

ZIREN: Succinct Arguments for MIPS32 Execution*

Stephen Duan^{†1, 2}, Jingping Yin¹, Blake Hou¹, Vanhger Pham¹, Ethan Zhu², and Ming Guo¹

¹ZKM Team

²GOAT Network

October 2026

Abstract

Proving MIPS32 execution requires preserving branch-delay semantics while controlling the cost of heterogeneous instruction constraints. We present ZIREN, a production zkVM for MIPS32r2 that supports 77 user-mode integer instructions and has proved Ethereum mainnet blocks end to end. Its arithmetisation embeds the program-counter pair in per-opcode instruction frames, eliminating the central CPU table. Each shard is proved with a single lookup argument, a zerocheck and a jagged commitment opened through batched WHIR; recursive aggregation uses an enumerated verifying-key allowlist.

We mechanically extract 104 determinism theorems from the prover’s constraint description and discharge them in Lean 4. The kernel-checked proofs cover 57 of the core machine’s 62 chips, including precompiles, and establish unique outputs for fixed inputs under the declared lookup-table assumptions. The exclusions are three preprocessed tables and two SHA-256 control chips. We also formulate a cost model that separates trace cells and bus interactions per guest instruction from hardware-dependent proving throughput, accounting for instruction frames, shard-boundary memory rows and precompile expansion.

On the evaluated Ethereum blocks, warm proving throughput reaches 12.0 MHz on one NVIDIA RTX 5090 GPU and 75.8 MHz on eight. Each proof stage has more than 100 bits of interactive soundness; a union bound for a tree with 149 core-schedule proofs and one root gives 93.5 bits of interactive composite security. End-to-end knowledge soundness remains conditional on five stated hypotheses, including enforcement of the recursion relation and computational binding of the cross-shard digest. No numerical security bound is established for that digest; the interactive estimates therefore do not quantify the full security of the deployed non-interactive proof.

Index terms: arithmetic circuits, formal verification, GPU acceleration, MIPS32, polynomial commitment schemes, round-by-round soundness, verifiable computation, zero-knowledge virtual machines.

1 Introduction

A *succinct argument for machine execution* convinces a verifier that a program, run on a given instruction-set architecture, produced a given output, with a proof far shorter and a check far cheaper than the execution. Such systems are commonly called zero-knowledge virtual machines (zkVMs), although the core execution proofs studied here are succinct but not hiding. The benchmark throughout is an Ethereum block of several hundred million instructions [Eth26b], a hash-intensive workload that we use as a benchmark rather than as the application.

1.1 Succinct arguments for machine execution

A zkVM fixes an instruction set, a register file and a memory model, and proves that a program for that machine ran correctly. Representative systems that prove RISC-V programs are SP1, OpenVM, ZisK, Jolt and Airbender [Suc24, Ope26a, Zis25, AST24, Mat25], the five systems in our design comparison. Such a

*Source code: <https://github.com/ProjectZKM/Ziren>. Measurements name the configuration they were taken on (§9.1).

[†]Corresponding author: stephen.d@zkm.io

system has two halves, a front end and a back end. The *arithmetisation* turns an execution into tables whose rows satisfy polynomial constraints and between which multiset relations hold; the *commitment* and its proximity test let the prover show that it knows such tables. The arithmetisation determines the trace cells and bus interactions per instruction; the commitment and proximity test determine the cost of encoding, hashing and opening those cells.

Motivation. ZIREN is a production zkVM for MIPS32. It proves 77 user-mode integer instructions of MIPS32r2 with the branch-delay slot (Appendix B, which also names the few it leaves out), and it has proved Ethereum mainnet blocks end to end in production; earlier proof systems address MIPS-*like* models or offer MIPS as one mode among several (§1.6). The specification is *stable*: its owner moved its roadmap to RISC-V in 2021 [Tur21, Har25], so the target ISA specification is fixed; changes to the arithmetisation still change its verifying keys. It is *uniform*: fixed 32-bit encodings in a handful of formats let one row carry an instruction’s whole frame. And its toolchains are *mature*: Rust, Go and C programs compile to the same little-endian ELF, and the architecture is widely deployed [MIP26]. The fault-proof software of Ethereum rollups, Optimism’s Cannon and Kona [Opt23, OP 24], can therefore be targeted, subject to compatible runtime conventions and image formats. On constrained devices, proofs today serve authentication and attestation [CJSC23, SD21, DDGM23, EH24]; a zkVM for the device’s instruction set makes its software execution the statement instead.

Challenges. Four properties shape the design. The instruction after a branch executes before control transfers,¹ so the next instruction address must be carried as machine state rather than inferred as `pc + 4`. The relations that hold the machine together are multiset relations *across* tables, so discharging them table by table would cost one argument per bus per table. A shard of the block workload commits up to about $3.6 \cdot 10^4$ columns whose heights run from 32 rows to millions. Separate commitments require many Merkle roots and authentication paths; a common height wastes space on padding. And a 32-bit word does not fit the 31-bit field, so every word is a vector of bytes.

1.2 Overview of Ziren

ZIREN represents each executed instruction by one row in its opcode-specific trace table (*chip*), with no central CPU table. Each chip embeds an *instruction frame*: shared columns and bus interactions that bind the instruction to the committed program, enforce register accesses at fixed sub-cycle positions and constrain the machine-state transition. The state includes both the current and next program counters, preserving the branch-delay slot (§5.3). The committed trace is thus the execution regrouped by opcode, the device of vRAM [ZGK⁺18], with the state bus as its permutation check.

One shard proof covers the chip rows and their bus interactions. Registers and memory share one offline memory-checking argument, carried across shards by an elliptic-curve digest; one LogUp-GKR instance checks all bus balances and one zerocheck checks all chip constraints; and the jagged sumcheck of Hemo et al. [HJR⁺25] reduces the claims on tens of thousands of columns of differing heights to one evaluation, which a single batched WHIR opening [ACFY24b] authenticates against every commitment. Recursion composes the shard proofs under a verifying-key allowlist. A block proof is

$$\text{Rec} \left[\text{FS} \left[\text{JaggedWHIR} \left[\text{LogUpGKR} \wedge \text{Zerocheck} \left[\mathcal{R}_{\text{shard}} \right] \right] \right] \right],$$

and Figure 1 draws the pipeline. Section 3 describes the system as a stack of these arguments ending in a compressed proof for a native verifier, and two verification routes for applications in EVM-compatible rollups, private computation and Bitcoin rollups: SNARK wrapping and garbled verification (§3.5).

Contributions. The commitment scheme, proximity test, lookup argument and recursion shape ZIREN builds on are prior work, used as published; our contributions are the machine construction, mechanical determinism proofs and a cost model.

¹For jumps and conditional branches alike, and after a conditional branch whether or not it is taken [MIP10]; an argument system must constrain the delay slot, since normalising it away in the emulator proves a different machine.

- *The machine.* An arithmetisation of MIPS32 with no CPU table, in which an instruction frame carries the delay slot as a program-counter pair (Sections 4 and 5); the concrete composition of the jagged reduction with WHIR, with its multi-root layout, parameter schedule and recursive verifier (Sections 6 and 7).
- *Determinism in Lean 4.* The constraint system of every chip the extraction models (57 chips) is taken from the Rust description the prover evaluates, and its determinism is stated as a Lean 4 theorem; a propagation analysis derives each output from the inputs, and the generator replays that derivation as a proof, one step per determined column, over a library of hand-proved gadget lemmas (§8.7). All 104 theorems are closed, with no axiom beyond Lean 4’s standard three.
- *Cost model.* A formulation of instruction efficiency in committed trace cells and bus interactions per guest instruction, separated from hardware-dependent proving throughput. It accounts for instruction frames, shard-boundary memory rows and precompile expansion, and relates the final proof size to proving cost through the query count, code rate, folding schedule and grinding budget, identifying parameter choices that reduce proof size (§1.3 and Section 9).

1.3 Costs

Cost model. ZIREN commits Merkle trees over Reed–Solomon encodings of a 31-bit field, where every cell costs the same whatever it holds; a small value is no cheaper than a large one, unlike under a commitment built from multi-scalar multiplication, where small values are cheaper [AST24]. The units of prover work are therefore trace cells and bus interactions, which determine the size of the constraint, commitment and lookup arguments. For an execution, let C be its guest-instruction count, A its trace area and I its total number of (row, interaction) pairs, as defined in §2.4. Let T be proving time in seconds under the measurement boundary of §9.1. We report

$$\text{instruction efficiency} = (A/C, I/C), \quad \text{proving throughput} = \frac{C}{10^6 T} \text{ MHz}.$$

For a fixed workload and sharding rule, the first pair describes the arithmetisation independently of the hardware; throughput also depends on the hardware processing rate and the execution and recursion stages that cannot overlap. The model counts the frame and opcode logic separately, includes shard-boundary memory rows, and charges the multiple accelerator rows generated by a single precompile call. Section 9 applies these measures to the instruction costs and block-level census.

Prover costs. An executed instruction pays for its frame before it computes anything (§5.8), and the operation adds from four columns (AddSub) to over a hundred (DivRem) among the arithmetic chips (§§5.9 and 5.10); block 25 955 640 commits about 59 cells and raises about 26 bus interactions per executed instruction. Cells per instruction is a property of the workload, not a constant: a precompile call is one guest cycle but many rows, and a shard closes on its committed area, so guest cycles, the unit block-proving dashboards report, do not measure prover work (§9.2). On the 15 consecutive Ethereum mainnet blocks retained in Table 11, warm proofs with Reth and Geth reach 12.0 MHz on one NVIDIA RTX 5090 and 75.8 MHz on eight; the Reth guest, at lower throughput, proves each block 1.8–3.9 times faster than Geth on one GPU (Table 11).

Verifier costs. The compressed proof is 284 KiB and its native verification takes a median of 25.2 ms on the evaluation host, with the in-circuit child-key allowlist checks enabled (§9.4). For a fixed root verifier program, the size is independent of the execution’s length and is set by the opening’s schedule (§9.4). Every proof stage has more than 100 bits of interactive soundness, and the *composite security* of a block, $-\log_2$ of the sum of the interactive error bounds over all tree nodes, is 93.5 bits for a tree with 149 core-schedule proofs and one compress root. These figures bound the *interactive* protocol; §8.2 gives the union-bound accounting and the level after Fiat–Shamir compilation.

1.4 Auditable soundness

An accepting proof implies a satisfying trace; that the trace is an execution of the advertised machine needs the arithmetisation to admit one behaviour per input, which the proof system cannot establish about itself. ZIREN audits its per-opcode constraints mechanically: it extracts the determinism of every chip it models as a theorem statement from the constraint description and discharges all 104 of them in Lean 4. A lookup-only design instead relies on tables fixed by the instruction set, which are easier to audit [AST24].

Theorem 1.1 (Conditional argument statement, informal). *Let **Emulate** be the MIPS32 execution relation of Section 4, and let the block proof be the protocol of Section 3 over the 31-bit field KoalaBear. Honest executions that halt with exit code zero produce accepting proofs (§4.4). Suppose that every node’s protocol is round-by-round knowledge sound, that child extractors accept the transcripts produced by parent extractors, that the recursion programs enforce the compose relation, and that Assumption 8.4 holds (§8.1). Then the recursion tree is knowledge sound for its satisfying witness rows in the random-oracle model, with the conditional error bound of Theorem 8.2; §8.3 separates this statement into a functional IOP, an interactive commit-and-open argument and its Fiat–Shamir compilation. Identifying those rows with a unique **Emulate** execution additionally requires the premises of Theorem 8.6, chip determinism, proved in Lean 4 for every extracted chip (§8.7), well-formed shard boundary rows, and real-row exhaustiveness (§8.1).*

Theorem 1.2 (Chip determinism, informal). *For every chip U of the core machine that the extraction models, there is a mechanically extracted Lean 4 theorem stating that two rows of U that satisfy its constraints and agree on their inputs (their bus inputs and the values the constraints leave to the prover by design) agree on their bus outputs. All 104 such theorems are closed (§8.7). Together with the bus arguments, chip determinism implies that an accepting shard proof determines, up to permutation, every row reachable from the entry state along the state and memory buses, given well-formed shard boundary rows, the initial values of the words the shard touches and those declared free values (Theorem 8.6); that this exhausts the real rows is argued, not proved (§1.7).*

1.5 Technical overview

We illustrate the arithmetisation on a single instruction. Consider a MIPS32 **and** executed at clock clk . It occupies one row of the **Bitwise** chip, which is 37 columns wide. Thirty columns form the instruction frame (§5.8), and two further columns hold the program-counter pair. Through the frame, the row sends $(\text{pc}, \text{opcode}, \dots)$ on the program bus; on the state bus it receives $(\text{shard}, \text{clk}, \text{pc}, \text{next_pc})$ and sends $(\text{shard}, \text{clk} + 5, \text{next_pc}, \text{next_pc} + 4)$; and it performs its three register accesses on the memory bus. The remaining five columns are four selectors and a gate. No polynomial constraint computes the result: it is established by four lookups into a 2^{16} -entry byte table, the decomposition of the original Jolt design (§5.2).

Besides the delay slot (§5.3), the general case raises two further difficulties: a limb identity that holds in $\mathbb{F}$ need not hold over the integers; and memory accesses cross shard boundaries (§5.5). Section 3 states the protocol that operates on these rows.

1.6 Related work

Table 1 places the systems discussed along the axes of this introduction. Its axes are design axes: no cross-system performance claim follows from them, and no same-hardware comparison is attempted. Machine-checked constraint-system correctness is compared only with the published verification work cited below.

Machine arithmetisation. ZIREN inherits MIPS32 and decomposes its execution into opcode chips, each with an instruction frame that carries the delay slot; it has no central CPU table (Table 1(a)). SP1 [Suc24] likewise decomposes RISC-V execution into opcode chips, with **LogUp-GKR** and recursive aggregation; Hypercube [Suc25a] adds a jagged commitment and shard-level lookup and zerocheck, and SP1 also has a just-in-time executor [Suc25b]. OpenVM [Ope26a] has no CPU chip either, but separates an instruction’s adapter from its core. Our frame supplies the MIPS32 state transitions and register accesses that SP1 implements with state-and-reader columns and OpenVM with adapters; Section 5.3 compares the three. We use SP1’s elliptic-curve cross-shard digest [RR24] and register shadow read, credited where they appear.

Table 1: Design axes of ZIREN and five representative systems at the listed releases. The prose below compares the machine, commitment and instruction-evaluation designs. Soundness regimes describe the project’s claim or, where marked, an external soundcalc analysis; they do not imply comparable security levels (Section 8.2).

(a) Machine arithmetisation				
System	ISA	Field	Instruction tables	Lookup
ZIREN	MIPS32	KoalaBear (31-bit)	Chip per opcode; instruction frames	LogUp-GKR
SP1	RV64IM	KoalaBear	Chip per opcode; readers	LogUp-GKR
Hypercube [Suc25a]				
ZisK [Zis25]	RV64IMA	Goldilocks	Micro-operations; PIL2	LogUp
OpenVM [Ope26a]	Own ISA; RV32IM transpiler	BabyBear	Executor chips; adapter and core	LogUp
Jolt [AST24]	RV64IMAC	BN254 scalar (Dory); 128-bit (Akita)	Twist and Shout; small R1CS	Twist and Shout
Airbender [Mat25]	RV32IM (restricted)	Mersenne-31	CPU trace; delegated precompiles	Logarithmic derivative
(b) Commitment and soundness				
System	Commitment / IOPP	Soundness regime	Release	
ZIREN	Jagged over WHIR	Unique decoding; Johnson bound at root; union-bound accounting	This paper	
SP1 Hypercube	Jagged over BaseFold	Unique decoding; proved proximity gaps	v6.8.0	
ZisK	PIL2 (e)STARK	Johnson bound (soundcalc) ^a	v1.3.0-alpha	
OpenVM	SWIRL: stacked WHIR	Unique decoding low in tree; list decoding above	v2.0.2	
Jolt	Dory (default); Akita (lattice)	Not stated	v0.3.0-alpha	
Airbender	DEEP STARK / FRI	Unique decoding / Johnson bound (soundcalc) ^a	v0.5.2	

^aThe soundcalc reports [Eth26a] analyse ZisK v0.16.1 and an Airbender configuration with 87 queries, rather than v0.5.2’s 53. They are not soundness claims for the listed releases.

Cairo [GPR21] and Valida [Lit24], by contrast, design an instruction set for proving rather than inherit a hardware one.

