{src}_2$):

Require: $\mathsf{src}_1, \mathsf{src}_2 \in \mathbb{F}$ ▷ src_1: 32-bit address, src_2: 32-bit word
Ensure: dst $\in \mathbb{F}$ (unused; set to 0)
Cost: 0 input, 0 $\mathbb{F}$ mult., 0 set, 1 kvs

1: kvs($\mathsf{src}_1, \mathsf{src}_2$) ▷ write src_2 to address src_1
2: dst := 0
3: **return** dst

single resource in isolation, but by balancing different types of non-linear constraints so as to minimize the number of universal units required — e.g., preferring additional restricted-set inputs over extra kvs gates, or vice versa, depending on which constraint type dominates for a particular ZEE instruction.

Pseudocode Notation. input() denotes a $\mathcal{P}$-provided unrestricted auxiliary input; set() denotes a $\mathcal{P}$-provided auxiliary input drawn from the restricted domain set; and kvs(idx, $\mathsf{val}_{\mathsf{new}}$) outputs the previous value $\mathcal{M}[\mathsf{idx}]$ and then sets $\mathcal{M}[\mathsf{idx}] := \mathsf{val}_{\mathsf{new}}$. Arithmetic operations are in $\mathbb{F}$, and checkzero(x) enforces that $x = 0$.

Arithmetic Gadgets. Arithmetic gadgets illustrate core techniques, including limb decomposition (i.e. decomposition of a larger word into several chunks), carry handling, and constraint-based enforcement of word-level semantics. We include here ADD (Gadget 1), MUL (Gadget 2) and EQZ (Gadget 3) as representative gadgets.

For each gadget, $\mathcal{P}$ supplies auxiliary inputs (e.g., carry bits, bounded limbs, or inverse elements). Zero constraints ensure that there exists a decomposition consistent with the intended 32-bit operation. Any accepting assignment therefore induces values that agree with the corresponding semantics in $\mathbb{Z}_{2^{32}}$, while any deviation from these semantics violates at least one enforced constraint.

Load and Store Gadgets. Load and store gadgets illustrate how ZEE200 realizes reads and writes to RAM via kvs gates, enforcing consistency of memory accesses across execution. We include LOAD (Gadget 4) and STORE (Gadget 5) as representative gadgets.

Correctness and soundness follow directly from the semantics of the kvs gate together with the enforced zero constraints.

Bitwise Logic via Lookup Tables. Unlike arithmetic instructions, bitwise operations such as XOR do not admit a compact representation using low-degree arithmetic constraints over $\mathbb{F}$. Instead, ZEE200 realizes these operations using limb decomposition and fixed lookup tables stored in $\mathcal{M}$, as depicted in Gadget 6.

Gadget 6 $\texttt{XOR}(\mathsf{src}_1, \mathsf{src}_2)$:

Require: $\mathsf{src}_1, \mathsf{src}_2 \in \mathbb{F}$ (intended to encode 32-bit words)
Ensure: $\mathsf{dst} \in \mathbb{F}$ (the bitwise XOR of src_1 and src_2)
Cost: 8 input, 0 $\mathbb{F}$ mult., 4 set, 8 kvs

1: $\mathsf{lo}_1 := \mathsf{set}(),\ \mathsf{hi}_1 := \mathsf{set}(), \mathsf{checkzero}(\mathsf{src}_1 - 2^{16}\mathsf{hi}_1 - \mathsf{lo}_1)$
2: $u_1 := \texttt{LOAD}(\mathsf{offset}_{\mathsf{HIGH8}} + \mathsf{lo}_1), u_0 := \mathsf{lo}_1 - 2^8 u_1$
3: $u_3 := \texttt{LOAD}(\mathsf{offset}_{\mathsf{HIGH8}} + \mathsf{hi}_1), u_2 := \mathsf{hi}_1 - 2^8 u_3$
4: $\mathsf{lo}_2 := \mathsf{set}(),\ \mathsf{hi}_2 := \mathsf{set}(), \mathsf{checkzero}(\mathsf{src}_2 - 2^{16}\mathsf{hi}_2 - \mathsf{lo}_2)$
5: $v_1 := \texttt{LOAD}(\mathsf{offset}_{\mathsf{HIGH8}} + \mathsf{lo}_2), v_0 := \mathsf{lo}_2 - 2^8 v_1$
6: $v_3 := \texttt{LOAD}(\mathsf{offset}_{\mathsf{HIGH8}} + \mathsf{hi}_2), v_2 := \mathsf{hi}_2 - 2^8 v_3$
7: $w_0 := \texttt{LOAD}(\mathsf{offset}_{\mathsf{XOR}} + 2^8 u_0 + v_0)$
8: $w_1 := \texttt{LOAD}(\mathsf{offset}_{\mathsf{XOR}} + 2^8 u_1 + v_1)$
9: $w_2 := \texttt{LOAD}(\mathsf{offset}_{\mathsf{XOR}} + 2^8 u_2 + v_2)$
10: $w_3 := \texttt{LOAD}(\mathsf{offset}_{\mathsf{XOR}} + 2^8 u_3 + v_3)$
11: $\mathsf{dst} := 2^{24} w_3 + 2^{16} w_2 + 2^8 w_1 + w_0$
12: **return** dst

5.3 Selected Optimizations

Maintaining Registers in the Key-Value Store. In the GTZK framework, each architectural register is represented by a circuit wire and incurs one multiplication constraint. As a result, basic blocks that touch only a few registers still incur cost proportional to the total number of registers. A simple optimization is to store registers in the KVS: read only the registers required by the block at entry, and write back the updated registers at exit.

Handling Immediate and Constant Operands. When an operand is a public constant, the corresponding gadget can often be simplified. We illustrate this with two examples. For assembly instruction such as MUL, if one operand is constant, the decomposition into two 16-bit limbs is free, and the corresponding field multiplications (with these limbs) reduce to linear operations. More substantial savings arise for bitwise instructions. Recall that bitwise operations decompose each operand into four 8-bit chunks followed by table lookups. When one operand is a constant, this decomposition is again free; moreover, if a constant chunk is all zeros or all ones, the corresponding bitwise computation reduces to a linear transformation (i.e., no lookup).

Utilizing Invalid Key Space. We can optimize word decomposition by exploiting unused key space in the KVS. Consider decomposing a 32-bit word x into four 8-bit chunks. The straightforward construction uses two set operations (together with a zero constraint) to extract 16-bit chunks, followed by two kvs lookups on the HIGH8 table (see Gadget 6). We can eliminate one set operation as follows. We first use a single set to obtain the high 16-bit value hi, and apply one kvs lookup to split hi into two 8-bit chunks. For the lower two chunks, we perform a *single* kvs lookup at key $\mathsf{offset}_{\mathsf{HIGH8}} + (x - 2^{16} \cdot \mathsf{hi})$. By placing the HIGH8 table in the contiguous key interval $[\mathsf{offset}_{\mathsf{HIGH8}}, \mathsf{offset}_{\mathsf{HIGH8}} + 2^{16})$ and leaving the surrounding keys unused, any successful lookup implies that the query key lies in this valid range, which in turn enforces that hi is consistent with x. This removes one set from the decomposition.

Saving an Extra Constraint in LOAD. In Gadget 4 for LOAD, $\mathcal{P}$ supplies an explicit input value x, which is enforced to equal the value stored at the accessed key. At the protocol level of GTZK CPU, this input is redundant: $\mathcal{P}$ commits to the old value $\mathsf{val}_{\mathsf{old}}$ and the returned value $\mathsf{val}_{\mathsf{new}}$, whose consistency with the

memory state is enforced by the ZK KVS. The additional input x is then constrained to satisfy $x = \mathsf{val}_{\mathsf{old}}$ and $x = \mathsf{val}_{\mathsf{new}}$, incurring an extra constraint. We can directly impose the single constraint $\mathsf{val}_{\mathsf{new}} = \mathsf{val}_{\mathsf{old}}$, eliminating the unnecessary input. Note, this optimization is protocol-level and cannot be captured by a local gadget redefinition.

6 Evaluation

Implementation. We implemented ZEE200 in C++ with 9 KLOC, measured by `cloc`, excluding the reused ZEE compiler frontend. Our implementation targets ≈40 bits of statistical security and over 100 bits of computational security. Increasing statistical security in ZEE200 is straightforward by instantiating the protocol over a larger prime field; this increases field-operation costs, but can also simplify the compilation of ZEE instructions into GTZK instructions. We leave the exploration of this trade-off to future work. We instantiate the $\mathcal{F}_{\mathsf{CPZK}}$ functionality using VOLE-based ZK over the prime field $\mathbb{F}_{2^{61}-1}$ via QuickSilver [YSWW21], as implemented in the EMP toolkit [WMK16], which we use in an almost black-box manner.

We remark that our implementation is intentionally modular. In particular, (1) the GTZK protocol is implemented via simple and well-defined interfaces, enabling future exploration of, e.g., alternative universal-unit profiles, and (2) the compilation pipeline from ZEE assembly to GTZK instructions is easily extensible to support additional assembly instructions.

Code Availability. Our open-source implementation is available at https://github.com/ainta/ZEE200.

Experimental Setup. Unless otherwise stated, we evaluate ZEE200 (and all baselines) on a 2021 ThinkPad X1 Carbon Gen 9 in a single-threaded configuration.[5] Both $\mathcal{P}$ and $\mathcal{V}$ run on the same machine and communicate via localhost; this setup is used solely for benchmarking purposes. Unless otherwise stated, we simulate a LAN setting with 1 Gbps bandwidth and 2 ms latency, following ZEE [HYDK21]. Unless otherwise stated, all experiments use the uniform universal-unit profile $(1, 1, 1, 1)$ for the underlying GTZK protocol.

Benchmarks. We reuse ZEE benchmarks [HYDK21], including: (i) the SIR-reported `sed` vulnerability [DER05], where $\mathcal{P}$ proves knowledge of an input that triggers the bug, and (ii) the CVE-reported `gzip` vulnerability [CVE05], where $\mathcal{P}$ similarly proves knowledge of a bug-triggering input. In addition, we include the following benchmarks:

1. `gzip` decompression, where $\mathcal{P}$ proves knowledge of a compressed `.gz` file whose decompression succeeds,
2. a recursive Fibonacci program, where $\mathcal{P}$ proves knowledge of an input n corresponding to the n-th Fibonacci number,
3. a mergesort program, where $\mathcal{P}$ proves correct execution of mergesort on a private n-length input array,
4. a SHA-256 program, where $\mathcal{P}$ proves knowledge of an input that maps to a specified digest after n iterations.

In all benchmarks, the parties agree *only* on the C program, which is compiled to (extended) ZEE assembly using the frontend of [HYDK21].

Metrics. Our primary metric is end-to-end execution time, defined as the time from proof initiation by $\mathcal{P}$ to acceptance by $\mathcal{V}$. We define the CPU speed as the number of executed ZEE assembly instructions divided by the end-to-end execution time. Since ZEE200 uses the extended ZEE assembly, instruction counts are measured with respect to this extended IS (Figure 1 and Figure 4).

[5] 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80 GHz and 16 GB physical RAM

Benchmark	Input	# ExZEE Instructions	ZK RAM Size (bytes)	ZEE200 Time (s)	ZEE200 Speed (KHz)	ZEE200 Comm.	ZEE Time (s)	Speed Up
sed	vuln payload	124.2 K	2^{15}	1.5	82.3	53MB	30.1	20×
fib	$n = 20$	481.1 K	2^{13}	2.7	172.5	136MB	44.5	17×
fib	$n = 23$	2.0 M	2^{13}	8.9	226.3	532MB	–	–
mergesort	$n = 500$	375.2 K	2^{17}	2.6	142.7	123MB	43.9	17×
mergesort	$n = 5000$	4.8 M	2^{17}	24.5	199.8	1.4GB	–	–
gzip (bug trigger)	vuln payload	18.8 K	2^{19}	1.1	16.8	21MB	5.2	5×
gzip (decompress)	22KB img gz	3.1 M	2^{19}	17.3	182.1	1.0 GB	–	–
sha256	10 blocks	152.4 K	2^{13}	1.6	91.3	68MB	36.1	23×
sha256	200 blocks	2.9 M	2^{13}	19.7	148.7	1.1GB	–	–

Table 1: End-to-end time and speed of ZEE200. We also report the communication cost of ZEE200, end-to-end time of the baseline ZEE and its slowdown relative to ZEE200 on the same benchmarks. For each benchmark, the ZK RAM size is set to the minimum required to support execution and is identical for ZEE200 and ZEE. Each data point is averaged over 8 runs (out of 10 total, excluding the fastest and slowest runs). Note, ZEE cannot execute some benchmarks — including those with the largest observed speedups (up to $54\times$) — due to the ThinkPad's limited RAM; these cases are marked by "–," and we rerun them for ZEE (and ZEE200) on an AWS server (Table 2).

6.1 ZEE200 Overall Performance

Table 1 reports the overall performance of ZEE200 on our benchmarks, compared against the baseline ZEE. We remark:

- For each benchmark, we configure the ZK RAM size to the minimum required to support assembly execution; this configuration is identical for ZEE200 and ZEE.
- ZEE200 incurs a one-time setup cost (e.g., initializing the ZK KVS), and hence achieves lower CPU speeds on short-running benchmarks — 16.8 KHz on the gzip bug trigger, and 82.3 KHz and 91.3 KHz on the sed and sha256 ($n = 10$) benchmarks, respectively. To explicate this fixed cost, we measure the execution time of an empty program (i.e., no instructions) with 2^{13}-byte ZK RAM, which runs in 0.9s. Still, even for short executions, ZEE200 achieves large speedups compared to ZEE.
- For sufficiently long executions, ZEE200 achieves a steady-state CPU speed of approximately 200 KHz. When measured in terms of non-extended ZEE assembly instructions, the effective speed is even higher. For example, the gzip decompression benchmark executes 8.9 M non-extended ZEE instructions, yielding an effective rate of 511 KHz. This explains our naming of ZEE200.
- ZEE cannot execute some benchmarks due to the limited physical RAM available on the ThinkPad (16 GB). This indicates that ZEE200 not only outperforms ZEE in efficiency, but also has a smaller physical memory footprint. Additional benchmarks and discussion are provided in Section 6.2.
  For a better performance comparison, we rerun the experiments in Table 1 on an AWS r6i.8xlarge instance — which has comparable CPU performance but significantly more physical RAM (256 GB) — allowing ZEE to complete these benchmarks. The result is shown in Table 2. On these larger benchmarks, ZEE200 outperforms ZEE by up to $54\times$.

Communication Cost. While reducing communication is not our main focus, we report ZEE200's communication in Table 1. Overall, ZEE200 incurs 250-400 bytes of communication per extended ZEE instruction. Table 3 reports ZEE200's performance under different network conditions, indicating room for performance improvement by multi-threading.

Benchmark	ZEE200		ZEE	Speed Up
	Time (s)	Speed (KHz)	Time (s)	
`sed` (bug trigger)	2.24	55.5	52.6	23×
`fib` ($n = 20$)	4.0	120.3	77.8	19×
`fib` ($n = 23$)	11.8	171.3	324.6	28×
`mergesort` ($n = 500$)	3.8	98.7	77.3	20×
`mergesort` ($n = 5000$)	32.7	149.7	1003.1	31×
`gzip` (bug trigger)	1.7	11.0	9.15	5×
`gzip` (decompress)	23.4	135.0	1262.1	54×
`sha256` (10 blocks)	2.5	61.0	62.8	25×
`sha256` (200 blocks)	25.6	114.9	1210.4	47×

Table 2: ZEE200 vs. ZEE on an AWS server. We rerun the benchmarks from Table 1 on an AWS EC2 server (`r6i.8xlarge` with 256 GB RAM) to provide sufficient physical RAM for ZEE. The experimental setup is identical to that of Table 1. ZEE200 does not require large physical RAM for these benchmarks; the extra memory is needed only for the ZEE baseline.

Bandwidth	Latency (ms)		
	2	50	100
100 Mbps	11.3 s	13.5 s	16.2 s
500 Mbps	3.5 s	5.8 s	9.6 s
1 Gbps	2.6 s	5.5 s	9.5 s

Table 3: End-to-end time under varying network conditions. Results are obtained using the `mergesort` ($n = 500$) benchmark. Each data point is averaged over 8 runs (out of 10 total, excluding the fastest and slowest runs).

6.2 ZEE200 with Varying ZK RAM Sizes

As discussed in Section 6.1, ZEE200 requires less physical RAM per party than ZEE. This is primarily because ZEE200 instantiates its ZK RAM using a constant-overhead construction [YH24], rather than the logarithmic-overhead construction used in ZEE. We measure the parties' RAM usage while executing an empty program under varying ZK RAM configurations, as shown in Table 4.

Note, ZEE200 supports larger ZK RAM sizes with little additional overhead. This is because, in ZEE200, parties incur cost linear in the RAM size. For example, with 2^{25}-byte ZK RAM, ZEE200 requires 22.0 seconds to execute the `sha256` ($n = 200$) benchmark, compared to 19.7 seconds with 2^{13}-byte ZK RAM.

6.3 Comparison with zkVM

Recall from Section 1.2 that zkVMs target a design point distinct from ZEE200. In particular, many zkVMs prioritize public verifiability and succinctness for blockchain-oriented applications, and do *not* provide zero knowledge by default — often leaking the execution path to $\mathcal{V}$. While generic zk-wrapping techniques can be applied to recover full ZK, they incur significant additional overhead. As a complementary point of reference, we evaluate two widely used zkVM implementations, SP1 and Jolt (latest versions), on a subset of our benchmark kernels.

ZK RAM Size (bytes)	2^{19}	2^{20}	2^{21}	2^{22}	2^{23}	2^{24}	2^{25}	2^{26}	2^{27}
ZEE200 (GB)	0.8	0.8	0.9	1.1	1.5	2.2	3.6	6.5	12.1
ZEE (GB)	0.5	0.6	0.9	1.9	4.1	7.9	–	–	–

Table 4: Physical RAM usage under varying ZK RAM size. We run an empty program on ZEE200 and ZEE while varying the ZK RAM size, and report total physical RAM summed over both parties (about half per party). ZEE succeeds up to 2^{24} but is killed by the OS at 2^{25}, whereas ZEE200 scales to 2^{27} and only fails at 2^{28} due to the 16 GB physical memory limit. Note that ZEE200 uses more RAM at small sizes due to lookup-table support.

Benchmark	SP1 (STARK)†	SP1 (Groth16-wrapped)	Jolt (default)†
`fib`, $n = 20$	141 (50×)	2220 (796×)	13 (5×)
`fib`, $n = 23$	251 (28×)	2501 (280×)	37 (4×)
`mergesort`, $n = 500$	145 (55×)	2212 (841×)	27 (10×)
`mergesort`, $n = 5000$	808 (33×)	3821 (156×)	238 (10×)
`sha256`, 10 blocks	115 (70×)	2193 (1313×)	26 (16×)
`sha256`, 200 blocks	926 (47×)	3531 (179×)	195 (10×)

Table 5: zkVM prover computation time. Entries report prover runtime in seconds; the parenthetical factor is the slowdown relative to ZEE200. † indicates the reported mode does not provide zero knowledge; Groth16-wrapped includes wrapper overhead.

Table 5 reports single-threaded prover times for these zkVMs on the same ThinkPad as ZEE200. For zkVMs, prover computation time dominates the end-to-end cost due to the succinctness property. For SP1, we report results for both the native STARK prover (which does not provide ZK) and the Groth16 zk-wrapped prover (which provides ZK). For Jolt, we report its default proving mode, which is succinct but does not provide ZK; to the best of our knowledge, a zk-wrapper is not currently available in Jolt.

Across all benchmarks, ZEE200 achieves orders-of-magnitude lower end-to-end time than zk-wrapped SP1, and $28-70\times$ and $4-16\times$ improvements over non-ZK configurations of SP1 and Jolt, respectively. We emphasize that ZEE200 and zkVMs ultimately target different application domains, and this comparison is intended only as a coarse performance reference rather than an apple-to-apple comparison. More direct comparisons between zkVMs and ZEE200 (e.g., via alternative $\mathcal{F}_{\mathsf{CPZK}}$ instantiations and Fiat-Shamir [FS87]) is an interesting direction for future work.

6.4 ZEE200 with Different Universal Profiles

As discussed in Section 5.2.4, the universal-unit profile has a direct impact on the performance of the GTZK CPU and, consequently, on ZEE200. We view a systematic exploration of this design space as a potential direction for future work. To motivate such exploration, we include two proof-of-concept experiments, shown as follows:

- Using the $(1, 2, 1, 2)$ profile, ZEE200 completes the `fib` ($n = 23$) in 8.5 s (vs. 8.9 s in Table 1).
- Using the $(1, 2, 1, 1)$ profile, ZEE200 completes the `gzip` (decompression) in 16.4 s (vs. 17.3 s in Table 1).

Acknowledgments

This work was supported in part by a Visa research award and NSF awards CNS-2246354 and CCF-2217070.

Statement on AI use. All research ideas, protocols, and proofs in this paper were developed without the use of AI. The authors used ChatGPT and Claude to assist with the implementation of the artifact and to check grammar and polish sentences in the paper. The authors reviewed all AI-assisted changes, verified the implementation and technical results, and take full responsibility for the paper's contents and correctness.

References

[AHIV17] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 2087–2104, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press.

[AST24] Arasu Arun, Srinath T. V. Setty, and Justin Thaler. Jolt: SNARKs for virtual machines via lookups. In Marc Joye and Gregor Leander, editors, *EUROCRYPT 2024, Part VI*, volume 14656 of *LNCS*, pages 3–33, Zurich, Switzerland, May 26–30, 2024. Springer, Cham, Switzerland.

[BBB+18] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy*, pages 315–334, San Francisco, CA, USA, May 21–23, 2018. IEEE Computer Society Press.

[BBMH+21] Carsten Baum, Lennart Braun, Alexander Munch-Hansen, Benoît Razet, and Peter Scholl. Appenzeller to brie: Efficient zero-knowledge proofs for mixed-mode arithmetic and Z2k. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 192–211, Virtual Event, Republic of Korea, November 15–19, 2021. ACM Press.

[BBMHS22] Carsten Baum, Lennart Braun, Alexander Munch-Hansen, and Peter Scholl. Moz$\mathbb{Z}_{2^k}$arella: Efficient vector-OLE and zero-knowledge proofs over $\mathbb{Z}_{2^k}$. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part IV*, volume 13510 of *LNCS*, pages 329–358, Santa Barbara, CA, USA, August 15–18, 2022. Springer, Cham, Switzerland.

[BCC+25] Dung Bui, Haotian Chu, Geoffroy Couteau, Xiao Wang, Chenkai Weng, Kang Yang, and Yu Yu. An efficient ZK compiler from SIMD circuits to general circuits. *Journal of Cryptology*, 38(1):10, January 2025.

[BCG+13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 90–108, Santa Barbara, CA, USA, August 18–22, 2013. Springer Berlin Heidelberg, Germany.

[BCG+14] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from Bitcoin. In

2014 IEEE Symposium on Security and Privacy, pages 459–474, Berkeley, CA, USA, May 18–21, 2014. IEEE Computer Society Press.

[BCG+19] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators: Silent OT extension and more. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 489–518, Santa Barbara, CA, USA, August 18–22, 2019. Springer, Cham, Switzerland.

[BCGI18] Elette Boyle, Geoffroy Couteau, Niv Gilboa, and Yuval Ishai. Compressing vector OLE. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 896–912, Toronto, ON, Canada, October 15–19, 2018. ACM Press.

[BCTV14] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In Kevin Fu and Jaeyeon Jung, editors, *USENIX Security 2014*, pages 781–796, San Diego, CA, USA, August 20–22, 2014. USENIX Association.

[BDOZ11] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In Kenneth G. Paterson, editor, *EUROCRYPT 2011*, volume 6632 of *LNCS*, pages 169–188, Tallinn, Estonia, May 15–19, 2011. Springer Berlin Heidelberg, Germany.

[BDSW23] Carsten Baum, Samuel Dittmer, Peter Scholl, and Xiao Wang. Sok: vector OLE-based zero-knowledge protocols. *DCC*, 91(11):3527–3561, 2023.

[Bea95] Donald Beaver. Precomputing oblivious transfer. In Don Coppersmith, editor, *CRYPTO'95*, volume 963 of *LNCS*, pages 97–109, Santa Barbara, CA, USA, August 27–31, 1995. Springer Berlin Heidelberg, Germany.

[BFKL94] Avrim Blum, Merrick L. Furst, Michael J. Kearns, and Richard J. Lipton. Cryptographic primitives based on hard learning problems. In Douglas R. Stinson, editor, *CRYPTO'93*, volume 773 of *LNCS*, pages 278–291, Santa Barbara, CA, USA, August 22–26, 1994. Springer Berlin Heidelberg, Germany.

[BG+23] Jeremy Bruestle, Paul Gafni, et al. Risc zero zkvm: scalable, transparent arguments of risc-v integrity. *Draft. July*, 29, 2023.

[BMRS21] Carsten Baum, Alex J. Malozemoff, Marc B. Rosen, and Peter Scholl. Mac'n'cheese: Zero-knowledge proofs for boolean and arithmetic circuits with nested disjunctions. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part IV*, volume 12828 of *LNCS*, pages 92–122, Virtual Event, August 16–20, 2021. Springer, Cham, Switzerland.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145, Las Vegas, NV, USA, October 14–17, 2001. IEEE Computer Society Press.

[CDS94] Ronald Cramer, Ivan Damgård, and Berry Schoenmakers. Proofs of partial knowledge and simplified design of witness hiding protocols. In Yvo Desmedt, editor, *CRYPTO'94*, volume 839 of *LNCS*, pages 174–187, Santa Barbara, CA, USA, August 21–25, 1994. Springer Berlin Heidelberg, Germany.

[CGG+24] Arka Rai Choudhuri, Sanjam Garg, Aarushi Goel, Sruthi Sekar, and Rohit Sinha. SublonK: Sublinear prover PlonK. *PoPETs*, 2024(3):314–335, July 2024.

[CHM+20] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Psi Vesely, and Nicholas P. Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part I*, volume 12105 of *LNCS*, pages 738–768, Zagreb, Croatia, May 10–14, 2020. Springer, Cham, Switzerland.

[CLOS02] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. In *34th ACM STOC*, pages 494–503, Montréal, Québec, Canada, May 19–21, 2002. ACM Press.

[CVE05] Common Vulnerabilities and Exposures. CVE-2005-1228. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-1228, 2005.

[DEGL+23] Samuel Dittmer, Karim Eldefrawy, Stéphane Graham-Lengrand, Steve Lu, Rafail Ostrovsky, and Vitor Pereira. Boosting the performance of high-assurance cryptography: Parallel execution and optimizing memory access in formally-verified line-point zero-knowledge. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *ACM CCS 2023*, pages 2098–2112, Copenhagen, Denmark, November 26–30, 2023. ACM Press.

[DER05] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[DILO22] Samuel Dittmer, Yuval Ishai, Steve Lu, and Rafail Ostrovsky. Improving line-point zero knowledge: Two multiplications for the price of one. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 829–841, Los Angeles, CA, USA, November 7–11, 2022. ACM Press.

[DIO21] Samuel Dittmer, Yuval Ishai, and Rafail Ostrovsky. Line-point zero knowledge and its applications. In Stefano Tessaro, editor, *ITC 2021*, volume 199 of *LIPIcs*, pages 5:1–5:24, Seattle, WA, USA, July 23–26, 2021. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

[DOT21] Cyprien Delpech de Saint Guilhem, Emmanuela Orsini, and Titouan Tanguy. Limbo: Efficient zero-knowledge MPCitH-based arguments. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 3022–3036, Virtual Event, Republic of Korea, November 15–19, 2021. ACM Press.

[DOTV22] Cyprien Delpech de Saint Guilhem, Emmanuela Orsini, Titouan Tanguy, and Michiel Verbauwhede. Efficient proof of RAM programs from any public-coin zero-knowledge system. In Clemente Galdi and Stanislaw Jarecki, editors, *SCN 22*, volume 13409 of *LNCS*, pages 615–638, Amalfi, Italy, September 12–14, 2022. Springer, Cham, Switzerland.

[DXNT24] Zijing Di, Lucas Xia, Wilson D. Nguyen, and Nirvan Tyagi. MuxProofs: Succinct arguments for machine computation from vector lookups. In Kai-Min Chung and Yu Sasaki, editors, *ASIACRYPT 2024, Part V*, volume 15488 of *LNCS*, pages 236–265, Kolkata, India, December 9–13, 2024. Springer, Singapore, Singapore.

[FDNZ21] Zhiyong Fang, David Darais, Joseph P. Near, and Yupeng Zhang. Zero knowledge static program analysis. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 2951–2967, Virtual Event, Republic of Korea, November 15–19, 2021. ACM Press.

[FKL+21] Nicholas Franzese, Jonathan Katz, Steve Lu, Rafail Ostrovsky, Xiao Wang, and Chenkai Weng. Constant-overhead zero-knowledge for RAM programs. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 178–191, Virtual Event, Republic of Korea, November 15–19, 2021. ACM Press.

[FS87] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 186–194, Santa Barbara, CA, USA, August 1987. Springer Berlin Heidelberg, Germany.

[GGHAK22] Aarushi Goel, Matthew Green, Mathias Hall-Andersen, and Gabriel Kaptchuk. Stacking sigmas: A framework to compose Σ-protocols for disjunctions. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part II*, volume 13276 of *LNCS*, pages 458–487, Trondheim, Norway, May 30 – June 3, 2022. Springer, Cham, Switzerland.

[GHAK24] Aarushi Goel, Mathias Hall-Andersen, and Gabriel Kaptchuk. Dora: A simple approach to zero-knowledge for RAM programs. In Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie, editors, *ACM CCS 2024*, pages 869–883, Salt Lake City, UT, USA, October 14–18, 2024. ACM Press.

[GHAKS23] Aarushi Goel, Mathias Hall-Andersen, Gabriel Kaptchuk, and Nicholas Spooner. Speed-stacking: Fast sublinear zero-knowledge proofs for disjunctions. In Carmit Hazay and Martijn Stam, editors, *EUROCRYPT 2023, Part II*, volume 14005 of *LNCS*, pages 347–378, Lyon, France, April 23–27, 2023. Springer, Cham, Switzerland.