Among MIPS systems, ZIREN arithmetises MIPS32 family by family, including the branch-delay slot. Zilch’s zMIPS verifies a MIPS-like instruction model and composes instruction proofs [MT21]; o1VM documents a segmented prover with MIPS and RISC-V modes [BL24]. Cannon [Opt23] is a single-step MIPS interpreter for interactive fault proofs whose suite is one of our conformance inputs, and Kona [OP 24] is the fault-proof program of the same ecosystem.

Commitment and soundness. ZIREN opens its jagged commitment with WHIR and reports soundness by summing component errors (Table 1(b)). Its query analysis uses unique decoding below the root and the Johnson bound at the root, both with proved bounds (§8.2). SP1 Hypercube instead opens with BaseFold and uses unique decoding with proved proximity-gap bounds [Suc26b]. OpenVM’s SWIRL combines stacked WHIR with LogUp-GKR [Ope26b], but pads columns to power-of-two heights with a univariate skip and reduces openings by a batch sumcheck fused with the lookup’s input layer. ZIREN rounds heights to a multiple of 32 and uses the jagged sumcheck. Its WHIR rounds fold several variables at a time, and KoalaBear rather than BabyBear changes the challenge extension and fingerprinting term. SWIRL composes round-by-round errors by their maximum per layer, which requires different accounting from our union bound. The external soundcalc analyses of ZisK and Airbender in Table 1(b) concern the configurations identified in the table note, rather than the listed releases.

The jagged paper anticipates WHIR in place of its FRI-based opening [HJR⁺25, Rem. 7.1]; our contribution is the concrete composition and its soundness accounting. Nexus 1.0 instead folded executions with Nova-family schemes in a tree of proof-carrying data [Nex24]; Nexus 3.0 replaced folding with a circle STARK over

Mersenne-31, a single trace with a CPU component and a logarithmic-derivative memory check [Nex25].

Instruction evaluation. ZIREN evaluates instructions with per-opcode constraints and uses lookups for byte-granular facts and buses (Table 1(a)). Jolt [AST24] instead evaluates instructions by lookups; since v0.2.0 it uses Twist and Shout [ST25] in place of the Lasso construction [STW24] of its original design, with a small R1CS instance for instruction fetch. The designs share three origins: the byte table comes from the original Jolt design’s decomposition of an instruction table (§5.2), since made unnecessary by Shout [ST25]; the chip-per-opcode layout uses the regrouping by instruction that Jolt, after vRAM [ZGK⁺18], identifies as one way to route a step to its table (§5.3); and the timestamped memory check comes from Spice [SAGL18], which the original Jolt design used before switching to Twist (§5.4). ZisK [Zis25] instead lowers RV64IMA execution to micro-operations constrained in PIL2. Airbender [Mat25] uses a CPU trace with delegated precompiles, a degree-two arithmetisation over Mersenne-31 and DEEP-FRI. RISC Zero [BGt23] similarly proves a trace of machine cycles under a fixed constraint set, selected per cycle by public control columns, with a sorted-column permutation for memory and a DEEP-ALI and FRI STARK; its 2023 draft also claims zero knowledge, which ZIREN’s core proofs do not provide. Section 9 evaluates ZIREN on its own implementation and hardware.

Verifiable RAM and regrouped traces. ZIREN uses three devices from the verifiable-RAM literature: Blum et al.’s offline memory check [BEG⁺94] with Spice’s timestamps [SAGL18], which verifies a read–write memory without reordering the trace and which ZIREN’s registers and memory use (§2.3); vRAM’s regrouping of the trace by instruction, proved a permutation [ZGK⁺18], which Ceno’s per-opcode design [LZZ⁺24, Scr26] published in multiset form and which the frame realises with the delay slot’s counter pair, a shard field and a proved chain lemma (§5.3); and Ceno’s initialisation of only the memory a run touches, which ZIREN shares (§5.4).

Primitives and verification. ZIREN uses Plonky3 [Pol24] for its field, Poseidon2 [GKS23], constraint interfaces and transforms. Circle STARKs [HLP24] reach smooth Mersenne-31 domains through the circle group, and Binius [DP23, DP24] avoids splitting words with binary towers; KoalaBear uses ordinary Reed–Solomon domains. The proximity line runs from FRI [BSBHR18] through DEEP-FRI [BSGKS20] and proximity gaps [BSCI⁺20] to BaseFold [ZCF23], STIR [ACFY24a] and WHIR [ACFY24b]; Ligerio [AHIV17] and Brakedown [GLS⁺21] are linear-time alternatives. The jagged commitment is that of Hemo et al. [HJR⁺25], with the branching-program evaluation of Holmgren and Rothblum [HR18]. The lookup is Haböck’s logarithmic derivative [Hab22] in its GKR realisation [PH23]; Plookup [GW20] and cq [EFG22] are univariate alternatives that do not transfer to this multilinear many-bus workload. Picus [PCW⁺23] introduced the determinism formulation and an SMT solver for it; we discharge it in Lean 4, which gives kernel-checked proofs. OpenVM’s verification project proves refinement of its RV32IM and precompile chips to Lean specifications [Ope26c], a stronger property than determinism, for a different ISA. The soundness accounting follows the WHIR analysis [ACFY24b], computed with `soundcalc` [Eth26a], in the style of the ethSTARK documentation [Sta23].

1.7 Scope and limitations

The evaluation and security accounting cover ZIREN through the compressed proof; the applications built on that proof (§3.5) are outside its scope. Soundness rests on a computational assumption for the cross-shard digest in addition to the information-theoretic bounds of the proximity test, and that assumption, unlike the rest of the argument, does not survive a quantum adversary (§8.8). The symbolic extraction and proof replay use the prover’s constraint interface; we have run it on the core machine, and the two SHA-256 control chips and the recursion machine’s chips, written against the same interface, are its next inputs rather than a new method. Well-formed shard boundary rows and real-row exhaustiveness remain open premises, and Theorem 8.6 itself is argued on paper (§8.1). The evaluation uses one workload class on one GPU model. The verifying-key allowlist is enumerated from the machine’s declared chip-set clusters, which are curated from its target workloads rather than exhaustive, so a shard outside them is refused rather than admitted (§3.3).

2 Technical preliminaries

This section fixes the field, the arguments and the chip model the rest of the paper builds on. Appendix C lists each result used from prior work and where this paper uses it.

2.1 Field and multilinear extensions

Field. All traces live over the prime field $\mathbb{F} = \text{KoalaBear}$, $p = 2^{31} - 2^{24} + 1$, whose multiplicative group has two-adicity 24, which bounds the largest Reed–Solomon domain (§9.4). Challenges and all sumcheck arithmetic use the extension $\mathbb{F}_{p^4} = \mathbb{F}[X]/(X^4 - 3)$, so a sumcheck round of degree d has soundness error $d/|\mathbb{F}_{p^4}| \approx d \cdot 2^{-124}$. Since $2^{32} > p$, a 32-bit word is not one field element, and every word in the arithmetisation is four field elements holding its little-endian bytes.

Multilinear extensions and sumcheck. For $f : \{0, 1\}^n \rightarrow \mathbb{F}$ the multilinear extension is $\tilde{f}(\mathbf{z}) = \sum_{\mathbf{b}} f(\mathbf{b}) \text{eq}(\mathbf{b}, \mathbf{z})$ with $\text{eq}(\mathbf{b}, \mathbf{z}) = \prod_i (b_i z_i + (1 - b_i)(1 - z_i))$. The sumcheck protocol [LFKN92] reduces a claim $\sum_{\mathbf{b} \in \{0, 1\}^n} g(\mathbf{b}) = \sigma$ about a low-degree g to a claim $g(\mathbf{r}) = \sigma'$ at a random $\mathbf{r} \in \mathbb{F}_{p^4}^n$ in n rounds of degree $\deg g$. We write a claim $\tilde{f}(\mathbf{z}) = v$ as the pair $(\mathbf{z}, v)$.

Zerocheck. To show $g(\mathbf{b}) = 0$ on a subcube, the verifier samples $\mathbf{r}$ and the prover runs a sumcheck on $\sum_{\mathbf{b}} \text{eq}(\mathbf{b}, \mathbf{r}) g(\mathbf{b})$, which is zero for every $\mathbf{r}$ exactly when g vanishes on the subcube, so a random $\mathbf{r}$ errs with probability at most the number of variables over $|\mathbb{F}_{p^4}|$, plus the sumcheck’s error. Section 5.7 proves all constraints of a shard by one such check.

2.2 Arguments and polynomial commitments

Definition 2.1 (Argument of knowledge). Let $\mathcal{R}$ be a relation over public parameters pp , a structure s common to many instances, an instance u and a witness w . An argument of knowledge for $\mathcal{R}$ is a tuple of algorithms ($\text{Setup}, \text{Prove}, \text{Verify}$): $\text{Setup}(\text{pp}, s)$ outputs a proving key and a verifying key (pk, vk) ; $\text{Prove}(\text{pk}, u, w)$ outputs a proof π ; and $\text{Verify}(\text{vk}, u, \pi)$ outputs a bit. It is *complete* if an honest proof for $(\text{pp}, s, u, w) \in \mathcal{R}$ always verifies, *knowledge sound* with error ε if for every efficient prover whose proof verifies with probability above ε an extractor outputs a w with $(\text{pp}, s, u, w) \in \mathcal{R}$, and *succinct* if the proof size and verification time are polylogarithmic in the size of the statement’s computation.

Here s is a program image, u the public values of an execution and w its input and hint streams (§4.4); the construction provides no zero knowledge.

Definition 2.2 (Multilinear polynomial commitment). A polynomial commitment scheme for multilinear polynomials is a tuple $(\text{Commit}, \text{Open}, \text{VerifyOpen})$. $\text{Commit}(\text{pp}, f)$ outputs a commitment C to an m -variate multilinear f and a prover state; Open is an interactive protocol, compiled by Fiat–Shamir, in which the prover convinces the verifier that $f(\mathbf{z}) = v$ for a given point $\mathbf{z} \in \mathbb{F}_{p^4}^m$ and value v , and $\text{VerifyOpen}(C, \mathbf{z}, v, \pi) \in \{0, 1\}$ is its verifier. It is *binding* if no efficient prover opens one commitment to two different polynomials, and *knowledge sound* if an accepting opening yields an f consistent with C and $f(\mathbf{z}) = v$.

A polynomial interactive oracle proof, in which the verifier only evaluates the prover’s polynomials at a few points, compiled with a polynomial commitment gives a succinct argument [BSCS16]; every argument of this paper is of that kind.

Reed–Solomon and constrained Reed–Solomon codes. For a smooth domain $L \subset \mathbb{F}^*$ (a coset of a subgroup of order 2^d) the code $\text{RS}[\mathbb{F}, L, m]$ is the set of functions $L \rightarrow \mathbb{F}$ that agree with a polynomial of degree $< 2^m$; its rate is $\rho = 2^m/|L|$. A codeword is also the evaluation of an m -variate multilinear $\hat{f}$ at the points $(x, x^2, x^4, \dots, x^{2^{m-1}})$, $x \in L$. Following Arnon et al. [ACFY24b], the *constrained* code $\text{CRS}[\mathbb{F}, L, m, \hat{w}, \sigma]$ is the subset of $\text{RS}[\mathbb{F}, L, m]$ whose multilinear satisfies $\sum_{\mathbf{b} \in \{0, 1\}^m} \hat{w}(\hat{f}(\mathbf{b}), \mathbf{b}) = \sigma$; the weight $\hat{w}(Z, \mathbf{X}) = Z \cdot \text{eq}(\mathbf{X}, \mathbf{z})$ expresses the claim $\hat{f}(\mathbf{z}) = \sigma$. A *fold* by a challenge α maps $f : L \rightarrow \mathbb{F}$ to $\text{Fold}(f, \alpha)(x^2) = \frac{f(x) + f(-x)}{2} + \alpha \frac{f(x) - f(-x)}{2x}$, a codeword of $\text{RS}[\mathbb{F}, L^{(2)}, m - 1]$ whenever f is a codeword, equal to the partial evaluation of $\hat{f}$ at $X_1 = \alpha$.

Hashing and transcripts. Merkle commitments and the Fiat–Shamir transcript [FS87] use a hash function H with a digest of d bits. The argument assumes collision resistance for the commitments and models the transcript hash as a random oracle. The hash instantiation determines the commitment and transcript computations and the hash constraints of the recursive verifier. Two hash instantiations illustrate the prover–verifier cost trade-off. The Poseidon2 permutation [GKS23] over $\mathbb{F}$ is algebraic, so a recursive verifier evaluates it in few constraints, and it costs more to compute on the prover. BLAKE3 [OANWO20] is a byte-oriented hash that is faster to compute and costs more constraints inside a recursive verifier. Only the first is implemented and measured: every commitment, transcript, key and timing in this paper uses Poseidon2 at width 16 with $d = 248$. Its S-box has degree three, so the schedule is eight full and twenty partial rounds; the thirteen partial rounds tabulated for a degree-seven S-box give less margin, and the measured configuration of §9.1 carried that count. The same permutation serves the transcript, commitment trees and recursion machine. Grinding b bits means searching for a nonce whose absorption yields b leading zero bits in the squeezed challenge; it multiplies the cost of the corresponding cheating strategy by 2^b .

2.3 Lookup arguments and memory checking

Logarithmic-derivative lookup arguments. A lookup argument proves that two multisets of tuples agree. In the logarithmic-derivative form of Haböck and its GKR realisation [Hab22, PH23], written **LogUp-GKR**, a tuple $\mathbf{f}$ on bus **bus** with multiplicity m contributes $m/(\alpha - \mathbf{bus} - \sum_{j \geq 1} \beta^j f_j)$ for random $\alpha, \beta \in \mathbb{F}_{p^4}$; senders and receivers must sum to the same value, which a layered GKR circuit [GKR08] computes and proves by sumcheck (§5.6).

Uses of the lookup argument. The buses ask this argument for three things, named by the distinctions introduced with Lasso [STW24] and used by Jolt [AST24]. An *indexed* lookup proves $a_i = T[b_i]$ for committed indices b_i ; an *unindexed* lookup proves only that each a_i occurs in T , as a range check does. An indexed lookup is a read from a read-only memory [ST25], so the program bus and the byte bus are indexed lookups into preprocessed tables, and range checks are unindexed. Registers and memory are read–write, where a lookup no longer suffices, since a read could be answered with a value written only later.

Definition 2.3 (Memory checking [ST25, §2.2]). Consider a memory of K cells, an initial content $init \in \mathbb{F}^K$ and T accesses in time order, access j carrying an address a_j , a returned value v_j and a written value v'_j (for a read, $v'_j = v_j$). The accesses are *consistent* if every v_j equals $v'_{j'}$ for the largest $j' < j$ with $a_{j'} = a_j$, or $init[a_j]$ if there is none. A *memory-checking argument* proves consistency of committed accesses; with no writes and public contents it is an indexed lookup.

Memory-checking arguments either reorder the trace by address or do not [ST25, App. A]. ZIREN’s does not: it is Blum et al.’s offline check [BEG⁺94] with Spice’s timestamps [SAGL18], which makes consistency the balance of one multiset relation plus a range fact per access (§5.4, Lemma 5.4). The timestamp is the frame’s (**shard**, **clk** + **pos**), already ordered by the state bus, so one range-checked difference suffices where Spice’s rule needs two. The fingerprinting error grows linearly in the number of accesses and cells [ST25, §2.7], and is included in the quantitative bounds of §8.2.

2.4 Chips, buses and determinism

ZIREN’s constraint system is a set of *chips*, tables whose rows satisfy local polynomial constraints and which communicate only through buses.

Definition 2.4 (Chip). A *chip* is a table over $\mathbb{F}$ of a fixed width w together with a set of polynomial constraints of degree at most three in the w columns of one row, a distinguished boolean column **is_real** gating every constraint, and a set of *interactions*. A row is *satisfying* if it satisfies every constraint.

Definition 2.5 (Interaction and bus). An interaction of a chip is a triple (**bus**, $\mathbf{f}$, m) where **bus** is a name, $\mathbf{f}$ is a tuple of linear expressions in the row’s columns and m is a linear expression in the columns (the multiplicity), marked as a *send* or a *receive*. Multiplicities are non-negative integers bounded well below p : senders use zero, one, or a value range-checked against the byte table, and the preprocessed **Program**, **Byte** and **Range** tables receive with use counts that the shard’s cycle budget bounds by 2^{25} . For a set of tables, the

multiset of a bus is $\{(\mathbf{f}(r))\}$ with multiplicity $\sum m(r)$ over the sends of all rows r of all chips, and likewise for the receives. A bus *balances* if its send and receive multisets are equal.

Two facts connect balance to the fraction identity over $\mathbb{F}_{p^4}$. The multiplicity bound stops the counters wrapping modulo p , so balance is a statement about integer multiplicities; the exact condition is a positive total below p and a negative total above $-p$ [Ope25, Cor. 3.6]. And the identity is checked after compressing tuples with sampled challenges, so an unbalanced bus can still satisfy it, with probability bounded by Schwartz–Zippel; Section 8.2 charges that error.

Definition 2.6 (Shard relation). A *shard* is a set of tables, one per chip, together with public values $\text{pv} = (\text{sh}, \text{entry}, \text{exit}, \text{cur}_{\text{in}}, \text{cur}_{\text{out}}, D, h, \text{code})$: its shard index; its entry and exit states, each a program-counter pair with the clock; the address cursors of the global memory tables before and after it; its cross-shard digest, a point of an elliptic curve over the degree-7 extension $\mathbb{F}_{p^7}$ (§5.5); the digest of committed output; and the exit code. It is *valid* for a verifying key vk if its preprocessed tables are those vk commits to, every real row is satisfying and every bus balances, where the public-values table contributes the entry state as a send and the exit state as a receive on the state bus, and the program bus is received by the preprocessed program table with the recorded multiplicities.

Definition 2.7 (Inputs, outputs and determinism of a chip). For a chip U , the *inputs* $I_U(r)$ of a row r are the values the row consumes from elsewhere: the tuples of its receives, the program tuple it fetches, the previous-value part of its memory accesses, and its *declared free values*, the values its constraints leave to the prover by design, which the statements of §8.7.2 list. The *outputs* $O_U(r)$ are the values it produces for elsewhere: the tuples of its sends other than the fetch, and the new-value part of its memory accesses. Fetch and previous value are sends in the sense of Definition 2.5, but inputs of the row, so the direction of an interaction is not the direction of the data. U is *deterministic* if for all satisfying real rows r, r' ,

$$I_U(r) = I_U(r') \Rightarrow O_U(r) = O_U(r').$$

This is the notion of Picus [PCW⁺23] adapted to the interaction language.

Guest, host and cost metrics. The *guest* is the MIPS32 program and its runtime, whose execution is proved; *guest cycles* count its executed instructions. The *host* runs the executor and the prover, and the verifier trusts nothing it computes. The *trace area* of a shard or block is the sum over chips of rows $\times$ columns of the main trace, in cells; the *(row, interaction) count* is the number of bus interactions over all rows, the leaf count of the lookup circuit; and *throughput* is proved guest cycles per second of wall-clock time. Every committed value lies in the base field and costs one cell whatever its width, so instruction efficiency is improved by removing cells, not narrowing them (§1.3).

3 Architecture

ZIREN proves one statement, the execution of a MIPS32 image (§4.4), and outputs a *compressed proof* of it. A *native verifier* checks that proof by running verifier software, either as native code or WebAssembly, as the consumers of the proofs published for real-time proving of Ethereum blocks do; its costs are proof bytes read and verification time. Algorithm 1 describes the steps that produce the compressed proof and Figure 1 draws it. The subsections below annotate the algorithm and state its core as two protocols, the shard argument (Protocol 3.1) and the block argument (Protocol 3.2), symbolic in the parameters of §3.4; Sections 4–7 construct the shard components and Section 8 states what they establish.

Two verification routes extend ZIREN to applications in EVM-compatible rollups, private computation and Bitcoin rollups (§3.5). Each starts from one shrink node: a pairing-based SNARK wrapper for an *on-chain verifier*, a contract that pays gas, and a binary stage and narrow recursion for a *garbled verifier*, a Boolean circuit that pays for its AND gates and input bits.

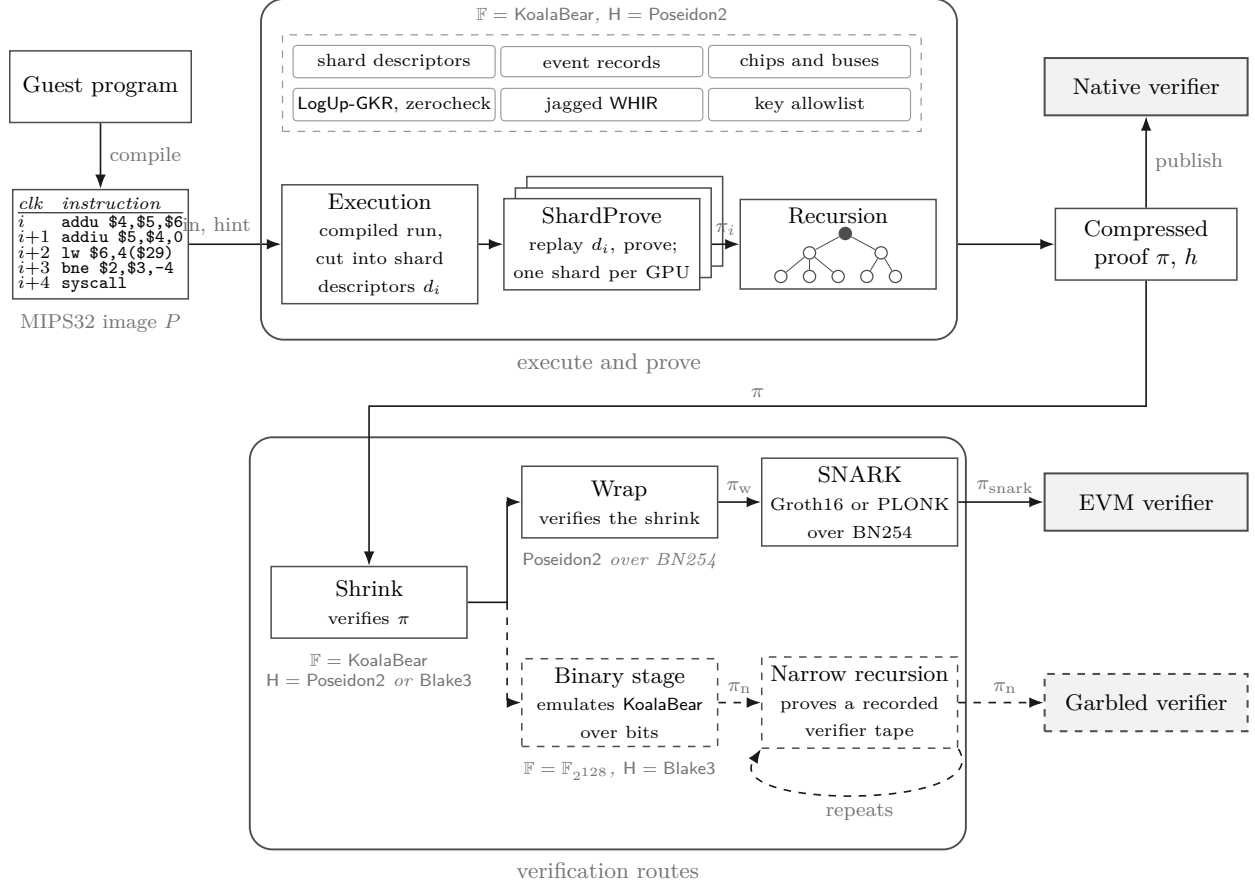


Figure 1: One proving stack produces a compressed proof for a native verifier, and two verification routes re-prove that proof for on-chain and garbled verifiers. Read left to right. ZIREN (top): a guest program compiles to a MIPS32 image; execution cuts one run into shards, which accelerators prove in parallel over KoalaBear with Poseidon2 commitments (the dashed panel lists the objects built along the way), and recursion composes the shard proofs into the compressed proof that a native verifier checks. Verification routes (bottom): a shrink node re-proves the compressed proof under the hash an application reads; the wrap closes it with a pairing-based SNARK for an on-chain verifier, and a binary stage over $\mathbb{F}_{2^{128}}$ and narrow recursion reach a garbled verifier.

Algorithm 1: ZIREN, up to the compressed proof. Lines 3–9 run as a pipeline (§3.3).

Input: a program image P ; input and hint streams $(in, hint)$

Output: the compressed proof π

```

1  $(pk, vk) \leftarrow \text{Setup}(P)$                                 // commit the program and the preprocessed tables
2  $(d_0, \dots, d_{N-1}) \leftarrow \text{Execute}(P, in, hint)$     // one compiled run, cut into shard descriptors
3 foreach  $i \in \{0, \dots, N-1\}$  do                        // in parallel, one shard per accelerator
4    $T_i \leftarrow \text{Replay}(d_i)$                             // re-execute shard  $i$  to its event record
5    $\pi_i \leftarrow \text{ShardProve}(pk, T_i)$                 // over KoalaBear, committed under Poseidon2
6  $\mathcal{S} \leftarrow (\text{Leaf}_{\sigma(\pi_i)}(\pi_i))_{i < N}$       // one leaf key per quantised shard shape
7 while  $|\mathcal{S}| > 1$  do
8    $\mathcal{S} \leftarrow \text{Compose}(\mathcal{S})$                       // one level; a node takes  $\leq 3$  contiguous children
9  $\pi \leftarrow \text{Root}(\mathcal{S})$                                 // arity one, under the compress schedule
// the native verifier runs  $\text{Verify}(vk, h, \pi)$  (Protocol 3.2)

```

3.1 Execution

Line 2 runs once: a just-in-time compiler runs the guest and cuts the run into shards, each described by the values its reads observe and its boundary registers, and a worker replays its shard under the interpreter, which alone emits the event record the prover consumes, so no event record crosses a process boundary (§§4.2 and 4.3). Nothing in the protocol depends on the compiler.

3.2 The shard argument

Line 5 proves one shard. Chips, with one row per executed instruction, and buses arithmetise it (Section 5); their constraints and buses are designed to hold exactly when the shard is a correct piece of the execution, and the premises that remain open are listed in §8.1. A bus argument **BA** (LogUp-GKR, §5.6) and a constraint argument **CA** (a zerocheck, §5.7) reduce every claim to column evaluations at one point; a jagged commitment **JC** reduces those to one evaluation of one dense polynomial (§6), which one **WHIR** opening authenticates (§7). Protocol 3.1 states the reduction. Each step ends with evaluation claims on committed columns, so the steps compose without the verifier reading a column, and the opening protocol consumes the reduced evaluation claim and commitment geometry rather than a separate claim for each chip column.

Protocol 3.1 (Shard argument). **Statement:** a verifying key committing to the program and the preprocessed tables, and the public values of one shard.

Witness: the shard’s execution record.

1. **Commit.** The prover arranges each chip’s events into rows, places the columns of each commitment round into one dense vector and commits to it. Round order is fixed and the per-chip dimensions enter the transcript, so the geometry is public before any challenge is drawn (Section 6).
2. **Buses.** The verifier draws the bus challenges. The prover runs the bus argument, which reduces every send and receive of every chip to column evaluations at one point (Section 5.6).
3. **Constraints.** The verifier draws the constraint challenges. The prover runs the constraint argument on the same point, so each column is opened once for both arguments (Section 5.7).
4. **Jagged reduction.** The column claims of the two previous steps, together with explicit zero claims for the gaps in the committed geometry, are batched and reduced to one evaluation of the dense polynomial (Section 6.2).
5. **Jagged assist.** The verifier does not materialise the indicator table the previous step reduced against. A second sumcheck over the public column boundaries certifies its evaluation, and any point-extension coordinates are sampled here, in an order that is itself part of the protocol (Section 6.3).
6. **Opening.** One batched opening authenticates that evaluation against every commitment of step 1 (Section 7).

Output: a shard proof for the public values **pv** of Definition 2.6.

Verifier checks. No quantity a soundness bound depends on is read from the proof beyond the declared heights, which the configuration caps (a column is at most 2^{22} rows, §6.6), and every algebraic identity the native verifier checks is also checked by the recursive verifier. The verifier (i) absorbs the verifying key, the public values and the per-chip dimensions before drawing any challenge; (ii) checks every sumcheck round of the bus argument, the constraint argument, the jagged sumcheck and the jagged assist, and each one’s closing identity, evaluating the constraint polynomials itself; (iii) checks Equation (5); (iv) runs the **WHIR** verifier with the query counts, folding factors and grinding bits of the schedule; and (v) checks that the public values are well formed. Preprocessed row counts come from the verifying key and the schedule from the verifier’s configuration.

Costs. Let A be a shard’s committed area, n_{int} its number of bus interactions, K its number of columns, ρ the initial encoding rate and L the largest Reed–Solomon encoding-domain size per stripe. For a fixed opening schedule, the prover performs $O(A + n_{\text{int}})$ field operations outside the commitment, $O(A/\rho)$ hashing work and $O((A/\rho) \log L)$ field operations for FFT-based encoding. The verifier checks $O(\log^2 n_{\text{int}})$ sumcheck rounds for the layered lookup circuit and $O(\log A)$ for each of the other three sumchecks, evaluates each chip’s constraints once, spends $O(mK)$ on the jagged assist’s closing identity (Theorem 6.2), and checks the WHIR queries, whose count is fixed by the schedule. The schedule sets the proof size (§9.4), and Section 9 gives the concrete values.

3.3 The block argument

Lines 6–9 compose the shard proofs into one proof of the arithmetised relation $\mathcal{R}_{\text{arith}}$ of §8.1, which §8.7 connects to the relation $\mathcal{R}$ of §4.4. Leaves and compose nodes are recursion programs proved with the same shard argument. A leaf verifies its core proof against the guest key; a compose node checks its children’s recursion keys against an allowlist enumerated from the machine’s declared shapes. Protocol 3.2 states these lines, and the native verifier, with the conditions each node checks.

For N shard proofs, the recursion pipeline is

$$\pi^{\text{compressed}} = \text{Root}(\text{ComposeTree}(\text{Normalize}(\pi_0^{\text{shard}}), \dots, \text{Normalize}(\pi_{N-1}^{\text{shard}}))). \quad (1)$$

In Equation (1), **Normalize** produces one leaf proof per shard, **ComposeTree** aggregates adjacent shard ranges with arity at most three, and **Root** proves a final compose of arity one under the compress schedule. Every recursion node carries the public-value tuple defined in §8.4; Algorithm 2 checks child compatibility and specifies the root’s additional conditions for closing the execution and memory argument.

Protocol 3.2 (Block proof, end to end). **Public parameters:** the field $\mathbb{F}$ and extension $\mathbb{F}_{p^4}$, the Poseidon2 transcript and Merkle trees, the WHIR schedule (Table 5), the bucket rule (§6.6), the compose arity (at most three), and the allowlist root root_K of the admissible leaf and compose keys.

Setup(P). Commit to the program ROM of image P and to the preprocessed **Program**, **Byte** and **Range** tables; output (pk, vk) .

Prove($\text{pk}, P, \text{in}, \text{hint}$).

1. **Execute and shard.** Execute P on (in, hint) and cut the run into shards $T_0, \dots, T_{N-1}$, shard T_i having index $i + 1$ and its record emitted by re-executing it (§4.3), at the first of the clock fence, the shard size, a chip’s height budget and the area budget; each shard’s public values pv_i are those of Definition 2.6 (§4.2).
2. **Prove the shards.** As each record is produced, prove it with the shard argument (Protocol 3.1): commit, bus argument, zerocheck, jagged reduction and assist, and one WHIR opening, giving π_i .
3. **Leaves.** Quantise each shard’s geometry into its class $\sigma_i = (\text{cluster}, \text{first}, b_p, b_m, \text{pad}(b_m))$. The leaf program of that class verifies π_i against vk and outputs the compose tuple of the range (§8.4).
4. **Compose.** As soon as contiguous ranges are complete, merge up to three of them with a compose node, which performs the checks of Algorithm 2.
5. **Root.** Once one node covers $T_0, \dots, T_{N-1}$, a final compose node of arity one that verifies that node’s proof, proved under the compress schedule, performs the root checks of Algorithm 2 and sets **complete**; the first shard’s leaf has already required $\text{entry}_0 = \text{start}(\text{vk})$ and starting cursors of zero (§8.4). Its proof, with the recursion key vk^{rec} that verifies it, is the compressed proof π .

Verify(vk, h, π).

1. Verify π under vk^{rec} with the shard verifier of §3.2 and the compress schedule, which the verifier fixes from the stage it expects and never reads from the proof.
2. Require vk^{rec} to be in the allowlist, the exported root to equal root_K , and the completion flag to be set (§8.4).
3. Require the exported guest key to equal vk and the exported output digest to equal h ; accept.