[GKK+12] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012*, pages 513–524, Raleigh, NC, USA, October 16–18, 2012. ACM Press.

[GMR85] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems (extended abstract). In *17th ACM STOC*, pages 291–304, Providence, RI, USA, May 6–8, 1985. ACM Press.

[HHK+24] Carmit Hazay, David Heath, Vladimir Kolesnikov, Muthuramakrishnan Venkitasubramaniam, and Yibin Yang. LogRobin++: Optimizing proofs of disjunctive statements in VOLE-based ZK. In Kai-Min Chung and Yu Sasaki, editors, *ASIACRYPT 2024, Part V*, volume 15488 of *LNCS*, pages 367–401, Kolkata, India, December 9–13, 2024. Springer, Singapore, Singapore.

[HK20a] David Heath and Vladimir Kolesnikov. A 2.1 KHz zero-knowledge processor with BubbleRAM. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 2055–2074, Virtual Event, USA, November 9–13, 2020. ACM Press.

[HK20b] David Heath and Vladimir Kolesnikov. Stacked garbling for disjunctive zero-knowledge proofs. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part III*, volume 12107 of *LNCS*, pages 569–598, Zagreb, Croatia, May 10–14, 2020. Springer, Cham, Switzerland.

[HY24] Carmit Hazay and Yibin Yang. Toward malicious constant-rate 2PC via arithmetic garbling. In Marc Joye and Gregor Leander, editors, *EUROCRYPT 2024, Part V*, volume 14655 of *LNCS*, pages 401–431, Zurich, Switzerland, May 26–30, 2024. Springer, Cham, Switzerland.

[HYDK21] David Heath, Yibin Yang, David Devecsery, and Vladimir Kolesnikov. Zero knowledge for everything and everyone: Fast ZK processor with cached ORAM for ANSI C programs. In *2021 IEEE Symposium on Security and Privacy*, pages 1538–1556, San Francisco, CA, USA, May 24–27, 2021. IEEE Computer Society Press.

[IKOS07] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In David S. Johnson and Uriel Feige, editors, *39th ACM STOC*, pages 21–30, San Diego, CA, USA, June 11–13, 2007. ACM Press.

[KS22] Abhiram Kothapalli and Srinath Setty. SuperNova: Proving universal machine executions without universal circuits. Cryptology ePrint Archive, Report 2022/1758, 2022.

[KS24] Abhiram Kothapalli and Srinath T. V. Setty. HyperNova: Recursive arguments for customizable constraint systems. In Leonid Reyzin and Douglas Stebila, editors, *CRYPTO 2024, Part X*, volume 14929 of *LNCS*, pages 345–379, Santa Barbara, CA, USA, August 18–22, 2024. Springer, Cham, Switzerland.

[KST22] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. Nova: Recursive zero-knowledge arguments from folding schemes. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part IV*, volume 13510 of *LNCS*, pages 359–388, Santa Barbara, CA, USA, August 15–18, 2022. Springer, Cham, Switzerland.

[LAH+22] Ning Luo, Timos Antonopoulos, William R. Harris, Ruzica Piskac, Eran Tromer, and Xiao Wang. Proving UNSAT in zero knowledge. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 2203–2217, Los Angeles, CA, USA, November 7–11, 2022. ACM Press.

[LHLA25] Mengling Liu, Yang Heng, Xingye Lu, and Man Ho Au. JesseQ: Efficient zero-knowledge proofs for circuits over any field. In Marina Blanton, William Enck, and Cristina Nita-Rotaru, editors, *2025 IEEE Symposium on Security and Privacy*, pages 3402–3420, San Francisco, CA, USA, May 12–15, 2025. IEEE Computer Society Press.

[LXY24a] Fuchun Lin, Chaoping Xing, and Yizhou Yao. Interactive line-point zero-knowledge with sublinear communication and linear computation. In Kai-Min Chung and Yu Sasaki, editors, *ASIACRYPT 2024, Part V*, volume 15488 of *LNCS*, pages 337–366, Kolkata, India, December 9–13, 2024. Springer, Singapore, Singapore.

[LXY24b] Fuchun Lin, Chaoping Xing, and Yizhou Yao. More efficient zero-knowledge protocols over $\mathbb{Z}_{2^k}$ via Galois rings. In Leonid Reyzin and Douglas Stebila, editors, *CRYPTO 2024, Part IX*, volume 14928 of *LNCS*, pages 424–457, Santa Barbara, CA, USA, August 18–22, 2024. Springer, Cham, Switzerland.

[LXZ21] Tianyi Liu, Xiang Xie, and Yupeng Zhang. zkCNN: Zero knowledge proofs for convolutional neural network predictions and accuracy. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 2968–2985, Virtual Event, Republic of Korea, November 15–19, 2021. ACM Press.

[LXZ+24] Tianyi Liu, Tiancheng Xie, Jiaheng Zhang, Dawn Song, and Yupeng Zhang. Pianist: Scalable zkRollups via fully distributed zero-knowledge proofs. In *2024 IEEE Symposium on Security and Privacy*, pages 1777–1793, San Francisco, CA, USA, May 19–23, 2024. IEEE Computer Society Press.

[LZZ+25] Tianyi Liu, Zhenfei Zhang, Yuncong Zhang, Wenqing Hu, and Ye Zhang. Ceno: Non-uniform, segment and parallel zero-knowledge virtual machine. *J. Cryptol.*, 38(2):17, 2025.

[MBKM19] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge SNARKs from linear-size universal and updatable structured reference strings. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 2111–2128, London, UK, November 11–15, 2019. ACM Press.

[NNOB12] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 681–700, Santa Barbara, CA, USA, August 19–23, 2012. Springer Berlin Heidelberg, Germany.

[SGRR19] Phillipp Schoppmann, Adrià Gascón, Leonie Reichert, and Mariana Raykova. Distributed vector-OLE: Improved constructions and implementation. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 1055–1072, London, UK, November 11–15, 2019. ACM Press.

[Suc24] Succinct Labs. SP1 zkVM. https://docs.succinct.xyz/docs/sp1/introduction, 2024. Accessed: 2026-01-12.

[WMK16] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit, 2016.

[WSR+15] Riad S. Wahby, Srinath T. V. Setty, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *NDSS 2015*, San Diego, CA, USA, February 8–11, 2015. The Internet Society.

[WYKW21] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. In *2021 IEEE Symposium on Security and Privacy*, pages 1074–1091, San Francisco, CA, USA, May 24–27, 2021. IEEE Computer Society Press.

[WYX+21] Chenkai Weng, Kang Yang, Xiang Xie, Jonathan Katz, and Xiao Wang. Mystique: Efficient conversions for zero-knowledge proofs with applications to machine learning. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 501–518. USENIX Association, August 11–13, 2021.

[Yan25] Yibin Yang. Justvengers: Batched VOLE ZK disjunctions in o(r+b+c) communication. In Benny Applebaum and Huijia (Rachel) Lin, editors, *Theory of Cryptography - 23rd International Conference, TCC 2025, Aarhus, Denmark, December 1-5, 2025, Proceedings, Part IV*, volume 16271 of *Lecture Notes in Computer Science*, pages 105–137. Springer, 2025.

[YH24] Yibin Yang and David Heath. Two shuffles make a RAM: Improved constant overhead zero knowledge RAM. In Davide Balzarotti and Wenyuan Xu, editors, *USENIX Security 2024*, Philadelphia, PA, USA, August 14–16, 2024. USENIX Association.

[YHH+23] Yibin Yang, David Heath, Carmit Hazay, Vladimir Kolesnikov, and Muthuramakrishnan Venkitasubramaniam. Batchman and robin: Batched and non-batched branching for interactive ZK. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *ACM CCS 2023*, pages 1452–1466, Copenhagen, Denmark, November 26–30, 2023. ACM Press.

[YHH+24] Yibin Yang, David Heath, Carmit Hazay, Vladimir Kolesnikov, and Muthuramakrishnan Venkitasubramaniam. Tight ZK CPU: Batched ZK branching with cost proportional to evaluated instruction. In Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie, editors, *ACM CCS 2024*, pages 3095–3109, Salt Lake City, UT, USA, October 14–18, 2024. ACM Press.

[YHKD22] Yibin Yang, David Heath, Vladimir Kolesnikov, and David Devecsery. EZEE: Epoch parallel zero knowledge for ANSI C. In *2022 IEEE European Symposium on Security and Privacy*, pages 109–123, Genoa, Italy, June 6–10, 2022. IEEE Computer Society Press.

[YPHK23] Yibin Yang, Stanislav Peceny, David Heath, and Vladimir Kolesnikov. Towards generic MPC compilers via variable instruction set architectures (VISAs). In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *ACM CCS 2023*, pages 2516–2530, Copenhagen, Denmark, November 26–30, 2023. ACM Press.

[YSWW21] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. QuickSilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 2986–3001, Virtual Event, Republic of Korea, November 15–19, 2021. ACM Press.

[YWL+20] Kang Yang, Chenkai Weng, Xiao Lan, Jiang Zhang, and Xiao Wang. Ferret: Fast extension for correlated OT with small communication. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1607–1626, Virtual Event, USA, November 9–13, 2020. ACM Press.

[ZGK+18] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. vRAM: Faster verifiable RAM with program-independent preprocessing. In *2018 IEEE Symposium on Security and Privacy*, pages 908–925, San Francisco, CA, USA, May 21–23, 2018. IEEE Computer Society Press.

A Deferred Preliminaries

A.1 Ideal $\mathcal{F}_{\mathsf{CPZK}}$

Figure 5 depicts the ideal functionality for commit-and-prove ZK.

A.2 VOLE-Based ZK

IT-MAC. *Information-theoretic message authentication codes* (IT-MACs) — named by [BDOZ11, NNOB12] — are a form of distributed correlated randomness between two parties (in our case, $\mathcal{P}$ and $\mathcal{V}$) that can be used as commitments. In IT-MACs over $\mathbb{F}$, $\mathcal{V}$ holds a uniformly sampled *global* key $\Delta \xleftarrow{\$} \mathbb{F}$. To commit to a value $x \in \mathbb{F}$, $\mathcal{V}$ samples a uniform *local* key $k_x \xleftarrow{\$} \mathbb{F}$, and $\mathcal{P}$ receives the corresponding MAC, computed as $m_x := k_x + x\Delta$. We denote the IT-MAC correlation of a value x as $[x]_\Delta := \langle (x, m_x), k_x \rangle_\Delta$. When clear from context, we omit Δ for simplicity. IT-MACs hold several useful properties:

1. **Hiding:** k_x and Δ, held by $\mathcal{V}$, are independent of the committed value x.
2. **Binding:** To open $[x]$, $\mathcal{P}$ sends x and m_x, and $\mathcal{V}$ accepts if $m_x = k_x + x\Delta$. To maliciously open $[x]$ as a different value $x' \neq x$ (i.e., forge x), $\mathcal{P}$ would need to guess Δ, succeeding with probability at most $\frac{1}{|\mathbb{F}|}$.
3. **Linear homomorphism:** IT-MACs support linear operations — such as addition, scalar multiplication, and constant addition — *without* any communication. Specifically, for any *public* constants $c_0, c_1, \ldots, c_n \in \mathbb{F}$, the parties can *locally* compute $[c_0 + c_1 x_1 + \cdots + c_n x_n]$ from $[x_1], \ldots, [x_n]$.[6] Note, this implies that an IT-MAC of a public constant can be generated *for free*.

VOLE. Random IT-MAC instances (over $\mathbb{F}$) can be viewed as the standard *vector oblivious linear evaluation* (VOLE) correlation functionality, defined as $\mathcal{F}_{\mathsf{VOLE}}$ in Figure 6. This functionality has been extensively studied in prior work (e.g., [BCGI18, BCG+19, SGRR19, YWL+20]). It can be realized efficiently, achieving even sublinear communication in n (e.g., [BCGI18]) with the *learning parity with noise* (LPN) assumption [BFKL94].

Lemma A.1 (Informal)**.** *Assuming the hardness of LPN, there exists a protocol that UC-emulates $\mathcal{F}_{\mathsf{VOLE}}$ (Figure 6) with $o(n)$ communication for each* `extend`.

It is standard [Bea95] to *consume* a random IT-MAC $[u]$ to generate an IT-MAC $[x]$ for a value x chosen by $\mathcal{P}$: (1) $\mathcal{P}$ sends $x - u$, and (2) parties *locally* compute $[x] := [u] + (x - u)$. Note that each random u can only be used once.

LPZK [DIO21, YSWW21]. VOLE-based ZK follows the typical "commit-and-prove" paradigm: $\mathcal{P}$ commits to its extended witness with IT-MACs, and later proves to $\mathcal{V}$ that committed values are consistent with the proof statement.

Given a circuit $\mathcal{C}$ (over $\mathbb{F}$), $\mathcal{P}$ commits to its inputs and the outputs of multiplication gates. Due to the linear homomorphism, parties can locally derive IT-MACs for all wires in $\mathcal{C}$. The remaining task is for $\mathcal{P}$ to prove:

1. **Satisfied circuit:** The IT-MAC on the output wire of $\mathcal{C}$ commits to a zero. This can be proved by simply opening the IT-MAC.

[6] For example, if $m_x = k_x + x\Delta$ and $m_y = k_y + y\Delta$, then $(m_x + m_y) = (k_x + k_y) + (x + y)\Delta$. Moreover, for any constant $c \in \mathbb{F}$, $\mathcal{P}$ sets $m_c = 0$ whereas $\mathcal{V}$ sets $k_c = -c\Delta$.

Functionality $\mathcal{F}_{\mathsf{CPZK}}$

$\mathcal{F}_{\mathsf{CPZK}}$, parameterized by a field $\mathbb{F}$, proceeds as follows, running with a prover $\mathcal{P}$, a verifier $\mathcal{V}$, and an adversary $\mathcal{S}$:

Commitments. On receiving $(\texttt{Commit}, \mathsf{cid}, x)$ from $\mathcal{P}$ where (i) there is no recorded tuple $(\mathsf{cid}, \cdot)$, and (ii) $x \in \mathbb{F}$: Record tuple (cid, x) and send $(\texttt{commit}, \mathsf{cid})$ to $\mathcal{V}$ and $\mathcal{S}$.

Linear Combination. On receiving $(\texttt{Linear}, \mathsf{cid}, \mathsf{cid}_1, \ldots, \mathsf{cid}_k, c_0, c_1, \ldots, c_k)$ from $\mathcal{P}$ where (i) there is no recorded tuple $(\mathsf{cid}, \cdot)$, (ii) each $\mathsf{cid}_{i\in[k]}$ has a recorded tuple, and (iii) $c_0, \ldots, c_k \in \mathbb{F}$:

1. Fetch recorded $(\mathsf{cid}_1, x_1), \ldots, (\mathsf{cid}_k, x_k)$.
2. Compute $x := c_0 + c_1 x_1 + \cdots + c_k x_k$. Record (cid, x).
3. Send $(\texttt{linear}, \mathsf{cid}, \mathsf{cid}_1, \ldots, \mathsf{cid}_k, c_0, \ldots, c_k)$ to $\mathcal{V}$ and $\mathcal{S}$.

Open. On receiving $(\texttt{Open}, \mathsf{cid})$ from $\mathcal{P}$ where cid has a recorded tuple, fetch (cid, x), send $(\texttt{open}, \mathsf{cid}, x)$ to $\mathcal{V}$ and $\mathcal{S}$.

Check. On receiving $(\texttt{Check}, \mathcal{C}, \mathsf{cid}_1, \ldots, \mathsf{cid}_n)$ from $\mathcal{P}$ where (i) $\mathcal{C} : \mathbb{F}^{(n)} \to \mathbb{F}^{(*)}$ is an arithmetic circuit, and (ii) each $\mathsf{cid}_{i\in[n]}$ has a recorded tuple: Fetch tuples $(\mathsf{cid}_1, x_1), \ldots, (\mathsf{cid}_n, x_n)$ and compute $\boldsymbol{y} := \mathcal{C}(x_1, \ldots, x_n)$. If $\boldsymbol{y} = 0^{(*)}$, send $(\texttt{check}, \mathcal{C}, \textbf{cid}, \texttt{true})$ to $\mathcal{V}$ and $\mathcal{S}$; else send $(\texttt{check}, \mathcal{C}, \textbf{cid}, \texttt{false})$ to $\mathcal{V}$ and $\mathcal{S}$.

Figure 5: Ideal functionality for commit-and-prove ZK. Each committed element is associated with a unique identifier cid. `Linear` operation allows $\mathcal{P}$ to generate a new commitment (associated with cid) via a *public affine function* over committed elements.

2. **Correct multiplication:** Each multiplication gate in $\mathcal{C}$ must be well-formed. For the k-th multiplication gate, parties hold three IT-MACs $[\ell_k], [r_k], [o_k]$ corresponding to the left input, right input, and output wires. $\mathcal{P}$ must prove in ZK that $\ell_k \cdot r_k = o_k$.

   The LPZK technique [DIO21, YSWW21] enables this proof to be performed over n such triples using only $\mathcal{O}(1)$ communication by using an extra VOLE correlation.

Although the LPZK technique achieves constant communication overhead for verifying multiple multiplication triples, $\mathcal{P}$ still needs to commit to the (extended) witness, incurring communication that scales linearly with the size of the circuit.

In summary, VOLE-based ZK provides a natural instantiation of the commit-and-prove functionality $\mathcal{F}_{\mathsf{CPZK}}$ (see Figure 5), leading to the following lemma:

Lemma A.2 (VOLE-based ZK [DIO21, YSWW21])**.** *There exists a protocol Π_{CPZK} that UC-realizes $\mathcal{F}_{\mathsf{CPZK}}$ (Figure 5) in the $\mathcal{F}_{\mathsf{VOLE}}$-hybrid model (Figure 6) secure against any static, computationally unbounded adversary with the following metric:*

- `Commit` *requires* 1 *field element of communication,* $\mathcal{O}(1)$ *field operations, and a VOLE correlation of length* 1.
- `Linear` *requires* no *communication and* $\mathcal{O}(k)$ *field operations.*
- `Open` *requires* 2 *field elements of communication and* $\mathcal{O}(1)$ *field operations, with soundness error at most* $\frac{1}{|\mathbb{F}|}$.
- *For circuit $\mathcal{C}$ with m outputs and $n_\times$ multiplication gates,* `Check` *requires $n_\times+3$ field elements of communication, $\mathcal{O}(|\mathcal{C}|)$ field operations, and a VOLE correlation of length $n_\times+1$, with soundness error at most* $\frac{n_\times+m+1}{|\mathbb{F}|}$.

Functionality $\mathcal{F}_{\mathsf{VOLE}}$

$\mathcal{F}_{\mathsf{VOLE}}$, parameterized by a field $\mathbb{F}$, proceeds as follows, running with a prover $\mathcal{P}$, a verifier $\mathcal{V}$ and an adversary $\mathcal{S}$:

Initialize. Upon receiving $(\texttt{init})$ from $\mathcal{P}$ and $\mathcal{V}$, if $\mathcal{V}$ is honest, sample $\Delta \xleftarrow{\$} \mathbb{F}$, else receive Δ from $\mathcal{S}$. Store Δ and send it to $\mathcal{V}$. Ignore subsequent $(\texttt{init})$.

Extend. Upon receiving $(\texttt{extend}, n)$ from $\mathcal{P}$ and $\mathcal{V}$:

- If $\mathcal{V}$ is honest, sample $\boldsymbol{k_u} \xleftarrow{\$} \mathbb{F}^n$, else receive $\boldsymbol{k_u} \in \mathbb{F}^n$ from $\mathcal{S}$.
- If $\mathcal{P}$ is honest, sample $\boldsymbol{u} \xleftarrow{\$} \mathbb{F}^n$ and compute $\boldsymbol{m_u} := \boldsymbol{k_u} + \boldsymbol{u} \cdot \Delta \in \mathbb{F}^n$, else receive $\boldsymbol{u} \in \mathbb{F}^n$ and $\boldsymbol{m_u} \in \mathbb{F}^n$ from $\mathcal{S}$ and compute $\boldsymbol{k_u} := \boldsymbol{m_u} - \boldsymbol{u} \cdot \Delta \in \mathbb{F}^n$.
- Send $(\boldsymbol{u}, \boldsymbol{m_u})$ to $\mathcal{P}$ and $\boldsymbol{k_u}$ to $\mathcal{V}$.

Figure 6: The VOLE correlation functionality.

- *Communication of* `Check` *for certain circuits can be improved at the cost of computation and soundness. In particular, for a circuit $\mathcal{C}$ that outputs m polynomials $f_1, \ldots, f_m$ where $f_{i\in[m]}$ is of degree-d_i,* `Check` *can be performed with:*
  - $\max_{i\in[m]} d_i + 1$ *field elements of communication;*
  - $\mathcal{O}\left((\max_{i\in[m]} d_i)^2 \cdot |\mathcal{C}|\right)$ *field operations;*
  - *a VOLE correlation of length $\mathcal{O}(\max_{i\in[m]} d_i)$; and*
  - *soundness error at most $\frac{m+\max_{i\in[m]} d_i - 1}{|\mathbb{F}|}$.*

  This is useful, e.g., when we want to prove that $\boldsymbol{a} \odot \boldsymbol{b} = \boldsymbol{0}$.

B Generalized Tight ZK CPU Protocol

In this section, we present the generalized tight ZK CPU protocol $\Pi_{\mathsf{G\text{-}ZKCPU}}$. The protocol is defined in the $\mathcal{F}_{\mathsf{CPZK\text{-}UROM\text{-}SET\text{-}KVS}}$-hybrid model and realizes the generalized tight ZK CPU ideal functionality $\mathcal{F}_{\mathsf{G\text{-}ZKCPU}}$ (see Figure 3).

The main steps toward achieving the target ideal functionality $\mathcal{F}_{\mathsf{G\text{-}ZKCPU}}$ follow the same abstraction methodology as the tight ZK CPU framework [YHH+24], and are outlined as follows:

$$\mathcal{F}_{\mathsf{CPZK}} \text{ (Figure 5)} \overset{\text{[YHH+24, YH24]}}{\Longrightarrow} \mathcal{F}_{\mathsf{CPZK\text{-}UROM\text{-}SET\text{-}KVS}} \overset{\text{this work}}{\Longrightarrow} \mathcal{F}_{\mathsf{G\text{-}ZKCPU}}$$

What is Generalized. Relative to [YHH+24], the generalized tight ZK CPU protocol extends the tight ZK CPU construction as follows:

- **Augmented UROM functionality.** We extend the read-only UROM functionality used in the tight ZK CPU construction (denoted $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$ in [YHH+24]) to a more expressive functionality $\mathcal{F}_{\mathsf{CPZK\text{-}UROM\text{-}SET\text{-}KVS}}$, which additionally supports non-linear constraints arising from set membership checks and read-write key-value store operations. These extensions are conceptually straightforward given prior CPZK-based ZK SET and ZK KVS constructions [YH24], and strictly subsume $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$.

- **Revised topology matrix definition.** To support the generalized instruction model, we revisit the *topology matrix* definition that is central to the tight ZK CPU protocol (with the underlying idea tracing back to [YHH+23]). This is required for two reasons: (i) to capture the universal-unit execution layout, which uniformly accounts for heterogeneous resource usage across the execution, and (ii) to incorporate additional non-linear resource interfaces, such as set-restricted inputs and key-value store gates, into the linear wiring constraints. In particular, accommodating these additional resource types necessitates a careful reorganization of how register values are passed across instruction boundaries. The resulting topology matrices preserve the same compressed linear-verification paradigm as in the original construction.

B.1 Additional Notation

We introduce several pieces of auxiliary notation that were originally introduced in [YHH+24] and are reused here to facilitate the formal protocol description.

- $\odot$ denotes the element-wise product.
- $\langle \cdot, \cdot \rangle$ denotes inner product.
- For a vector $\boldsymbol{a}$, we denote $(a_i, \ldots, a_j)$ as $\boldsymbol{a}[i:j]$ or $\boldsymbol{a}_{[i:j]}$.
- $\mathcal{F}_{\mathsf{CPZK}}$ (see Figure 5) allows $\mathcal{P}$ to commit to field elements (over $\mathbb{F}$) and then prove that evaluating a particular circuit on the committed values yields a vector of 0's. We denote by $\mathsf{com}(\alpha)$ a cryptographic commitment to $\alpha \in \mathbb{F}$, and naturally extend this notation to vectors (e.g., $\mathsf{com}(\boldsymbol{\alpha})$).
- For a vector $\boldsymbol{p} \in \mathbb{F}^n$ where $n \in \mathbb{Z}^+$, we say that $\boldsymbol{p}$ is a *boundary string* if and only if $\boldsymbol{p} \in \{0,1\}^{n-1} \| 1$. We note that it is efficient to check whether $\mathsf{com}(\boldsymbol{p})$ commits to a valid boundary string. Namely, given $\mathsf{com}(\boldsymbol{p})$, $\mathcal{P}$ opens p_n to prove it is 1, and $\mathcal{P}$ proves $\boldsymbol{p} \odot (\mathbf{1} - \boldsymbol{p}) = \mathbf{0}$ (i.e., each $p_{i \in [n]}$ is either 0 or 1).
- We use $\mathsf{HW}(\boldsymbol{p})$ to denote the *Hamming weight* of a boundary string. That is, the number of ones in $\boldsymbol{p}$.

We now introduce two functions Partition and Filter that we use as analysis tools. We remark that we *never* run them inside ZK.

Partition. Consider a length-n boundary string $\boldsymbol{p}$. $\boldsymbol{p}$ specifies a *partition* of a length-n vector $\boldsymbol{v}$ into $\mathsf{HW}(\boldsymbol{p})$ subvectors. We define a function Partition:

$$\begin{aligned} \boldsymbol{p} &= (\overbrace{0,\ldots,0,1}^{n_1}, \overbrace{0,\ldots,0,1}^{n_2}, \overbrace{0,\ldots,0,1}^{n_3}, \ldots), \boldsymbol{v} \in \mathbb{F}^n \\ \Rightarrow \mathsf{Partition}(\boldsymbol{p}, \boldsymbol{v}) &= \left(\boldsymbol{v}^{(1)}, \ldots, \boldsymbol{v}^{(\mathsf{HW}(\boldsymbol{p}))}\right) \text{ such that} \\ \boldsymbol{v}^{(1)} &= (v_1, \ldots, v_{n_1}), \boldsymbol{v}^{(2)} = (v_{n_1+1}, \ldots, v_{n_1+n_2}), \cdots \end{aligned}$$

Filter. A length-n boundary string $\boldsymbol{p}$ also specifies a way to *filter* a length-n vector $\boldsymbol{v}$ into a length-$\mathsf{HW}(\boldsymbol{p})$ vector. We define a function Filter:

$$\begin{aligned} \boldsymbol{p} &= (\overbrace{0,\ldots,0,1}^{n_1}, \overbrace{0,\ldots,0,1}^{n_2}, \overbrace{0,\ldots,0,1}^{n_3}, \ldots), \boldsymbol{v} \in \mathbb{F}^n \\ \Rightarrow \mathsf{Filter}(\boldsymbol{p}, \boldsymbol{v}) &= (v_{n_1}, v_{n_1+n_2}, v_{n_1+n_2+n_3}, \ldots, v_n) \end{aligned}$$

Expanding Random Challenges. In our protocol, $\mathcal{V}$ will issue random challenges, which will be composed with $\mathcal{P}$'s chosen boundary string. We consider the following composition:

- For some public challenge $\chi \in \mathbb{F}$, let $\sigma_1 := 1$, and for each $i \in [n-1]$ in order, let $\sigma_{i+1} := \sigma_i(1-p_i)+\chi^i p_i$. That is,

$$\begin{aligned} \boldsymbol{p} &= (\overbrace{0,\ldots,0,1}^{n_1},\overbrace{0,\ldots,0,1}^{n_2},\overbrace{0,\ldots,0,1}^{n_3},\ldots) \\ \Rightarrow \boldsymbol{\sigma} &= (\underbrace{1,\ldots,1}_{n_1},\underbrace{\chi^{n_1},\ldots,\chi^{n_1}}_{n_2},\underbrace{\chi^{n_1+n_2},\ldots,\chi^{n_1+n_2}}_{n_3},\ldots) \end{aligned}$$

  We denote this procedure by $\boldsymbol{\sigma} := \mathsf{Expand}_1(\boldsymbol{p}, \chi)$.

Starting from $\mathsf{com}(\boldsymbol{p})$, we can compute commitments to the above composition (i.e., $\mathsf{com}(\boldsymbol{\sigma})$) each via a circuit with $n-1$ MULTs.

B.2 The $\mathcal{F}_{\mathsf{CPZK\text{-}UROM\text{-}SET\text{-}KVS}}$-Hybrid Model

Our protocol is defined in the $\mathcal{F}_{\mathsf{CPZK\text{-}UROM\text{-}SET\text{-}KVS}}$-hybrid model, depicted in Figure 7. This hybrid extends the standard commit-and-prove zero-knowledge abstraction with ideal access to unbalanced non-zero-end read-only memory (UROM), set membership checks (SET), and a sequential key-value store (KVS).

The original tight ZK CPU protocol of [YHH+24] is defined in the $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$-hybrid model, which can be viewed as a special case of $\mathcal{F}_{\mathsf{CPZK\text{-}UROM\text{-}SET\text{-}KVS}}$ that provides UROM access but does not support SET or KVS operations. In particular, [YHH+24] shows how $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$ can be realized from $\mathcal{F}_{\mathsf{CPZK}}$ using only the standard Schwartz-Zippel lemma.

Independently, [YH24] shows that both SET and KVS functionalities can be expressed and securely enforced within the $\mathcal{F}_{\mathsf{CPZK}}$-hybrid model, again relying only on the standard Schwartz-Zippel lemma. Since the $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$ functionality already exposes the full $\mathcal{F}_{\mathsf{CPZK}}$ interface, it is therefore natural to augment $\mathcal{F}_{\mathsf{CPZK\text{-}UROM}}$ with SET and KVS support, yielding the $\mathcal{F}_{\mathsf{CPZK\text{-}UROM\text{-}SET\text{-}KVS}}$-hybrid model.