Admissible keys. The allowlist contains 6 494 leaf and compose keys determined by the declared chip clusters and committed shapes. The clusters are curated from target workloads rather than exhaustive over chip sets, so a shard outside them has no admissible key and is refused. Section 8.4 states the checks that bind the compressed proof to the advertised machine.

Steps 2 to 4 of **Prove** overlap: a shard is proved as soon as its record exists and a compose node as soon as its children are proved, with trace generation and proof-of-work on the accelerators. Only the root uses the schedule tuned for size (§9.4). The start-up interval and the serial composes over the last shards remain (Table 10).

3.4 Parameters

The protocol is symbolic in the following parameters; numeric values are fixed where each component is built.

- The base field $\mathbb{F}$ and the challenge extension $\mathbb{F}_{p^4}$, whose size bounds every term of the form $\deg / |\mathbb{F}_{p^4}|$; the permutation of the transcript and commitment trees, with its round counts (§2), and the trees' arity.
- The committed geometry: the stacking height $H = 2^\ell$, the row cube 2^n and the bucket rule (§6.6), which together fix the set of recursion programs a prover can produce.
- The proximity test: the initial rate ρ , per-round folding factors, query counts, out-of-domain samples, the grinding bits of each query phase and of the bus challenges (Table 5), and the decoding regime.
- Recursion: the compose arity and the height of the tree of admissible keys (§3.3).
- The machine: the shard size, clock fence and per-chip height budgets (§4), and the guest-accelerator set, which fixes the chip set.
- The cross-shard digest: the curve and its extension, the interaction bound N_{int} and the multiplicity bound B (§8.5).

3.5 Applications

ZIREN execution proofs support private computation and rollups on Ethereum and Bitcoin. Figure 1 (bottom) shows SNARK wrapping for private computation and garbled verification for Bitcoin rollups. Both routes preserve the execution statement of π .

Private computation. A Groth16 or PLONK circuit over BN254 verifies a wrapped form of the compressed proof while hashing in its native field. An on-chain verifier checks the resulting SNARK proof with a constant number of pairings [Gro16, GWC19]. The inner proof is a private witness to the Groth16 or PLONK circuit. A zero-knowledge instantiation hides that witness beyond what follows from the public execution statement [Gro16, GWC19]. The current circuit exposes the guest-key hash, the committed-values digest and the allowlist root; privacy is relative to these public inputs and to any application data published separately.

Ethereum rollups. A rollup can run its execution client as a MIPS32 guest and use ZIREN to prove its state transitions. For Ethereum rollups, a verifier contract checks the execution proof as part of settlement. Each rollup supplies its own data-availability and settlement protocol.

Bitcoin rollups. ZIREN execution proofs can be combined with a BitVM bridge and dispute protocol [Bit26]. The execution proof establishes the state transition; the bridge protocol determines how it authorises a withdrawal or resolves a dispute. The garbled-verification route develops a hash-based verifier intended for such protocols.

A garbled verifier is a Boolean circuit evaluated under garbling [Yao86], as in protocols that check a computation on Bitcoin only when it is disputed [Lin23]. Evaluating a garbled verifier on an invalid proof reveals a secret that serves as the fraud proof [Eag25]; later protocols build bridges on this [WAA⁺26], make the verification of pairing-based SNARKs far cheaper [EL26, GKS⁺26], and shrink the on-chain footprint of

Table 2: From the three MIPS32 encodings to the decoded tuple of Definition 4.2. Field widths are in bits, most significant first; *off* and *imm* are 16-bit, *target* 26-bit and *sa* 5-bit. immediates are extended when decoded, as the opcode dictates.

Format	Fields	Instructions	(<i>op_a</i> , <i>op_b</i> , <i>op_c</i>)
R	<i>op</i> 6 <i>rs</i> 5 <i>rt</i> 5 <i>rd</i> 5 <i>sa</i> 5 <i>func</i> 6	register ALU	(<i>rd</i> , <i>rs</i> , <i>rt</i>)
		shift by <i>sa</i>	(<i>rd</i> , <i>rt</i> , <i>sa</i>)
		jr , jalc	(<i>rd</i> , <i>rs</i> , $-$)
I	<i>op</i> 6 <i>rs</i> 5 <i>rt</i> 5 <i>imm</i> 16	immediate ALU, loads, stores	(<i>rt</i> , <i>rs</i> , $\text{ext}(\text{imm})$)
		branches	(<i>rs</i> , <i>rt</i> , $\text{sext}(\text{off}) \ll 2$)
J	<i>op</i> 6 <i>target</i> 26	j , jal	(0 or 31, <i>target</i> $\ll 2$, $-$)

malicious security [KTA⁺26]. They garble the verifier of a pairing-based SNARK or of a designated-verifier variant of one, where this route garbles a hash-based verifier over a binary field. A garbled verifier pays for its AND gates and for its input bits, while XOR gates are free [KS08, ZRE15]. Over a 31-bit prime field an addition is a carry chain and a multiplication a reduction modulo p ; over $\mathbb{F}_{2^{128}}$ an addition is a XOR, so this route moves the arithmetic to a binary field and the hash to one built from word additions, XORs and rotations. The binary stage commits bit columns as packed elements of $\mathbb{F}_{2^{128}}$ and opens them through ring switching [DP23, DP24], emulating the shrink verifier’s KoalaBear arithmetic over bits.

Its verifier is not rewritten as a circuit. A recorder runs it over a value type that appends every multiplication, inversion, bit decomposition and Blake3 hash to a tape, and every comparison as the condition it found, so the tape is fixed by the proof’s shape and key and enforces the control flow of the run that recorded it. A narrow machine native to $\mathbb{F}_{2^{128}}$ proves a run of the tape, with a ledger of read cells and tables for arithmetic, bit rewiring and Blake3, and narrow recursion repeats the step on each narrow proof’s own verifier. The garbled circuit evaluates the last recorded tape on the bits of the last narrow proof; it pays AND gates for field products, Blake3 compressions and the per-claim tensor algebra of ring switching, and takes as input every value the proof opens and every Merkle path. The narrow machine follows from that cost: few tables, narrow ones, none reading its next row, power-of-two widths, and a low-rate opening.

4 MIPS32 and the execution relation

This section gives enough of MIPS32 to follow the paper and states the relation ZIREN proves; the specification is the architecture manual [MIP10]. The guest has no floating-point unit, privileged state or interrupts; it executes the 77 instructions of Appendix B and reaches the outside world only through **SYSCALL**.

Definition 4.1 (Machine state). The machine state is $(\text{pc}, \text{next_pc}, R, M)$. R holds 32 general registers of 32 bits, register 0 reading as zero, together with the HI/LO pair written by multiplication and division; M is a little-endian, byte-addressed memory over a 32-bit address space. pc is the address of the instruction about to execute and next_pc the address of the one that executes after it, which is not in general $\text{pc} + 4$.

Definition 4.2 (Instruction format). Every instruction of the guest is decoded, once and before proving, into a tuple $(\text{opcode}, \text{op_a}, \text{op_b}, \text{op_c}, \text{imm_b}, \text{imm_c})$. op_a is the index of the register the instruction writes (or, for a store or branch, reads); op_b and op_c are each either a register index or an immediate already sign- or zero-extended to 32 bits, as the flags imm_b and imm_c say.

The decoded form abstracts MIPS32’s three encodings as Jolt’s five-tuple abstracts RISC-V’s [AST24]; the correspondence (Table 2) is fixed by the opcode and recomputable by anyone from the image. Figure 2 gives the transition of one instruction.

Features relevant to the arithmetisation. *Branch-delay slots:* a sequential instruction advances the counter pair as $(\text{pc}, \text{next_pc}) \mapsto (\text{next_pc}, \text{next_pc} + 4)$ and a transfer sets next_next_pc , so the successor of an instruction in a delay slot is the branch target and every row carries next_pc as a witnessed column. The pair is architectural state that crosses every shard boundary (§3.3). *Unaligned and narrow accesses:* LB, LBU, LH, LHU, SB, SH address bytes and half-words, and LWL/LWR/SWL/SWR merge part of an unaligned word with a

Step($pc, next_pc, R, M$) $\rightarrow (pc', next_pc', R', M')$ or reject

1. *Fetch.* ($opcode, op_a, op_b, op_c, imm_b, imm_c$) $\leftarrow Decode(M[pc])$; if the word does not decode, reject.
2. *Operands.* $b \leftarrow op_b$ if imm_b else $R[op_b]$; $c \leftarrow op_c$ if imm_c else $R[op_c]$; $a \leftarrow R[op_a]$.
3. *Execute*, by family (§§5.9 and 5.10):
 - ALU, shift, bit-field, move: $R[op_a] \leftarrow f_{opcode}(a, b, c)$; multiplication and division also set HI; division by zero: reject.
 - load: $addr \leftarrow b + c$; $R[op_a] \leftarrow$ the selected bytes of $M[addr]$, extended; store: $M[addr] \leftarrow$ the selected bytes of a merged into it.
 - branch or jump: $t \leftarrow$ the target ($b, next_pc + b$, or $next_pc + c$); $taken \leftarrow$ the condition on a, b (true for a jump); a linking jump sets $R[op_a] \leftarrow next_pc + 4$.
 - **teq** with $a = b$: reject. **syscall**: the call of §5.10.4, which may halt; a call number the machine does not implement: reject (Appendix B).
4. *Register 0.* $R'[0] \leftarrow 0$.
5. *Advance.* $next_next_pc \leftarrow t$ if $taken$ else $next_pc + 4$; return $(next_pc, next_next_pc, R', M')$.

Figure 2: The transition function of the guest machine. Step 5 departs from RISC-V: a transfer sets the *second* component of the pair, so the instruction in its delay slot still executes. A transfer in a delay slot, which the manual leaves unpredictable, gets the meaning the same rule gives it. In ZIREN the instruction frame performs fetch, operand reads, register writes and advance; the memory argument the memory access; and the chip’s own columns only the function f .

register; memory is word-granular, so these select and merge bytes in-row, in chips of their own (§5.9.6). *Bit manipulation*: EXT, INS, WSBH, SEB/SEH, ROTR, CLO/CLZ and the conditional moves, common in hashing and serialisation code, are supported natively.

4.1 Formatting the program

Before proving, the image is decoded (Definition 4.2) into a preprocessed *program table* keyed by address, with immediates already extended. LO and HI are registers 32 and 33, so moves between them and the general registers need no chip, and two further slots hold the program break and heap pointer used only by memory-management syscalls. The program table and the other preprocessed tables are committed at setup into the verifying key (§4.4).

4.2 Clock, shards and the execution record

Execution is divided into *shards*. Within a shard a clock clk advances by five per instruction, and by a bounded number more for a syscall (§5.10.4). An instruction executes at clk , and its accesses occupy five *sub-cycle positions* $clk + pos$, one per role (§5.3), so $(shard, clk + pos)$ identifies and timestamps every access. The executor fences the clock below 2^{25} , capping a shard at about 6.71 M cycles; the range-checked limbs the circuit witnesses for it (§5.3) prove it only below 2^{26} , which still excludes wrap-around. A shard closes at the first limit reached: that fence, the configured shard size of a few million cycles, a chip’s height budget, or a committed-area budget of $4.6 \cdot 10^8$ cells. On the blocks of Table 11, the precompile-heavy Reth guest’s shards close on their area after 2.3 million cycles on average, while many of the Geth guest’s run to the clock fence; Section 9 reports the resulting shard counts.

At a shard boundary the emulator emits, for every word touched in the shard, the value and timestamp of its first and last access, the leaves of the cross-shard memory argument. A shard record holds per-chip event lists, this ledger, the row counts and the public values of Definition 2.6, through which shards and recursion layers are chained; a deferred-proof digest beside them is zero here, since verifying other proofs inside the guest is out of scope.

4.3 The execution relation

Definition 4.3 (Execution). An *execution* of a program image P on streams (in, hint) is a sequence of machine states $\sigma_0, \sigma_1, \dots, \sigma_T$ in which σ_0 is the state the image defines, with the counter pair $(\text{entry}, \text{entry} + 4)$, the registers the image initialises (the stack pointer among them) and zero in the others, and the image’s memory, $\sigma_{i+1} = \text{Step}(\sigma_i)$ for every $i < T$ (Figure 2), no step rejects, and step T is a halting system call with exit code e .

Write $\text{Emulate}(P, \text{in}, \text{hint}) = (\text{out}, e)$ for the output stream and exit status of that execution. It is deterministic: no step reads a clock, a random source or host state other than the streams. Where the manual leaves room, or describes behaviour a single-threaded guest cannot observe, **Emulate** fixes a choice and the arithmetisation proves the same one: **add** and **addi** do not trap on signed overflow and behave as **addu** and **addiu**. **j** and **jal** take the region bits of the program counter as zero, which is exact for text below 2^{28} . **ll** places no reservation and **sc** always succeeds. And a trap, a division by zero, or an unrecognised instruction ends the execution with an error, which is not a halt the relation admits, so such an execution has no proof.

The relation is realised twice: an interpreter, which implements every path and alone emits the event record the prover consumes, and a just-in-time compiler for output-only runs and for the per-shard descriptors workers replay. Byte-identical descriptors on the tested corpus and the conformance evidence of §8.7.1 connect them, without proving equivalence for every program. The compiler matters for rate, since a host feeding several accelerators is bound by the executor (§§3.1 and 9.3).

Incompleteness of the arithmetisation. The arithmetisation refuses some executions the relation admits, which costs completeness, never soundness [GPR21, Note 1]. A shard’s clock is bounded by 2^{25} , so an execution is cut into shards; a cut never falls between a transfer and its delay slot. The only halt proved is exit code zero. A load or store must address memory above the 36 register slots, and every address and program counter must be a canonical field element, below p . And a shard whose chip set lies outside the machine’s declared clusters has no admissible verifying key (§3.3).

4.4 The statement being proved

With P a program image, in an input stream and hint a host-supplied *hint* stream, ZIREN proves the relation

$$\mathcal{R} = \{((\text{vk}, h), (P, \text{in}, \text{hint})) : \text{vk} = \text{Setup}(P) \wedge \exists \text{out} : \text{Emulate}(P, \text{in}, \text{hint}) = (\text{out}, 0) \wedge h = H(\text{out})\},$$

where **Setup** maps an image to its verifying key, a Merkle commitment to the program ROM and the preprocessed tables of Section 5, and H is SHA-256 over the committed output. (vk, h) is public, and vk binds the program image; (in, hint) is the witness and P is fixed by vk , though without blinding it is not confidential. Only exit code zero is admitted (§4.3); **Emulate** accepts a non-zero exit, and one Cannon test program does exactly that and cannot be proved (§8.7). Supported honest executions within the configured resource limits that halt with exit code zero produce accepting proofs; extraction of (in, hint) is conditional on Theorem 8.2 and on the trace-to-execution premises of Theorem 8.6.

5 The arithmetisation: frame, buses and chips

Figure 3 shows instruction chips connected by buses to a program ROM, a memory subsystem and lookup tables. Every relation between chips (Definition 2.4), or between rows of one chip, is an interaction on a named bus, and the lookup argument of Section 5.6 proves that every bus balances (Definition 2.5). No constraint reads the next row: the state bus carries everything one row passes to the next. The guest-accelerator set is a configuration choice: it fixes which precompile chips a shard may contain, and adding one changes the chip set and the verifying keys. This section builds the parts every chip shares and what one row costs (§§5.1–5.8), then gives the chips instruction by instruction, for the base instruction set (§5.9) and for the extended instruction set and system calls (§5.10).

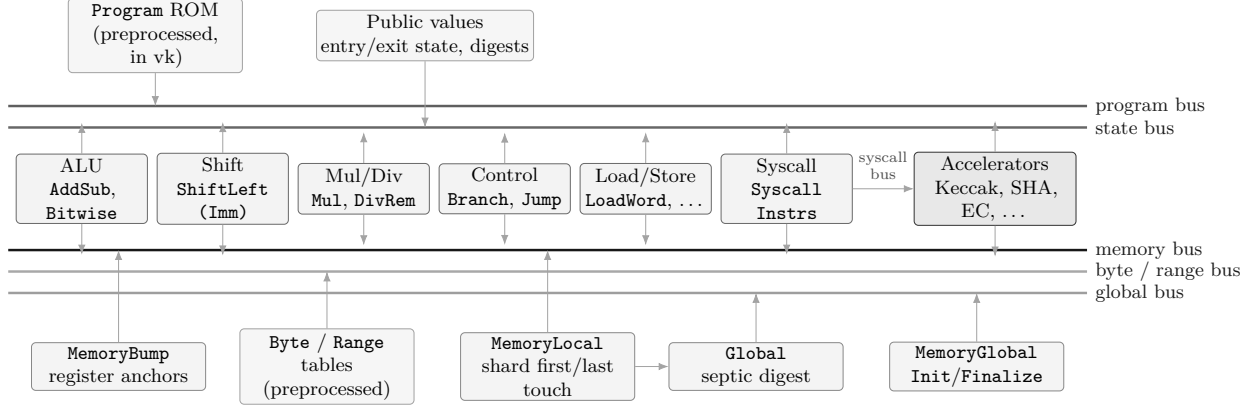


Figure 3: Each executed instruction occupies one opcode-chip row, and every relation between chips crosses a named bus. The program and lookup tables are preprocessed; public values close the state chain and carry the cross-shard digests.

5.1 Chips, words and padding

Appendix A catalogues the chips with their widths. Three families dominate the trace area of an Ethereum block: the memory-instruction chips, the ALU chips, and the **Global** chip that carries the cross-shard memory argument. Immediate forms of the ALU instructions have chips of their own, whose rows have one register read fewer and are about a tenth narrower.

Notation for words. A 32-bit word x is carried as four byte columns (x_0, x_1, x_2, x_3) , least significant first, and denotes the integer $\text{int}(x) = \sum_{i < 4} 2^{8i} x_i$; since $2^{32} > p$ that integer is never formed as a single field element. Its sign bit x_s is the top bit of x_3 , and $x_{<s}$ is the word with that bit cleared. A *byte-checked* word is one whose four columns are each proved to lie in $[0, 256)$ by the byte table. Signed instructions read a word in two’s complement, $\text{int}(x) - 2^{32} x_s$.

Lemma 5.1 (Limbs determine integers). *Let $x_0, \dots, x_{\ell-1}$ be field elements proved to lie in $[0, 2^{b_i})$ and let $y \in [0, B)$, with $2^{b_0+\dots+b_{\ell-1}} \leq p$ and $B \leq p$. If $\sum_i 2^{b_0+\dots+b_{i-1}} x_i = y$ holds in $\mathbb{F}$, it holds over the integers, and the x_i are the unique such limbs of y .*

Proof. Both sides are integers in $[0, p)$ that agree modulo p , so they are equal; uniqueness is that of positional notation. $\square$

This is Cairo’s instruction-decoding theorem [GPR21, Thm. 1] in general form, and every word-level constraint of the machine must meet its hypothesis. A word held as one element cannot, and neither can two 16-bit half-words checked only against their sum in $\mathbb{F}$, which is why the syscall bus carries the half-words themselves (§5.10).

Padding. Row counts are rounded up to a multiple of 32, not to a power of two; the jagged commitment of Section 6 commits the added rows without rounding every chip to a power-of-two height. Padding rows have $\text{is_real} = 0$, and every constraint and interaction multiplicity is gated so that they are vacuous.

5.2 Preprocessed tables

The byte table has 2^{16} rows indexed by a byte pair and one multiplicity column per operation (bitwise operations, shifts, comparisons and 8/16-bit range checks); every carry, borrow, comparison and range check in the machine is an interaction with it. A range table serves the clock limbs, and the program table holds the decoded program image. Preprocessed tables are part of the verifying key (§4.4), which therefore binds the image; OpenVM instead commits the program as a cached trace bound at the root of its recursion, so its verifying key is program-independent [Ope26a, §4.4].

The byte table as a decomposed instruction table. One lookup into the 2^{64} -entry table of a 32-bit bitwise operation becomes four lookups into the 2^{16} -entry byte table, indexed by a byte pair: the decomposition the original Jolt design applies to instruction tables through Lasso [AST24, STW24], taken at $W = 32$ with four 8-bit chunks. Two things separate the settings. Lasso never commits its subtables, because the verifier evaluates their multilinear extensions itself [STW24]; a 2^{16} -row table is small enough to commit once in the verifying key, and LogUp-GKR then charges the table side one multiplicity column per operation over 2^{16} rows per shard, about $6.6 \cdot 10^5$ cells, under 0.2% of a shard’s $4.6 \cdot 10^8$ -cell area budget (§4.2). And Jolt routes every instruction through such tables, addition and multiplication included; this arithmetisation evaluates those by constraints with byte-checked carries and uses the table only for byte-granular facts (§§5.9 and 5.10).

5.3 The instruction frame

A conventional zkVM arithmetisation has a central *CPU chip*: one row per cycle fetches the instruction, performs the register accesses, chains the program counter and forwards a decoded instruction to an opcode chip, which spends a second row on it. Like SP1 and OpenVM for RISC-V [Suc24, Ope26a], ZIREN has no such table. Each instruction chip embeds an *instruction frame* and performs those duties itself, so one executed instruction is one row of one chip. The instruction frame comprises shared columns and bus interactions that constrain instruction identity, execution time, register accesses and machine-state transitions. Opcode-specific columns constrain the relation between the operand values and the result. The frame consists of:

- **Program fetch.** The row sends $(pc, opcode, op_a, op_b, op_c, imm_b, imm_c)$ on the *program* bus, and the preprocessed program table receives it with the multiplicity recorded at trace generation, so the instruction comes from the image committed in the verifying key.
- **Register accesses.** Reads of op_c and op_b (unless immediate) and the read–write of op_a at sub-cycle positions 1, 2 and 3; position 0 is the memory access and 4 a write of HI. They use the register variant of the memory argument (§5.4). An op_a_0 flag pins a write to register 0 to zero, so chips bind their result through a $(1 - op_a_0)$ factor.
- **State bus.** The row receives $(shard, clk, pc, next_pc)$ and sends $(shard, clk + \Delta, next_pc, next_next_pc)$, where Δ is five, one per sub-cycle position, or a larger bounded value for a syscall (§5.10.4). The public-values table sends the shard’s entry state and receives its exit state.
- **Clock.** clk is witnessed as a 16-bit and a 10-bit limb, both range-checked, so that timestamp differences can use the same limb widths in the memory argument.

Lemma 5.2 (State-bus chain). *In a valid shard (Definition 2.6) with entry state s_0 and exit state s_N , the real instruction rows can be ordered $r_1, \dots, r_N$ so that r_i receives s_{i-1} and sends s_i , with $clk(s_i) > clk(s_{i-1})$; every real instruction row is in the chain, and N is the number of real instruction rows.*

Proof. Each real instruction row receives exactly one state tuple and sends exactly one, with a strictly larger clock, and the public-values table sends s_0 and receives s_N . Balance of the state bus means the multiset of sent tuples equals the multiset of received tuples. Consider the directed graph whose vertices are the distinct state tuples and whose edges are the rows (from received tuple to sent tuple). Every vertex other than s_0 and s_N has in-degree equal to out-degree, s_0 has out-degree one more than in-degree and s_N the reverse, so the edge multiset decomposes into one $s_0 \rightarrow s_N$ path and cycles. A cycle is impossible because the clock strictly increases along every edge. Hence the rows form a single path from s_0 to s_N . $\square$

Figure 4 shows three consecutive rows and the buses that bind them. The frame comes in R-type (two register reads), I-type (a word immediate, used by the immediate ALU, memory and branch chips) and shift-amount (a 5-bit immediate) variants, and a universal variant for chips whose opcodes mix operand shapes. It byte-checks the word written to op_a , the single place where a register value is range-checked (§5.4).

Example 5.3 (A delay slot on the state bus). Take a taken `beq` at address `0x100` whose target is `0x200`, the `addiu` in its delay slot at `0x104`, and the instruction at the target. With the shard s and the branch’s clock clk , the three rows exchange these tuples on the state bus:

Row	Chip	receives (shard, clk, pc, next_pc)	sends
beq	Branch	(s, clk, 0x100, 0x104)	(s, clk + 5, 0x104, 0x200)
addiu	AddSubImm	(s, clk + 5, 0x104, 0x200)	(s, clk + 10, 0x200, 0x204)
target	any	(s, clk + 10, 0x200, 0x204)	(s, clk + 15, 0x204, ...)

The branch does not change its own successor, 0x104; it sets the one after it. The `addiu` row is an ordinary row that happens to receive `next_pc = 0x200` and sends (0x200, 0x204) by the constraint every sequential row obeys. Not taken, the branch would have sent (0x104, 0x108) and nothing else would change.

The frame beside other instruction interfaces. SP1 gives each instruction chip a column group that chains (clk, pc) and an R-, I- or J-type register reader that fetches the instruction and reads its operands [Suc24]; the frame’s R-type, I-type and shift-amount variants correspond to those readers. OpenVM factors an instruction executor into an *adapter*, which owns every program, execution and memory interaction, and a *core*, which computes [Ope26a]. The frame fills the same role for MIPS32. It differs from both in carrying the delay slot, and from OpenVM in four respects. The state is (shard, clk, pc, next_pc) rather than (pc, t). Time advances by a fixed five per instruction with a fixed position per access, where OpenVM advances one tick per access; in both, the accesses of step i sit strictly between the states received and sent,

$$\text{clk}_i < \text{clk}_i + 1 < \text{clk}_i + 2 < \text{clk}_i + 3 < \text{clk}_i + 4 < \text{clk}_{i+1} = \text{clk}_i + 5,$$

Registers are shadow-read into the shard and use a six-column access (§5.4), where OpenVM’s are ordinary memory in a separate address space. And OpenVM’s connector chip is here the public-values table.

Both designs state their buses by the same kind of invariant: (shard, clk, pc, next_pc) is sent on the state bus exactly when it is the machine state at that clock, and (shard, clk, a, v) is on the memory bus exactly when address a held v after its access at that time.

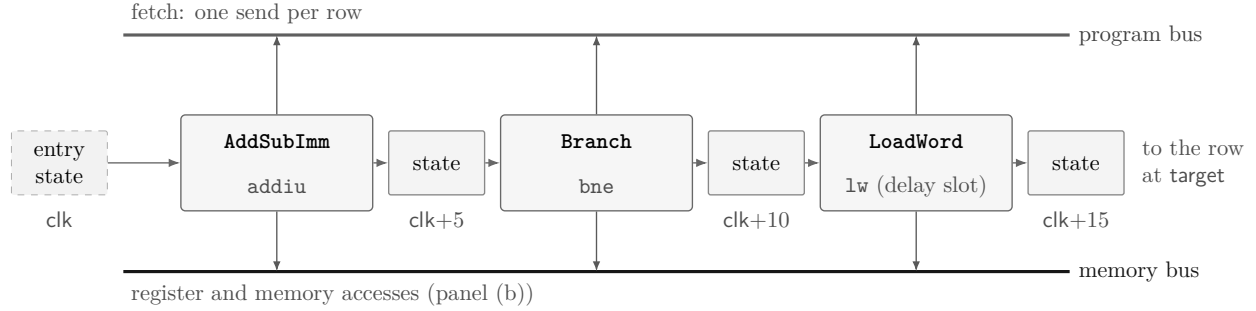
The frame as a reordering of the trace. A per-step circuit that can execute any instruction must either contain every instruction’s logic at every step or be told which one runs. vRAM [ZGK⁺18] took the second route: the prover announces how often each instruction executed, commits the trace regrouped by instruction, and proves by a permutation check that it is a reordering of a time-ordered trace; Jolt names the same device as one way to route its lookups by opcode [AST24], and Ceno’s per-opcode design gives it in multiset form, as a set equality over (program counter, clock) state records [LZZ⁺24, §6.2]. The frame realises this regrouping with a program-counter pair and a shard field in the state. The chips are the groups, their heights are the instruction counts and enter the transcript before any challenge, and the state bus is the permutation check: Lemma 5.2 says the grouped rows re-assemble into one time-ordered chain. The regrouping is of instructions only; the memory argument does not reorder the trace (§2.3).

Regrouping by opcode makes the trace shape depend on the instruction mix. A circuit that carries every instruction at every step has a trace shape that depends only on its length, as RISC Zero’s does [BGt23], or on its length and configured builtins, as Cairo’s does [GPR21, §2.8], which makes Cairo’s verifier program-independent and its recursion simple [GPR21, §2.2.4]. Here the shape depends on which counts are non-zero, so the key of the program that verifies a shard is a function of its chip set, and Section 3.3 enumerates those sets.

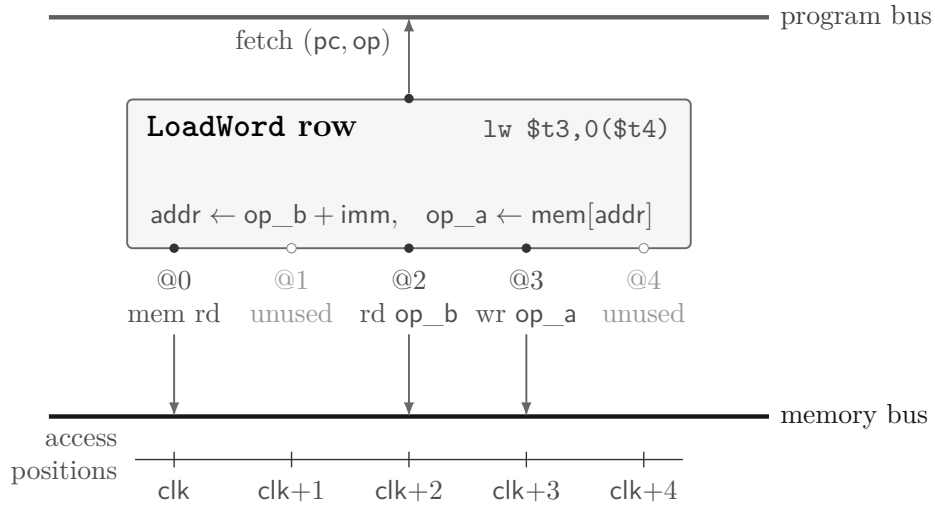
5.4 Registers and memory

Registers and memory share one offline memory-checking argument [BEG⁺94] in the multiset style used by SP1 [Suc24]. The state of address a is the tuple (shard, clk, a, value) of its last access. An access at time (s, c) sends the previous tuple (s', c', a, v') and receives the new tuple (s, c, a, v) on the *memory* bus, with $v = v'$ for a read. Every access enforces $(s', c') < (s, c)$ by witnessing $c - c' - 1$, or $s - s' - 1$ when the shards differ (a one-bit `compare_clk` selector chooses), as a range-checked 16-bit limb and 10-bit high limb, which covers a $\text{clk} + \text{pos}$ timestamp under the 25-bit clock fence. Figure 5 follows one address through two shards.

Lemma 5.4 (Memory argument). *The memory bus with the timestamp constraints is a memory-checking argument in the sense of Definition 2.3, up to the fingerprinting error of the bus argument and provided the bus’s multiplicities satisfy the bound of Definition 2.5. Concretely, in a valid shard with at most one*



(a) The state chain across three consecutive cycles: each row receives its state, sends the next, fetches its instruction from the program ROM and places its accesses on the memory bus.



(b) One row enlarged: the five sub-cycle positions, memory at clk , then op_c , op_b , op_a and HI . Filled pins are accesses the opcode uses; a load has no op_c read, its offset being an immediate, and no HI write.

Figure 4: The instruction frame binds consecutive opcode rows into one state chain (panel a) while assigning register and memory accesses to fixed sub-cycle positions (panel b). Each memory access sends the previous tuple (shard , $\text{clk}+\text{pos}$, addr , value) and receives the new one. In panel a the **LoadWord** row sits in the branch's delay slot, so its successor arrives as the branch target rather than $\text{pc}+4$; the rows are an **AddSubImm**, a **Branch** computing $\text{taken} \leftarrow [a \neq b]$ on registers rs and rt , and the aligned four-lane load of panel b.

initial-state row and one final-state row per address, for every address a the accesses to a form a chain ordered by $(\text{shard}, \text{clk})$ in which each access consumes exactly the tuple produced by the previous access to a , starting from the tuple supplied by the initial-state row of a and ending at the tuple consumed by its final-state row; in particular every read returns the value of the last write.

Proof. Each access sends its previous tuple and receives its new tuple on the memory bus, and its constraints enforce that the new timestamp strictly exceeds the previous one. Balance of the memory bus, restricted to the tuples with address a , gives a graph as in Lemma 5.2 with the initial-state row as source and the final-state row as sink; strictly increasing timestamps exclude cycles, so the accesses form one path, and along the path a read carries its previous value forward unchanged. $\square$

Registers. A dedicated chip, **MemoryBump**, inserts, for every register at its first touch in a shard, a shadow read at $(\text{shard}, 0)$, a device SP1 introduced [Suc24]. Since clk restarts at zero in every shard and real register accesses sit at positions 1..4, the previous access of a register is always in the current shard. The register variant of the access columns therefore drops `prev_shard`, the compare selector and one limb, re-derived linearly from the others: 9 columns become 6 on every register access.