Formally, we have the following lemma:

Lemma B.1. *Fix a field $\mathbb{F}$. Let the UROM be initialized with B non-zero-end vectors $\boldsymbol{u}^{(1)},\ldots,\boldsymbol{u}^{(B)}$, where $|\boldsymbol{u}^{(i)}| = n^{(i)}_{\mathsf{urom}}$. Let a UROM read produce a concatenated output vector of total length m_{urom}. Let the SET be initialized with n_{set} elements, and let a SET read query m_{set} elements. Let the KVS be initialized with n_{kvs} pairs, and let a read-write invocation perform m_{kvs} sequential read-write operations.*

Then there exists a protocol $\Pi_{\mathsf{CPZK\text{-}UROM\text{-}SET\text{-}KVS}}$ that UC-realizes $\mathcal{F}_{\mathsf{CPZK\text{-}UROM\text{-}SET\text{-}KVS}}$ (Figure 7) in the $\mathcal{F}_{\mathsf{CPZK}}$-hybrid model (Figure 5), with perfect zero-knowledge and soundness error at most

$$\frac{\max\{m_{\mathsf{urom}}, n^{(1)}_{\mathsf{urom}},\ldots,n^{(B)}_{\mathsf{urom}}\}+B+m_{\mathsf{urom}}+n_{\mathsf{set}}+m_{\mathsf{set}}+n_{\mathsf{kvs}}+3m_{\mathsf{kvs}}}{|\mathbb{F}|},$$

against any static, computationally unbounded adversary.

Moreover, the protocol achieves the following efficiency in the $\mathcal{F}_{\mathsf{CPZK}}$-hybrid model (omitting several public values sent by $\mathcal{P}$ for formalization purposes):

- *$\mathcal{V}$ sends $\mathcal{O}(1)$ field elements to $\mathcal{P}$.*
- *$\mathcal{O}(B + m_{\mathsf{urom}} + n_{\mathsf{set}} + m_{\mathsf{set}} + n_{\mathsf{kvs}} + m_{\mathsf{kvs}})$ `Commit` hybrid calls.*
- *$\mathcal{O}\left(\sum_{i\in[B]} n^{(i)}\right)$ local field operations.*
- *B `Linear` hybrid calls to commit public constants.*
- *1 `Open` hybrid call to open public constants.*
- *8 `Check` hybrid calls, invoking the following circuits:*

1. *Three circuits with at most m_{urom} multiplication gates each.*
2. *One circuit with at most $3m_{\mathsf{urom}}$ multiplication gates.*
3. *One circuit with at most $2B + 2m_{\mathsf{urom}}$ multiplication gates.*
4. *One circuit with at most $n_{\mathsf{set}} + 2m_{\mathsf{set}}$ multiplication gates.*
5. *One circuit with at most $n_{\mathsf{kvs}} + 2m_{\mathsf{kvs}}$ multiplication gates.*
6. *One circuit with at most $3m_{\mathsf{kvs}}$ multiplication gates.*

Proof. The result follows by straightforward composition of the constructions of [YHH+24] and [YH24]. □

B.3 Generalized Topology Matrices

We revisit a core component of the original tight ZK CPU protocol — the *topology matrix* — and redefine it for our generalized tight ZK CPU execution model.

Using Topology Matrices to Prove Circuit Satisfiability. To motivate the definition of a topology matrix, it is instructive to first see how it can be used to prove satisfiability of a *single* arithmetic circuit, without yet invoking the (generalized) tight ZK CPU.

Let $\mathcal{C}$ be a fixed arithmetic circuit over a field $\mathbb{F}$ with n_{in} input wires, $n_\times$ multiplication gates, and a single output wire whose value must be 0 for the circuit to be satisfied. W.l.o.g., we assume

$$n \;:=\; n_{\mathsf{in}} \;=\; n_\times + 2$$

The proof proceeds as follows.

The prover $\mathcal{P}$ commits to:

- the circuit inputs $\mathsf{in}_1, \ldots, \mathsf{in}_n$, and
- multiplication witnesses $(\ell_i, r_i, o_i)_{i\in[n]}$ satisfying $\ell_i \cdot r_i = o_i$ for all $i \in [n]$.

These multiplication witnesses are interpreted as follows:

- **Constant** 1**.** The first triple encodes a constant wire:

$$(\ell_1, r_1, o_1) = (1, 1, 1)$$

- **Multiplication gates.** For each $i \in \{2, \ldots, n-1\}$, the triple (ℓ_i, r_i, o_i) corresponds to one multiplication gate of $\mathcal{C}$.
- **Output check.** The last triple enforces circuit satisfaction:

$$(\ell_n, r_n, o_n) = (1, 0, 0)$$

The committed values are arranged into the interleaved vector

$$\boldsymbol{w} \;:=\; (\mathsf{in}_1, o_1, \mathsf{in}_2, o_2, \ldots, \mathsf{in}_n, o_n) \;\in\; \mathbb{F}^{2n}$$

The structure of $\mathcal{C}$ induces a *topology matrix* $\boldsymbol{T} \in \mathbb{F}^{2n\times 2n}$ such that

$$\boldsymbol{T} \cdot \boldsymbol{w}^T \;=\; (\ell_1, r_1, \ell_2, r_2, \ldots, \ell_n, r_n)^T \tag{2}$$

That is, each left and right input of every multiplication gate is expressed as a fixed linear combination of the circuit inputs and multiplication outputs (r_n encodes $\mathcal{C}$'s output).

To complete the proof, it suffices for $\mathcal{P}$ to prove to $\mathcal{V}$, in ZK:

- all committed multiplication triples satisfy $\ell_i \cdot r_i = o_i$;

Functionality $\mathcal{F}_{\text{CPZK-UROM-SET-KVS}}$

$\mathcal{F}_{\text{CPZK-UROM-SET-KVS}}$, parameterized by a field $\mathbb{F}$, runs with a prover $\mathcal{P}$, a verifier $\mathcal{V}$, and an adversary $\mathcal{S}$:

CPZK

The functionality supports all instructions of $\mathcal{F}_{\text{CPZK}}$ (see Figure 5).

Unbalanced Non-Zero-End Read-Only Memory

Initialize UROM. On input $(\texttt{InitUROM}, \boldsymbol{u}^{(1)}, \ldots, \boldsymbol{u}^{(B)})$ from $\mathcal{P}$, where for each $i \in [B]$ we have $\boldsymbol{u}^{(i)} = (u_1^{(i)}, \ldots, u_{n^{(i)}}^{(i)})$ and each $u_j^{(i)}$ was previously recorded as a cid (i.e., the unbalanced vectors were committed):

1. For each $i \in [B]$, fetch $(u_1^{(i)}, x_1^{(i)}), \ldots, (u_{n^{(i)}}^{(i)}, x_{n^{(i)}}^{(i)})$ and let $\boldsymbol{x}^{(i)} := (x_1^{(i)}, \ldots, x_{n^{(i)}}^{(i)})$. Abort if $\exists i \in [B]$ such that $x_{n^{(i)}}^{(i)} = 0$.
2. Create a key-value store X with $X[i] := \boldsymbol{x}^{(i)}$ for all $i \in [B]$, and set $f_{\mathsf{urom}} := \texttt{honest}$.
3. Send $(\texttt{initurom}, \boldsymbol{u}^{(1)}, \ldots, \boldsymbol{u}^{(B)})$ to $\mathcal{V}$ and $\mathcal{S}$.

Ignore subsequent `InitUROM` calls.

Set Program. On input $(\texttt{SetProg}, \mathbf{cid}^{(p)}, \mathbf{cid}^{(\mathsf{id})})$ from $\mathcal{P}$, where $|\mathbf{cid}^{(p)}| = |\mathbf{cid}^{(\mathsf{id})}| = m \in \mathbb{Z}^+$ and each $\mathsf{cid}_i^{(p)}, \mathsf{cid}_i^{(\mathsf{id})}$ was recorded: fetch $(\mathsf{cid}_i^{(p)}, p_i)$ and $(\mathsf{cid}_i^{(\mathsf{id})}, \mathsf{id}_i)$ for all $i \in [m]$, record $\boldsymbol{p}$ and $\mathbf{id}$, and abort unless $\boldsymbol{p} \in \{0,1\}^{m-1} \| 1$. If the check passes, send $(\texttt{setprog}, \mathbf{cid}^{(p)}, \mathbf{cid}^{(\mathsf{id})})$ to $\mathcal{V}$ and $\mathcal{S}$. Ignore subsequent `SetProg` calls.

Read UROM. On input $(\texttt{ReadUROM}, \mathbf{cid}^{(d)}, \boldsymbol{d})$ from $\mathcal{P}$ such that: (i) `InitUROM` and `SetProg` were executed; (ii) $|\mathbf{cid}^{(d)}| = |\boldsymbol{d}| = |\boldsymbol{p}| = |\mathbf{id}| = m$; (iii) no tuple $(\mathsf{cid}_i^{(d)}, \cdot)$ is recorded yet for any $i \in [m]$; and (iv) each $d_i \in \mathbb{F}$: record $(\mathsf{cid}_i^{(d)}, d_i)$ for all $i \in [m]$.

Index-range check. If $\mathcal{P}$ is corrupted and $\exists i \in [m]$ with $\mathsf{id}_i \notin [B]$, set $f_{\mathsf{urom}} := \texttt{cheating}$.

Content check. For each pair $(\boldsymbol{x}, \boldsymbol{y})$ in order, where $\boldsymbol{x} \in \mathsf{Partition}(\boldsymbol{p}, \boldsymbol{d})$ and $\boldsymbol{y} \in \mathsf{Partition}(\boldsymbol{p}, \mathbf{id})$ (there are $\mathsf{HW}(\boldsymbol{p})$ pairs): if $\mathcal{P}$ is corrupted and either $\mathsf{last}(\boldsymbol{x}) = 0$ or $X[\mathsf{last}(\boldsymbol{y})] \neq \boldsymbol{x}$, set $f_{\mathsf{urom}} := \texttt{cheating}$.

Send $(\texttt{readurom}, \mathbf{cid}^{(d)})$ to $\mathcal{V}$ and $\mathcal{S}$. Ignore subsequent `ReadUROM` calls.

Check UROM. On input $(\texttt{CheckUROM})$ from $\mathcal{P}$ after `ReadUROM`: if $\mathcal{P}$ is corrupted and $\mathcal{S}$ sends `Cheat`, set $f_{\mathsf{urom}} := \texttt{cheating}$. Send $(\texttt{checkurom}, f_{\mathsf{urom}})$ to $\mathcal{V}$ and $\mathcal{S}$.

Set

Initialize SET. On input $(\texttt{InitSET}, y_1, \ldots, y_n)$ from $\mathcal{P}$ where each $y_i \in \mathbb{F}$: set $Y := \{y_1, \ldots, y_n\}$ and $f_{\mathsf{set}} := \texttt{honest}$, and send $(\texttt{initset}, y_1, \ldots, y_n)$ to $\mathcal{V}$ and $\mathcal{S}$. Ignore subsequent `InitSET` calls.

Read SET. On input $(\texttt{ReadSET}, \mathsf{cid}_1, \ldots, \mathsf{cid}_m, x_1, \ldots, x_m)$ from $\mathcal{P}$ such that: (i) `InitSET` was executed; (ii) no tuple $(\mathsf{cid}_i, \cdot)$ is recorded yet for any $i \in [m]$; and (iii) each $x_i \in \mathbb{F}$: record (cid_i, x_i) for all $i \in [m]$.

Membership check. If $\mathcal{P}$ is corrupted and $\exists i \in [m]$ with $x_i \notin Y$, set $f_{\mathsf{set}} := \texttt{cheating}$.

Send $(\texttt{readset}, \mathsf{cid}_1, \ldots, \mathsf{cid}_m)$ to $\mathcal{V}$ and $\mathcal{S}$. Ignore subsequent `ReadSET` calls.

Check SET. On input $(\texttt{CheckSET})$ from $\mathcal{P}$ after `ReadSET`: if $\mathcal{P}$ is corrupted and $\mathcal{S}$ sends `Cheat`, set $f_{\mathsf{set}} := \texttt{cheating}$. Send $(\texttt{checkset}, f_{\mathsf{set}})$ to $\mathcal{V}$ and $\mathcal{S}$.

Key-Value Store

Initialize KVS. On input $(\texttt{InitKVS}, \mathsf{key}_1, \ldots, \mathsf{key}_n, \mathsf{val}_1, \ldots, \mathsf{val}_n)$ from $\mathcal{P}$ such that: (i) each $\mathsf{key}_i, \mathsf{val}_i \in \mathbb{F}$; and (ii) $\forall i \neq j \in [n]$, $\mathsf{key}_i \neq \mathsf{key}_j$: set $Z[\mathsf{key}_i] := \mathsf{val}_i$ for all $i \in [n]$ and $f_{\mathsf{kvs}} := \texttt{honest}$, and send $(\texttt{initkvs}, \mathsf{key}_1, \ldots, \mathsf{key}_n, \mathsf{val}_1, \ldots, \mathsf{val}_n)$ to $\mathcal{V}$ and $\mathcal{S}$. Ignore subsequent `InitKVS` calls.

Read-Write KVS. On input $(\texttt{ReadWriteKVS}, \mathbf{cid}^{(k)}, \boldsymbol{k}, \mathbf{cid}^{(v)}, \boldsymbol{v}, \mathbf{cid}^{(s)}, \boldsymbol{s})$ from $\mathcal{P}$ such that: (i) `InitKVS` was executed; (ii) no tuple $(\mathsf{cid}_i^{(k)}, \cdot)$, $(\mathsf{cid}_i^{(v)}, \cdot)$, or $(\mathsf{cid}_i^{(s)}, \cdot)$ is recorded yet for any $i \in [m]$; and (iii) each $k_i, v_i, s_i \in \mathbb{F}$: record all tuples $(\mathsf{cid}_i^{(k)}, k_i)$, $(\mathsf{cid}_i^{(v)}, v_i)$, $(\mathsf{cid}_i^{(s)}, s_i)$.

Sequential consistency and update. For each $i \in [m]$ in increasing order:

1. *Key existence check.* If $\mathcal{P}$ is corrupted and $k_i \notin \mathsf{dom}(Z)$, set $f_{\mathsf{kvs}} := \texttt{cheating}$.
2. *Read consistency check.* If $\mathcal{P}$ is corrupted and $Z[k_i] \neq v_i$, set $f_{\mathsf{kvs}} := \texttt{cheating}$.
3. *Write update.* Update $Z[k_i] := s_i$.

Send $(\texttt{readwritekvs}, \mathbf{cid}^{(k)}, \mathbf{cid}^{(v)}, \mathbf{cid}^{(s)})$ to $\mathcal{V}$ and $\mathcal{S}$. Ignore subsequent `ReadWriteKVS` calls.

Check KVS. On input $(\texttt{CheckKVS})$ from $\mathcal{P}$ after `ReadWriteKVS`: if $\mathcal{P}$ is corrupted and $\mathcal{S}$ sends `Cheat`, set $f_{\mathsf{kvs}} := \texttt{cheating}$. Send $(\texttt{checkkvs}, f_{\mathsf{kvs}})$ to $\mathcal{V}$ and $\mathcal{S}$.

Figure 7: Ideal functionality for commit-and-prove zero-knowledge supporting (i) unbalanced non-zero-end read-only memory, (ii) set membership checks, and (iii) a read-write KVS.

- $o_1 = 1$ (enforcing the constant wire);
- $o_n = 0$ (enforcing circuit satisfaction); and
- the linear constraint Equation (2) holds.

The linear constraint Equation (2) consists of $2n$ field equalities. Using the standard random linear combination technique, $\mathcal{V}$ samples a uniform challenge $\chi \xleftarrow{\$} \mathbb{F}$, and $\mathcal{P}$ proves the single equality

$$\langle \boldsymbol{gT}, \boldsymbol{w} \rangle \;=\; \langle \boldsymbol{g}, \boldsymbol{u} \rangle, \boldsymbol{g} := (1, \chi, \ldots, \chi^{2n-1}), \boldsymbol{u} := (\ell_1, r_1, \ldots, \ell_n, r_n) \tag{3}$$

Equivalently, defining the *public topology vector* $\mathbf{tv} \;:=\; \boldsymbol{gT} \in \mathbb{F}^{2n}$, it suffices to prove the single inner-product relation

$$\langle \mathbf{tv}, \boldsymbol{w} \rangle \;=\; \langle \boldsymbol{g}, \boldsymbol{u} \rangle \tag{4}$$

By the Schwartz–Zippel lemma, if $\boldsymbol{T}\boldsymbol{w}^T \neq \boldsymbol{u}$, then Equation (3) (and Equation (4)) fails except with probability at most $(2n-1)/|\mathbb{F}|$.

From a Single Circuit to a Register-Free Tight ZK CPU. We now explain how the above view extends to the *register-free* case of the original tight ZK CPU, where an execution is a *sequence* of instruction circuits (in particular, standard arithmetic circuits), but there is no state passed between consecutive instructions.

Concretely, let the public instruction set be $\{\mathcal{C}_1, \ldots, \mathcal{C}_B\}$, where each $\mathcal{C}_i$ has its own size $n^{(i)} = n_{\mathsf{in}}^{(i)} = n_{\times}^{(i)} + 2$ and induces a topology matrix $\boldsymbol{T}^{(i)} \in \mathbb{F}^{2n^{(i)} \times 2n^{(i)}}$ as in Equation (2). (Note, we still have one checking zero output.) An execution is specified by a private sequence of instruction indices $(\mathsf{id}_1, \ldots, \mathsf{id}_\tau) \in [B]^\tau$; $\mathcal{V}$ learns only the total length $n \;:=\; \sum_{t=1}^{\tau} n^{(\mathsf{id}_t)}$.

The "hidden circuit". If the executed sequence were public, the corresponding composed circuit would be the concatenation of the τ instructions, and its topology matrix would be block diagonal:

$$\boldsymbol{T} \;=\; \mathrm{diag}\big(\boldsymbol{T}^{(\mathsf{id}_1)}, \ldots, \boldsymbol{T}^{(\mathsf{id}_\tau)}\big) \;\in\; \mathbb{F}^{2n \times 2n} \tag{5}$$

$\mathcal{P}$ commits to the global interleaved witness $\boldsymbol{w} \in \mathbb{F}^{2n}$ and the global multiplication inputs $\boldsymbol{u} \in \mathbb{F}^{2n}$ (defined exactly as in the single-circuit case, but for the concatenated execution), proves all multiplication constraints $\ell_i r_i = o_i$, and proves the compressed linear relation

$$\langle \boldsymbol{gT}, \boldsymbol{w} \rangle \;=\; \langle \boldsymbol{g}, \boldsymbol{u} \rangle, \qquad \boldsymbol{g} = (1, \chi, \ldots, \chi^{2n-1})$$

The difficulty is that in the tight ZK CPU we must keep $\boldsymbol{T}$ (hence the executed sequence) hidden from $\mathcal{V}$.

Topology vectors and segment boundaries. Define the *per-instruction topology vectors*

$$\mathbf{tv}^{(i)} \;:=\; (1, \chi, \ldots, \chi^{2n^{(i)}-1})\, \boldsymbol{T}^{(i)} \;\in\; \mathbb{F}^{2n^{(i)}} \qquad i \in [B]$$

which are public once χ is fixed. Because $\boldsymbol{T}$ is block diagonal, the *global* topology vector $\mathbf{tv} := \boldsymbol{gT} \in \mathbb{F}^{2n}$ has the structured form

$$\mathbf{tv} \;=\; \mathbf{tv}^{(\mathsf{id}_1)} \,\|\, \chi^{2n^{(\mathsf{id}_1)}} \mathbf{tv}^{(\mathsf{id}_2)} \,\|\, \cdots \|\, \chi^{2\sum_{j<\tau} n^{(\mathsf{id}_j)}} \mathbf{tv}^{(\mathsf{id}_\tau)} \tag{6}$$

Thus, proving the compressed relation reduces to producing a commitment to the (global) $\mathbf{tv}$ consistent with some hidden segmentation and hidden instruction choices, and then proving the single inner-product equality $\langle \mathbf{tv}, \boldsymbol{w} \rangle = \langle \boldsymbol{g}, \boldsymbol{u} \rangle$.

To express the hidden segmentation, $\mathcal{P}$ commits to a boundary string $\boldsymbol{p} \in \{0,1\}^{2n-1} \| 1$ whose 1 entries mark the ends of instruction segments (in the $2n$-length, two-wires-per-gate view), and to an index vector

$\mathbf{idx} \in [B]^{2n}$ whose entries at boundaries specify which instruction ends there (all non-boundary positions are dummy). That is, the intended sequence is $\mathsf{Filter}(\boldsymbol{p}, \mathbf{idx})$, whereas $\mathsf{Partition}(\boldsymbol{p}, \mathbf{tv})$ specifies how $\mathbf{tv}$ is split into per-instruction chunks.

Enforcing constant-one and zero-output wires. The boundary string $\boldsymbol{p}$ is also used to enforce the local correctness conditions of each (private) instruction chunk. Specifically, for every chunk start marked by $\boldsymbol{p}$, the first multiplication output is constrained to equal 1, supplying the constant wire of that instruction; for every chunk end, the last multiplication output is constrained to equal 0, enforcing instruction satisfaction. These constraints are imposed conditionally on $\boldsymbol{p}$ and verified in ZK, without revealing instruction boundaries.

Committing to $\mathbf{tv}$ *without revealing the sequence.* Equation (6) shows that $\mathbf{tv}$ is obtained by (1) selecting the appropriate public vectors $\mathbf{tv}^{(b)}$ according to the hidden indices at boundaries, and (2) applying the appropriate χ-power *shift* to each selected chunk. The tight ZK CPU therefore reduces to a "load-and-concatenate" primitive over *variable-length* public vectors, driven by the committed $(\boldsymbol{p}, \boldsymbol{id})$, which can be realized essentially via the ZK unbalanced ROM formalized in Figure 7.

A Simpler View of Register Passing. We now extend the register-free tight ZK CPU to support registers. Our presentation deviates from the construction in the original tight ZK CPU protocol [YHH+24]; the resulting view is conceptually simpler and, more importantly, serves as a convenient stepping stone toward the generalized tight ZK CPU developed in this work. From now on, we treat the distinguished constant-1 wire as an additional register that is propagated unchanged across all instructions.

Register encoding inside one instruction. Fix an instruction $\mathcal{C}_i$ with m registers. We define $\mathcal{C}_i$ so that its topology matrix constrains, in addition to the regular multiplication gates and the final zero-check, the *last* $m+1$ *pairs of multiplication inputs* (left and right wires) to encode the outgoing register state together with a constant-1 wire. Concretely, these $m+1$ multiplications are forced to have left input 1 and right input equal to the corresponding value in $(1\|\mathbf{st}_{\mathsf{out}}^{(i)})$, so these multiplication outputs equal $(1\|\mathbf{st}_{\mathsf{out}}^{(i)})$ as well. Crucially, these $m+1$ outputs are *not consumed* by the computation of $\mathcal{C}_i$.

How values flow across instructions. When instructions are concatenated, the above unused outputs become the *first* $m+1$ *multiplication outputs* of the subsequent instruction. They serve as that instruction's incoming constant-1 wire and register values. Thus, register passing is realized by a simple split across the boundary: the current instruction constrains the multiplication *inputs* that encode its outgoing registers, while the next instruction consumes the corresponding multiplication *outputs* as its incoming state. No explicit cross-instruction wiring is needed; each instruction constrains only the wires it is responsible for.

Global offset interpretation. Equivalently, over the entire execution, register passing corresponds to a *public, fixed shift* by $m+1$ positions between multiplication inputs and multiplication outputs. In the global sequence of multiplication triples, the $m+1$ pairs of inputs (ℓ, r) that encode a register state appear exactly $m+1$ positions earlier than the outputs o that carry the same values forward. This shift is independent of the executed instruction sequence and therefore does not reveal instruction boundaries: the verifier observes only a single contiguous sequence of multiplication triples, and the reuse of certain outputs only by subsequent topology constraints is indistinguishable from ordinary intra-circuit wiring (with boundaries hidden elsewhere in the protocol).

Formally, fix an instruction $\mathcal{C}_i$ with m registers, $n_{\times}^{(i)}$ multiplication gates (denoted $\mathrm{MULT}_1, \ldots, \mathrm{MULT}_{n_{\times}^{(i)}}$), and $n_{\mathsf{in}}^{(i)}$ (unrestricted) input wires. W.l.o.g., assume $n^{(i)} := n_{\mathsf{in}}^{(i)} = n_{\times}^{(i)} + m + 2$.

We augment $\mathcal{C}_i$ with an additional $m+2$ *auxiliary* multiplications, denoted $\mathrm{MULT}_{n_{\times}^{(i)}+1}, \ldots, \mathrm{MULT}_{n_{\times}^{(i)}+m+2}$. For each auxiliary multiplication, the left input wire is fixed to the constant 1. The right input wires are defined as follows:

1. $\mathrm{MULT}_{n_\times^{(i)}+1}$ encodes the instruction's zero-check wire (i.e., the output of $\mathcal{C}_i$ that must be 0 for satisfaction).

2. $\mathrm{MULT}_{n_\times^{(i)}+2}$ encodes the distinguished constant-1 wire that is propagated across the execution.

3. For each $k \in [m]$, $\mathrm{MULT}_{n_\times^{(i)}+2+k}$ encodes the k-th outgoing register value.

Consequently, the outputs of these auxiliary multiplications equal their right inputs, and the last $m+1$ multiplication outputs of the instruction encode $(1 \| \mathbf{st}_{\mathsf{out}})$.

The instruction $\mathcal{C}_i$ induces a public topology matrix $\boldsymbol{T}^{(i)} \in \mathbb{F}^{2n^{(i)} \times 2n^{(i)}}$ whose rows define the left and right input wires of all multiplication gates $\mathrm{MULT}_1, \ldots, \mathrm{MULT}_{n_\times^{(i)}+m+2}$:

$$\begin{pmatrix} \text{define left input of } \mathrm{MULT}_1 \\ \text{define right input of } \mathrm{MULT}_1 \\ \vdots \\ \text{define left input of } \mathrm{MULT}_{n_\times^{(i)}+m+2} \\ \text{define right input of } \mathrm{MULT}_{n_\times^{(i)}+m+2} \end{pmatrix} \tag{7}$$

Now suppose $\mathcal{P}$ wishes to prove the execution of $\mathcal{C}_1$ followed by $\mathcal{C}_2$, where the initial registers are $\boldsymbol{x}$ and the final registers are $\boldsymbol{y}$ ($\boldsymbol{x}, \boldsymbol{y}$ are public). $\mathcal{P}$ can commit to $n = n^{(1)} + n^{(2)}$ inputs and $n + m + 1$ MULT triples and show:

$$\begin{aligned} &\boldsymbol{T} \times (\mathsf{in}_1, o_1, \ldots, \mathsf{in}_n, o_n)^T \\ &= (\ell_{m+2}, r_{m+2}, \ldots, \ell_{n+m+1}, r_{n+m+1})^T \end{aligned} \quad \Bigg| \quad \boldsymbol{T} := \begin{pmatrix} \boldsymbol{T}^{(1)} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{T}^{(2)} \end{pmatrix}$$

and $o_1 = 1, \boldsymbol{o}_{2:m+1} = \boldsymbol{x}, \boldsymbol{\ell}_{n+1:n+m+1} = \boldsymbol{1}, \boldsymbol{r}_{n+1:n+m+1} = (1, \boldsymbol{y})$. (The zero-checking outputs are checked separately via $\boldsymbol{p}$, as before.) The block-diagonal matrix $\boldsymbol{T}$ is private, but the compressed topology vector $(1, \chi, \chi^2, \ldots)\boldsymbol{T}$ can be constructed using ZK UROM, as before.

Generalized Topology Matrices. We now lift the topology-matrix view from the original tight ZK CPU to the generalized execution model introduced in this work.

Recall that, in the original tight ZK CPU, the role of the topology matrix is purely to enforce *linear wiring constraints*: it specifies how the left and right inputs of multiplication gates are expressed as fixed linear combinations of committed inputs and multiplication outputs. All non-linear correctness conditions (i.e., multiplication consistency) are enforced separately, and the topology matrix itself never encodes non-linear constraints.

Our generalized setting follows the same separation of concerns. Using the $\mathcal{F}_{\mathsf{CPZK\text{-}UROM\text{-}SET\text{-}KVS}}$-hybrid model, $\mathcal{P}$ can commit to (i) unrestricted inputs, (ii) multiplication triples (ℓ, r, o), (iii) set-restricted inputs, and (iv) key-value store triples $(\mathsf{idx}, \mathsf{val}_{\mathsf{old}}, \mathsf{val}_{\mathsf{new}})$, with the semantic correctness of multiplication, set membership, and KVS read-write behavior enforced directly by the ideal functionality. As a result, once these values are committed, they may be treated uniformly as field elements whose correctness is already guaranteed.

This allows us to reuse the topology-matrix abstraction almost verbatim. In the generalized topology matrix, rows no longer define only the left and right inputs of multiplication gates, but instead also define the index and value inputs of KVS gates. Moreover, such wires may be defined as linear combinations of unrestricted inputs, set-restricted inputs, multiplication outputs, and the values read from the key-value store. All such constraints remain linear.

The idea of a fixed universal-unit layout fits in naturally. It amounts to fixing a public ordering of all constrained wires, so that both parties agree on how the rows and columns of the topology matrix are interpreted.

A generalized instruction may designate *multiple* wires whose values are required to be zero. This situation is discussed in [YHH+24], and we adopt the same approach here. Specifically, rather than enforcing each zero constraint individually, all designated zero wires are aggregated into a single constraint using a random linear combination. After the verifier samples a uniform challenge $\gamma \xleftarrow{\$} \mathbb{F}$, the aggregated zero-check wire is defined as a linear combination of the designated zero wires with coefficients $(1, \gamma, \gamma^2, \ldots)$ in a fixed public order. The topology matrix then includes a single γ-dependent (but public) row defining this aggregated zero-check wire, which is enforced to equal zero. Soundness follows directly from the Schwartz-Zippel lemma.

In the generalized setting, register passing continues to be realized via auxiliary multiplication gates as described. Each instruction introduces additional multiplications whose outputs encode the outgoing register values together with the distinguished constant-1 wire. No explicit wiring across instruction boundaries is required.

Formally, fix a generalized instruction $\mathcal{I}_i$ with m registers, defined as in Definition 4.1. The protocol $\Pi_{\mathsf{G\text{-}ZKCPU}}$ is parameterized by a universal-unit profile $(u_{\mathsf{in}}, u_{\mathsf{set}}, u_{\mathsf{kvs}}, u_{\times})$, which specifies how different resource types are packed into a fixed execution layout.

For instruction $\mathcal{I}_i$, let $n^{(i)}_{\mathsf{in}}$ denote the number of unrestricted auxiliary inputs, $n^{(i)}_{\mathsf{set}}$ the number of set-restricted inputs, $n^{(i)}_{\mathsf{kvs}}$ the number of key-value store gates, and $n^{(i)}_{\times}$ the number of multiplication gates. The size $n^{(i)}$ of the instruction is induced by the universal-unit profile as defined in Equation (1). W.l.o.g., we assume that $\mathcal{I}_i$ is fully packed under the profile, i.e.,

$$n^{(i)}_{\mathsf{in}} = n^{(i)} \cdot u_{\mathsf{in}}, n^{(i)}_{\mathsf{set}} = n^{(i)} \cdot u_{\mathsf{set}}, n^{(i)}_{\mathsf{kvs}} = n^{(i)} \cdot u_{\mathsf{kvs}}, n^{(i)}_{\times} = n^{(i)} \cdot u_{\times} - m - 2$$

Under this convention, the $m{+}2$ subtraction in the definition of $n^{(i)}_{\times}$ accounts for auxiliary multiplication gates used for register passing and zero checking. Concretely, we view instruction $\mathcal{I}_i$ as consisting of multiplication gates $\mathrm{MULT}_1, \ldots, \mathrm{MULT}_{n^{(i)}_{\times}}$ that implement the instruction's internal semantics, together with $m{+}2$ auxiliary multiplications $\mathrm{MULT}_{n^{(i)}_{\times}+1}, \ldots, \mathrm{MULT}_{n^{(i)}_{\times}+m+2}$. For each auxiliary multiplication, the left input wire is fixed to the constant 1, while the right input wire encodes (i) the aggregated zero-checking value, (ii) the distinguished constant-1 wire, or (iii) one of the m outgoing register values. As before, the outputs of these auxiliary multiplications are not consumed within $\mathcal{I}_i$ itself and instead serve as the incoming register values of the subsequent instruction.

We view instruction $\mathcal{I}_i$ as consisting of KVS gates $\mathrm{KVS}_1, \ldots, \mathrm{KVS}_{n^{(i)}_{\mathsf{kvs}}}$ that implement the instruction's internal semantics.

Let $n^{(i)}_{\mathsf{row}} := n^{(i)} \cdot (2u_{\times} + 2u_{\mathsf{kvs}})$ and $n^{(i)}_{\mathsf{column}} := n^{(i)} \cdot (u_{\mathsf{in}} + u_{\mathsf{set}} + u_{\mathsf{kvs}} + u_{\times})$. The instruction $\mathcal{I}_i$ induces a public topology matrix $\boldsymbol{T}^{(i)} \in \mathbb{F}^{n^{(i)}_{\mathsf{row}} \times n^{(i)}_{\mathsf{column}}}$ of the following schematic form:

$$\begin{pmatrix} u_{\times} \text{ rows defining left inputs of } \mathrm{MULT}_{(t-1)u_{\times}+1:tu_{\times}} \\ u_{\times} \text{ rows defining right inputs of } \mathrm{MULT}_{(t-1)u_{\times}+1:tu_{\times}} \\ u_{\mathsf{kvs}} \text{ rows defining index inputs of } \mathrm{KVS}_{(t-1)u_{\mathsf{kvs}}+1:tu_{\mathsf{kvs}}} \\ u_{\mathsf{kvs}} \text{ rows defining value inputs of } \mathrm{KVS}_{(t-1)u_{\mathsf{kvs}}+1:tu_{\mathsf{kvs}}} \\ \vdots \\ (\text{the above four blocks repeated for each unit } t \in [n^{(i)}]) \end{pmatrix} \tag{8}$$

Now suppose $\mathcal{P}$ wishes to prove the execution of instruction $\mathcal{I}_1$ followed by $\mathcal{I}_2$, where the initial register state is $\boldsymbol{x}$ and the final register state is $\boldsymbol{y}$ (both public). Let $n := n^{(1)} + n^{(2)}$.

$\mathcal{P}$ commits to nu_{in} unrestricted inputs (denoted by $\mathbf{in}$), nu_{set} set-restricted inputs (denoted by $\boldsymbol{b}$), nu_{kvs} key-value store read-write triples (denoted by $\boldsymbol{k}, \boldsymbol{s}, \boldsymbol{v}$), and $nu_{\times} + m + 1$ multiplication triples (denoted by $\boldsymbol{\ell}, \boldsymbol{r}, \boldsymbol{o}$).

These commitments define a vector $\mathsf{LEFT} \in \mathbb{F}^{n^{(i)}_{\mathsf{column}}}$ and a vector $\mathsf{RIGHT} \in \mathbb{F}^{n^{(i)}_{\mathsf{row}}}$, constructed according to the fixed universal-unit layout as follows. For each unit $t \in [n]$, define $\mathsf{LEFT}^{(t)} \in \mathbb{F}^{u_{\mathsf{in}}+u_{\mathsf{set}}+u_{\mathsf{kvs}}+u_{\times}}$ as:

$$\mathbf{in}_{[(t-1)u_{\mathsf{in}}+1:tu_{\mathsf{in}}]} \parallel \boldsymbol{b}_{[(t-1)u_{\mathsf{set}}+1:tu_{\mathsf{set}}]} \parallel \boldsymbol{v}_{[(t-1)u_{\mathsf{kvs}}+1:tu_{\mathsf{kvs}}]} \parallel \boldsymbol{o}_{[(t-1)u_{\times}+1:tu_{\times}]}$$

Let $\mathsf{LEFT} := \mathsf{LEFT}^{(1)} \parallel \cdots \parallel \mathsf{LEFT}^{(n)}$. Similarly, define for each unit $t \in [n]$, define $\mathsf{RIGHT}^{(t)} \in \mathbb{F}^{2u_{\mathsf{kvs}}+2u_{\times}}$ as:

$$\boldsymbol{k}_{[(t-1)u_{\mathsf{kvs}}+1:tu_{\mathsf{kvs}}]} \parallel \boldsymbol{s}_{[(t-1)u_{\mathsf{kvs}}+1:tu_{\mathsf{kvs}}]} \parallel \boldsymbol{\ell}_{[(t-1)u_{\times}+m+2:tu_{\times}+m+1]} \parallel \boldsymbol{r}_{[(t-1)u_{\times}+m+2:tu_{\times}+m+1]}$$

Let $\mathsf{RIGHT} := \mathsf{RIGHT}^{(1)} \parallel \cdots \parallel \mathsf{RIGHT}^{(n)}$.

It suffices for $\mathcal{P}$ to show in ZK:

$$\boldsymbol{T} \times \mathsf{LEFT} = \mathsf{RIGHT} \;\Big|\; \boldsymbol{T} := \begin{pmatrix} \boldsymbol{T}^{(1)} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{T}^{(2)} \end{pmatrix}$$

and $o_1 = 1, \boldsymbol{o}_{2:m+1} = \boldsymbol{x}, \boldsymbol{\ell}_{n+1:n+m+1} = \boldsymbol{1}, \boldsymbol{r}_{n+1:n+m+1} = (1, \boldsymbol{y})$. (The zero-checking outputs are checked separately via $\boldsymbol{p}$, as before.) The block-diagonal matrix $\boldsymbol{T}$ is private, but the compressed topology vector $(1, \chi, \chi^2, \ldots)\boldsymbol{T}$ can be constructed using ZK UROM, as before.

Finally, we fix the top-right entry of each topology matrix to be 1, so that every induced topology vector ends with 1. This technical normalization simplifies the use of UROM; see [YHH+24] for details.

B.4 Formal $\Pi_{\mathsf{G\text{-}ZKCPU}}$

We now provide the formal description of our protocol $\Pi_{\mathsf{G\text{-}ZKCPU}}$. $\Pi_{\mathsf{G\text{-}ZKCPU}}$ is defined in the $\mathcal{F}_{\mathsf{CPZK\text{-}UROM\text{-}SET\text{-}KVS}}$-hybrid model; see Figures 8 to 10. The protocol follows the topology-matrix/topology-vector paradigm of the tight ZK CPU protocol [YHH+24] and reuses its compression and boundary-hiding machinery essentially unchanged. The main modifications are listed as follows.

- *Additional committed objects.* In addition to unrestricted inputs and multiplication triples, $\mathcal{P}$ commits to (i) inputs restricted to the public set set and (ii) the read-write triples (k, v, s) associated with all kvs gates. Their correctness (set membership and key-value consistency) is enforced directly by $\mathcal{F}_{\mathsf{CPZK\text{-}UROM\text{-}SET\text{-}KVS}}$. See Items 3 and 4.
- *Universal-unit layout.* All committed values are arranged according to the fixed universal-unit profile $(u_{\mathsf{in}}, u_{\mathsf{set}}, u_{\mathsf{kvs}}, u_{\times})$, which induces a public execution layout and determines how per-instruction resources are packed. See Item 9e.
- *γ-dependent zero aggregation.* Each instruction may designate multiple zero wires. As in [YHH+24], these are aggregated into a single zero-check wire using a verifier challenge $\gamma \xleftarrow{\$} \mathbb{F}$ (a random linear combination with coefficients $(1, \gamma, \gamma^2, \ldots)$ in a fixed public order). Consequently, each instruction's topology matrix (and hence topology vector) is fully determined only after γ is sampled. See Items 7 and 8.

Protocol $\Pi_{\text{G-ZKCPU}}$, Part I

$\Pi_{\text{G-ZKCPU}}$ runs with a prover $\mathcal{P}$, a verifier $\mathcal{V}$, and is parameterized by a field $\mathbb{F}$, a register dimension $m \in \mathbb{N}$, a public set $\mathsf{set} \subseteq \mathbb{F}$, a positive integer B, and B instructions (per Definition 4.1) $\mathcal{I}_1, \ldots, \mathcal{I}_B$ (with each $\mathcal{I}_i.\mathsf{set} = \mathsf{set}$), an initial register state $\mathbf{st}^{(0)} \in \mathbb{F}^m$, a final register state $\mathbf{st}^{(\mathsf{final})} \in \mathbb{F}^m$, an initial global key-value map $\mathcal{M}$, and a universal-unit profile $(u_{\mathsf{in}}, u_{\mathsf{set}}, u_{\mathsf{kvs}}, u_{\times})$.

For each $i \in [B]$, instruction $\mathcal{I}_i$ has $n_{\mathsf{in}}^{(i)}$ unrestricted auxiliary inputs, $n_{\mathsf{set}}^{(i)}$ inputs drawn from set, $n_{\mathsf{kvs}}^{(i)}$ kvs gates, and $n_{\times}^{(i)}$ multiplication gates. Its size $n^{(i)}$ is induced by the universal-unit profile as defined in Equation (1). W.l.o.g., we assume that $\mathcal{I}_i$ is fully packed under the profile, i.e., $n_{\mathsf{in}}^{(i)} = n^{(i)} \cdot u_{\mathsf{in}}, n_{\mathsf{set}}^{(i)} = n^{(i)} \cdot u_{\mathsf{set}}, n_{\mathsf{kvs}}^{(i)} = n^{(i)} \cdot u_{\mathsf{kvs}}, n_{\times}^{(i)} = n^{(i)} \cdot u_{\times} - m - 2$. (This assumption is without loss of generality, as any instruction that uses fewer constraints of a given type can be padded with dummy operations.)

For each $i \in [B]$, instruction $\mathcal{I}_i$ induces a topology matrix $\boldsymbol{T}^{(i)}$ under the fixed universal-unit layout, which encodes all linear wiring constraints relating unrestricted inputs, set inputs, multiplication outputs, and key-value store outputs to the corresponding constrained wires (multiplication inputs and key-value store inputs). The matrix $\boldsymbol{T}^{(i)}$ has $n_{\mathsf{row}}^{(i)} := n^{(i)}(2u_{\times} + 2u_{\mathsf{kvs}})$ rows and $n_{\mathsf{column}}^{(i)} := n^{(i)}(u_{\mathsf{in}} + u_{\mathsf{set}} + u_{\mathsf{kvs}} + u_{\times})$ columns. All rows of $\boldsymbol{T}^{(i)}$ are fully determined by $\mathcal{I}_i$ and the layout, except for a single row corresponding to the global zero-checking constraint, which aggregates the designated zero wires of $\mathcal{I}_i$ and is defined only after $\mathcal{V}$ samples the challenge γ (in Item 6). See Appendix B.3 for details on the topology matrix.

$\Pi_{\text{G-ZKCPU}}$ proceeds:

1. $\mathcal{P}$ **claims the size of the execution.** On receiving $(\texttt{Prove}, \tau, i_1, \ldots, i_\tau, \mathbf{in}_1, \ldots, \mathbf{in}_\tau)$, $\mathcal{P}$ computes and sends $n := \sum_{j \in [\tau]} n^{(i_j)}$.
2. $\mathcal{P}$ **commits to unrestricted inputs and multiplication triples.** Let n be as in Item 1 and define $N_{\mathsf{in}} := n \cdot u_{\mathsf{in}}$ and $N_{\times} := n \cdot u_{\times}$. $\mathcal{P}$ constructs the following vectors: $\mathbf{in} \in \mathbb{F}^{N_{\mathsf{in}}}$ and $\boldsymbol{\ell}, \boldsymbol{r}, \boldsymbol{o} \in \mathbb{F}^{N_{\times}}$.

   $\mathcal{P}$ sets $\mathbf{in}, \boldsymbol{\ell}, \boldsymbol{r}, \boldsymbol{o}$ to be empty. Let the m registers be $\mathbf{st}_j$ after executing the first j instructions. Note, $\mathbf{st}_0 = \mathbf{st}^{(0)}$. For each $j \in [\tau]$ in sequence:

   (a) **Extract the unrestricted inputs.** Parse the received input vector as $\mathbf{in}_j = \mathbf{in}_j^{(\mathsf{set})} \parallel \mathbf{in}_j^{(\mathsf{normal})}$, where $|\mathbf{in}_j^{(\mathsf{set})}| = n_{\mathsf{set}}^{(i_j)}$ and $|\mathbf{in}_j^{(\mathsf{normal})}| = n_{\mathsf{in}}^{(i_j)}$. Append $\mathbf{in} := \mathbf{in} \| \mathbf{in}_j^{(\mathsf{normal})}$.

   (b) **Encode the constant-1 wire.** Let $\boldsymbol{\ell} := \boldsymbol{\ell} \| 1$, $\boldsymbol{r} := \boldsymbol{r} \| 1$, $\boldsymbol{o} := \boldsymbol{o} \| 1$.

   (c) **Encode register inputs as multiplications.** For each $k \in [m]$, let $\boldsymbol{\ell} := \boldsymbol{\ell} \| 1$, $\boldsymbol{r} := \boldsymbol{r} \| \mathbf{st}_{j-1}[k]$, $\boldsymbol{o} := \boldsymbol{o} \| \mathbf{st}_{j-1}[k]$.

   (d) **Encode multiplication gates of $\mathcal{I}_{i_j}.\mathcal{C}$.** For each multiplication gate in $\mathcal{I}_{i_j}.\mathcal{C}$ (in the same order induced by the packed layout underlying $\boldsymbol{T}^{(i_j)}$), let $(\mathsf{val}^{(\ell)}, \mathsf{val}^{(r)}, \mathsf{val}^{(o)})$ be its left/right/output wire values obtained by evaluating $\mathcal{I}_{i_j}.\mathcal{C}(\mathbf{st}_{j-1} \parallel \mathbf{in}_j)$, and append $\boldsymbol{\ell} := \boldsymbol{\ell} \| \mathsf{val}^{(\ell)}$, $\boldsymbol{r} := \boldsymbol{r} \| \mathsf{val}^{(r)}$, $\boldsymbol{o} := \boldsymbol{o} \| \mathsf{val}^{(o)}$.

   (e) **Encode the global zero-checking multiplication.** Let $\boldsymbol{\ell} := \boldsymbol{\ell} \| 1$, $\boldsymbol{r} := \boldsymbol{r} \| 0$, $\boldsymbol{o} := \boldsymbol{o} \| 0$.

   $\mathcal{P}$ generates fresh cids as $\mathbf{cid}^{(\mathsf{in})}$ of length N_{in} and $\mathbf{cid}^{(\ell/r/o)}$ each of length $N_{\times}$. For each $t \in [N_{\mathsf{in}}]$, $\mathcal{P}$ sends $(\texttt{Commit}, \mathsf{cid}_t^{(\mathsf{in})}, \mathsf{in}_t)$ to $\mathcal{F}_{\text{CPZK-UROM-SET-KVS}}$. For each $t \in [N_{\times}]$, $\mathcal{P}$ sends $(\texttt{Commit}, \mathsf{cid}_t^{(\ell/r/o)}, (\ell/r/o)_t)$ to $\mathcal{F}_{\text{CPZK-UROM-SET-KVS}}$. Note, $\mathcal{V}$ receives $(\texttt{commit}, \mathsf{cid}_t^{(\mathsf{in})})$ and $(\texttt{commit}, \mathsf{cid}_t^{(\ell/r/o)})$ from $\mathcal{F}_{\text{CPZK-UROM-SET-KVS}}$.

Figure 8: Protocol $\Pi_{\text{G-ZKCPU}}$ (Part I) in the $\mathcal{F}_{\text{CPZK-UROM-SET-KVS}}$-hybrid model.

Protocol $\Pi_{\mathsf{G\text{-}ZKCPU}}$, Part II

3. $\mathcal{P}$ **initializes SET on the public set and commits to the restricted set inputs.** Let n be as in Item 1 and define $N_{\mathsf{set}} := n \cdot u_{\mathsf{set}}$.

   Initialize SET on the public set set. Let $\mathsf{set} = \{y_1, \ldots, y_{|\mathsf{set}|}\}$ be the public set. $\mathcal{P}$ sends $(\texttt{InitSET}, y_1, \ldots, y_{|\mathsf{set}|})$ to $\mathcal{F}_{\mathsf{CPZK\text{-}UROM\text{-}SET\text{-}KVS}}$, and $\mathcal{V}$ receives $(\texttt{initset}, \widetilde{y_1}, \ldots, \widetilde{y_{|\mathsf{set}|}})$. If $\mathsf{set} \neq \{\widetilde{y_1}, \ldots, \widetilde{y_{|\mathsf{set}|}}\}$, then $\mathcal{V}$ outputs $(\texttt{prove}, \texttt{false}, n)$ and aborts.

   Prove membership of per-execution set inputs. $\mathcal{P}$ constructs a length-N_{set} vector $\boldsymbol{b}$ by, for each $j \in [\tau]$ in order, parsing the already-received input as $\mathbf{in}_j = \mathbf{in}_j^{(\mathsf{set})} \| \mathbf{in}_j^{(\mathsf{normal})}$ with $|\mathbf{in}_j^{(\mathsf{set})}| = n_{\mathsf{set}}^{(i_j)}$, and setting $\boldsymbol{b} := \mathbf{in}_1^{(\mathsf{set})} \| \cdots \| \mathbf{in}_\tau^{(\mathsf{set})}$. $\mathcal{P}$ generates N_{set} fresh identifiers $\mathbf{cid}^{(\mathsf{set})}$ and sends $(\texttt{ReadSET}, \mathbf{cid}^{(\mathsf{set})}, \boldsymbol{b})$ to $\mathcal{F}_{\mathsf{CPZK\text{-}UROM\text{-}SET\text{-}KVS}}$, followed by $(\texttt{CheckSET})$. If $\mathcal{V}$ does not receive $(\texttt{checkset}, \texttt{honest})$ from $\mathcal{F}_{\mathsf{CPZK\text{-}UROM\text{-}SET\text{-}KVS}}$, then $\mathcal{V}$ outputs $(\texttt{prove}, \texttt{false}, n)$ and aborts. Note, $\mathcal{V}$ receives $(\texttt{readset}, \mathbf{cid}^{(\mathsf{set})})$ from $\mathcal{F}_{\mathsf{CPZK\text{-}UROM\text{-}SET\text{-}KVS}}$.

4. $\mathcal{P}$ **initializes KVS on the public map and commits to all KVS triples.** Let n be as in Item 1 and define $N_{\mathsf{kvs}} := n \cdot u_{\mathsf{kvs}}$.

   Initialize KVS on the public map $\mathcal{M}$. Let $\mathcal{M} = \{(\mathsf{key}_1, \mathsf{val}_1), \ldots, (\mathsf{key}_{|\mathcal{M}|}, \mathsf{val}_{|\mathcal{M}|})\}$ be the public initial key-value map, where all key_i are distinct. $\mathcal{P}$ sends $(\texttt{InitKVS}, \mathsf{key}_1, \ldots, \mathsf{key}_{|\mathcal{M}|}, \mathsf{val}_1, \ldots, \mathsf{val}_{|\mathcal{M}|})$ to $\mathcal{F}_{\mathsf{CPZK\text{-}UROM\text{-}SET\text{-}KVS}}$, and $\mathcal{V}$ receives $(\texttt{initkvs}, \widetilde{\mathsf{key}_1}, \ldots, \widetilde{\mathsf{key}_{|\mathcal{M}|}}, \widetilde{\mathsf{val}_1}, \ldots, \widetilde{\mathsf{val}_{|\mathcal{M}|}})$. If $\mathcal{M} \neq \{(\widetilde{\mathsf{key}_1}, \widetilde{\mathsf{val}_1}), \ldots, (\widetilde{\mathsf{key}_{|\mathcal{M}|}}, \widetilde{\mathsf{val}_{|\mathcal{M}|}})\}$, then $\mathcal{V}$ outputs $(\texttt{prove}, \texttt{false}, n)$ and aborts.

   Encode kvs *gates of the execution and prove consistency.* $\mathcal{P}$ sets $\boldsymbol{k}, \boldsymbol{v}, \boldsymbol{s}$ to be empty. Let the m registers be $\mathbf{st}_j$ after executing the first j instructions. For each $j \in [\tau]$ in sequence, consider instruction $\mathcal{I}_{i_j}$. For each kvs gate in $\mathcal{I}_{i_j}.\mathcal{C}$ (in the same order induced by the packed layout underlying $\boldsymbol{T}^{(i_j)}$), let $(\mathsf{idx}, \mathsf{val}_{\mathsf{new}}, \mathsf{val}_{\mathsf{old}})$ be its index, write-value, and read-value wires obtained by evaluating $\mathcal{I}_{i_j}.\mathcal{C}(\mathbf{st}_{j-1} \,\|\, \mathbf{in}_j)$, and append $\boldsymbol{k} := \boldsymbol{k} \| \mathsf{idx}$, $\boldsymbol{v} := \boldsymbol{v} \| \mathsf{val}_{\mathsf{old}}$, and $\boldsymbol{s} := \boldsymbol{s} \| \mathsf{val}_{\mathsf{new}}$. After processing all instructions, the vectors $\boldsymbol{k}, \boldsymbol{v}, \boldsymbol{s}$ have length N_{kvs}. $\mathcal{P}$ generates fresh identifiers $\mathbf{cid}^{(k)}, \mathbf{cid}^{(v)}, \mathbf{cid}^{(s)}$ and sends $(\texttt{ReadWriteKVS}, \mathbf{cid}^{(k)}, \boldsymbol{k}, \mathbf{cid}^{(v)}, \boldsymbol{v}, \mathbf{cid}^{(s)}, \boldsymbol{s})$ to $\mathcal{F}_{\mathsf{CPZK\text{-}UROM\text{-}SET\text{-}KVS}}$, followed by $(\texttt{CheckKVS})$. If $\mathcal{V}$ does not receive $(\texttt{checkkvs}, \texttt{honest})$, then $\mathcal{V}$ outputs $(\texttt{prove}, \texttt{false}, n)$ and aborts. Note, $\mathcal{V}$ receives $(\texttt{readwritekvs}, \mathbf{cid}^{(k)}, \mathbf{cid}^{(v)}, \mathbf{cid}^{(s)})$ from $\mathcal{F}_{\mathsf{CPZK\text{-}UROM\text{-}SET\text{-}KVS}}$.

5. $\mathcal{P}$ **commits to $\boldsymbol{p}$, $\mathbf{id}$ marking instruction boundaries and identities.** $\mathcal{P}$ constructs two length-n vectors, where (a) $\boldsymbol{p}$ is a boundary string with a single 1 at the end of each instruction block, and (b) $\mathbf{id} \in [B]^n$ records the instruction index on each position, such that $\mathsf{Filter}(\boldsymbol{p}, \mathbf{id}) = \{i_1, \ldots, i_\tau\}$:

$$\boldsymbol{p} = (\overbrace{0, \ldots, 0, 1}^{n^{(i_1)}}, \overbrace{0, \ldots, 0, 1}^{n^{(i_2)}}, \ldots, \overbrace{0, \ldots, 0, 1}^{n^{(i_\tau)}}), \quad \mathbf{id} = (\overbrace{i_1, \ldots, i_1}^{n^{(i_1)}}, \overbrace{i_2, \ldots, i_2}^{n^{(i_2)}}, \ldots, \overbrace{i_\tau, \ldots, i_\tau}^{n^{(i_\tau)}}).$$

   $\mathcal{P}$ generates $2n$ fresh cids as $\mathbf{cid}^{(p/\mathsf{id})}$ and, $\forall j \in [n]$, sends $(\texttt{Commit}, \mathsf{cid}_j^{(p/\mathsf{id})}, p_j/\mathsf{id}_j)$ to $\mathcal{F}_{\mathsf{CPZK\text{-}UROM\text{-}SET\text{-}KVS}}$. Note, $\mathcal{V}$ receives $(\texttt{commit}, \mathsf{cid}_j^{(p/\mathsf{id})})$ from $\mathcal{F}_{\mathsf{CPZK\text{-}UROM\text{-}SET\text{-}KVS}}$.

Figure 9: Protocol $\Pi_{\mathsf{G\text{-}ZKCPU}}$ (Part II) in the $\mathcal{F}_{\mathsf{CPZK\text{-}UROM\text{-}SET\text{-}KVS}}$-hybrid model.

Protocol $\Pi_{\text{G-ZKCPU}}$, Part III

6. **$\mathcal{V}$ issues uniform challenges $\chi, \gamma \xleftarrow{\$} \mathbb{F}$ to (i) compress topology matrices to topology vectors and (ii) aggregate zero checks.** $\mathcal{V}$ on receiving n from $\mathcal{P}$, $\mathcal{V}$ waits for all commitment identifiers produced in the previous steps from $\mathcal{F}_{\text{CPZK-UROM-SET-KVS}}$. $\mathcal{V}$ samples $\chi, \gamma \xleftarrow{\$} \mathbb{F}$ uniformly at random and sends (χ, γ) to $\mathcal{P}$. Here, χ is used exactly as in the tight ZK CPU protocol to derive topology vectors from topology matrices, while γ is used to define, for each instruction $\mathcal{I}_i$, a single aggregated zero-check wire as a random linear combination of the designated zero wires in $\mathcal{I}_i.\mathsf{zero}$ with coefficients $(1, \gamma, \gamma^2, \ldots)$ in a fixed public order.

7. **$\mathcal{P}$ and $\mathcal{V}$ initialize ZK UROM using B topology vectors.** For each $i \in [B]$, let $\boldsymbol{T}^{(i)}(\gamma)$ denote the *fully determined* topology matrix induced by instruction $\mathcal{I}_i$ under the fixed universal-unit layout, after instantiating the aggregated zero-check row using γ, with $n^{(i)}_{\mathsf{row}}$ rows and $n^{(i)}_{\mathsf{column}}$ columns. $\mathcal{P}$ and $\mathcal{V}$ compute the corresponding topology vector

$$\mathbf{tv}^{(i)} := (1, \chi, \ldots, \chi^{n^{(i)}_{\mathsf{row}}-1}) \cdot \boldsymbol{T}^{(i)}(\gamma) \ \in \ \mathbb{F}^{n^{(i)}_{\mathsf{column}}}.$$

   For each $i \in [B]$ and each $j \in [n^{(i)}_{\mathsf{column}}]$, $\mathcal{P}$ generates a fresh identifier $u^{(i)}_j$ and sends $(\texttt{Linear}, u^{(i)}_j, \mathsf{tv}^{(i)}_j)$ to $\mathcal{F}_{\text{CPZK-UROM-SET-KVS}}$. $\mathcal{V}$ receives $(\texttt{linear}, u^{(i)}_j, \widetilde{\mathsf{tv}}^{(i)}_j)$ and, if $\widetilde{\mathsf{tv}}^{(i)}_j \neq \mathsf{tv}^{(i)}_j$, outputs $(\texttt{prove}, \texttt{false}, n)$ and aborts.

   Finally, $\mathcal{P}$ sends $(\texttt{InitUROM}, \boldsymbol{u}^{(1)}, \ldots, \boldsymbol{u}^{(B)})$ to $\mathcal{F}_{\text{CPZK-UROM-SET-KVS}}$. If $\mathcal{V}$ does not receive the corresponding $(\texttt{initurom}, \boldsymbol{u}^{(1)}, \ldots, \boldsymbol{u}^{(B)})$ message, then $\mathcal{V}$ outputs $(\texttt{prove}, \texttt{false}, n)$ and aborts.

8. **$\mathcal{P}$ uses committed p, id to set the UROM and read the concatenated topology vector.** Let $u_{\mathsf{col}} := u_{\mathsf{in}} + u_{\mathsf{set}} + u_{\mathsf{kvs}} + u_{\times}$ and $n_{\mathsf{col}} := n \cdot u_{\mathsf{col}}$.