Shard-local memory. Within a shard, the first access to a word consumes a tuple no access in the shard produced, and the last produces one nothing in the shard consumes. A shard-boundary chip, **MemoryLocal**, records, for every word the shard touches, its initial and final $(\text{shard}, \text{clk}, \text{value})$; it receives the initial tuple on the memory bus, sends the final one, and forwards both as *global* interactions (§5.5) to be matched across shards. A shard of an Ethereum block executed by the Reth client touches a few hundred thousand words, mostly a hot working set touched again in the next shard, and these two global rows per touched word are a large share of trace area (§9.2), which is why shard size matters.

Memory initialisation and finalisation. The **MemoryGlobalInit** and **MemoryGlobalFinalize** chips carry the initial contents of the words outside the program image (hint data and zero-initialised words) and the final contents of every word; an initial tuple carries timestamp $(0, 0)$, below every access. Both witness the word as 32 boolean columns, so every value that enters memory is byte-shaped by construction. Outside the image, only words the execution touches appear, as in Ceno [LZZ⁺24, §5.1]. Addresses are 32-bit decomposed and proved strictly increasing along the table, and the address cursors are public values, so the recursion checks that consecutive tables tile the address space. The program image is bound by the verifying key: its digest (§5.5), the sum of the image’s initial tuples, is a value of the key that the first shard’s leaf adds to the cross-shard digest, so every image word is finalised.

Byte checks. Every value that enters a register is byte-checked by the frame at the write, and every value a precompile writes to memory is byte-checked or bit-decomposed by the precompile chip; memory-instruction rows therefore omit the two byte lookups per word that accesses otherwise carry.

Comparison with Twist and Shout. Twist and Shout [ST25] checks memory with one-hot addresses and increments far more cheaply than timestamped checking, but its costs assume commitments in which zeros are free or bits pack, and it notes that committing to zeros is not free under a hash-based commitment [ST25, §2.9]: a one-hot register address alone would be 32 cells against the six of a register access here. Across shards, Twist must commit the memory’s state at each boundary, which it leaves open [ST25, §2.9]; the digest below pays only for the words a shard touches.

5.5 Cross-shard interactions

Interactions whose two sides live in different shards (the initial and final tuples of every shard-touched word, memory initialisation and finalisation, and syscall pairings) cannot be balanced by a per-shard fraction sum. Each shard proof therefore publishes a commitment to its cross-shard state, and the recursion checks that the commitments of the shards compose; the per-shard argument is the same whatever that commitment is. There are four options: two multiset hashes, over an elliptic curve and over a lattice, a global challenge, and Merkle roots over touched memory. They differ in what a shard publishes, what composition checks and

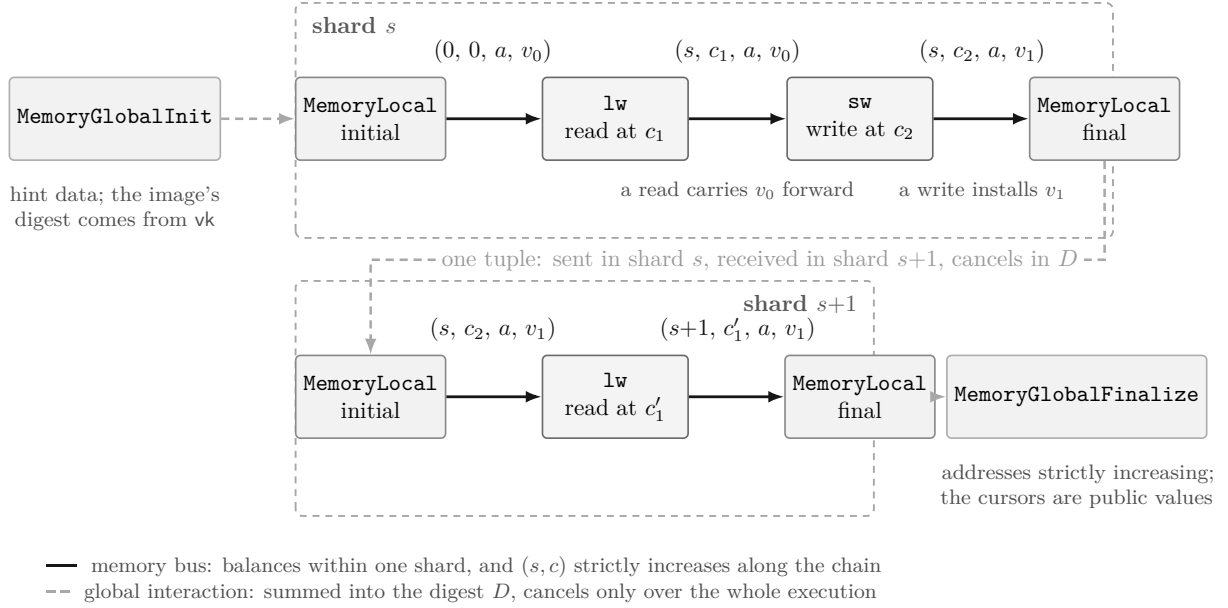


Figure 5: The chain of one address a across a shard boundary. A shard proof balances only tuples whose two sides it contains, so the shard-boundary rows stand in for the accesses it cannot see: the initial row supplies the tuple the shard’s first access consumes, and the final row consumes the tuple its last access produces. These are the same tuple on either side of the boundary, carried with opposite signs into the digest D of §5.5, so the chain closes over the execution without any shard proof depending on another.

what soundness rests on, and they trade committed area against the order in which shards can be proved (Table 3); three are implemented and measured, and the lattice hash is costed. The configuration selects one option according to its security assumptions, trace area and scheduling costs. The measurements of Section 9 and the soundness accounting of Section 8 are stated for the elliptic-curve multiset hash.

Option 1: elliptic-curve multiset hash. Each interaction tuple is mapped into a commutative group, sends add its image and receives subtract it, each shard publishes its running sum, its *cross-shard digest* D , and the root compose node checks that the sums over the execution add to the identity: the multiset hash of Clarke et al. [CDvD⁺03] on the incremental hashing of Bellare and Micciancio [BM97]. On an elliptic curve [MSTA17], which SP1 introduced to zkVM memory checking [RR24], a tuple $\mathbf{m} \in \mathbb{F}^7$ is lifted to a point of $E : y^2 = x^3 + 3\zeta x - 3$ over $\mathbb{F}_{p^7} = \mathbb{F}[\zeta]/(\zeta^7 + 2\zeta - 8)$. One row per interaction witnesses the point, checks the curve equation and a sign band for y that separates sends from receives, and adds the point to a running sum on an accumulation bus; the sum starts at a public offset point D_{off} , so two digests combine as $D + D' - D_{\text{off}}$ and the empty digest is D_{off} . The *lift* applies no hash: the message is the x -coordinate, up to a prover-chosen byte *offset* that makes the curve’s right-hand side a square, which keeps the chip at a few dozen columns; §8.5 gives what soundness then rests on.

Option 2: lattice multiset hash. The group can instead be a lattice, as in LtHash [LKMW19], the instantiation of Bellare and Micciancio’s incremental hash over a product of small rings, which ZisK uses for its challenge derivation [Mas25]. A tuple $\mathbf{m}$ is hashed to $h(\mathbf{m}) \in \mathbb{F}^n$, sends add $h(\mathbf{m})$ and receives subtract it, each shard publishes its running sum, and composition checks that the sums over the execution add to zero, exactly as for the curve. A collision is a short integer relation among hash outputs (§8.8). The cost is h : it must be constrained in the chip for every interaction, and it must behave as a random function, since a linear h makes any two multisets with equal message sums collide, which is what our prototype with a fixed public linear map did. A map strong enough for the lattice assumption needs $n = 800$ or more over the 31-bit field and at least $19.6\times$ today’s committed area by our count, so this option is costed, not built.

Table 3: The four options for cross-shard interactions. Times are relative to the elliptic-curve multiset hash on the same GPUs (warm, compressed proofs, $H = \text{Poseidon2}$); the global challenge was measured on one block, Merkle roots over six and twelve blocks with the two builds alternated (ABBA), the block count in parentheses.

Option	a shard publishes	composition checks	soundness rests on	1 GPU	4 GPUs	other costs
1. Elliptic-curve multiset hash	a running sum of points	the sums add to the identity	Assumption 8.4	–	–	
2. Lattice multiset hash	a running sum of vectors	the sums add to zero	short integer relations	$\geq 19.6\times$	area	not built
3. Global challenge	a LogUp-GKR fraction sum	the sums add to zero	the hash chain, LogUp-GKR	+29 %	+42 %	a commit pass before proving
4. Merkle roots	memory roots before and after	the roots chain from the image root	collision resistance of H	–16.6 % (6)	+3.4 % (12)	–47 % shards; +20 GB per worker

Option 3: global challenge. Every shard commits its main trace first; one **LogUp-GKR** challenge is then drawn from a hash chain over all the commitments, each shard proves its cross-shard interactions as **LogUp-GKR** fractions under that challenge and publishes their sum, and the recursion checks that the sums over the execution add to zero. Soundness rests on the hash chain and on the **LogUp-GKR** argument the shards already use, and the option adds no hashing to the machine. The price is order: every shard must be committed before any can be proved, which on the GPU pipeline means a commit pass ahead of the proving pass.

Option 4: Merkle roots over touched memory. Memory is a Merkle tree under H , a height-27 tree of four-word leaves (guest memory lies below 2^{31}) and a height-4 tree of registers joined under one root, and each shard publishes the root before it and the root after it. A boundary unit recomputes, for every leaf the shard touches, its path in the initial tree and in the final tree: it sends the leaf’s initial words on the memory bus and receives its final words, so the shard’s accesses balance inside the shard, and the untouched siblings along the paths enter both trees through one lookup. A per-word initialisation bit replaces the initialisation table, and precompile and syscall pairings balance in the shard that issues them, the executor closing a shard before a precompile that would give its recursion leaf an area no recursion shape admits. Composition checks that each shard’s outgoing root is the next shard’s incoming root and that the first is the image root the verifying key fixes. Soundness rests only on the collision resistance of H ; the accumulation rows and cross-shard memory units leave the machine, the Merkle rows take their place, and every measured block proves through compression with the root chain closed.

5.6 The bus argument

One **LogUp-GKR** instance, the bus argument, proves all interactions of a shard on the program, state, memory, byte, range, syscall, global and accumulation buses. After the trace is committed, the verifier samples $\alpha, \beta \in \mathbb{F}_{p^4}$, and the fraction of each interaction (§2.3) is a leaf of a layered GKR circuit whose transitions sumcheck proves from the root down; the final layer leaves evaluation claims on the participating columns. A **Reth** shard has on the order of 10^8 leaves, one per (row, interaction) pair, so interactions per row cost separately from columns (§9.3). Its soundness is $\deg / |\mathbb{F}_{p^4}|$ per sumcheck round plus the fingerprint’s collision probability, which grows with the number of leaves (§8.2); the schedule of Table 5 grinds the **LogUp-GKR** challenge by 22 bits.

5.7 The constraint argument

The constraint system is uniform: a chip imposes the same constraints on every row, so nothing about it is committed and the verifier evaluates the constraint polynomials itself at the closing point. All chips’ constraints are proved by *one* zerocheck. The verifier samples η to combine chips and μ to combine a chip’s constraints, so that $C_c(\mathbf{X}) = \sum_k \mu^k c_{c,k}(\text{row}(\mathbf{X}))$ must vanish on chip c ’s real rows, and the prover proves $\sum_b \text{eq}(\mathbf{b}, \mathbf{r}) \sum_c \eta^c C_c(\mathbf{b}) = 0$ for a random $\mathbf{r}$ by sumcheck over the tallest chip’s variables, with shorter

chips handled by an analytic “virtual $\geq$ ” factor rather than padding. Constraint degree three gives round polynomials of degree four. The GPU’s constraint kernels are compiled per chip from the description the verifier uses, except for the widest accelerator chips, which are interpreted (Table 9), and host verification catches any mismatch. The zerocheck is seeded from the **LogUp-GKR** openings, so the two share one opening of each column; what remains is about $3.6 \cdot 10^4$ (chip, column, point, value) claims per shard, which the commitment layer reduces to one.

5.8 The cost of a row

Counting a step’s committed elements shows that an instruction’s cost lies in the state its row reads and writes, not in the function it computes. The R-type frame is 30 columns, 22 of them for the three register accesses, the I-type frame 27 and the shift-amount frame 24 (Table 12). With the program-counter pair each chip owns, an instruction pays 32 columns before it computes anything; **Bitwise** then adds five and **AddSub** four. The six-column register access saves three columns on every access of every instruction. §§5.9 and 5.10 report each chip against this floor, and Section 9.2 turns the widths into a cost per executed instruction.

5.9 Chips for the base instruction set

For each instruction family we give the chip, its frame, how the row fixes the result, and the lookups the chip adds to its frame’s. Operands follow Definition 4.2: a is the word written to (or, for stores and branches, read from) register **op_a**, and b and c are the words of **op_b** and **op_c**, from registers or decoded immediates, in the notation of §5.1. Every result is gated by $(1 - \text{op_a_0})$, which the tables omit.

Frames. The frame’s interactions are paid on every row (§5.3): 20 for the R-type frame and 16 for the I-type and shift-amount frames, itemised in Appendix A. The *universal* frame is the R-type frame with its reads gated by the immediate flags. The four frames play the part of OpenVM’s six RV32IM adapters [Ope26a]; loads, stores and branches share the I-type frame. Widths are from Table 13, and lookups are counted beyond the frame’s.

5.9.1 Logical instructions

Instructions	Chip (width), frame	Result	Added lookups
and, or, xor, nor	Bitwise (37), R	$a_i = \text{OP}(b_i, c_i), i < 4$	4 byte-table
andi, ori, xori	BitwiseImm (33), I	$a_i = \text{OP}(b_i, c_i), i < 4$	4 byte-table

The bitwise result is fixed by lookups rather than by a polynomial result constraint: each byte pair is one lookup into the byte table, whose operation index is a selector-weighted sum of the opcode flags (§5.2). The immediates of **andi**, **ori** and **xori** are zero-extended at decode, and **nor** has no immediate form.

5.9.2 Arithmetic instructions

Instructions	Chip (width), frame	Result	Added lookups
add, addu	AddSub (36), R	$a = b + c \bmod 2^{32}$	none
sub, subu	AddSub (36), R	$a + c = b \bmod 2^{32}$	none
addi, addiu	AddSubImm (33), I	$a = b + \text{sxt}(imm) \bmod 2^{32}$	none
lui	AddSubImm (33), I	$a = 0 + (imm \ll 16)$	none
mfhi, mflo, mthi, mtlo	AddSubImm (33), I	$a = b + 0$	none

Addition is checked byte by byte with carries that are not columns: the carry out of byte i is the linear expression $\theta_i = (b_i + c_i - a_i + \theta_{i-1})/256$, $\theta_{-1} = 0$, and the chip asserts $\theta_i(\theta_i - 1) = 0$; subtraction asserts the same of $a + c = b$. The frame byte-checks the result, so the family adds four columns (two selectors, two gates) and no lookup. Jolt instead forms $b + c$ as one element of a 256-bit field and looks up its low bits [AST24], which $2^{32} > p$ rules out here. **lui** adds an immediate shifted at decode to register 0, and the moves to and from HI and LO add zero (§4.1). **add** and **addi** decode as **addu** and **addiu** (§4.3).

5.9.3 Comparisons

Instructions	Chip (width), frame	Result	Added lookups
sltu, slt	Lt (50), R	$a = x_s(1 - y_s) + \text{eq}(x_s, y_s) \text{LTU}(x_{<s}, y_{<s})$	2 AND, 1 LTU
sltiu, slti	LtImm (47), I	as above, $y = \text{sxt}(imm)$	2 AND, 1 LTU

With $x = b$ and $y = c$ the result is Jolt’s signed less-than, while for the unsigned forms the sign bits are not split off and LTU compares the full words [AST24]. Two AND lookups clear the top bit of each high byte, which gives $x_{<s}$, $y_{<s}$ and, by difference, the sign bits. The unsigned part is decomposed differently: Jolt sums over chunks, weighting each chunk’s less-than by the equality of every chunk above it, a less-than and an equality lookup per chunk; Lt witnesses a one-hot flag for the most significant differing byte, asserts by constraints that the bytes above it agree and, through a witnessed inverse, that the flagged bytes differ, and spends one LTU lookup on the flagged pair. The upper three result bytes are zero.