   Commit a shared zero. $\mathcal{P}$ and $\mathcal{V}$ use $\texttt{Linear}$ in the $\mathcal{F}_{\text{CPZK-UROM-SET-KVS}}$-hybrid model to generate a commitment $\mathsf{cid}^{(\mathsf{zero})}$ to 0.

   Set the UROM program on the column timeline. Define two length-n_{col} identifier vectors $\mathbf{cid}^{(p,\mathsf{col})}$ and $\mathbf{cid}^{(\mathsf{id},\mathsf{col})}$ by, for each $t \in [n]$,

$$\mathbf{cid}^{(p,\mathsf{col})}\big[(t-1)u_{\mathsf{col}} + 1 : tu_{\mathsf{col}}\big] \ := \ (\underbrace{\mathsf{cid}^{(\mathsf{zero})}, \ldots, \mathsf{cid}^{(\mathsf{zero})}}_{u_{\mathsf{col}}-1}, \mathsf{cid}^{(p)}_t),$$

$$\mathbf{cid}^{(\mathsf{id},\mathsf{col})}\big[(t-1)u_{\mathsf{col}} + 1 : tu_{\mathsf{col}}\big] \ := \ (\underbrace{\mathsf{cid}^{(\mathsf{id})}_t, \ldots, \mathsf{cid}^{(\mathsf{id})}_t}_{u_{\mathsf{col}}}).$$

   $\mathcal{P}$ sends $(\texttt{SetProg}, \mathbf{cid}^{(p,\mathsf{col})}, \mathbf{cid}^{(\mathsf{id},\mathsf{col})})$ to $\mathcal{F}_{\text{CPZK-UROM-SET-KVS}}$. Upon receiving $(\texttt{setprog}, \widetilde{\mathbf{cid}}^{(p,\mathsf{col})}, \widetilde{\mathbf{cid}}^{(\mathsf{id},\mathsf{col})})$, $\mathcal{V}$ aborts with $(\texttt{prove}, \texttt{false}, n)$ if $\widetilde{\mathbf{cid}}^{(p,\mathsf{col})} \neq \mathbf{cid}^{(p,\mathsf{col})}$ or $\widetilde{\mathbf{cid}}^{(\mathsf{id},\mathsf{col})} \neq \mathbf{cid}^{(\mathsf{id},\mathsf{col})}$.

   Read UROM and check honesty. $\mathcal{P}$ constructs the length-n_{col} vector $\boldsymbol{d} := \mathbf{tv}^{(i_1)} \| \mathbf{tv}^{(i_2)} \| \cdots \| \mathbf{tv}^{(i_\tau)} \in \mathbb{F}^{n_{\mathsf{col}}}$, generates fresh identifiers $\mathbf{cid}^{(d)}$ of length n_{col}, and sends $(\texttt{ReadUROM}, \mathbf{cid}^{(d)}, \boldsymbol{d})$ to $\mathcal{F}_{\text{CPZK-UROM-SET-KVS}}$, followed by $(\texttt{CheckUROM})$. If $\mathcal{V}$ does not receive $(\texttt{checkurom}, \texttt{honest})$, then $\mathcal{V}$ outputs $(\texttt{prove}, \texttt{false}, n)$ and aborts. Note, $\mathcal{V}$ receives $(\texttt{readurom}, \mathbf{cid}^{(d)})$ from $\mathcal{F}_{\text{CPZK-UROM-SET-KVS}}$.

Figure 10: Protocol $\Pi_{\text{G-ZKCPU}}$ (Part III) in the $\mathcal{F}_{\text{CPZK-UROM-SET-KVS}}$-hybrid model.

Protocol $\Pi_{\text{G-ZKCPU}}$, Part IV

9. **$\mathcal{V}$ checks multiplication correctness, zero-check consistency, anchors to public init/final states, and verifies all linear constraints.**

   (a) **$\mathcal{V}$ checks that the committed multiplication triples are well-formed.** $\mathcal{P}$ sends $(\texttt{Check}, \mathcal{C}^{\mathsf{mul}}, \mathbf{cid}^{(\ell)}, \mathbf{cid}^{(r)}, \mathbf{cid}^{(o)})$ to $\mathcal{F}_{\text{CPZK-UROM-SET-KVS}}$, where $\mathcal{C}^{\mathsf{mul}}$ is a circuit with inputs $\boldsymbol{\ell}, \boldsymbol{r}, \boldsymbol{o} \in \mathbb{F}^{N_\times}$ that outputs $\ell_t \cdot r_t - o_t$ for each $t \in [N_\times]$. If $\mathcal{V}$ does not receive $(\texttt{check}, \mathcal{C}^{\mathsf{mul}}, \mathbf{cid}^{(\ell)}, \mathbf{cid}^{(r)}, \mathbf{cid}^{(o)}, \texttt{true})$ from $\mathcal{F}_{\text{CPZK-UROM-SET-KVS}}$, then $\mathcal{V}$ outputs $(\texttt{prove}, \texttt{false}, n)$ and aborts.

   (b) **$\mathcal{V}$ checks that the first $m{+}1$ multiplication outputs encode $(1\|\mathbf{st}^{(0)})$.** For each $t \in [m{+}1]$, $\mathcal{P}$ sends $(\texttt{Open}, \mathsf{cid}_t^{(o)})$ to $\mathcal{F}_{\text{CPZK-UROM-SET-KVS}}$. For each $t \in [m{+}1]$, $\mathcal{V}$ receives $(\texttt{open}, \mathsf{cid}_t^{(o)}, \widetilde{o}_t)$ from $\mathcal{F}_{\text{CPZK-UROM-SET-KVS}}$. If $\widetilde{o}_1 \neq 1$ or there exists $k \in [m]$ such that $\widetilde{o}_{k+1} \neq \mathbf{st}^{(0)}[k]$, then $\mathcal{V}$ outputs $(\texttt{prove}, \texttt{false}, n)$ and aborts.

   (c) **$\mathcal{P}$ appends public final-state tail commitments to $\mathbf{cid}^{(\ell)}$ and $\mathbf{cid}^{(r)}$.** Let $N_\times := n \cdot u_\times$. $\mathcal{P}$ and $\mathcal{V}$ extend the existing identifier vectors $\mathbf{cid}^{(\ell)}$ and $\mathbf{cid}^{(r)}$ (from Item 2) by $m{+}1$ fresh identifiers as follows.

   Append $(m{+}1)$ ones to $\mathbf{cid}^{(\ell)}$. For each $t \in [m{+}1]$, $\mathcal{P}$ samples a fresh identifier $\mathsf{cid}_{N_\times+t}^{(\ell)}$ and sends $(\texttt{Linear}, \mathsf{cid}_{N_\times+t}^{(\ell)}, 1)$ to $\mathcal{F}_{\text{CPZK-UROM-SET-KVS}}$. For each $t \in [m{+}1]$, $\mathcal{V}$ aborts with $(\texttt{prove}, \texttt{false}, n)$ if $\mathcal{V}$ does not receive $(\texttt{linear}, \mathsf{cid}_{N_\times+t}^{(\ell)}, 1)$ from $\mathcal{F}_{\text{CPZK-UROM-SET-KVS}}$.

   Append $(1\|\mathbf{st}^{(\mathsf{final})})$ to $\mathbf{cid}^{(r)}$. $\mathcal{P}$ samples a fresh identifier $\mathsf{cid}_{N_\times+1}^{(r)}$ and sends $(\texttt{Linear}, \mathsf{cid}_{N_\times+1}^{(r)}, 1)$ to $\mathcal{F}_{\text{CPZK-UROM-SET-KVS}}$. $\mathcal{V}$ aborts with $(\texttt{prove}, \texttt{false}, n)$ if $\mathcal{V}$ does not receive $(\texttt{linear}, \mathsf{cid}_{N_\times+1}^{(r)}, 1)$ from $\mathcal{F}_{\text{CPZK-UROM-SET-KVS}}$. For each $k \in [m]$, $\mathcal{P}$ samples a fresh identifier $\mathsf{cid}_{N_\times+1+k}^{(r)}$ and sends $(\texttt{Linear}, \mathsf{cid}_{N_\times+1+k}^{(r)}, \mathbf{st}^{(\mathsf{final})}[k])$ to $\mathcal{F}_{\text{CPZK-UROM-SET-KVS}}$. For each $k \in [m]$, $\mathcal{V}$ receives $(\texttt{linear}, \mathsf{cid}_{N_\times+1+k}^{(r)}, \widetilde{z}_k)$ from $\mathcal{F}_{\text{CPZK-UROM-SET-KVS}}$ and aborts with $(\texttt{prove}, \texttt{false}, n)$ if $\widetilde{z}_k \neq \mathbf{st}^{(\mathsf{final})}[k]$.

   (d) **$\mathcal{V}$ checks that the zero-checking multiplication output is zero at instruction boundaries.** $\mathcal{P}$ sends $(\texttt{Check}, \mathcal{C}^{\mathsf{op}}, \mathbf{cid}^{(o)}, \mathbf{cid}^{(p)})$ to $\mathcal{F}_{\text{CPZK-UROM-SET-KVS}}$, where $\mathcal{C}^{\mathsf{op}}$ is a circuit with inputs $\boldsymbol{o} \in \mathbb{F}^{N_\times}$ and $\boldsymbol{p} \in \mathbb{F}^n$ that outputs $p_t \cdot o_{t \cdot u_\times}$ for each $t \in [n]$. If $\mathcal{V}$ does not receive $(\texttt{check}, \mathcal{C}^{\mathsf{op}}, \mathbf{cid}^{(o)}, \mathbf{cid}^{(p)}, \texttt{true})$ from $\mathcal{F}_{\text{CPZK-UROM-SET-KVS}}$, then $\mathcal{V}$ outputs $(\texttt{prove}, \texttt{false}, n)$ and aborts.

   (e) **$\mathcal{V}$ verifies all remaining linear wiring constraints via a single compressed check.**

   Let $u_{\mathsf{col}} := u_{\mathsf{in}} + u_{\mathsf{set}} + u_{\mathsf{kvs}} + u_\times$, $n_{\mathsf{col}} := n \cdot u_{\mathsf{col}}$, $u_{\mathsf{row}} := 2u_{\mathsf{kvs}} + 2u_\times$, and $n_{\mathsf{row}} := n \cdot u_{\mathsf{row}}$.
   $\mathcal{P}$ sends $(\texttt{Check}, \mathcal{C}^{\mathsf{lin}}, \mathbf{cid}^{(\mathsf{in})}, \mathbf{cid}^{(\mathsf{set})}, \mathbf{cid}^{(v)}, \mathbf{cid}^{(o)}, \mathbf{cid}^{(k)}, \mathbf{cid}^{(s)}, \mathbf{cid}^{(\ell)}, \mathbf{cid}^{(r)}, \mathbf{cid}^{(d)}, \mathbf{cid}^{(p)})$ to $\mathcal{F}_{\text{CPZK-UROM-SET-KVS}}$, where $\mathcal{C}^{\mathsf{lin}}$ is a circuit with inputs $\mathbf{in} \in \mathbb{F}^{n \cdot u_{\mathsf{in}}}$, $\boldsymbol{b} \in \mathbb{F}^{n \cdot u_{\mathsf{set}}}$, $\boldsymbol{v} \in \mathbb{F}^{n \cdot u_{\mathsf{kvs}}}$, $\boldsymbol{o} \in \mathbb{F}^{n \cdot u_\times}$, $\boldsymbol{k} \in \mathbb{F}^{n \cdot u_{\mathsf{kvs}}}$, $\boldsymbol{s} \in \mathbb{F}^{n \cdot u_{\mathsf{kvs}}}$, $\boldsymbol{\ell} \in \mathbb{F}^{n \cdot u_\times + m + 1}$, $\boldsymbol{r} \in \mathbb{F}^{n \cdot u_\times + m + 1}$, $\boldsymbol{d} \in \mathbb{F}^{n_{\mathsf{col}}}$, $\boldsymbol{p} \in \mathbb{F}^n$, that outputs the single field element $\mathsf{sum}_L - \mathsf{sum}_R$ defined as follows.

   Let $\boldsymbol{p}^{(\mathsf{col})} \in \mathbb{F}^{n_{\mathsf{col}}}$ be the column-expansion of $\boldsymbol{p}$: for each $t \in [n]$, the block $\boldsymbol{p}^{(\mathsf{col})}[(t-1)u_{\mathsf{col}} + 1 : tu_{\mathsf{col}}]$ equals $(0, \ldots, 0, p_t)$. Let $\boldsymbol{\sigma} := \mathsf{Expand}_1(\boldsymbol{p}^{(\mathsf{col})}, \chi) \in \mathbb{F}^{n_{\mathsf{col}}}$.

   Left-hand sum. For each $t \in [n]$, define the per-unit slices

   $$\mathbf{in}^{(t)} := \mathbf{in}[u_{\mathsf{in}}(t-1) + 1 : u_{\mathsf{in}}t], \quad \boldsymbol{b}^{(t)} := \boldsymbol{b}[u_{\mathsf{set}}(t-1) + 1 : u_{\mathsf{set}}t]$$

   $$\boldsymbol{v}^{(t)} := \boldsymbol{v}[u_{\mathsf{kvs}}(t-1) + 1 : u_{\mathsf{kvs}}t], \quad \boldsymbol{o}^{(t)} := \boldsymbol{o}[u_\times(t-1) + 1 : u_\times t]$$

   Set $\boldsymbol{L}_t := \mathbf{in}^{(t)} \| \boldsymbol{b}^{(t)} \| \boldsymbol{v}^{(t)} \| \boldsymbol{o}^{(t)} \in \mathbb{F}^{u_{\mathsf{col}}}$. Let $\mathsf{LEFT} := \boldsymbol{L}_1 \| \cdots \| \boldsymbol{L}_n \in \mathbb{F}^{n_{\mathsf{col}}}$ and define $\mathsf{sum}_L := \langle \boldsymbol{\sigma} \odot \boldsymbol{d},\ \mathsf{LEFT} \rangle$.

   Right-hand sum. For each $t \in [n]$, define the per-unit slices

   $$\boldsymbol{k}^{(t)} := \boldsymbol{k}[u_{\mathsf{kvs}}(t-1) + 1 : u_{\mathsf{kvs}}t], \quad \boldsymbol{s}^{(t)} := \boldsymbol{s}[u_{\mathsf{kvs}}(t-1) + 1 : u_{\mathsf{kvs}}t]$$

   $$\boldsymbol{\ell}^{(t)} := \boldsymbol{\ell}[u_\times(t-1) + m + 2 : u_\times(t-1) + m + 1 + u_\times], \quad \boldsymbol{r}^{(t)} := \boldsymbol{r}[u_\times(t-1) + m + 2 : u_\times(t-1) + m + 1 + u_\times]$$

   Set $\boldsymbol{R}_t := \boldsymbol{k}^{(t)} \| \boldsymbol{s}^{(t)} \| \boldsymbol{\ell}^{(t)} \| \boldsymbol{r}^{(t)} \in \mathbb{F}^{u_{\mathsf{row}}}$. Let $\mathsf{RIGHT} := \boldsymbol{R}_1 \| \cdots \| \boldsymbol{R}_n \in \mathbb{F}^{n_{\mathsf{row}}}$ and define $\mathsf{sum}_R := \big\langle (1, \chi, \ldots, \chi^{n_{\mathsf{row}}-1}),\ \mathsf{RIGHT} \big\rangle$.

   If $\mathcal{V}$ receives $(\texttt{check}, \mathcal{C}^{\mathsf{lin}}, \mathbf{cid}^{(\mathsf{in})}, \mathbf{cid}^{(\mathsf{set})}, \mathbf{cid}^{(v)}, \mathbf{cid}^{(o)}, \mathbf{cid}^{(k)}, \mathbf{cid}^{(s)}, \mathbf{cid}^{(\ell)}, \mathbf{cid}^{(r)}, \mathbf{cid}^{(d)}, \mathbf{cid}^{(p)}, \texttt{true})$ from $\mathcal{F}_{\text{CPZK-UROM-SET-KVS}}$, then $\mathcal{V}$ outputs $(\texttt{prove}, \texttt{true}, n)$; otherwise, $\mathcal{V}$ outputs $(\texttt{prove}, \texttt{false}, n)$ and aborts.

Figure 11: Protocol $\Pi_{\text{G-ZKCPU}}$ (Part IV) in the $\mathcal{F}_{\text{CPZK-UROM-SET-KVS}}$-hybrid model.

- *Register passing via auxiliary multiplications.* Register values (including the constant-1 wire) are propagated across instructions using auxiliary multiplication gates, following the register-passing construction described earlier. See Items 9c and 9e.

We emphasize that these changes do not alter the core linear-algebraic structure: after fixing (χ, γ), $\mathcal{V}$ still checks all wire soldering constraints via a single compressed linear relation between the committed values and the (shifted) concatenated topology vectors read from ZK UROM.

The formal theorem is stated as follows.

Theorem B.2. *Protocol $\Pi_{\mathsf{G\text{-}ZKCPU}}$ (Figures 8 to 11) UC-realizes $\mathcal{F}_{\mathsf{G\text{-}ZKCPU}}$ (Figure 3) in the $\mathcal{F}_{\mathsf{CPZK\text{-}UROM\text{-}SET\text{-}KVS}}$-hybrid model (Figure 7) with perfect zero-knowledge and soundness error at most*

$$\frac{2n(u_{\mathsf{kvs}} + u_{\times}) - B - 1 + \sum_{i=1}^{B} |\mathcal{I}_i.\mathsf{zero}|}{|\mathbb{F}|},$$

against any static, computationally unbounded adversary.

Proof. We need to show completeness (trivial, omitted); soundness (constructing the simulator $\mathcal{S}$ for $\mathcal{P}^*$); and zero knowledge (constructing the simulator $\mathcal{S}$ for $\mathcal{V}^*$).

Intuition (Proof Sketch). The security proof of $\Pi_{\mathsf{G\text{-}ZKCPU}}$ follows the same high-level structure as the tight ZK CPU protocol [YHH+24]. Perfect zero knowledge is immediate in the $\mathcal{F}_{\mathsf{CPZK\text{-}UROM\text{-}SET\text{-}KVS}}$-hybrid model: the verifier learns only commitment identifiers and boolean check outcomes (always `true` if $\mathcal{P}$ is honest), while all semantic constraints — such as multiplication correctness, set membership, and key-value store consistency — are enforced within the ideal-hybrid functionality $\mathcal{F}_{\mathsf{CPZK\text{-}UROM\text{-}SET\text{-}KVS}}$. Soundness is governed by a single algebraic invariant. If the prover's committed execution does not correspond to a valid execution of $\mathcal{F}_{\mathsf{G\text{-}ZKCPU}}$, then the linear wiring constraints encoded by the topology matrices are violated (conditioned on all other checks being satisfied). The verifier checks these constraints only through a random linear compression using a challenge χ; by the Schwartz-Zippel lemma, an invalid execution passes this check with negligible probability. The only additional step in the generalized setting is the aggregation of multiple zero wires using an independent challenge γ, which contributes an additional negligible soundness term. Overall, the protocol reduces correctness to linear algebra and randomness, and inherits perfect zero knowledge and negligible soundness error from the tight ZK CPU framework.

We now proceed with the formal proof.

Malicious $\mathcal{V}^*$ (zero knowledge). The simulator for a malicious $\mathcal{V}^*$ is straightforward in the $\mathcal{F}_{\mathsf{CPZK\text{-}UROM\text{-}SET\text{-}KVS}}$-hybrid model. The simulator learns the execution length n from its interaction with $\mathcal{F}_{\mathsf{G\text{-}ZKCPU}}$ and forwards the same value to $\mathcal{V}^*$, matching the real protocol (where $\mathcal{P}$ sends n in Item 1).

In a real execution, $\mathcal{V}^*$ receives only (i) commitment identifiers output by $\mathcal{F}_{\mathsf{CPZK\text{-}UROM\text{-}SET\text{-}KVS}}$ through its various procedures (e.g., `Commit`, `ReadUROM`), (ii) the value n, and (iii) boolean outcomes of the corresponding `Check`/`Open` invocations. No committed values are revealed beyond the explicitly opened public anchors.

Crucially, when the prover is honest, *all* invocations of `Check` within $\mathcal{F}_{\mathsf{CPZK\text{-}UROM\text{-}SET\text{-}KVS}}$ return `true`. Accordingly, the simulator samples fresh identifiers whenever $\mathcal{F}_{\mathsf{CPZK\text{-}UROM\text{-}SET\text{-}KVS}}$ would return them and simply relays `true` for every `Check` (including messages such as $(\texttt{checkurom}, \texttt{honest})$, $(\texttt{checkset}, \texttt{honest})$, and $(\texttt{checkkvs}, \texttt{honest})$). No circuit evaluation or witness-dependent computation is required.

The challenges χ, γ may be chosen adversarially by $\mathcal{V}^*$; this does not affect simulation, since the $\mathcal{V}$'s view consists solely of public randomness, commitment identifiers, and check outcomes. Hence, the simulated view is *perfectly* distributed as in the real protocol.

Malicious $\mathcal{P}^*$ (soundness). The simulator interacts with $\mathcal{F}_{\text{G-ZKCPU}}$, runs $\mathcal{P}^*$ as a subroutine, and emulates the hybrid $\mathcal{F}_{\text{CPZK-UROM-SET-KVS}}$ for her. In particular, the simulator emulates an honest real-world verifier $\mathcal{V}$ interacting with $\mathcal{P}^*$. Since $\mathcal{V}$ has *no* private input, $\mathcal{S}$ can emulate $\mathcal{V}$ trivially. Thus, the main task is to correctly simulate the output delivered to the ideal verifier.

Concretely, $\mathcal{S}$ proceeds as follows. If the emulated $\mathcal{V}$ outputs $(\texttt{prove}, \texttt{false}, n)$, then $\mathcal{S}$ configures $\mathcal{F}_{\text{G-ZKCPU}}$ (via Item 4 in Figure 3) to output $(\texttt{prove}, \texttt{false}, n)$ to the ideal verifier. Otherwise, if the emulated $\mathcal{V}$ outputs $(\texttt{prove}, \texttt{true}, n)$, this implies that $\mathcal{P}^*$ has passed all checks in $\Pi_{\text{G-ZKCPU}}$.

Since $\mathcal{P}^*$ submits all commitments via $\mathcal{F}_{\text{CPZK-UROM-SET-KVS}}$ (which is emulated by $\mathcal{S}$), the simulator can extract all committed values. In particular, $\mathcal{S}$ extracts $\mathbf{in}$ (from Item 2), $\boldsymbol{b}$ (from Item 3), and $\boldsymbol{p}, \mathbf{id}$ (from Item 5).

Conditioned on the emulated $\mathcal{V}$ outputting `true`, the following properties must hold:

1. $\boldsymbol{p}$ is a length-n boundary string. Otherwise, the `SetProg` procedure (in $\mathcal{F}_{\text{CPZK-UROM-SET-KVS}}$) would reject.
2. $\mathbf{id} \in [B]^n$. Otherwise, the `ReadUROM` procedure would reject.
3. The vector $\boldsymbol{d}$ equals $\mathbf{tv}^{(i_1)} \| \cdots \| \mathbf{tv}^{(i_\tau)}$, where each $\mathbf{tv}^{(j)}$ is the topology vector of instruction j (see Item 7) and $(i_1, \ldots, i_\tau) := \mathsf{Filter}(\boldsymbol{p}, \mathbf{id})$. Otherwise, `ReadUROM` would reject.
4. All entries of $\boldsymbol{b}$ lie in set. Otherwise, `ReadSET` would reject.

The simulator $\mathcal{S}$ then sends $(\texttt{Prove}, \tau, i_1, \ldots, i_\tau, \boldsymbol{w})$ to $\mathcal{F}_{\text{G-ZKCPU}}$, where $\tau := \mathsf{HW}(\boldsymbol{p})$ and $\boldsymbol{w}$ is obtained by partitioning $\mathbf{in}$ and $\boldsymbol{b}$ into τ blocks and reordering them according to $(i_1, \ldots, i_\tau)$.

It remains to show that $\mathcal{F}_{\text{G-ZKCPU}}$ outputs $(\texttt{prove}, \texttt{false}, n)$ with only negligible probability. This establishes soundness. To see this, note the following additional facts (recall, conditioned over the emulated $\mathcal{V}$ outputting `true`):

5. $\boldsymbol{\ell} \odot \boldsymbol{r} = \boldsymbol{o}$. Otherwise, the `Check` of $\mathcal{C}^{\mathsf{mul}}$ would reject (see Item 9a), and the emulated $\mathcal{V}$ would output `false`.
6. For every $t \in [n]$, we have $p_t \cdot o_{t \cdot u_\times} = 0$. Equivalently, $o_{t \cdot u_\times} = 0$ at every instruction boundary ($p_t = 1$). Otherwise, the `Check` of $\mathcal{C}^{\mathsf{op}}$ would reject (see Item 9d), and the emulated $\mathcal{V}$ would output `false`.
7. The compressed linear wiring check accepts, i.e., $\mathsf{sum}_L = \mathsf{sum}_R$ for the values defined inside $\mathcal{C}^{\mathsf{lin}}$. Otherwise, the `Check` of $\mathcal{C}^{\mathsf{lin}}$ would reject (see Item 9e), and the emulated $\mathcal{V}$ would output `false`.

If the ideal-world $\mathcal{V}$ outputs `false`, then the witness extracted by $\mathcal{S}$ does not correspond to a valid execution of $\mathcal{F}_{\text{G-ZKCPU}}$. Conditioned on facts (1)–(6) above, all local consistency checks have passed, and the (malicious) prover has fixed deterministic vector $\mathsf{LEFT} \in \mathbb{F}^{n_{\text{col}}}$ and $\mathsf{RIGHT} \in \mathbb{F}^{n_{\text{row}}}$ inside $\mathcal{C}^{\mathsf{lin}}$, as well as the corresponding (hidden) topology matrix $\boldsymbol{T}$.

We first account for the additional aggregation challenge γ. For each instruction $\mathcal{I}_i$, let $(z_1^{(i)}, \ldots, z_{q_i}^{(i)})$ be its designated zero wires in a fixed public order, where $q_i := |\mathcal{I}_i.\mathsf{zero}|$. The protocol defines the corresponding zero-checking row of $\boldsymbol{T}^{(i)}$ via the aggregated wire $z_{\mathsf{agg}}^{(i)}(\gamma) := \sum_{j=1}^{q_i} \gamma^{j-1} z_j^{(i)}$. If $(z_1^{(i)}, \ldots, z_{q_i}^{(i)}) \neq \mathbf{0}$, then $z_{\mathsf{agg}}^{(i)}(\gamma)$ is a nonzero univariate polynomial in γ of degree at most $q_i - 1$. Since $\gamma \xleftarrow{\$} \mathbb{F}$ is sampled uniformly *after* $\mathcal{P}^*$ has fixed all committed values — and hence all $z_j^{(i)}$ — the Schwartz-Zippel lemma implies $\Pr_\gamma\big[z_{\mathsf{agg}}^{(i)}(\gamma) = 0\big] \leq \frac{q_i - 1}{|\mathbb{F}|}$. By a union bound over all $i \in [B]$, except with probability at most

$\frac{\sum_{i=1}^{B}(|\mathcal{I}_i.\mathsf{zero}|-1)}{|\mathbb{F}|}$, the aggregation is faithful, i.e., $z^{(i)}_{\mathsf{agg}}(\gamma) = 0$ implies $(z^{(i)}_1, \ldots, z^{(i)}_{q_i}) = \mathbf{0}$ for all i. In the remainder, we condition on this event.

Since the extracted witness is invalid, the linear wiring constraints induced by $\boldsymbol{T}$ are violated, i.e.,

$$\boldsymbol{T} \cdot \mathsf{LEFT} \neq \mathsf{RIGHT}.$$

Equivalently, letting $\Delta := \boldsymbol{T} \cdot \mathsf{LEFT} - \mathsf{RIGHT} \in \mathbb{F}^{n_{\mathsf{row}}}$, we have $\Delta \neq \mathbf{0}$.

Nevertheless, fact (7) asserts that the verifier accepted the compressed linear check, which is equivalent to

$$\langle (1, \chi, \ldots, \chi^{n_{\mathsf{row}}-1}),\ \Delta \rangle = 0.$$

Since Δ is a fixed nonzero vector determined before χ is sampled, the left-hand side is a nonzero univariate polynomial in χ of degree at most $n_{\mathsf{row}} - 1$. By the Schwartz-Zippel lemma, this equality holds with probability at most $\frac{n_{\mathsf{row}}-1}{|\mathbb{F}|}$.

To sum up, combining the two bounds gives overall soundness error at most $\frac{n_{\mathsf{row}}-1+\sum_{i=1}^{B}(|\mathcal{I}_i.\mathsf{zero}|-1)}{|\mathbb{F}|}$. □

Efficiency of $\Pi_{\mathsf{G\text{-}ZKCPU}}$. The protocol achieves the following efficiency in the $\mathcal{F}_{\mathsf{CPZK\text{-}UROM\text{-}SET\text{-}KVS}}$-hybrid model (omitting several public values sent by $\mathcal{P}$ for formalization purposes):

- $\mathcal{V}$ sends $\mathcal{O}(1)$ field elements to $\mathcal{P}$.
- $nu_{\mathsf{in}} + 3nu_{\times} + 3n$ `Commit` hybrid calls.
- $\mathcal{O}\left(\sum_{i\in[B]} |\mathcal{I}_i.\mathcal{C}|\right)$ local field operations via technique in [YHH+23].
- $2m + 3 + \sum_{i\in[B]} n^{(i)}(u_{\mathsf{in}} + u_{\mathsf{set}} + u_{\mathsf{kvs}} + u_{\times})$ `Linear` hybrid calls to commit public constants.
- $m + 1$ `Open` hybrid call to open public constants.
- `InitSET` hybrid call on $|\mathsf{set}|$ elements and `ReadSET` hybrid call on nu_{set} elements.
- `InitKVS` hybrid call on $|\mathcal{M}|$ pairs and `ReadKVS` hybrid call on nu_{kvs} triples.
- `InitUROM` hybrid call on B vectors of length $n^{(i\in[B])}(u_{\mathsf{in}} + u_{\mathsf{set}} + u_{\mathsf{kvs}} + u_{\times})$ and `ReadUROM` hybrid to load $n(u_{\mathsf{in}} + u_{\mathsf{set}} + u_{\mathsf{kvs}} + u_{\times})$ elements.
- 3 `Check` hybrid calls, invoking the following circuits:
  1. $\mathcal{C}^{(\mathsf{mul})}$ with at most $nu_{\times}$ multiplication gates.
  2. $\mathcal{C}^{(\mathsf{op})}$ with at most n multiplication gates.
  3. $\mathcal{C}^{(\mathsf{lin})}$ with at most $3n(u_{\mathsf{in}} + u_{\mathsf{set}} + u_{\mathsf{kvs}} + u_{\times})$ MULT gates.

Finally, we remark that Theorem 4.2 follows direct composition of Lemmas B.1 and A.2 and Theorem B.2.

C Assembly Instruction Gadget Constructions

This section collects the gadget specifications that were deferred from the main body. See Appendix D for formal security proofs.

Omitted Gadgets. We omit several straightforward gadget constructions, including:

- Gadgets that involve only unconditional register moves, i.e., `MOV` and `JMP`.
- Gadgets that perform register moves conditioned on the result of `EQZ`, i.e., `CMOV` and `BNZ`.
- Gadgets whose definitions closely mirror those of other gadgets, i.e., `PC` (identical to `ADD`) and `AND`/`OR` (implemented as `XOR` using lookup tables).
- Gadgets that are simple no-ops, i.e., `HALT`, `QED`, and `VMCALL`.

Gadget 7 BYTEDECOMP(u):

Require: $\mathsf{u} \in \mathbb{F}$ (intended to encode a 32-bit word)
Ensure: $(u_0, u_1, u_2, u_3) \in \mathbb{F}^4$ (each u_i an 8-bit word and $u_0 + 2^8 u_1 + 2^{16} u_2 + 2^{24} u_3 = \mathsf{u}$)
Cost: 0 input, 0 $\mathbb{F}$ mult., 1 set, 2 kvs

1: $\mathsf{hi} := \mathsf{set}()$
2: $\mathsf{lo} := \mathsf{u} - 2^{16}\mathsf{hi}$ ▷ lo is constrained by later LOAD
3: $u_1 := \mathtt{LOAD}(\mathsf{offset}_{\mathsf{HIGH8}} + \mathsf{lo}),\ u_0 := \mathsf{lo} - 2^8 u_1$
4: $u_3 := \mathtt{LOAD}(\mathsf{offset}_{\mathsf{HIGH8}} + \mathsf{hi}),\ u_2 := \mathsf{hi} - 2^8 u_3$
5: **return** (u_0, u_1, u_2, u_3)

Gadget 8 LO2(src):

Require: $\mathsf{src} \in \mathbb{F}$ (intended to encode a 32-bit word)
Ensure: $\mathsf{dst} \in \mathbb{F}$ ($\mathsf{dst} = \mathsf{src} \bmod 4$)
Cost: 0 input, 0 $\mathbb{F}$ mult., 1 set, 1 kvs

1: $\mathsf{hi} := \mathsf{set}()$
2: $\mathsf{lo} := \mathsf{src} - 2^{16}\mathsf{hi}$ ▷ lo is constrained by later LOAD
3: $\mathsf{dst} := \mathtt{LOAD}(\mathsf{offset}_{\mathsf{lo2}} + \mathsf{lo})$
4: **return** dst

C.1 Auxiliary Gadgets

We describe two auxiliary gadgets that support common low-level operations shared across multiple instruction gadgets. These gadgets do not correspond to standalone assembly instructions, but instead provide reusable subcircuits for word-level decomposition and low-bit extraction.

C.2 Gadgets without KVS Access

We next present gadget constructions that do not interact with the key-value store (KVS). Their correctness is nontrivial and will be established formally in the gadget inspection (Appendix D). This subsection serves as the starting point of our inspection pipeline, before turning to gadgets that additionally involve KVS access.

C.3 Gadgets with KVS Access

We finally specify gadget constructions that interact with the key-value store (KVS). As with the previous subsection, their correctness is nontrivial and will be formally established in the gadget inspection (Appendix D).

Gadget 9 MSB(src):

Require: src $\in \mathbb{F}$ (intended to encode a 32-bit word)
Ensure: dst $\in \mathbb{F}$ (the most-significant bit of src)
Cost: 1 input, 1 $\mathbb{F}$ mult., 2 set, 0 kvs

1: $c :=$ input(), checkzero($c(c-1)$) ▷ carry bit $c \in \{0, 1\}$
2: hi $:=$ set(), lo $:=$ set() ▷ 16-bit limbs
3: checkzero($2\mathsf{src} - 2^{32}c - 2^{16}\mathsf{hi} - \mathsf{lo}$)
4: dst $:= c$
5: **return** dst

Gadget 10 SUB($\mathsf{src}_1, \mathsf{src}_2$):

Require: $\mathsf{src}_1, \mathsf{src}_2 \in \mathbb{F}$ (intended to encode 32-bit words)
Ensure: dst $\in \mathbb{F}$ (low 32 bits of $\mathsf{src}_1 - \mathsf{src}_2$)
Cost: 1 input, 1 $\mathbb{F}$ mult., 2 set, 0 kvs

1: $c :=$ input(), checkzero($c(c-1)$) ▷ no-borrow bit $c \in \{0, 1\}$
2: hi $:=$ set(), lo $:=$ set() ▷ 16-bit limbs
3: checkzero($\mathsf{src}_1 - \mathsf{src}_2 + 2^{32} - 2^{32}c - 2^{16}\mathsf{hi} - \mathsf{lo}$)
4: dst $:= 2^{16}\mathsf{hi} + \mathsf{lo}$
5: **return** dst

Gadget 11 UDIV($\mathsf{src}_1, \mathsf{src}_2$):

Require: $\mathsf{src}_1, \mathsf{src}_2 \in \mathbb{F}$ (intended to encode 32-bit words, $\mathsf{src}_2 \neq 0$)
Ensure: dst $\in \mathbb{F}$ (the quotient $\lfloor \mathsf{src}_1 / \mathsf{src}_2 \rfloor$)
Cost: 0 input, 2 $\mathbb{F}$ mult., 8 set, 0 kvs

1: $b_0, b_1 :=$ set(), checkzero($\mathsf{src}_2 - b_0 - 2^{16}b_1$) ▷ the denominator
2: $c_0, c_1 :=$ set(), $c := c_0 + 2^{16}c_1$ ▷ c is the quotient
3: $d_0, d_1 :=$ set(), $d := d_0 + 2^{16}d_1$ ▷ d is the remainder
4: $z := b_1 \cdot c_1$, checkzero(z) ▷ $b \cdot c < 2^{48}$
5: $t := \mathsf{src}_2 \cdot c$ ▷ $t = \mathsf{src}_2 \cdot c$
6: checkzero($\mathsf{src}_1 - t - d$) ▷ $\mathsf{src}_1 = \mathsf{src}_2 \cdot c + d$
7: $e_0, e_1 :=$ set(), $e := e_0 + 2^{16}e_1$
8: checkzero($\mathsf{src}_2 - d - 1 - e$) ▷ $0 \leq d < \mathsf{src}_2$
9: dst $:= c$
10: **return** dst

Gadget 12 UREM($\mathsf{src}_1, \mathsf{src}_2$):

Require: $\mathsf{src}_1, \mathsf{src}_2 \in \mathbb{F}$ (intended to encode 32-bit words, $\mathsf{src}_2 \neq 0$)
Ensure: $\mathsf{dst} \in \mathbb{F}$ (the remainder $\mathsf{src}_1 \bmod \mathsf{src}_2$)
Cost: 0 input, 2 $\mathbb{F}$ mult., 8 set, 0 kvs

1: $b_0, b_1 := \mathsf{set}()$, $\mathsf{checkzero}(\mathsf{src}_2 - b_0 - 2^{16} b_1)$ ▷ the denominator
2: $c_0, c_1 := \mathsf{set}()$, $c := c_0 + 2^{16} c_1$ ▷ c is the quotient
3: $d_0, d_1 := \mathsf{set}()$, $d := d_0 + 2^{16} d_1$ ▷ d is the remainder
4: $z := b_1 \cdot c_1$, $\mathsf{checkzero}(z)$ ▷ $b \cdot c < 2^{48}$
5: $t := \mathsf{src}_2 \cdot c$ ▷ $t = b \cdot c$
6: $\mathsf{checkzero}(\mathsf{src}_1 - t - d)$ ▷ $\mathsf{src}_1 = b \cdot c + d$
7: $e_0, e_1 := \mathsf{set}()$, $e := e_0 + 2^{16} e_1$
8: $\mathsf{checkzero}(\mathsf{src}_2 - d - 1 - e)$ ▷ $0 \leq d < \mathsf{src}_2$
9: $\mathsf{dst} := d$
10: **return** dst

Gadget 13 ULT($\mathsf{src}_1, \mathsf{src}_2$):

Require: $\mathsf{src}_1, \mathsf{src}_2 \in \mathbb{F}$ (intended to encode 32-bit words)
Ensure: $\mathsf{dst} \in \mathbb{F}$ ($\mathsf{dst} = 1$ iff $\mathsf{src}_1 < \mathsf{src}_2$, else 0; unsigned)
Cost: 1 input, 3 $\mathbb{F}$ mult., 4 set, 0 kvs

1: $c := \mathsf{input}()$, $\mathsf{checkzero}(c(c-1))$ ▷ comparison bit $c \in \{0, 1\}$
2: $d := \mathsf{src}_2 - \mathsf{src}_1 - 1$
3: $d_0, d_1 := \mathsf{set}()$, $\mathsf{checkzero}(c \cdot (d - d_0 - 2^{16} d_1))$ ▷ enforce $d \in [0, 2^{32})$ if $c = 1$
4: $e := \mathsf{src}_1 - \mathsf{src}_2$
5: $e_0, e_1 := \mathsf{set}()$, $\mathsf{checkzero}((1-c) \cdot (e - e_0 - 2^{16} e_1))$ ▷ enforce $e \in [0, 2^{32})$ if $c = 0$
6: $\mathsf{dst} := c$
7: **return** dst

Gadget 14 LT($\mathsf{src}_1, \mathsf{src}_2$):

Require: $\mathsf{src}_1, \mathsf{src}_2 \in \mathbb{F}$ (intended to encode 32-bit words)
Ensure: $\mathsf{dst} \in \mathbb{F}$ ($\mathsf{dst} = 1$ iff $\mathsf{src}_1 < \mathsf{src}_2$, else 0; signed)
Cost: 3 input, 5 $\mathbb{F}$ mult., 8 set, 0 kvs

1: $u_0, u_1 := \mathsf{set}()$, $u := u_0 + 2^{16} u_1$ ▷ $u \equiv \mathsf{src}_1 + 2^{31} \pmod{2^{32}}$
2: $o_u := \mathsf{input}()$, $\mathsf{checkzero}(o_u(o_u - 1))$
3: $\mathsf{checkzero}(\mathsf{src}_1 + 2^{31} - u - 2^{32} o_u)$
4: $v_0, v_1 := \mathsf{set}()$, $v := v_0 + 2^{16} v_1$ ▷ $v \equiv \mathsf{src}_2 + 2^{31} \pmod{2^{32}}$
5: $o_v := \mathsf{input}()$, $\mathsf{checkzero}(o_v(o_v - 1))$
6: $\mathsf{checkzero}(\mathsf{src}_2 + 2^{31} - v - 2^{32} o_v)$
7: $\mathsf{dst} := \mathsf{ULT}(u, v)$
8: **return** dst

Gadget 15 $\mathsf{INPUT}()$:

Require: next value in prover input queue (intended to encode a 32-bit word)
Ensure: $\mathsf{dst} \in \mathbb{F}$ (the consumed input, encoding a 32-bit word)
Cost: 0 input, 0 $\mathbb{F}$ mult., 2 set, 0 kvs

1: $\mathsf{hi} := \mathsf{set}(),\ \mathsf{lo} := \mathsf{set}()$
2: $\mathsf{dst} := 2^{16}\mathsf{hi} + \mathsf{lo}$
3: **return** dst

Gadget 16 $\mathsf{POW2}(\mathsf{k})$:

Require: $\mathsf{k} \in \mathbb{F}$ (intended to encode an integer in $\{0, \ldots, 31\}$)
Ensure: $\mathsf{dst} \in \mathbb{F}$ ($\mathsf{dst} = 2^{\mathsf{k}}$)
Cost: 0 input, 0 $\mathbb{F}$ mult., 0 set, 1 kvs

1: $\mathsf{dst} := \mathsf{LOAD}(\mathsf{offset}_{\mathsf{POW2}} + \mathsf{k})$
2: **return** dst

Gadget 17 $\mathsf{SRA}(\mathsf{src}, \mathsf{sh})$:

Require: $\mathsf{src}, \mathsf{sh} \in \mathbb{F}$ (intended to encode a 32-bit word and $\mathsf{sh} \in \{0, \ldots, 31\}$)
Ensure: $\mathsf{dst} \in \mathbb{F}$ ($\mathsf{dst} = \mathsf{src} \gg \mathsf{sh}$; signed)
Cost: 2 input, 5 $\mathbb{F}$ mult., 10 set, 1 kvs

1: $s := \mathsf{MSB}(\mathsf{src})$
2: $\mathsf{pow} := \mathsf{POW2}(\mathsf{sh})$ ▷ $\mathsf{pow} = 2^{\mathsf{sh}}$
3: $q := \mathsf{UDIV}(\mathsf{src}, \mathsf{pow})$
4: $t := \mathsf{input}(), \mathsf{checkzero}(t \cdot \mathsf{pow} - 2^{32})$ ▷ $t = 2^{32-\mathsf{sh}}$
5: $\mathsf{dst} := q + s \cdot (2^{32} - t)$
6: **return** dst

Gadget 18 $\mathsf{XOR}(\mathsf{src}_1, \mathsf{src}_2)$:

Require: $\mathsf{src}_1, \mathsf{src}_2 \in \mathbb{F}$ (intended to encode 32-bit words)
Ensure: $\mathsf{dst} \in \mathbb{F}$ (the bitwise XOR of src_1 and src_2)
Cost: 0 input, 0 $\mathbb{F}$ mult., 4 set, 8 kvs

1: $(u_0, u_1, u_2, u_3) := \mathsf{BYTEDECOMP}(\mathsf{src}_1)$
2: $(v_0, v_1, v_2, v_3) := \mathsf{BYTEDECOMP}(\mathsf{src}_2)$
3: $w_0 := \mathsf{LOAD}(\mathsf{offset}_{\mathsf{XOR}} + 2^8 u_0 + v_0)$
4: $w_1 := \mathsf{LOAD}(\mathsf{offset}_{\mathsf{XOR}} + 2^8 u_1 + v_1)$
5: $w_2 := \mathsf{LOAD}(\mathsf{offset}_{\mathsf{XOR}} + 2^8 u_2 + v_2)$
6: $w_3 := \mathsf{LOAD}(\mathsf{offset}_{\mathsf{XOR}} + 2^8 u_3 + v_3)$
7: $\mathsf{dst} := 2^{24} w_3 + 2^{16} w_2 + 2^8 w_1 + w_0$
8: **return** dst

Gadget 19 ULOADB(addr):

Require: addr $\in \mathbb{F}$ (intended to encode a 32-bit byte address)
Ensure: dst $\in \mathbb{F}$ (zero-extended byte at address addr)
Cost: 3 input, 8 $\mathbb{F}$ mult., 2 set, 4 kvs

1: $r :=$ LO2(addr) ▷ $r =$ addr $\bmod 4$
2: addr$' :=$ addr $- r$
3: word $:=$ LOAD(addr$'$)
4: $(a_0, a_1, a_2, a_3) :=$ BYTEDECOMP(word)
5: $b_0, b_1, b_2 :=$ input() ▷ selector bits
6: checkzero$(b_i(b_i - 1))$ for $i \in \{0, 1, 2\}$
7: $b_3 := 1 - b_0 - b_1 - b_2,$ checkzero$(b_3(b_3 - 1))$
8: checkzero$(3 - 3b_0 - 2b_1 - b_2 - r)$
9: dst $:= b_0a_0 + b_1a_1 + b_2a_2 + b_3a_3$
10: **return** dst

Gadget 20 ULOADH(addr):

Require: addr $\in \mathbb{F}$ (intended to encode a 32-bit byte address)
Ensure: dst $\in \mathbb{F}$ (zero-extended halfword at address addr)
Cost: 1 input, 3 $\mathbb{F}$ mult., 3 set, 2 kvs

1: $r :=$ LO2(addr)
2: addr$' :=$ addr $- r$
3: word $:=$ LOAD(addr$'$)
4: $h_0 :=$ set(), $h_1 :=$ set(), checkzero(word $- 2^{16}h_1 - h_0$)
5: $b :=$ input(), checkzero$(b(b - 1))$
6: $b_0 := 1 - b,\ b_1 := b$
7: checkzero$(2b_1 - r)$ ▷ $r \in \{0, 2\}$
8: dst $:= b_0h_0 + b_1h_1$
9: **return** dst

Gadget 21 LOADB(addr):

Require: addr $\in \mathbb{F}$ (intended to encode a 32-bit byte address)
Ensure: dst $\in \mathbb{F}$ (sign-extended byte at address addr)
Cost: 4 input, 9 $\mathbb{F}$ mult., 3 set, 4 kvs

1: byte $:=$ ULOADB(addr) ▷ byte $\in [0, 2^8)$
2: msb $:=$ input(), checkzero(msb(msb $- 1$)) ▷ bit 7 of byte
3: tmp_scaled $:=$ set() ▷ tmp_scaled $= 512 \cdot$ (byte $\bmod 2^7$)
4: checkzero$(512 \cdot$ byte $-$ tmp_scaled $- 2^{16}$msb$)$
5: dst $:=$ byte $+$ msb $\cdot (2^{32} - 2^8)$
6: **return** dst

Gadget 22 LOADH(addr):

Require: addr $\in \mathbb{F}$ (intended to encode a 32-bit byte address)
Ensure: dst $\in \mathbb{F}$ (sign-extended halfword at address addr)
Cost: 2 input, 4 $\mathbb{F}$ mult., 4 set, 2 kvs

1: half := ULOADH(addr) ▷ half $\in [0, 2^{16})$
2: msb := input(), checkzero(msb(msb $-$ 1)) ▷ bit 15 of half
3: tmp_scaled := set() ▷ tmp_scaled $= 2 \cdot (\text{half} \bmod 2^{15})$
4: checkzero($2 \cdot$ half $-$ tmp_scaled $- 2^{16}$msb)
5: dst := half + msb $\cdot (2^{32} - 2^{16})$
6: **return** dst

Gadget 23 STOREB(addr, src):

Require: addr, src $\in \mathbb{F}$ (intended: byte address, 32-bit word)
Ensure: dst $\in \mathbb{F}$ (unused; set to 0)
Cost: 4 input, 8 $\mathbb{F}$ mult., 3 set, 6 kvs

1: r := LO2(addr) ▷ $r = \text{addr} \bmod 4$
2: addr$'$:= addr $- r$
3: word := LOAD(addr$'$)
4: (a_0, a_1, a_2, a_3) := BYTEDECOMP(word)
5: hi := set(), lo := input(), checkzero(src $- 2^{16}$hi $-$ lo)
6: v_1 := LOAD(offset$_{\text{HIGH8}}$ + lo), v := lo $- 2^8 v_1$ ▷ $v = \text{src} \bmod 2^8$
7: b_0, b_1, b_2 := input() ▷ selector bits
8: checkzero($b_i(b_i - 1)$) for $i \in \{0, 1, 2\}$
9: $b_3 := 1 - b_0 - b_1 - b_2$, checkzero($b_3(b_3 - 1)$)
10: checkzero($3 - 3b_0 - 2b_1 - b_2 - r$)
11: $a'_i := a_i + b_i \cdot (v - a_i)$ for $i \in \{0, 1, 2, 3\}$
12: new := $a'_0 + 2^8 a'_1 + 2^{16} a'_2 + 2^{24} a'_3$
13: STORE(addr$'$, new)
14: dst := 0
15: **return** dst

Gadget 24 STOREH(addr, src):

Require: addr, src $\in \mathbb{F}$ (intended: byte address, 32-bit word)
Ensure: dst $\in \mathbb{F}$ (unused; set to 0)
Cost: 1 input, 3 $\mathbb{F}$ mult., 5 set, 3 kvs

1: r := LO2(addr) ▷ $r = \text{addr} \bmod 4$
2: addr$'$:= addr $- r$, word := LOAD(addr$'$)
3: h_0 := set(), h_1 := set(), checkzero(word $- 2^{16} h_1 - h_0$)
4: hi := set(), v := set(), checkzero(src $- 2^{16}$hi $- v$) ▷ $v = \text{src} \bmod 2^{16}$
5: b := input(), checkzero($b(b - 1)$)
6: $b_0 := 1 - b$, $b_1 := b$
7: checkzero($2b_1 - r$) ▷ $r \in \{0, 2\}$
8: $h'_0 := h_0 + b_0 \cdot (v - h_0)$, $h'_1 := h_1 + b_1 \cdot (v - h_1)$, new := $h'_0 + 2^{16} h'_1$
9: STORE(addr$'$, new), dst := 0
10: **return** dst

D Gadget Inspection

In this section, we prove correctness of each gadget by inspection. Concretely, under the stated preconditions, we show that any assignment satisfying the gadget constraints induces the intended ZEE instruction semantics and preserves all explicitly assumed invariants. Therefore, when the preconditions hold and the invariants are satisfied initially, these correctness guarantees compose across gadget invocations.

We do *not* distinguish correctness and soundness at the gadget level: each inspection establishes that satisfying the gadget constraints is *sufficient* to realize the specified semantics (and to preserve the assumed invariants, when applicable), independently of the (potentially malicious) prover's strategy. ZK is orthogonal to these local semantic arguments.

Each gadget inspection follows a uniform structure. We first state a *claim* (semantics and invariant preservation under explicit preconditions), then prove it assuming the gadget constraints are satisfied. We include optional *facts* and *remarks* to record useful variations (e.g., alternative constructions) and to document how undefined behaviors (e.g., division by zero or misaligned accesses) make the constraints unsatisfiable and hence are rejected.

D.1 Field Encoding and No-Wraparound Facts

We formally define notation for embedding integers into $\mathbb{F}_p$ and record basic "no-wraparound" facts that allow us to lift field equalities to equalities over $\mathbb{Z}$ whenever decoded values stay below p; these facts are used throughout the gadget inspections.

Let $\mathbb{F}_p$ be a prime field of characteristic p.

Define the canonical embedding

$$\mathsf{Em} : \mathbb{Z} \to \mathbb{F}_p, \qquad \mathsf{Em}(x) := (x \bmod p) \in \mathbb{F}_p$$

Define the canonical decoding

$$\mathsf{De} : \mathbb{F}_p \to [0, p) \subset \mathbb{Z}$$

as the unique integer $\mathsf{De}(x) \in [0, p)$ such that

$$x = \mathsf{Em}(\mathsf{De}(x))$$

Fact D.1 (Inverse on bounded range)**.** *For every integer $t \in [0, p)$, we have $\mathsf{De}(\mathsf{Em}(t)) = t$.*

Fact D.2 (No-wraparound decoding of linear combinations)**.** *Let $a_1, \ldots, a_k \in \mathbb{F}_p$ and let $\lambda_1, \ldots, \lambda_k \in \mathbb{Z}_{\geq 0}$. Define $S := \sum_{i=1}^{k} \lambda_i \, \mathsf{De}(a_i) \in \mathbb{Z}$. If $S < p$, then*

$$\mathsf{De}\Big(\sum_{i=1}^{k} \mathsf{Em}(\lambda_i) \, a_i \Big) = S,$$

where the sum inside $\mathsf{De}(\cdot)$ is taken in $\mathbb{F}_p$.

Proof. $\forall i$, $a_i = \mathsf{Em}(\mathsf{De}(a_i))$ by definition of De. Thuerefore, $\sum_i \mathsf{Em}(\lambda_i) a_i = \mathsf{Em}(S)$ in $\mathbb{F}_p$. Since $S < p$, $\mathsf{De}(\mathsf{Em}(S)) = S$. □

Fact D.3 (No-wraparound decoding of products)**.** *Let $a_1, \ldots, a_k \in \mathbb{F}_p$ and define $P := \prod_{i=1}^{k} \mathsf{De}(a_i) \in \mathbb{Z}$. If $P < p$, then*

$$\mathsf{De}\Big(\prod_{i=1}^{k} a_i \Big) = P,$$

where the product inside $\mathsf{De}(\cdot)$ *is taken in* $\mathbb{F}_p$.

Proof. $\forall i$, $a_i = \mathsf{Em}(\mathsf{De}(a_i))$ by definition of De. Thus $\prod_i a_i = \mathsf{Em}\Big(\prod_i \mathsf{De}(a_i)\Big) = \mathsf{Em}(P)$ in $\mathbb{F}_p$. Since $P < p$, $\mathsf{De}(\mathsf{Em}(P)) = P$. □

Let $\mathbb{W} := [0, 2^{32}) \subset \mathbb{Z}$. Throughout, we assume $p > 2^{32}$. Under this assumption, the canonical embedding Em is injective on $\mathbb{W}$, and hence $\mathsf{De}(\mathsf{Em}(x)) = x$ for all $x \in \mathbb{W}$. Accordingly, when we say that an element $y \in \mathbb{F}_p$ represents a machine word, we mean that $\mathsf{De}(y) \in \mathbb{W}$.

D.2 Gadgets without KVS Access

In this section, we inspect gadgets that do not access the key-value store, starting with auxiliary facts.

Fact D.4 (Restricted-set inputs)**.** *We clarify the meaning of obtaining an element from* $\mathsf{set}()$ *in our gadget representation when the restricted set is fixed to* $\mathsf{Em}([0, 2^{16})) \subset \mathbb{F}_p$.

Semantically, a call to $\mathsf{set}()$ *introduces a field element* $y \in \mathbb{F}_p$ *chosen by the prover* $\mathcal{P}$*, subject to the constraint that*

$$\mathsf{De}(y) \in [0, 2^{16})$$

That is, while $\mathcal{P}$ *is free to choose the value of* y *as a field element, any satisfying assignment to the gadget enforces that* y *decodes to an integer in* $[0, 2^{16})$.

Fact D.5 (Bit auxiliary inputs)**.** *A call to* $\mathsf{input}()$ *introduces an unrestricted field element* $c \in \mathbb{F}_p$ *chosen by* $\mathcal{P}$*. If the gadget additionally enforces* $\mathsf{checkzero}(c(c-1))$*, then any satisfying assignment must have* $\mathsf{De}(c) \in \{0, 1\}$.

Invariants. Since the gadgets in this section do not access the key-value store, the only invariants we maintain are local typing invariants: inputs representing machine words decode to elements of $\mathbb{W}$, and each gadget produces outputs that also decode to elements of $\mathbb{W}$. This invariant holds trivially at initialization, since the register state is public and consists of valid machine words. (Even when the initial register state is committed, $\mathcal{P}$ can instead prove this property using ZK SET at negligible cost.)

ADD (Gadget 1).

Claim. Let $\mathsf{src}_1, \mathsf{src}_2 \in \mathbb{F}_p$ satisfy $\mathsf{De}(\mathsf{src}_1), \mathsf{De}(\mathsf{src}_2) \in \mathbb{W}$. Assume $p > 2^{33}$ and that $\mathsf{set}()$ ranges over $\mathsf{Em}([0, 2^{16}))$. Then for any assignment satisfying the constraints in Gadget 1, the output $\mathsf{dst} \in \mathbb{F}_p$ satisfies $\mathsf{De}(\mathsf{dst}) = \big(\mathsf{De}(\mathsf{src}_1) + \mathsf{De}(\mathsf{src}_2)\big) \bmod 2^{32}$ (and $\mathsf{De}(\mathsf{dst}) \in \mathbb{W}$).

Proof. Since $\mathsf{De}(\mathsf{src}_1), \mathsf{De}(\mathsf{src}_2) \in [0, 2^{32} - 1]$, we have $\mathsf{De}(\mathsf{src}_1) + \mathsf{De}(\mathsf{src}_2) \in [0, 2^{33} - 2] < p$.

Since $\mathsf{De}(c) \in \{0, 1\}$ (Fact D.5), $\mathsf{De}(\mathsf{hi}), \mathsf{De}(\mathsf{lo}) \in [0, 2^{16} - 1]$ (Fact D.4), we have $2^{32}\mathsf{De}(c) + 2^{16}\mathsf{De}(\mathsf{hi}) + \mathsf{De}(\mathsf{lo}) \in [0, 2^{33} - 1] < p$.

By Fact D.2, we have

$$\mathsf{De}(\mathsf{src}_1 + \mathsf{src}_2) = \mathsf{De}(\mathsf{src}_1) + \mathsf{De}(\mathsf{src}_2)$$

$$\mathsf{De}(2^{32}c + 2^{16}\mathsf{hi} + \mathsf{lo}) = 2^{32}\mathsf{De}(c) + 2^{16}\mathsf{De}(\mathsf{hi}) + \mathsf{De}(\mathsf{lo})$$

Since the constraint $\mathsf{checkzero}(\mathsf{src}_1 + \mathsf{src}_2 - 2^{32}c - 2^{16}\mathsf{hi} - \mathsf{lo})$ is satisfied, we have the field equality

$$\mathsf{src}_1 + \mathsf{src}_2 = 2^{32}c + 2^{16}\mathsf{hi} + \mathsf{lo} \quad \text{in } \mathbb{F}_p$$

and therefore (applying De to both sides)

$$2^{32}\mathsf{De}(c) + 2^{16}\mathsf{De}(\mathsf{hi}) + \mathsf{De}(\mathsf{lo}) = \mathsf{De}(\mathsf{src}_1) + \mathsf{De}(\mathsf{src}_2)$$

Reducing modulo 2^{32} and because $2^{16}\mathsf{De}(\mathsf{hi}) + \mathsf{De}(\mathsf{lo}) \in [0, 2^{32})$, we obtain

$$2^{16}\mathsf{De}(\mathsf{hi}) + \mathsf{De}(\mathsf{lo}) = \big(\mathsf{De}(\mathsf{src}_1) + \mathsf{De}(\mathsf{src}_2)\big) \bmod 2^{32}$$

Finally, the gadget outputs $\mathsf{dst} = 2^{16}\mathsf{hi} + \mathsf{lo}$. Since $2^{16}\mathsf{De}(\mathsf{hi}) + \mathsf{De}(\mathsf{lo}) < p$, another application of Fact D.2 yields

$$\mathsf{De}(\mathsf{dst}) = 2^{16}\mathsf{De}(\mathsf{hi}) + \mathsf{De}(\mathsf{lo}) = \big(\mathsf{De}(\mathsf{src}_1) + \mathsf{De}(\mathsf{src}_2)\big) \bmod 2^{32}$$

as desired. □

Fact. The above proof also suggests that when p is sufficiently large, the carry wire c in ADD can be replaced by a restricted-set input (in particular, a 16-bit limb when $p = 2^{61} - 1$).