5.9.4 Shifts and rotates

Instructions	Chip (width), frame	Result	Added lookups
sll	ShiftLeftImm (45), shift	$a = b \cdot 2^k \bmod 2^{32}$, $k = sa$	4 range
sllv	ShiftLeft (62), R	$a = b \cdot 2^k \bmod 2^{32}$, $k = c \bmod 32$	4 range
srl, sra, rotr	ShiftRightImm (83), shift	low word of $(\hat{b} \gg k)$, $k = sa$	1 MSB, 8 ShrCarry, 16 range
srlv, srav, rotrv	ShiftRight (89), R	low word of $(\hat{b} \gg k)$, $k = c \bmod 32$	1 MSB, 8 ShrCarry, 16 range

A shift by $k = 8q + r$ is a shift by r bits followed by a relabelling of whole bytes by q . The shift amount is decomposed into bits, of which the low five are used (the manual’s reduction modulo 32), and one-hot selectors for $q \in [0, 4)$ and $r \in [0, 8)$ are tied to them. A left shift multiplies by 2^r with witnessed per-byte carries, $\rho_i = 2^r b_i + \theta_{i-1} - 256\theta_i$, then moves byte i to byte $i + q$; the shift-amount form computes 2^r as $(1 + s_0)(1 + 3s_1)(1 + 15s_2)$ with no one-hot. A right shift first extends b to eight bytes $\hat{b}$, whose upper four are zero for **srl**, 0xFF times the sign bit for **sra**, and a copy of b for **rotr**, so one right shift leaves the rotation in the low half. The bit part uses eight ShrCarry lookups, each mapping a byte and a 3-bit shift to the shifted byte and the bits shifted out: Jolt’s shift decomposition [AST24], a subtable indexed by a chunk and the shift amount, with the byte part done by relabelling.

5.9.5 Branches and jumps

Instructions	Chip (width), frame	next_next_pc	Added lookups
beq, bne	Branch (56), I	$\text{next_pc} + \text{off}$ if taken, else $\text{next_pc} + 4$	≤ 9 of 13
bgez, bgtz, blez, bltz	Branch (56), I	as above, comparing with register 0	≤ 9 of 13
j, jal	Jump (57), universal	$\text{target} \ll 2$	3 LTU
jr, jalr	Jump (57), universal	b	3 LTU
bal	Jump (57), universal	$\text{next_pc} + \text{off}$	3 LTU, 6 range

A transfer sets only **next_next_pc**, the second component of the pair it sends; its delay-slot instruction is the next row of the chain (Example 5.3). **Branch** compares a (register **rs**) with b (register **rt**, or register 0 for the compare-with-zero forms). Equality is two is-zero tests, by witnessed inverses, on the differences of the 16-bit halves; the sign of a is one MSB lookup, gated to the compare-with-zero forms; and $a > 0$ is $(1 - a_s)(1 - [a = 0])$. The offset is shifted and sign-extended at decode, so the taken target is one in-row addition with a byte-checked result, and one LTU lookup per counter word proves it canonical. **Jump** writes the link $\text{next_pc} + 4$, which is $\text{pc} + 8$ through the chain, and takes its target from the immediate, from register b , or from an addition for **bal**. As in the executor, **j** and **jal** take the region bits as zero (§4.3); the guest toolchain links text below 2^{28} .

5.9.6 Loads and stores

Instructions	Chip (width), frame	Result	Added interactions
lw, ll	LoadWord (48), I	$a = M[addr]$	9
sw, sc	StoreWord (52), I	$M[addr] \leftarrow a$; sc also sets $a = 1$	9
lb, lbu, lh, lhu	LoadNarrow (59), I	selected byte or half, sign- or zero-extended	10
sb, sh	StoreNarrow (55), I	byte or half of a merged into $M[addr]$	9
lwl, lwr, swl, swr	MemoryUnaligned (59), I	byte-wise merge of $M[addr]$ and a	9

The five memory chips share one block of columns and nine interactions. The effective address $b + \text{sext}(off)$ is one in-row addition with range-checked result bytes; one LTU proves it canonical and a second, with an is-zero test on its upper bytes, that it lies above the 36 register addresses; and one AND lookup extracts its two low bits, from which the aligned address is linear. The access is one word-granular access at sub-cycle position 0 (two memory-bus interactions and two timestamp lookups), and the word is not byte-checked again (§5.4). The word forms assert that the low bits are zero; the narrow and unaligned forms turn them into one-hot offset flags and select or merge bytes polynomially, and a narrow signed load adds one MSB lookup and fills with 0xFF times the bit. **ll** and **sc** follow the choices of §4.3.

OpenVM likewise accesses aligned four-cell blocks and selects bytes within the row [Ope26a, §3.2.4], where the original Jolt design spends one memory-checking operation per byte [AST24]; the price is the offset flags and merge columns that make the narrow and unaligned chips the widest of the family.

5.10 Chips for the extended instruction set and system calls

Each of these instructions is still one row of one chip, which carries as columns the work a lookup-based design spreads over a sequence of virtual instructions [AST24], and **HI** is written by a second access at sub-cycle position 4. That access raises eight interactions, counted as “an access” in the tables below: a send and a receive on the memory bus, two timestamp lookups, and two byte lookups each for its previous and its new word.

5.10.1 Multiplication

Instructions	Chip (width), frame	Result	Added lookups
mul	Mul (74), R	$a = b \cdot c \bmod 2^{32}$	14, and an access
mult, multu	Mul (74), R	$(HI, LO) = b \cdot c$, signed or unsigned	14, and an access
madd, maddu, msub, msubu	MiscInstrs (432), universal	$(HI, LO) \pm b \cdot c$	26, and an access

A 64-bit product exceeds the field, so it is formed as a convolution of bytes. Both operands are extended to eight bytes, with 0xFF times the sign bit for the signed forms **mult**, **madd** and **msub** (two MSB lookups) and zeros otherwise; the sign extension does in the row what Jolt’s virtual sequence for signed high multiplication does [AST24]. The uncarried convolution $m_k = \sum_{i+j=k} b_i c_j$, $k < 8$, is tied to witnessed product bytes π_k and carries θ_k by $m_k + \theta_{k-1} = \pi_k + 256 \theta_k$, with carries range-checked to 16 bits and product bytes to 8. The low word $\pi_{0..3}$ goes to **op_a** (LO for **mult** and **multu**, rd for **mul**) and the high word to **HI**. The multiply-accumulate forms add the same product to the 64-bit (HI, LO) with a byte-carry addition, and prove a subtraction as the addition that undoes it.

5.10.2 Division and remainder

Instructions	Chip (width), frame	Result	Added lookups
div, divu	DivRem (163), R	$LO = q$, $HI = r$ with $b = qc + r$, $ r < c $	36, and an access

As in Jolt [AST24], the prover supplies the quotient q and remainder r as advice and the row checks them; Jolt’s check is eight virtual instructions; here it is 131 columns beyond the frame. qc is formed by the **Mul** convolution, signed for **div**, and r , sign-extended to eight bytes, is added with byte carries; the low word of the sum must equal b and the high word its sign extension. The remainder takes the dividend’s sign, and $|r| < |c|$ compares absolute values, each proved by an addition to zero when negative, with the one-flag

comparison of Lt. The overflowing case $b = -2^{31}$, $c = -1$ is detected by two word-equality tests and exempts the high word. Division by zero is an execution error (§4.3), and the row asserts $c \neq 0$.

5.10.3 Bit manipulation and conditional moves

Instructions	Chip (width), frame	Result	Added lookups
clz, clo	CloClz (93), I	leading zeros of b , or of $\neg b$	28
movz, movn	MovCond (49), universal	$a = b$ if $[c = 0]$ (resp. $\neq$), else a unchanged	none
wsbh	MovCond (49), universal	$a = (b_1, b_0, b_3, b_2)$	none
seb, seh	MiscInstrs (432), universal	b_0 or (b_0, b_1) , sign-extended	1 MSB
ext, ins	MiscInstrs (432), universal	bit-field extract, insert	up to 113
teq	MiscInstrs (432), universal	no trap: $a \neq b$	none

clo is **clz** of the complemented word. For **clz**, the zero word yields 32; otherwise the row proves, with the right-shift construction of §5.9.4 inlined, that shifting b right by $31 - a$ leaves exactly 1, and an LTU lookup bounds a by 32. The conditional moves test $c = 0$ by two witnessed inverses on its 16-bit halves and select between b and the previous value of **op_a**, which the frame’s read–write access carries; **wsbh** is a byte permutation. **ext** is a left shift followed by a right shift, and **ins** rotates the destination, shifts, adds the shifted source and rotates back. These rare opcodes share **MiscInstrs**, whose opcode-specific witness fields reuse column positions through a union, so the row is as wide as **ins** and every lookup is gated by an opcode selector. **teq** is provable only when it does not trap.

5.10.4 System calls

Instructions	Chip (width), frame	Effect	Added interactions
syscall	SyscallInstrs (79), R	sends id , b , c as 16-bit halves	2 LTU, 2 syscall-bus
Linux ABI calls	SysLinux (135), none	brk , mmap , read , write , ...	up to 50, gated
accelerators	per precompile, none	a hash or curve operation in memory	per chip

SYSCALL is the one place a row hands work to another table. The call number in **\$v0** is read as bytes: the low two are the call identifier, the third says whether the call is sent to a table, and the fourth is a count of extra clock ticks, by which the frame advances the clock beyond five so that a precompile’s memory accesses fit before the successor. The two arguments travel on the syscall bus as exact 16-bit halves with a Linux flag. A halting call sets the next program counter to zero and exposes its first argument as the public exit code; a commit call compares its argument with the public output digest. **SysLinux** receives the Linux-ABI calls and performs their memory accesses at the calling cycle, and the precompile chips of Appendix A receive accelerator calls, routed across shards through the digest or, with Merkle roots, proved in the shard that issues them (§5.5). The Jolt design [AST24] has no precompile tables; cryptographic operations there are compiled into sequences of ordinary and virtual instructions.

6 The jagged commitment

After the bus argument and the zerocheck of a shard (§§5.6 and 5.7), the verifier holds evaluation claims on individual chip columns (§5.7), with heights from 32 rows to millions. Committing each column or table separately costs one Merkle root and one query phase per commitment, and puts every table’s height into the verifier, so that each height profile needs a verifier circuit of its own [HJR⁺25, p. 4]. The jagged polynomial commitment of Hemo et al. [HJR⁺25] commits the non-zero part of all columns as one dense multilinear and reduces every column claim to one evaluation of it. ZIREN uses its basic construction [HJR⁺25, Thm. 1.4] with its jagged assist [HJR⁺25, Thm. 1.5]; it uses neither the “fancy” variant for columns of equal height (§6 there) nor committed heights proved by GKR (§1.1.2 there), because the machine’s columns are too heterogeneous for grouping into power-of-two tables to pay.

Sections 6.1–6.3 restate the construction in our notation. Sections 6.4–6.6 describe what ZIREN adds: several ordered commitments opened as one, the stacked layout of the opening, and a quantised geometry that makes the verifier a function of a small class rather than of a shard’s row counts. A chip can therefore be

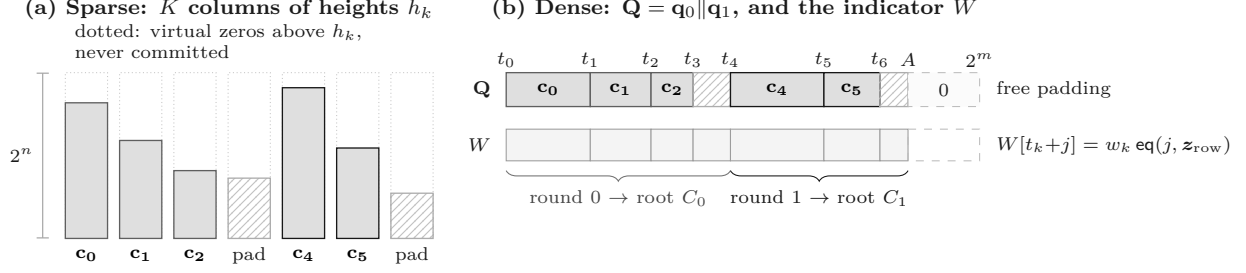


Figure 6: The jagged commitment. (a) The sparse view: K columns of different heights inside a 2^n -row box; the dotted cells above each column are virtual zeros, never committed; hatched columns, c_3 and c_6 , are the synthetic padding of §6.4 and count among the K columns. (b) The dense representation $\mathbf{Q}$: the columns laid end to end, located by the public prefix sums t_k , split into two ordered input rounds with separate Merkle roots, and extended to 2^m by implicit zeros (the dashed tail, which has no column and so no weight in W); the indicator W is aligned with it cell for cell. Protocol 6.3 reduces the K column claims to one opening of $\mathbf{Q}$.

Table 4: Notation of this section against that of Hemo et al. [HJR⁺25].

Here	[HJR ⁺ 25]	Meaning
$K, \kappa = \lceil \log_2 K \rceil$	$2^k, k$	number of columns
$n, m, M = 2^m$	n, m, M	log height of the row cube; log and size of the dense domain
h_k, t_k (exclusive)	h_y, t_y (inclusive)	column heights and prefix sums
$\mathbf{Q}, \tilde{\mathbf{Q}}$	$q, \hat{q}$	dense representation
W	$f_t(z_r, z_c, \cdot)$	the indicator of Equation (2)
$\mathbf{z}_{\text{row}}, \mathbf{z}_c, \mathbf{z}^*$	z_r, z_c, i	row point, column point, dense point
BP, u_k	$\hat{g}, (t_{y-1}, t_y)$	branching program and its boundary input

widened, added or removed without changing the layer above it; the chip revisions of §9.2 left this protocol unchanged. Figure 6 shows the construction end to end.

6.1 Jagged functions and the dense representation

Let there be K columns $\mathbf{c}_0, \dots, \mathbf{c}_{K-1}$ with heights $h_0, \dots, h_{K-1} \leq 2^n$, viewed as one function $p : \{0, 1\}^n \times \{0, 1\}^\kappa \rightarrow \mathbb{F}$, $\kappa = \lceil \log_2 K \rceil$, with $p(x, k) = \mathbf{c}_k[x]$ for $x < h_k$ and $p(x, k) = 0$ otherwise. Such a function is *jagged* [HJR⁺25, Def. 1.1]. Its non-zero part is the *dense representation*

$$\mathbf{Q} = \mathbf{c}_0 \| \mathbf{c}_1 \| \dots \| \mathbf{c}_{K-1}, \quad t_0 = 0, \quad t_{k+1} = t_k + h_k, \quad A = t_K,$$

padded with zeros to $M = 2^m$, $m = \lceil \log_2 A \rceil$, and written $\tilde{\mathbf{Q}}$ as a multilinear in m variables. Column k occupies $[t_k, t_{k+1})$ in $\mathbf{Q}$, and the prefix sums are public: they enter the transcript before any challenge. $\mathbf{Q}$ and the prefix sums determine p [HJR⁺25, Fact 1.3], so committing to $\tilde{\mathbf{Q}}$ commits to every column at the cost of the real area A rather than $K \cdot 2^n$.

Table 4 maps our notation to the paper's. Two differences are inessential: our prefix sums are exclusive (t_k starts column k), and a boundary takes $m + 1$ bits because $t_K = A$ may equal 2^m . Heights may reach 2^n rather than $2^n - 1$, which is harmless because only boundaries are encoded.

6.2 The jagged reduction

Let $\mathbf{z}_{\text{row}} \in \mathbb{F}_{p^4}^n$ be the common row point left by the zerocheck, and y_k the claimed evaluation of column k there, high zeros included:

$$y_k = \sum_{j=0}^{h_k-1} Q[t_k + j] \text{eq}(j, \mathbf{z}_{\text{row}}) = \tilde{p}(\mathbf{z}_{\text{row}}, k).$$

The verifier samples a column point $\mathbf{z}_c \in \mathbb{F}_{p^4}^\kappa$, which turns the K claims into the single claim

$$T = \sum_{k < K} \text{eq}(k, \mathbf{z}_c) y_k = \tilde{p}(\mathbf{z}_{\text{row}}, \mathbf{z}_c)$$

at an error of $\kappa/|\mathbb{F}_{p^4}|$ by Schwartz–Zippel; this is the input of the basic jagged construction. With $w_k = \text{eq}(k, \mathbf{z}_c)$, the length- M table

$$W[t_k + j] = w_k \text{eq}(j, \mathbf{z}_{\text{row}}) \quad (0 \leq j < h_k), \quad W[i] = 0 \quad \text{elsewhere},$$

is the paper’s indicator $f_t(\mathbf{z}_{\text{row}}, \mathbf{z}_c, \cdot)$ on the Boolean cube [HJR⁺25, Eqs. (3)–(4)], and

$$T = \sum_k w_k \sum_{j < h_k} Q[t_k + j] \text{eq}(j, \mathbf{z}_{\text{row}}) = \sum_{i \in \{0,1\}^m} \tilde{Q}(i) \tilde{W}(i). \quad (2)$$

$\tilde{W}$ agrees with the product of equality polynomials only on the cube [HJR⁺25, Rem. 3.1], so the verifier cannot evaluate it by formula; §6.3 supplies it.