MUL (Gadget 2).

Claim. Let $\mathsf{src}_1, \mathsf{src}_2 \in \mathbb{F}_p$ satisfy $\mathsf{De}(\mathsf{src}_1), \mathsf{De}(\mathsf{src}_2) \in \mathbb{W}$. Assume $p > 2^{49}$ and that $\mathsf{set}()$ ranges over $\mathsf{Em}([0, 2^{16}))$. Then for any assignment satisfying the constraints in Gadget 2, the output $\mathsf{dst} \in \mathbb{F}_p$ satisfies $\mathsf{De}(\mathsf{dst}) = \mathsf{De}(\mathsf{src}_1)\mathsf{De}(\mathsf{src}_2) \bmod 2^{32}$ (and $\mathsf{De}(\mathsf{dst}) \in \mathbb{W}$).

Proof. The gadget enforces

$$\mathsf{src}_1 = 2^{16}\mathsf{hi}_1 + \mathsf{lo}_1 \quad \text{in } \mathbb{F}_p$$
$$\mathsf{src}_2 = 2^{16}\mathsf{hi}_2 + \mathsf{lo}_2 \quad \text{in } \mathbb{F}_p$$
$$\mathsf{lo}_1 \cdot \mathsf{src}_2 + 2^{16} \cdot \mathsf{hi}_1 \cdot \mathsf{lo}_2 = 2^{48}c + 2^{32}u_2 + 2^{16}u_1 + u_0 \quad \text{in } \mathbb{F}_p$$

By Facts D.4 and D.5, we have $\mathsf{De}(c) \in \{0, 1\}$ and $\mathsf{De}(\mathsf{hi}_1), \mathsf{De}(\mathsf{lo}_1), \mathsf{De}(\mathsf{hi}_2), \mathsf{De}(\mathsf{lo}_2), \mathsf{De}(u_0), \mathsf{De}(u_1), \mathsf{De}(u_2) \in [0, 2^{16})$. Since $\mathsf{De}(\mathsf{src}_1), \mathsf{De}(\mathsf{src}_2) \in \mathbb{W}$ and $p > 2^{49}$, the hypotheses of Facts D.2 and D.3 are satisfied, yielding

$$\mathsf{De}(\mathsf{src}_1) = 2^{16}\mathsf{De}(\mathsf{hi}_1) + \mathsf{De}(\mathsf{lo}_1) \quad \text{in } \mathbb{Z}$$
$$\mathsf{De}(\mathsf{src}_2) = 2^{16}\mathsf{De}(\mathsf{hi}_2) + \mathsf{De}(\mathsf{lo}_2) \quad \text{in } \mathbb{Z}$$

$$\mathsf{De}(\mathsf{lo}_1)\mathsf{De}(\mathsf{src}_2) + 2^{16}\mathsf{De}(\mathsf{hi}_1)\mathsf{De}(\mathsf{lo}_2)$$
$$= 2^{48}\mathsf{De}(c) + 2^{32}\mathsf{De}(u_2) + 2^{16}\mathsf{De}(u_1) + \mathsf{De}(u_0) \quad \text{in } \mathbb{Z}$$

Multiplying the first two equalities in $\mathbb{Z}$ yields $\mathsf{De}(\mathsf{src}_1)\mathsf{De}(\mathsf{src}_2)$

$$= 2^{32}\mathsf{De}(\mathsf{hi}_1)\mathsf{De}(\mathsf{hi}_2) + 2^{16}\mathsf{De}(\mathsf{hi}_1)\mathsf{De}(\mathsf{lo}_2) + \mathsf{De}(\mathsf{lo}_1)\mathsf{De}(\mathsf{src}_2)$$

This implies $\mathsf{De}(\mathsf{src}_1)\mathsf{De}(\mathsf{src}_2) \bmod 2^{32}$

$$= 2^{16}\mathsf{De}(\mathsf{hi}_1)\mathsf{De}(\mathsf{lo}_2) + \mathsf{De}(\mathsf{lo}_1)\mathsf{De}(\mathsf{src}_2) \bmod 2^{32}$$

Reducing modulo 2^{32} on the third equality in $\mathbb{Z}$ and because $2^{16}\mathsf{De}(u_1) + \mathsf{De}(u_0) \in [0, 2^{32})$, we obtain

$$2^{16}\mathsf{De}(u_1) + \mathsf{De}(u_0) = \mathsf{De}(\mathsf{lo}_1)\mathsf{De}(\mathsf{src}_2) + 2^{16}\mathsf{De}(\mathsf{hi}_1)\mathsf{De}(\mathsf{lo}_2) \bmod 2^{32}$$

Finally, the gadget outputs $\mathsf{dst} = 2^{16}u_1 + u_0$. Since $2^{16}\mathsf{De}(u_1) + \mathsf{De}(u_0) < p$, another application of Fact D.2 yields

$$\mathsf{De}(\mathsf{dst}) = 2^{16}\mathsf{De}(u_1) + \mathsf{De}(u_0) = \mathsf{De}(\mathsf{src}_1)\mathsf{De}(\mathsf{src}_2) \bmod 2^{32}$$

as desired. □

Fact. The above proof also suggests that when p is sufficiently large, the carry wire c in MUL can be replaced by a restricted-set input (in particular, a 8-bit limb when $p = 2^{61} - 1$).

EQZ (Gadget 3).

Claim. Let $\mathsf{src} \in \mathbb{F}_p$. Then for any assignment satisfying the constraints in Gadget 3, the output $\mathsf{dst} \in \mathbb{F}_p$ satisfies $\mathsf{dst} \in \{0, 1\} \subset \mathbb{F}_p$, $\mathsf{dst} = 1$ i.f.f. $\mathsf{src} = 0$ (and $\mathsf{De}(\mathsf{dst}) \in \{0, 1\} \subset \mathbb{W}$).

Proof. By Fact D.5, $z \in \{0, 1\} \subset \mathbb{F}_p$. The gadget enforces

$$1 - z = \mathsf{src} \cdot w \quad \text{and} \quad z \cdot \mathsf{src} = 0 \quad \text{in } \mathbb{F}_p$$

If $\mathsf{src} = 0$, the left equation forces $z = 1$. If $\mathsf{src} \neq 0$, the right equation forces $z = 0$, and the left equation is satisfied by $w = \mathsf{src}^{-1}$.

Finally, since the gadget outputs $\mathsf{dst} = z$, the claim follows. □

MSB (Gadget 9).

Claim. Let $\mathsf{src} \in \mathbb{F}_p$ satisfy $\mathsf{De}(\mathsf{src}) \in \mathbb{W}$. Assume $p > 2^{33}$ and that $\mathsf{set}()$ ranges over $\mathsf{Em}([0, 2^{16}))$. Then for any assignment satisfying the constraints in Gadget 9, the output $\mathsf{dst} \in \mathbb{F}_p$ satisfies $\mathsf{De}(\mathsf{dst}) = \left\lfloor \mathsf{De}(\mathsf{src})/2^{31} \right\rfloor$ (and $\mathsf{De}(\mathsf{dst}) \in \{0, 1\} \subset \mathbb{W}$).

Proof. The gadget enforces

$$2\mathsf{src} = 2^{32}c + 2^{16}\mathsf{hi} + \mathsf{lo} \quad \text{in } \mathbb{F}_p$$

By Facts D.4 and D.5, we have $\mathsf{De}(c) \in \{0, 1\}$ and $2^{16}\mathsf{De}(\mathsf{hi}) + \mathsf{De}(\mathsf{lo}) \in [0, 2^{32})$. Since $\mathsf{De}(\mathsf{src}) \in \mathbb{W}$ and $p > 2^{33}$, the hypotheses of Fact D.2 are satisfied, yielding

$$2\mathsf{De}(\mathsf{src}) = 2^{32}\mathsf{De}(c) + 2^{16}\mathsf{De}(\mathsf{hi}) + \mathsf{De}(\mathsf{lo}) \quad \text{in } \mathbb{Z}$$

Dividing by 2^{32} and taking floors, we obtain

$$\mathsf{De}(c) = \left\lfloor 2\mathsf{De}(\mathsf{src})/2^{32} \right\rfloor = \left\lfloor \mathsf{De}(\mathsf{src})/2^{31} \right\rfloor$$

Finally, since the gadget outputs $\mathsf{dst} = c$, the claim follows. □

SUB (Gadget 10).

Claim. Let $\mathsf{src}_1, \mathsf{src}_2 \in \mathbb{F}_p$ satisfy $\mathsf{De}(\mathsf{src}_1), \mathsf{De}(\mathsf{src}_2) \in \mathbb{W}$. Assume $p > 2^{34}$ and that $\mathsf{set}()$ ranges over $\mathsf{Em}([0, 2^{16}))$. Then for any assignment satisfying the constraints in Gadget 10, the output $\mathsf{dst} \in \mathbb{F}_p$ satisfies $\mathsf{De}(\mathsf{dst}) = \left(2^{32} + \mathsf{De}(\mathsf{src}_1) - \mathsf{De}(\mathsf{src}_2)\right) \bmod 2^{32}$ (and $\mathsf{De}(\mathsf{dst}) \in \mathbb{W}$).

Proof. The gadget enforces

$$\mathsf{src}_1 + 2^{32} = \mathsf{src}_2 + 2^{32}c + 2^{16}\mathsf{hi} + \mathsf{lo} \quad \text{in } \mathbb{F}_p$$

By Facts D.4 and D.5, we have $\mathsf{De}(c) \in \{0, 1\}$ and $2^{16}\mathsf{De}(\mathsf{hi}) + \mathsf{De}(\mathsf{lo}) \in [0, 2^{32})$. Since $\mathsf{De}(\mathsf{src}_1), \mathsf{De}(\mathsf{src}_2) \in \mathbb{W}$ and $p > 2^{34}$, the hypotheses of Fact D.2 are satisfied, yielding

$$\mathsf{De}(\mathsf{src}_1) + 2^{32} = \mathsf{De}(\mathsf{src}_2) + 2^{32}\mathsf{De}(c) + 2^{16}\mathsf{De}(\mathsf{hi}) + \mathsf{De}(\mathsf{lo}) \quad \text{in } \mathbb{Z}$$

equivariant to

$$2^{32} + \mathsf{De}(\mathsf{src}_1) - \mathsf{De}(\mathsf{src}_2) = 2^{32}\mathsf{De}(c) + 2^{16}\mathsf{De}(\mathsf{hi}) + \mathsf{De}(\mathsf{lo}) \quad \text{in } \mathbb{Z}$$

Reducing modulo 2^{32} and because $2^{16}\mathsf{De}(\mathsf{hi}) + \mathsf{De}(\mathsf{lo}) \in [0, 2^{32})$, we obtain

$$2^{16}\mathsf{De}(\mathsf{hi}) + \mathsf{De}(\mathsf{lo}) = \left(2^{32} + \mathsf{De}(\mathsf{src}_1) - \mathsf{De}(\mathsf{src}_2)\right) \bmod 2^{32}$$

Finally, the gadget outputs $\mathsf{dst} = 2^{16}\mathsf{hi} + \mathsf{lo}$. Since $2^{16}\mathsf{De}(\mathsf{hi}) + \mathsf{De}(\mathsf{lo}) < p$, another application of Fact D.2 yields

$$\mathsf{De}(\mathsf{dst}) = 2^{16}\mathsf{De}(\mathsf{hi}) + \mathsf{De}(\mathsf{lo}) = \left(2^{32} + \mathsf{De}(\mathsf{src}_1) - \mathsf{De}(\mathsf{src}_2)\right) \bmod 2^{32}$$

as desired. □

Fact. The above proof also suggests that when p is sufficiently large, the carry wire c in SUB can be replaced by a restricted-set input (in particular, a 16-bit limb when $p = 2^{61} - 1$).

UDIV (Gadget 11).

Claim. Let $\mathsf{src}_1, \mathsf{src}_2 \in \mathbb{F}_p$ satisfy $\mathsf{De}(\mathsf{src}_1) \in \mathbb{W}$, $\mathsf{De}(\mathsf{src}_2) \in \mathbb{W} \setminus \{0\}$. Assume $p > 2^{49}$ and that set() ranges over $\mathsf{Em}([0, 2^{16}))$. Then for any assignment satisfying the constraints in Gadget 11, the output $\mathsf{dst} \in \mathbb{F}_p$ satisfies $\mathsf{De}(\mathsf{dst}) = \lfloor \mathsf{De}(\mathsf{src}_1)/\mathsf{De}(\mathsf{src}_2) \rfloor$ (and $\mathsf{De}(\mathsf{dst}) \in \mathbb{W}$).

Proof. The gadget enforces

$$\mathsf{src}_2 = b_0 + 2^{16} b_1 \quad \text{and} \quad c = c_0 + 2^{16} c_1 \quad \text{and} \quad b_1 c_1 = 0 \quad \text{in } \mathbb{F}_p$$

By Fact D.4, we have $\mathsf{De}(b_0), \mathsf{De}(b_1), \mathsf{De}(c_0), \mathsf{De}(c_1) \in [0, 2^{16})$. Since $\mathsf{De}(\mathsf{src}_2) \in \mathbb{W}$ and $p > 2^{49}$, the hypotheses of Fact D.2 are satisfied, yielding

$$\mathsf{De}(\mathsf{src}_2) = \mathsf{De}(b_0) + 2^{16}\mathsf{De}(b_1) \quad \text{in } \mathbb{Z}$$

$$\mathsf{De}(c) = \mathsf{De}(c_0) + 2^{16}\mathsf{De}(c_1) \quad \text{and} \quad \mathsf{De}(b_1)\mathsf{De}(c_1) = 0 \quad \text{in } \mathbb{Z}$$

This implies $\mathsf{De}(\mathsf{src}_2)\mathsf{De}(c) =$

$$\mathsf{De}(b_0)\mathsf{De}(c_0) + 2^{16}\mathsf{De}(b_1)\mathsf{De}(c_0) + 2^{16}\mathsf{De}(b_0)\mathsf{De}(c_1) \quad \text{in } \mathbb{Z}$$

Hence, $\mathsf{De}(\mathsf{src}_2)\mathsf{De}(c) < 2^{49} - 2^{33}$. The gadget also enforces

$$d = d_0 + 2^{16} d_1 \quad \text{and} \quad e = e_0 + 2^{16} e_1 \quad \text{and} \quad \mathsf{src}_2 = d + e + 1 \quad \text{in } \mathbb{F}_p$$

By Fact D.4, we have $\mathsf{De}(d_0), \mathsf{De}(d_1), \mathsf{De}(e_0), \mathsf{De}(e_1) \in [0, 2^{16})$. Since $\mathsf{De}(\mathsf{src}_1) \in \mathbb{W}$ and $p > 2^{49}$, the hypotheses of Fact D.2 are satisfied, yielding

$$\mathsf{De}(\mathsf{src}_2) = \mathsf{De}(d) + \mathsf{De}(e) + 1 \quad \text{in } \mathbb{Z}$$

This implies

$$0 \le \mathsf{De}(d) < \mathsf{De}(\mathsf{src}_2) \tag{9}$$

The gadget also enforces

$$\mathsf{src}_1 = \mathsf{src}_2 \cdot c + d \quad \text{in } \mathbb{F}_p$$

By Fact D.4, we have $\mathsf{De}(d_0), \mathsf{De}(d_1) \in [0, 2^{16})$, implying $\mathsf{De}(d) < 2^{32}$. Since $\mathsf{De}(\mathsf{src}_2) \cdot \mathsf{De}(c) < 2^{49} - 2^{33}$ and $p > 2^{49}$, the hypotheses of Facts D.2 and D.3 are satisfied, yielding

$$\mathsf{De}(\mathsf{src}_1) = \mathsf{De}(\mathsf{src}_2) \cdot \mathsf{De}(c) + \mathsf{De}(d) \quad \text{in } \mathbb{Z}$$

Combined with Equation (9) yieds $\mathsf{De}(c) = \lfloor \mathsf{De}(\mathsf{src}_1)/\mathsf{De}(\mathsf{src}_2) \rfloor$.

Finally, since the gadget outputs $\mathsf{dst} = c$, the claim follows. □

Remark. The gadget is complete when $\mathsf{De}(\mathsf{src}_2) \in \mathbb{W} \setminus \{0\}$. When $\mathsf{De}(\mathsf{src}_2) = 0$, the constraint $\mathsf{src}_2 = d + e + 1$ is unsatisfiable, hence any proof is rejected.

UREM (Gadget 12).

Claim. Let $\mathsf{src}_1, \mathsf{src}_2 \in \mathbb{F}_p$ satisfy $\mathsf{De}(\mathsf{src}_1) \in \mathbb{W}, \mathsf{De}(\mathsf{src}_2) \in \mathbb{W} \setminus \{0\}$. Assume $p > 2^{49}$ and that $\mathsf{set}()$ ranges over $\mathsf{Em}([0, 2^{16}))$. Then for any assignment satisfying the constraints in Gadget 12, the output $\mathsf{dst} \in \mathbb{F}_p$ satisfies $\mathsf{De}(\mathsf{dst}) = \mathsf{De}(\mathsf{src}_1) \bmod \mathsf{De}(\mathsf{src}_2)$ (and $\mathsf{De}(\mathsf{dst}) \in \mathbb{W}$).

Proof. This follows directly from UDIV's proof. □

Remark. The gadget is complete when $\mathsf{De}(\mathsf{src}_2) \in \mathbb{W} \setminus \{0\}$. When $\mathsf{De}(\mathsf{src}_2) = 0$, the constraint $\mathsf{src}_2 = d + e + 1$ is unsatisfiable, hence any proof is rejected.

ULT (Gadget 13).

Claim. Let $\mathsf{src}_1, \mathsf{src}_2 \in \mathbb{F}_p$ satisfy $\mathsf{De}(\mathsf{src}_1), \mathsf{De}(\mathsf{src}_2) \in \mathbb{W}$. Assume $p > 2^{33}$ and that $\mathsf{set}()$ ranges over $\mathsf{Em}([0, 2^{16}))$. Then for any assignment satisfying the constraints in Gadget 13, the output $\mathsf{dst} \in \{0, 1\} \subset \mathbb{F}_p$ satisfies $\mathsf{dst} = 1$ i.f.f. $\mathsf{De}(\mathsf{src}_1) < \mathsf{De}(\mathsf{src}_2)$ (and $\mathsf{De}(\mathsf{dst}) \in \{0, 1\} \subset \mathbb{W}$).

Proof. By Fact D.5, $c \in \{0, 1\} \subset \mathbb{F}_p$. The gadget enforces

$$c(\mathsf{src}_2 - \mathsf{src}_1 - d_0 - 2^{16} d_1 - 1) = 0 \quad \text{in } \mathbb{F}_p$$

$$(1 - c)(\mathsf{src}_1 - \mathsf{src}_2 - e_0 - 2^{16} e_1) = 0 \quad \text{in } \mathbb{F}_p$$

If $c = 0$, then $\mathsf{src}_1 - \mathsf{src}_2 - e_0 - 2^{16} e_1 = 0 \in \mathbb{F}_p$. By Fact D.4, we have $\mathsf{De}(e_0), \mathsf{De}(e_1) \in [0, 2^{16})$. Since $\mathsf{De}(\mathsf{src}_1), \mathsf{De}(\mathsf{src}_2) \in \mathbb{W}$ and $p > 2^{33}$, the hypotheses of Fact D.2 are satisfied, yielding

$$\mathsf{De}(\mathsf{src}_1) = \mathsf{De}(\mathsf{src}_2) + \mathsf{De}(e_0) + 2^{16}\mathsf{De}(e_1) \quad \text{in } \mathbb{Z}$$

This equality admits a solution i.f.f. $\mathsf{De}(\mathsf{src}_1) \geq \mathsf{De}(\mathsf{src}_2)$.

If $c = 1$, then $\mathsf{src}_2 - \mathsf{src}_1 - d_0 - 2^{16} d_1 - 1 = 0 \in \mathbb{F}_p$. By Fact D.4, we have $\mathsf{De}(d_0), \mathsf{De}(d_1) \in [0, 2^{16})$. Since $\mathsf{De}(\mathsf{src}_1), \mathsf{De}(\mathsf{src}_2) \in \mathbb{W}$ and $p > 2^{33}$, the hypotheses of Fact D.2 are satisfied, yielding

$$\mathsf{De}(\mathsf{src}_2) = \mathsf{De}(\mathsf{src}_1) + \mathsf{De}(d_0) + 2^{16}\mathsf{De}(d_1) + 1 \quad \text{in } \mathbb{Z}$$

This equality admits a solution i.f.f. $\mathsf{De}(\mathsf{src}_1) < \mathsf{De}(\mathsf{src}_2)$.

Finally, since the gadget outputs $\mathsf{dst} = c$, the claim follows. □

LT (Gadget 14).

Claim. Let $\mathsf{src}_1, \mathsf{src}_2 \in \mathbb{F}_p$ satisfy $\mathsf{De}(\mathsf{src}_1), \mathsf{De}(\mathsf{src}_2) \in \mathbb{W}$. Assume $p > 2^{33}$ and that $\mathsf{set}()$ ranges over $\mathsf{Em}([0, 2^{16}))$. Then for any assignment satisfying the constraints in Gadget 14, the output $\mathsf{dst} \in \{0, 1\} \subset \mathbb{F}_p$ satisfies $\mathsf{dst} = 1$ i.f.f. $(\mathsf{De}(\mathsf{src}_1) + 2^{31}) \bmod 2^{32} < (\mathsf{De}(\mathsf{src}_2) + 2^{31}) \bmod 2^{32}$ (and $\mathsf{De}(\mathsf{dst}) \in \{0, 1\} \subset \mathbb{W}$).

Proof. By Fact D.4, $\mathsf{De}(u_0), \mathsf{De}(u_1), \mathsf{De}(v_0), \mathsf{De}(v_1) \in [0, 2^{16})$, hence $\mathsf{De}(u), \mathsf{De}(v) \in \mathbb{W}$ for $u = u_0 + 2^{16} u_1$ and $v = v_0 + 2^{16} v_1$. By Fact D.5, $o_u, o_v \in \{0, 1\} \subset \mathbb{F}_p$.

The gadget enforces

$$\mathsf{src}_1 + 2^{31} = u + 2^{32} o_u \quad \text{and} \quad \mathsf{src}_2 + 2^{31} = v + 2^{32} o_v \quad \text{in } \mathbb{F}_p$$

Since $\mathsf{De}(\mathsf{src}_1), \mathsf{De}(\mathsf{src}_2) \in \mathbb{W}$ and $p > 2^{33}$, the hypotheses of Fact D.2 are satisfied, yielding

$$\mathsf{De}(\mathsf{src}_1) + 2^{31} = \mathsf{De}(u) + 2^{32}\mathsf{De}(o_u) \quad \text{in } \mathbb{Z}$$

$$\mathsf{De}(\mathsf{src}_2) + 2^{31} = \mathsf{De}(v) + 2^{32}\mathsf{De}(o_v) \quad \text{in } \mathbb{Z}$$

Reducing both equalities modulo 2^{32} and using $\mathsf{De}(u), \mathsf{De}(v) \in [0, 2^{32})$ gives

$$\mathsf{De}(u) = \left(\mathsf{De}(\mathsf{src}_1) + 2^{31}\right) \bmod 2^{32} \quad \mathsf{De}(v) = \left(\mathsf{De}(\mathsf{src}_2) + 2^{31}\right) \bmod 2^{32}$$

Finally, the gadget outputs $\mathsf{dst} = \mathtt{ULT}(u, v)$. By correctness of ULT (as shown in its inspection; note, $\mathsf{De}(u), \mathsf{De}(v) \in \mathbb{W}$), we have

$$\mathsf{dst} = 1 \text{ i.f.f. } \mathsf{De}(u) < \mathsf{De}(v)$$

Hence, we have

$$\mathsf{dst} = 1 \text{ i.f.f. } \left(\mathsf{De}(\mathsf{src}_1) + 2^{31}\right) \bmod 2^{32} < \left(\mathsf{De}(\mathsf{src}_2) + 2^{31}\right) \bmod 2^{32}$$

as desired. □

INPUT (Gadget 15).

Claim. For any assignment satisfying the constraints in Gadget 15, the output $\mathsf{dst} \in \mathbb{F}_p$ satisfies $\mathsf{De}(\mathsf{dst}) \in \mathbb{W}$.

Proof. This is trivial by Facts D.2 and D.4. □

Remark. Conversely, for any $x \in \mathbb{W}$, the prover may realize $\mathsf{dst} = \mathsf{Em}(x)$ by choosing $(\mathsf{hi}, \mathsf{lo})$ as the 16-bit limbs of x.

D.3 Gadgets with KVS Access

In this section, we inspect gadgets that access the key-value store, starting with auxiliary facts.

Fact D.6 (KVS gate semantics)**.** *Let $\mathcal{M} : \mathsf{Dom}(\mathcal{M}) \subseteq \mathbb{F}_p \to \mathbb{F}_p$ denote the global key-value store. A call to $\mathsf{kvs}(\mathsf{idx}, \mathsf{val}_{\mathsf{new}})$, with $\mathsf{idx}, \mathsf{val}_{\mathsf{new}} \in \mathbb{F}_p$, introduces an output wire $\mathsf{val}_{\mathsf{old}} \in \mathbb{F}_p$. For any satisfying assignment (i.e., when the key idx exists in the global store), $\mathsf{val}_{\mathsf{old}}$ equals the value currently stored at key idx in the global key-value store. The store is then updated so that the value at key idx becomes $\mathsf{val}_{\mathsf{new}}$, while all other keys remain unchanged.*

Fact D.7 (RAM layout in the KVS)**.** *To model program memory using the key-value store, we reserve the key range $\mathsf{Em}([0, 2^{32})) \subset \mathbb{F}_p$ for memory addressing. Within this range, we adopt a word-aligned layout: keys in*

$$\mathsf{Em}\left(4 \cdot [0, 2^{30})\right)$$

represent 32-bit words stored at the corresponding memory addresses, i.e., integers whose two least significant bits are zero. All other keys in $\mathsf{Em}([0, 2^{32}))$ are outside the domain of the key-value store.

Fact D.8 (Lookup-table key ranges in the KVS)**.** *We fix public offsets*

$$\mathsf{offset}_T \in \mathbb{F}_p$$

for each table $T \in \{\mathsf{XOR}, \mathsf{AND}, \mathsf{OR}, \mathsf{POW}, \mathsf{LO2}, \mathsf{HIGH8}\}$*. Let* $\mathsf{off}_T := \mathsf{De}(\mathsf{offset}_T) \in [0, p)$*. Define the corresponding (decoded) key ranges:*

$$\mathcal{K}_T := \begin{cases} \{\mathsf{off}_T + 2^8 a + b : a, b \in [0, 2^8)\} & T \in \{\mathsf{XOR}, \mathsf{AND}, \mathsf{OR}\} \\ \{\mathsf{off}_T + a : a \in [0, 2^{16})\} & T = \{\mathsf{HIGH8}, \mathsf{LO2}\} \\ \{\mathsf{off}_T + a : a \in [0, 32)\} & T = \mathsf{POW} \end{cases}$$

Assume the following:

- ***Disjoint from RAM / no wraparound.*** *For every* T *and every* $k \in \mathcal{K}_T$*, we have* $2^{32} \le k < p$*.*
- ***No overlap.*** *The sets* $\{\mathcal{K}_T\}_T$ *are pairwise disjoint.*
- ***Existence.*** *For every* T *and every* $k \in \mathcal{K}_T$*, the entry* $\mathcal{M}[\mathsf{Em}(k)]$ *is defined. That is,* $\mathsf{Em}(k)$ *is inside the domain of the key-value store.*

KVS invariants. Throughout gadget inspection, we maintain the following global invariants on the key-value store $\mathcal{M}$.

- **RAM well-typedness**. For every RAM key $k \in \mathsf{Em}(4 \cdot [0, 2^{30}))$, $\mathcal{M}[k]$ represents a machine word, i.e., $\mathsf{De}(\mathcal{M}[k]) \in \mathbb{W}$.
- **Lookup-table correctness.** For every lookup table T and every key $k \in \mathsf{Em}(\mathcal{K}_T)$, the entry $\mathcal{M}[k]$ is defined and equals the value prescribed by the semantics of table T as specified in Section 5.2. In particular, all lookup-table entries decode to elements of $\mathbb{W}$.

These invariants are trivially satisfied at initialization, since the contents of RAM and all lookup tables are public and fixed at setup. All gadgets are designed to preserve these invariants: RAM updates maintain word-typedness, while lookup-table entries are treated as read-only. Note, gadgets inspected in Appendix D.2 do not involve KVS operations and therefore satisfy these invariants trivially.

LOAD (Gadget 4).

Claim. Let $\mathsf{src} \in \mathbb{F}_p$ satisfy $\mathsf{src} \in \mathsf{Dom}(\mathcal{M})$. Assume the KVS invariants hold. Then for any assignment satisfying the constraints in Gadget 4, the output $\mathsf{dst} \in \mathbb{F}_p$ satisfies $\mathsf{dst} = \mathcal{M}[\mathsf{src}]$, the key-value store $\mathcal{M}$ remains unchanged (inferring the KVS invariants still hold and $\mathsf{De}(\mathsf{dst}) \in \mathbb{W}$).

Proof. By Fact D.6, the call $\mathsf{kvs}(\mathsf{src}, \mathsf{new})$ returns $\mathsf{dst} = \mathcal{M}[\mathsf{src}]$ and updates the store so that $\mathcal{M}[\mathsf{src}] := \mathsf{new}$. The constraint $\mathsf{checkzero}(\mathsf{new} - \mathsf{dst})$ enforces $\mathsf{new} = \mathsf{dst}$, hence the update is a no-op and $\mathcal{M}$ is unchanged. By the KVS invariants, $\mathsf{De}(\mathcal{M}[\mathsf{src}]) \in \mathbb{W}$, and therefore $\mathsf{De}(\mathsf{dst}) \in \mathbb{W}$. □

Remark. Although the LOAD instruction is specified at the assembly level to take a 32-bit memory address, the above correctness argument only requires that src lie in $\mathsf{Dom}(\mathcal{M})$. This more general form is intentional: the same subcircuit is reused to read lookup-table entries, whose keys lie outside the RAM address range.

STORE (Gadget 5).

Claim. Let $\mathsf{src}_1, \mathsf{src}_2 \in \mathbb{F}_p$ satisfy $\mathsf{De}(\mathsf{src}_1), \mathsf{De}(\mathsf{src}_2) \in \mathbb{W}$ and $\mathsf{src}_1 \in \mathsf{Dom}(\mathcal{M})$. Assume the KVS invariants hold. Then for any assignment satisfying the constraints in Gadget 5, the output $\mathsf{dst} \in \mathbb{F}_p$ satisfies $\mathsf{dst} = 0$ (inferring $\mathsf{De}(\mathsf{dst}) \in \mathbb{W}$), the key-value store $\mathcal{M}$ is updated as $\mathcal{M}[\mathsf{src}_1] := \mathsf{src}_2$ and the KVS invariants still hold.

Proof. Correctness follows immediately from the Fact D.6. It remains to argue preservation of the KVS invariants.

Since $\mathsf{De}(\mathsf{src}_1) \in \mathbb{W} = [0, 2^{32})$, while every lookup-table key decodes to an integer in $[2^{32}, p)$ by Fact D.8, the updated key src_1 cannot belong to any lookup-table range. Hence all lookup-table entries are unchanged and remain correct. Moreover, $\mathsf{De}(\mathsf{src}_2) \in \mathbb{W}$ by assumption, so the updated RAM entry remains well-typed, and all other RAM entries are unchanged. Therefore, the KVS invariants continue to hold. □

BYTEDECOMP (Gadget 7).

Claim. Let $\mathsf{u} \in \mathbb{F}_p$ satisfy $\mathsf{De}(\mathsf{u}) \in \mathbb{W}$. Assume the KVS invariants hold and that $\mathsf{set}()$ ranges over $\mathsf{Em}([0, 2^{16}))$. Let $\mathsf{off} := \mathsf{De}(\mathsf{offset}_{\mathsf{HIGH8}})$, where $\mathsf{off} \geq 2^{32}$ by Fact D.8, and assume $\mathsf{off} + 2^{32} < p$.

Assume furthermore that within the decoded window $[\mathsf{off} - 2^{32} + 1,\ \mathsf{off} + 2^{32} - 1]$ the key-value store domain contains exactly the HIGH8 table keys, i.e., $\mathsf{Em}\big([\mathsf{off} - 2^{32} + 1,\ \mathsf{off} + 2^{32} - 1]\big) \cap \mathsf{Dom}(\mathcal{M}) = \mathsf{Em}\big([\mathsf{off},\ \mathsf{off} + 2^{16} - 1]\big)$. This implies that, for any $x \in \mathbb{F}_p$ and any integer $s \in [-2^{32} + 1, 2^{32} - 1]$ such that $x = \mathsf{Em}(s)$, we have

$$\mathsf{offset}_{\mathsf{HIGH8}} + x \in \mathsf{Dom}(\mathcal{M}) \implies s \in [0, 2^{16}) \text{ and } \mathsf{De}(x) = s$$

Then for any satisfying assignment to Gadget 7, the outputs $(u_0, u_1, u_2, u_3) \in \mathbb{F}_p^4$ satisfy $\mathsf{De}(u_i) \in [0, 2^8)$ for all $i \in \{0, 1, 2, 3\}$ and

$$\mathsf{De}(\mathsf{u}) = \mathsf{De}(u_0) + 2^8\mathsf{De}(u_1) + 2^{16}\mathsf{De}(u_2) + 2^{24}\mathsf{De}(u_3)$$

Moreover, $\mathcal{M}$ is unchanged and the KVS invariants still hold.

Proof. By Fact D.4, $\mathsf{De}(\mathsf{hi}) \in [0, 2^{16})$. By correctness of LOAD and the HIGH8 table (KVS invariants), we have

$$\mathsf{De}(u_2), \mathsf{De}(u_3) \in [0, 2^8) \quad \text{and} \quad \mathsf{De}(\mathsf{hi}) = \mathsf{De}(u_2) + 2^8\mathsf{De}(u_3)$$

Let $s := \mathsf{De}(\mathsf{u}) - 2^{16}\mathsf{De}(\mathsf{hi}) \in \mathbb{Z}$. Since $\mathsf{De}(\mathsf{u}) \in [0, 2^{32})$ and $\mathsf{De}(\mathsf{hi}) \in [0, 2^{16})$, we have $s \in [-2^{32} + 1,\ 2^{32} - 1]$. Moreover,

$$\mathsf{lo} \ = \ \mathsf{u} - 2^{16}\mathsf{hi} \ = \ \mathsf{Em}(\mathsf{De}(\mathsf{u})) - \mathsf{Em}(2^{16})\,\mathsf{Em}(\mathsf{De}(\mathsf{hi})) \ = \ \mathsf{Em}(s)$$

where we used $\mathsf{u} = \mathsf{Em}(\mathsf{De}(\mathsf{u}))$ and $\mathsf{hi} = \mathsf{Em}(\mathsf{De}(\mathsf{hi}))$.

By correctness of LOAD, the call $u_1 := \mathsf{LOAD}(\mathsf{offset}_{\mathsf{HIGH8}} + \mathsf{lo})$ is well-defined, hence $\mathsf{offset}_{\mathsf{HIGH8}} + \mathsf{lo} \in \mathsf{Dom}(\mathcal{M})$. Together with the window assumption (applied to $x := \mathsf{lo} = \mathsf{Em}(s)$) we obtain

$$s \in [0, 2^{16}) \quad \text{and} \quad \mathsf{De}(\mathsf{lo}) = s$$

Again by correctness of LOAD and the HIGH8 table, we have

$$\mathsf{De}(u_0), \mathsf{De}(u_1) \in [0, 2^8) \quad \text{and} \quad \mathsf{De}(\mathsf{lo}) = \mathsf{De}(u_0) + 2^8\mathsf{De}(u_1)$$

Finally, since $\mathsf{u} = \mathsf{lo} + 2^{16}\mathsf{De}(\mathsf{hi})$ and $\mathsf{De}(\mathsf{lo}) + 2^{16}\mathsf{hi} < p$, an application of Fact D.2 yields

$$\mathsf{De}(\mathsf{u}) = \mathsf{De}(u_0) + 2^8\mathsf{De}(u_1) + 2^{16}\mathsf{De}(u_2) + 2^{24}\mathsf{De}(u_3)$$

Finally, the gadget performs exactly two LOAD calls and no other KVS-related operations. By correctness of LOAD, $\mathcal{M}$ is unchanged and the KVS invariants continue to hold. □

LO2 (Gadget 8).

Claim. Let $\mathsf{src} \in \mathbb{F}_p$ satisfy $\mathsf{De}(\mathsf{src}) \in \mathbb{W}$. Assume the KVS invariants hold and that $\mathsf{set}()$ ranges over $\mathsf{Em}([0, 2^{16}))$. Let $\mathsf{off} := \mathsf{De}(\mathsf{offset}_{\mathsf{LO2}})$, where $\mathsf{off} \geq 2^{32}$ by Fact D.8, and assume $\mathsf{off} + 2^{32} < p$.

Assume furthermore that within the decoded window

$$[\mathsf{off} - 2^{32} + 1,\ \mathsf{off} + 2^{32} - 1]$$

the key-value store domain contains exactly the LO2 table keys, i.e., $\mathsf{Em}\big([\mathsf{off} - 2^{32} + 1,\ \mathsf{off} + 2^{32} - 1]\big) \cap \mathsf{Dom}(\mathcal{M}) = \mathsf{Em}\big([\mathsf{off},\ \mathsf{off} + 2^{16} - 1]\big)$.

Then for any satisfying assignment to Gadget 8, the output $\mathsf{dst} \in \mathbb{F}_p$ satisfies

$$\mathsf{De}(\mathsf{dst}) = \mathsf{De}(\mathsf{src}) \bmod 4 \quad \text{and} \quad \mathsf{De}(\mathsf{dst}) \in [0, 4)$$

Moreover, $\mathcal{M}$ is unchanged and the KVS invariants still hold.

Proof. The proof follows directly from the BYTEDECOMP inspection. As there, writing $\mathsf{lo} = \mathsf{src} - 2^{16}\mathsf{hi} = \mathsf{Em}(s)$ with $s = \mathsf{De}(\mathsf{src}) - 2^{16}\mathsf{De}(\mathsf{hi})$, the fact that the LOAD is defined forces $s \in [0, 2^{16})$ and hence $\mathsf{De}(\mathsf{lo}) = s$. By correctness of the LO2 table, $\mathsf{De}(\mathsf{dst}) = \mathsf{De}(\mathsf{src}) \bmod 4$. The single LOAD call leaves $\mathcal{M}$ unchanged, so the KVS invariants continue to hold. □

POW2 (Gadget 16).

Claim. Let $\mathsf{k} \in \mathbb{F}_p$ satisfy $\mathsf{De}(\mathsf{k}) \in [0, 32)$. Assume the KVS invariants hold. Let $\mathsf{off} := \mathsf{De}(\mathsf{offset}_{\mathsf{POW2}})$, where $\mathsf{off} \geq 2^{32}$ by Fact D.8, and assume $\mathsf{off} + 32 < p$. Then for any satisfying assignment to Gadget 16, the output $\mathsf{dst} \in \mathbb{F}_p$ satisfies

$$\mathsf{De}(\mathsf{dst}) = 2^{\mathsf{De}(\mathsf{k})} \in \mathbb{W}$$

Moreover, $\mathcal{M}$ is unchanged and the KVS invariants still hold.

Proof. This follows directly from correctness of LOAD and the POW2 lookup table. □

Remark. If the decoded window $[\mathsf{De}(\mathsf{offset}_{\mathsf{POW2}}),\ \mathsf{De}(\mathsf{offset}_{\mathsf{POW2}}) + 2^{32} - 1]$ contains exactly the POW2 table keys, then any satisfying assignment must have $\mathsf{De}(\mathsf{k}) \in [0, 32)$; otherwise the LOAD is undefined and the proof is rejected.

SRA (Gadget 17).

Claim. Let $\mathsf{src}, \mathsf{sh} \in \mathbb{F}_p$ satisfy $\mathsf{De}(\mathsf{src}) \in \mathbb{W}$ and $\mathsf{De}(\mathsf{sh}) \in [0, 32)$. Assume $p > 2^{49}$ and that $\mathsf{set}()$ ranges over $\mathsf{Em}([0, 2^{16}))$. Assume the KVS invariants hold. Then for any satisfying assignment to Gadget 17, the output $\mathsf{dst} \in \mathbb{F}_p$ satisfies

$$\mathsf{De}(\mathsf{dst}) = \left\lfloor \frac{\mathsf{De}(\mathsf{src})}{2^{\mathsf{De}(\mathsf{sh})}} \right\rfloor \quad \text{if } \mathsf{De}(\mathsf{src}) < 2^{31}$$

and

$$\mathsf{De}(\mathsf{dst}) = \left\lfloor \frac{\mathsf{De}(\mathsf{src})}{2^{\mathsf{De}(\mathsf{sh})}} \right\rfloor + 2^{32} - 2^{32-\mathsf{De}(\mathsf{sh})} \quad \text{otherwise}$$

i.e., dst equals the signed right shift src $\gg$ sh, with $\mathsf{De}(\mathsf{dst}) \in \mathbb{W}$. Moreover, $\mathcal{M}$ is unchanged and the KVS invariants still hold.

Proof. By correctness of MSB, the bit $\mathsf{De}(s) \in \{0,1\}$ equals the sign bit of $\mathsf{De}(\mathsf{src}) \in \mathbb{W}$. By correctness of POW2, we have $\mathsf{De}(\mathsf{pow}) = 2^{\mathsf{De}(\mathsf{sh})} \in \mathbb{W} \setminus \{0\}$. By correctness of UDIV, the value

$$\mathsf{De}(q) = \lfloor \mathsf{De}(\mathsf{src}) / \mathsf{De}(\mathsf{pow}) \rfloor \in \mathbb{W}$$

The constraint checkzero$(t \cdot \mathsf{pow} - 2^{32})$ enforces $\mathsf{De}(t) = 2^{32-\mathsf{De}(\mathsf{sh})}$. Note that the gadget outputs

$$\mathsf{dst} = q + s \cdot (2^{32} - t).$$

If $s = 0$ (i.e., $\mathsf{De}(\mathsf{src}) < 2^{31}$), then $\mathsf{dst} = q$, which trivially ensures $\mathsf{De}(\mathsf{dst}) = \left\lfloor \frac{\mathsf{De}(\mathsf{src})}{2^{\mathsf{De}(\mathsf{sh})}} \right\rfloor \in \mathbb{W}$. If $s = 1$ (i.e., $\mathsf{De}(\mathsf{src}) \geq 2^{31}$), then $\mathsf{dst} = q + 2^{32} - 2^{32-\mathsf{De}(\mathsf{sh})}$. Now since $\mathsf{De}(q) \in [0, 2^{32})$ and $2^{32} - 2^{32-\mathsf{De}(\mathsf{sh})} \in [0, 2^{32})$ and $p > 2^{49}$, applying Fact D.2 yields $\mathsf{De}(\mathsf{dst}) = \left\lfloor \frac{\mathsf{De}(\mathsf{src})}{2^{\mathsf{De}(\mathsf{sh})}} \right\rfloor + 2^{32} - 2^{32-\mathsf{De}(\mathsf{sh})}$, which lies in $\mathbb{W}$.

Finally, the gadget performs exactly one POW2 call and no other KVS-related operations. By correctness of POW2, $\mathcal{M}$ is unchanged and the KVS invariants continue to hold. □

Remark. Under the same window assumption for the POW2 table as above, any satisfying assignment to SRA must satisfy $\mathsf{De}(\mathsf{sh}) \in [0, 32)$, since otherwise the LOAD inside POW2 is undefined and the proof is rejected.

XOR (Gadget 18).

Claim. Let $\mathsf{src}_1, \mathsf{src}_2 \in \mathbb{F}_p$ satisfy $\mathsf{De}(\mathsf{src}_1), \mathsf{De}(\mathsf{src}_2) \in \mathbb{W}$. Assume the KVS invariants hold and that set() ranges over $\mathsf{Em}([0, 2^{16}))$. Let $\mathsf{off} := \mathsf{De}(\mathsf{offset}_{\mathsf{HIGH8}})$, where $\mathsf{off} \geq 2^{32}$ by Fact D.8, and assume $\mathsf{off} + 2^{32} < p$.

Assume furthermore that within the decoded window $[\mathsf{off} - 2^{32} + 1,\ \mathsf{off} + 2^{32} - 1]$ the key-value store domain contains exactly the HIGH8 table keys, i.e., $\mathsf{Em}\big([\mathsf{off} - 2^{32} + 1,\ \mathsf{off} + 2^{32} - 1]\big) \cap \mathsf{Dom}(\mathcal{M}) = \mathsf{Em}\big([\mathsf{off},\ \mathsf{off} + 2^{16} - 1]\big)$.

Then for any satisfying assignment to Gadget 18, the output $\mathsf{dst} \in \mathbb{F}_p$ satisfies $\mathsf{De}(\mathsf{dst}) \in \mathbb{W}$ and

$$\mathsf{De}(\mathsf{dst}) = \mathsf{De}(\mathsf{src}_1) \mathbin{\mathsf{xor}} \mathsf{De}(\mathsf{src}_2)$$

where xor denotes bitwise XOR on 32-bit words. Moreover, $\mathcal{M}$ is unchanged and the KVS invariants continue to hold.

Proof. This follows immediately from the correctness of LOAD, BYTEDECOMP, and the XOR lookup table. Moreover, since the gadget performs only LOAD and BYTEDECOMP calls, $\mathcal{M}$ is unchanged and the KVS invariants continue to hold. □

ULOADB (Gadget 19).

Claim. Let $\mathsf{addr} \in \mathbb{F}_p$ satisfy $\mathsf{De}(\mathsf{addr}) \in \mathbb{W}$. Assume the KVS invariants hold and that the hypotheses of the LO2 and BYTEDECOMP inspections hold. Then for any satisfying assignment to Gadget 19, the output $\mathsf{dst} \in \mathbb{F}_p$ satisfies

$$\mathsf{De}(\mathsf{dst}) = \text{the byte stored at address } \mathsf{De}(\mathsf{addr})$$

That is, let $m := \mathsf{De}(\mathsf{addr}) \bmod 4$ and $W := \mathsf{De}(\mathcal{M}[\mathsf{addr} - \mathsf{Em}(m)])$, then $\mathsf{De}(\mathsf{dst}) =$ the m-th byte of W. Moreover, $\mathsf{De}(\mathsf{dst}) \in [0, 2^8) \subset \mathbb{W}$, $\mathcal{M}$ is unchanged and the KVS invariants continue to hold.

Proof. By correctness of LO2, we have $\mathsf{De}(r) = \mathsf{De}(\mathsf{addr}) \bmod 4$. Thus $\mathsf{addr}' = \mathsf{addr} - r$ is the word-aligned address containing the target byte. By correctness of LOAD, $\mathsf{word} = \mathcal{M}[\mathsf{addr}']$ such that $\mathsf{De}(\mathsf{word}) \in \mathbb{W}$. By correctness of BYTEDECOMP, (a_0, a_1, a_2, a_3) are exactly encoding the four bytes of word.

The constraints on (b_0, b_1, b_2, b_3) enforce that exactly one b_i equals 1, with index determined by r, and all others equal 0. Hence $\mathsf{dst} = a_r$, which is precisely the byte at address $\mathsf{De}(\mathsf{addr})$, zero-extended to a word. Hence $\mathsf{dst} \in [0, 2^8) \subset \mathbb{W}$.

Finally, all KVS accesses in ULOADB boil down to LOADs, so $\mathcal{M}$ is unchanged and the KVS invariants continue to hold. □

ULOADH (Gadget 20).

Claim. Let $\mathsf{addr} \in \mathbb{F}_p$ satisfy $\mathsf{De}(\mathsf{addr}) \in \mathbb{W}$. Assume the KVS invariants hold, that $\mathsf{set}()$ ranges over $\mathsf{Em}([0, 2^{16}))$ and that the hypotheses of the LO2 inspection hold. Then for any satisfying assignment to Gadget 20, the output $\mathsf{dst} \in \mathbb{F}_p$ satisfies

$$\mathsf{De}(\mathsf{dst}) = \text{the halfword stored at address } \mathsf{De}(\mathsf{addr})$$

That is, let $m := \mathsf{De}(\mathsf{addr}) \bmod 4$ and $W := \mathsf{De}(\mathcal{M}[\mathsf{addr} - \mathsf{Em}(m)])$, then

$$\mathsf{De}(\mathsf{dst}) = \begin{cases} W \bmod 2^{16}, & m = 0, \\ \lfloor W / 2^{16} \rfloor, & m = 2, \end{cases}$$

Moreover, $\mathsf{De}(\mathsf{dst}) \in [0, 2^{16}) \subset \mathbb{W}$, $\mathcal{M}$ is unchanged and the KVS invariants continue to hold.

Proof. By correctness of LO2, we have $\mathsf{De}(r) = \mathsf{De}(\mathsf{addr}) \bmod 4$. Thus $\mathsf{addr}' = \mathsf{addr} - r$ is the word-aligned address containing the target halfword. By correctness of LOAD, $\mathsf{word} = \mathcal{M}[\mathsf{addr}']$ such that $\mathsf{De}(\mathsf{word}) \in \mathbb{W}$.

By Facts D.2 and D.4, the constraint $\mathsf{checkzero}(\mathsf{word} - 2^{16} h_1 - h_0)$ enforces the decomposition

$$\mathsf{De}(\mathsf{word}) = \mathsf{De}(h_0) + 2^{16} \mathsf{De}(h_1)$$

where $\mathsf{De}(h_0), \mathsf{De}(h_1) \in [0, 2^{16})$.

By Fact D.5, $b \in \{0, 1\}$, and $\mathsf{checkzero}(2b_1 - r)$ enforces $r \in \{0, 2\}$ with $b_1 = 1$ iff $r = 2$. Hence $b_0 = 1 - b$ and $b_1 = b$ select h_0 when $r = 0$ and h_1 when $r = 2$. Therefore, $\mathsf{dst} = b_0 h_0 + b_1 h_1$ is exactly the halfword stored at address $\mathsf{De}(\mathsf{addr})$ and $\mathsf{De}(\mathsf{dst}) \in [0, 2^{16})$.

Finally, all KVS accesses in ULOADH boil down to LOADs, so $\mathcal{M}$ is unchanged and the KVS invariants continue to hold. □

Remark. The constraint $\mathsf{checkzero}(2b_1 - r)$ enforces $r \in \{0, 2\}$. Hence, if $\mathsf{De}(\mathsf{addr}) \bmod 4 \in \{1, 3\}$, the constraints in ULOADH are unsatisfiable and any proof is rejected.

LOADB (Gadget 21).

Claim. Under the same assumptions as in the ULOADB inspection, LOADB returns the byte at address $\mathsf{De}(\mathsf{addr})$ with 8-bit two's-complement sign extension to a 32-bit word. Moreover, $\mathcal{M}$ is unchanged and the KVS invariants continue to hold.

Proof. By correctness of ULOADB, byte satisfies $\mathsf{De}(\mathsf{byte}) \in [0, 2^8)$ and equals the byte stored at address $\mathsf{De}(\mathsf{addr})$. By Fact D.5, $\mathsf{msb} \in \{0, 1\}$. The gadget enforces

$$2^9\mathsf{byte} = \mathsf{tmp_scaled} + 2^{16}\mathsf{msb} \quad \text{in } \mathbb{F}_p$$

By Fact D.4, $\mathsf{De}(\mathsf{tmp_scaled}) \in [0, 2^{16})$, applying Fact D.2 yields

$$2^9\mathsf{De}(\mathsf{byte}) = \mathsf{De}(\mathsf{tmp_scaled}) + 2^{16}\mathsf{De}(\mathsf{msb}) \quad \text{in } \mathbb{Z}$$

This implies

$$\mathsf{De}(\mathsf{msb}) = \left\lfloor \mathsf{De}(\mathsf{byte})/2^7 \right\rfloor$$

i.e., msb equals the most significant bit of the byte. The gadget then outputs

$$\mathsf{dst} = \mathsf{byte} + \mathsf{msb} \cdot (2^{32} - 2^8),$$

which equals b when $b < 2^7$ and equals $b + 2^{32} - 2^8$ otherwise, exactly implementing sign extension, inferring $\mathsf{De}(\mathsf{dst}) \in \mathbb{W}$.

All KVS accesses in LOADB occur inside ULOADB and are LOADs; hence $\mathcal{M}$ is unchanged and the KVS invariants continue to hold. □

LOADH (Gadget 22).

Claim. Under the same assumptions as in the ULOADH inspection, LOADH returns the halfword at address $\mathsf{De}(\mathsf{addr})$ with 16-bit two's-complement sign extension to a 32-bit word. Moreover, $\mathcal{M}$ is unchanged and the KVS invariants continue to hold.

Proof. This follows directly from the proof of LOADB. □

Remark. The constraint $\mathsf{checkzero}(2b_1 - r)$ inside ULOADH enforces $r \in \{0, 2\}$. Hence, if $\mathsf{De}(\mathsf{addr}) \bmod 4 \in \{1, 3\}$, the constraints in ULOADH are unsatisfiable and any proof is rejected.

STOREB (Gadget 23).

Claim. Let $\mathsf{addr}, \mathsf{src} \in \mathbb{F}_p$ satisfy $\mathsf{De}(\mathsf{addr}), \mathsf{De}(\mathsf{src}) \in \mathbb{W}$. Assume the KVS invariants hold and that the hypotheses of the LO2, LOAD, BYTEDECOMP, and STORE inspections hold.

Let $m := \mathsf{De}(\mathsf{addr}) \bmod 4$ and $\mathsf{val} := \mathsf{De}(\mathsf{src}) \bmod 2^8$. Then for any satisfying assignment to Gadget 23, the output $\mathsf{dst} \in \mathbb{F}_p$ satisfies $\mathsf{De}(\mathsf{dst}) = 0$, and the key-value store is updated so that the m-th byte of $\mathcal{M}[\mathsf{addr} - \mathsf{Em}(m)]$ is set to val, while all other bytes are unchanged. Moreover, the KVS invariants remain valid.

Proof. This follows immediately from the inspection of ULOADB; the only additional step is enforcing $\mathsf{De}(v) = \mathsf{De}(\mathsf{src}) \bmod 2^8$.

This is ensured as follows. The gadget ensures $\mathsf{lo} = \mathsf{src} - 2^{16}\mathsf{hi}$. Let $s := \mathsf{De}(\mathsf{src}) - 2^{16}\mathsf{De}(\mathsf{hi}) \in \mathbb{Z}$. Since $\mathsf{De}(\mathsf{src}) \in [0, 2^{32})$ and $\mathsf{De}(\mathsf{hi}) \in [0, 2^{16})$ (Fact D.4), we have $s \in [-2^{32} + 1,\ 2^{32} - 1]$. Moreover,

$$\mathsf{lo} \;=\; \mathsf{src} - 2^{16}\mathsf{hi} \;=\; \mathsf{Em}(\mathsf{De}(\mathsf{src})) - \mathsf{Em}(2^{16})\,\mathsf{Em}(\mathsf{De}(\mathsf{hi})) \;=\; \mathsf{Em}(s)$$

where we used $\mathsf{src} = \mathsf{Em}(\mathsf{De}(\mathsf{src}))$ and $\mathsf{hi} = \mathsf{Em}(\mathsf{De}(\mathsf{hi}))$.

By correctness of LOAD, the call $v_1 := \mathsf{LOAD}(\mathsf{offset}_{\mathsf{HIGH8}} + \mathsf{lo})$ is well-defined, hence $\mathsf{offset}_{\mathsf{HIGH8}} + \mathsf{lo} \in \mathsf{Dom}(\mathcal{M})$. Together with the window assumption (applied to $x := \mathsf{lo} = \mathsf{Em}(s)$) we obtain

$$s \in [0, 2^{16}) \quad \text{and} \quad \mathsf{De}(\mathsf{lo}) = s$$

By correctness of LOAD and the HIGH8 table, we have

$$\mathsf{De}(v) = \mathsf{De}(\mathsf{lo}) \bmod 2^8 = \mathsf{De}(\mathsf{src}) \bmod 2^8$$

where we implicitly apply Fact D.2.

Moreover, note that $\mathsf{De}(\mathsf{new}) \in \mathbb{W}$ and other KVS accesses in STOREB boil down to LOADs, by the correctness of STORE, the KVS invariants continue to hold. □

STOREH (Gadget 24).

Claim. Let $\mathsf{addr}, \mathsf{src} \in \mathbb{F}_p$ satisfy $\mathsf{De}(\mathsf{addr}), \mathsf{De}(\mathsf{src}) \in \mathbb{W}$. Assume the KVS invariants hold, that $\mathsf{set}()$ ranges over $\mathsf{Em}([0, 2^{16}))$ and that the hypotheses of the LO2 inspection hold.

Let $m := \mathsf{De}(\mathsf{addr}) \bmod 4$ and $\mathsf{val} := \mathsf{De}(\mathsf{src}) \bmod 2^{16}$. Then for any satisfying assignment to Gadget 24, the output $\mathsf{dst} \in \mathbb{F}_p$ satisfies $\mathsf{De}(\mathsf{dst}) = 0$, and the key-value store is updated so that the $\frac{m}{2}$-th halfword of $\mathcal{M}[\mathsf{addr} - \mathsf{Em}(m)]$ is set to val, while the other halfword is unchanged. Moreover, the KVS invariants remain valid.

Proof. This follows immediately from the inspection of ULOADH; the only additional step is enforcing $\mathsf{De}(v) = \mathsf{De}(\mathsf{src}) \bmod 2^{16}$, which is straightforward. □

Remark. The constraint $\mathsf{checkzero}(2b_1 - r)$ enforces $r \in \{0, 2\}$. Hence, if $\mathsf{De}(\mathsf{addr}) \bmod 4 \in \{1, 3\}$, the constraints in STOREH are unsatisfiable and any proof is rejected.

D.4 Composition of Gadget Inspections

We conclude by explaining how the above gadget-level inspections compose into an end-to-end correctness statement.

Each gadget inspection establishes a form of *local correctness*: under its explicit preconditions, any satisfying assignment to the gadget constraints (i) realizes the intended ZEE instruction semantics, and (ii) preserves the invariants it assumes (local typing invariants for gadgets without KVS access, and additionally the KVS invariants for gadgets with KVS access). Consequently, for any sequence of gadget invocations whose preconditions hold at each step, correctness follows by a straightforward induction argument.

Note, all preconditions (and initial invariants) used throughout the inspections are simultaneously satisfied under the concrete parameter choices used in ZEE200. For example, for our field choice $p = 2^{61} - 1$, all numeric no-wraparound requirements on p appearing in the claims are satisfied.

Together with the ideal GTZK functionality $\mathcal{F}_{\mathsf{G\text{-}ZKCPU}}$, these compositional correctness guarantees imply that ZEE200 correctly executes ZEE assembly programs (on the GTZK CPU).