Theorem 6.1 (Basic jagged [HJR⁺25, Thm. 1.4]). *There is an m -round protocol in which the prover holds $\mathbf{Q}$ and the verifier holds only the prefix sums, $(\mathbf{z}_{\text{row}}, \mathbf{z}_c)$ and T , and at whose end the verifier either rejects or outputs a point $\mathbf{z}^* \in \mathbb{F}_{p^4}^m$ and a value v_Q . Completeness: if $T = \tilde{p}(\mathbf{z}_{\text{row}}, \mathbf{z}_c)$ then $v_Q = \tilde{Q}(\mathbf{z}^*)$. Soundness: if $T \neq \tilde{p}(\mathbf{z}_{\text{row}}, \mathbf{z}_c)$, then except with probability $2m/|\mathbb{F}_{p^4}|$ the verifier rejects or $v_Q \neq \tilde{Q}(\mathbf{z}^*)$. Efficiency: the prover performs at most $5 \cdot 2^m + 2^n + K$ multiplications; the verifier’s work is an arithmetic circuit determined by m and K alone, dominated by one evaluation of $\tilde{W}(\mathbf{z}^*)$.*

The protocol is the sumcheck for a product of two multilinear [HJR⁺25, Lemma 2.3] applied to Equation (2), the *jagged sumcheck*. Each round polynomial has degree two and is sent as its values at $0, 1, 2$, one value more than the paper’s accounting [HJR⁺25, App. A]. With challenges $\mathbf{z}^* = (r_0, \dots, r_{m-1})$ in sampling order, least-significant variable first, the closing claim is

$$T_{\text{final}} = \tilde{Q}(\mathbf{z}^*) \tilde{W}(\mathbf{z}^*). \quad (3)$$

The proof carries $v_Q = \tilde{Q}(\mathbf{z}^*)$, which the opening of §6.5 authenticates; the verifier obtains $\tilde{W}(\mathbf{z}^*)$ from the public prefix sums.

6.3 Evaluating the indicator: the jagged assist

Materialising W would cost $\Theta(M)$ extension-field elements. Instead, the indicator is a sum over columns of a function of each column’s boundaries [HJR⁺25, Claim 3.2.1, Eq. (5)],

$$\tilde{W}(\mathbf{z}^*) = \sum_{k < K} w_k \tilde{g}(\mathbf{z}_{\text{row}}, \mathbf{z}^*, t_k, t_{k+1}), \quad g(a, b, c, d) = 1 \iff b < d \wedge b = a + c,$$

where g is computed by a width-four read-once branching program that reads one bit of each of (a, b, c, d) at a time, least significant first, tracking the carry of $a + c$ and whether $b < d$ [HJR⁺25, Claim 3.2.2]. Its multilinear extension is evaluated by the dynamic program of Holmgren and Rothblum [HR18], stated and proved as in Hemo et al. [HJR⁺25, Lemma 4.2]. Points are passed to the program most-significant coordinate first [HJR⁺25, p. 5] while the sumcheck samples least-significant first, so the program’s point is $\text{rev}(\mathbf{z}^*)$. Evaluated directly, the indicator costs $O(mK)$ [HJR⁺25, Prop. 3.2], too much for a recursion circuit at $K \approx 3.6 \cdot 10^4$.

The *jagged assist* [HJR⁺25, Thm. 1.5] delegates it. With $u_k = \text{bits}_{\text{BE}}(t_k) \parallel \text{bits}_{\text{BE}}(t_{k+1})$ the $2(m+1)$ boundary bits of column k , the prover proves

$$\tilde{W}(\mathbf{z}^*) = \sum_{x, y \in \{0,1\}^{m+1}} \sum_{k=0}^{K-1} w_k \text{eq}((x, y), u_k) \text{BP}(\mathbf{z}_{\text{row}}, \text{rev}(\mathbf{z}^*), x, y) \quad (4)$$

by one degree-two sumcheck over the $2(m+1)$ boundary variables of Equation (4). This is the batch evaluation protocol of Hemo et al. [HJR⁺25, Lemma 5.1] specialised: the K evaluations are already combined by the weights w_k rather than by fresh random coefficients, so no further randomness is needed and the error is that of the sumcheck alone (Theorem 6.2).

Theorem 6.2 (Jagged assist [HJR⁺25, Thm. 1.5, Lemma 5.1], as instantiated). Completeness: *if $\widetilde{W}(\mathbf{z}^*)$ is claimed correctly, the verifier accepts*. Soundness: *otherwise it rejects except with probability $4(m+1)/|\mathbb{F}_{p^4}|$* . Efficiency: *the verifier evaluates BP once, $48(m+1)$ multiplications, and the K equality factors $\sum_k w_k \mathbf{eq}(\boldsymbol{\rho}, u_k)$ at the closing point $\boldsymbol{\rho}$, $O(mK)$ operations; the prover’s cost is $O(mw^2(K+w))$ operations for a program of width w [HJR⁺25, Lemma 4.6].*

The second verifier term dominates: it is linear in the number of columns, about $2 \cdot 10^6$ extension multiplications at the core shard’s m and K , against a few thousand for the branching program, consistent with the paper’s $m \cdot 2^k$ [HJR⁺25, p. 7]. The native verifier computes the value by the closed form and replays the assist transcript; the recursion circuit checks the assist’s closing identity explicitly.

6.4 Several commitments opened as one

The construction of Hemo et al. [HJR⁺25] assumes one dense oracle. ZIREN commits a shard in several ordered *input-commitment rounds*, preprocessed first and main last, each with its own root, and opens them together; each round uses the stacked layout of SP1 Hypercube [Suc25a], which the paper’s concrete evaluation also uses, cutting the dense side into 2^8 blocks of 2^{21} cells [HJR⁺25, §7].

Round r contains columns $\mathbf{c}_{r,0}, \dots, \mathbf{c}_{r,K_r-1}$ of real area $a_r = \sum_k h_{r,k}$, committed over an area $A_r \geq a_r$ that is a multiple of the stacking height $H = 2^\ell$:

$$\mathbf{q}_r = \mathbf{c}_{r,0} \parallel \dots \parallel \mathbf{c}_{r,K_r-1} \parallel \mathbf{0}^{A_r - a_r}, \quad \mathbf{Q} = \mathbf{q}_0 \parallel \mathbf{q}_1 \parallel \dots \parallel \mathbf{q}_{R-1}, \quad A = \sum_{r < R} A_r.$$

The reduction of §6.2 runs on this logical $\mathbf{Q}$; the Merkle roots $C_0, \dots, C_{R-1}$ authenticate the input rounds separately. Their fixed order determines the stripe order used for batching and authentication. Each gap $A_r - a_r$ is represented by explicit synthetic zero columns of zero claim and height at most 2^n , so the prefix sums describe every committed cell and only the suffix from A to M is implicit. That unallocated suffix is the “free padding” of Hemo et al. [HJR⁺25, fn. 4], on which the indicator vanishes because g requires $b < d$. Under a recursion area pin the gap is split into a fixed number of columns, zero-height ones allowed, so the recursion program’s column count does not depend on a child’s row counts.

Each round is cut into $S_r = A_r/H$ stripes of H consecutive entries under its own root. With $S = \sum_r S_r$ and $b = \lceil \log_2 \max(S, 1) \rceil$, the opening domain has $\ell + b$ variables, which equals m for the nonempty stacked layout defined above. Its point is $(\mathbf{z}_{\text{stack}}, \mathbf{z}_{\text{batch}})$: ℓ coordinates select a position inside a stripe and b interpolate the ordered list of stripes across all rounds.

6.5 One opening across all roots

If the reduction point has fewer than $\ell + b$ coordinates, the prover appends Fiat–Shamir coordinates after the assist, each contributing a factor $1 - r_j$; in the multi-round layout above, $m = \ell + b$, so no coordinates are appended. For round r and stripe s let $e_{r,s} = \widetilde{\mathbf{q}_{r,s}}(\mathbf{z}_{\text{stack}})$. The prover reports these in round-major, stripe-major order; the verifier zero-pads the list to 2^b and checks

$$v_Q = (e_{0,0}, \dots, e_{0,S_0-1}, \widetilde{e_{1,0}, \dots, e_{R-1,S_{R-1}-1}})(\mathbf{z}_{\text{batch}}). \quad (5)$$

One stacked WHIR or BaseFold proof then authenticates every $e_{r,s}$ against $C_0, \dots, C_{R-1}$ at the common stack point, so the reduction, assist and opening each run once while each input round retains its own Merkle root. All rounds must share the stacking height and PCS mode; a mixed transcript is rejected. The implementation also offers BaseFold as the opening; every configuration in this paper uses WHIR.

The step order, the round order and the reversal convention fix the challenges; changing any of them is a protocol change.

Soundness. The three algebraic steps contribute $\kappa/|\mathbb{F}_{p^4}|$ (column batching), $2m/|\mathbb{F}_{p^4}|$ (jagged sumcheck, Theorem 6.1) and $4(m+1)/|\mathbb{F}_{p^4}|$ (assist, Theorem 6.2), which together form the row “reduction to the dense polynomial” of Table 6. The paper states plain soundness of each reduction; the round-by-round form Theorem 8.2 assumes follows because each is a sumcheck, round-by-round sound with the same per-round errors [CY24]. The opening’s terms are those of Section 7, and all of them are accounted in §8.2.

Protocol 6.3 (Jagged opening).

Public input: ordered roots $C_0, \dots, C_{R-1}$, the prefix sums $(t_k)_k$, a row point $\mathbf{z}_{\text{row}}$ and column claims $(y_k)_{k < K}$ at $\mathbf{z}_{\text{row}}$.

Witness: the committed rounds $\mathbf{q}_0, \dots, \mathbf{q}_{R-1}$.

Prover $\mathcal{P}$		Verifier $\mathcal{V}$
1.	$\xleftarrow{\mathbf{z}_c}$	Sample $\mathbf{z}_c \in \mathbb{F}_{p^4}^\kappa$ and set $T = \sum_k \text{eq}(k, \mathbf{z}_c) y_k$.
2. Run the jagged sumcheck on $\langle \tilde{Q}, \tilde{W} \rangle = T$ (m rounds, Theorem 6.1); send $v_Q = \tilde{Q}(\mathbf{z}^*)$.	$\xrightarrow{\text{rounds}, v_Q}$	Check every round; obtain $\mathbf{z}^*$ and T_{final} .
3. Run the jagged assist at $\text{rev}(\mathbf{z}^*)$ ($2(m+1)$ rounds, Theorem 6.2).	$\xrightarrow{\text{rounds}}$	Obtain $v_W = \tilde{W}(\mathbf{z}^*)$; check $T_{\text{final}} = v_Q v_W$ (3).
4.	$\xleftarrow{\mathbf{z}_{\text{ext}}}$	Sample the point-extension coordinates, if any.
5. Send the stripe evaluations $(e_{r,s})_{r,s}$.	$\xrightarrow{(e_{r,s})}$	Check (5).
6. Open every stripe at the common stack point in one batched stacked WHIR or BaseFold proof.	$\xleftarrow{\text{opening}}$	Accept if and only if the opening verifies against $C_0, \dots, C_{R-1}$.

6.6 Quantised committed geometry

The stacking height is $H = 2^{21}$, so a round is committed as a whole number of 2^{21} -cell blocks, and a column is at most 2^{22} rows, which bounds the gap one synthetic column can absorb. A round with $\nu = \lceil a_r/H \rceil$ blocks of real data is committed in a *bucket* of $\hat{\nu}$ blocks, where $\hat{\nu} = \nu$ for $\nu \leq 4$ and otherwise the next multiple of eight:

1, 2, 3, 4, 8, 16, 24, ... blocks.

For an unpinned round, which is every core round, the gap is split into

$$\left\lceil \frac{(\hat{\nu} - \hat{\nu}_{\text{prev}}) H}{2^{22}} \right\rceil \text{ columns,} \quad \hat{\nu}_{\text{prev}} = \text{the bucket below } \hat{\nu},$$

at least one: the number the widest gap of the bucket needs, so the count depends on the bucket alone, and a narrower gap leaves trailing columns short or empty. A pinned round, as in the compress machine, uses a fixed column count.

Both rules make a shard's committed geometry, and hence the recursion program that verifies it and that program's key, a function of the chip set, the first-shard flag and the two buckets rather than of exact row counts. Rounding up to the next block, or letting the padding-column count follow the gap, would make every row profile a distinct verifier program. Under the quantised rule the key space is small enough to enumerate offline, which lets a verifier hold a fixed allowlist covering every shard shape the declared clusters admit (Section 3.3). The overhead is the padding inside a bucket, committed and opened like any other cell.

The reduction's memory stays proportional to the real data: W is never materialised (§6.3); the suffix from A to M is treated as zero rather than allocated, which is why inter-round padding must be explicit and the suffix need not be; and the stripe evaluations of Equation (5) are read off the committed interleaved layout, so a round is traversed once to commit and once to open.

7 The WHIR opening

WHIR [ACFY24b] opens the jagged commitment of Section 6 (the commitment JC of §3.2). It authenticates the claim $v_Q = \tilde{Q}(\mathbf{z}^*)$ left by the jagged reduction, as an interactive oracle proof of proximity (IOPP) for

Table 5: The core and compress Jagged-WHIR schedules. Analysis is the regime queries are counted in. Queries are per committed round, followed by the final query phase. OOD entries count samples after committed folds; the fold grind applies to every folding challenge. The stripe bound covers both ordered input-commitment rounds.

Profile	first rate	analysis	folds	queries	OOD	query grind	fold grind	batch grind	LogUp-GKR grind	stripe bound	final variables
core	2^{-2}	unique decoding	[3, 6, 6]	[124, 88], 85	[2, 2]	22 bits	0	14 bits	22 bits	544	6
compress	2^{-3}	Johnson, $m = 6$	[2, 6, 6]	[58, 28], 19	[2, 2]	26 bits	27 bits	27 bits	22 bits	64	7

constrained Reed-Solomon codes, whose multilinear also satisfies a stated weighted sum. This section gives the protocol as implemented and the commitment layout; the soundness of the schedule is accounted in §8.2. The schedule is the opening’s principal parameter: at a fixed soundness level it trades proof size against prover time.

7.1 The protocol

WHIR tests membership of $f : L \rightarrow \mathbb{F}$ in $\text{CRS}[\mathbb{F}, L, m, \hat{w}, \sigma]$ (Section 2). An iteration with folding factor k runs k sumcheck rounds, sends a folded oracle re-encoded at the round’s rate, answers out-of-domain samples, answers t shift queries by opening 2^k symbols each, and recurses on a constrained code with k fewer variables; the final polynomial is sent in the clear. The out-of-domain samples distinguish WHIR from BaseFold: they pin the folded oracle to a unique nearby codeword before the shift queries are drawn. At the same provable level WHIR needs fewer queries in later rounds than BaseFold and a handful of committed rounds rather than one per variable, while sharing its prover skeleton (encode, Merkle-commit, fold).

7.2 Commitment layout

ZIREN uses the *interleaved* commitment: a polynomial in ν variables is reshaped into a $2^{\nu-k_0} \times 2^{k_0}$ matrix whose column c is the stride- 2^{k_0} slice $f[h \cdot 2^{k_0} + c]$; each column is Reed-Solomon encoded independently (one DFT per column), and, for a single stripe, Merkle leaf j holds the 2^{k_0} column values at domain point x_j . Folding a leaf by the round’s challenge gives the folded polynomial at x_j , so a query answer enters the next sumcheck as an ordinary constraint whose weight folds in closed form.

The layout is applied separately to every input-commitment round of Section 6.4. Each round retains its own Merkle root; one batched WHIR opening authenticates all input rounds against their respective roots. Within each input round’s Merkle tree, a leaf of the first oracle holds one coset row from every stripe under that root, $S_r \times 2^{k_0}$ field elements. The first folding factor k_0 is therefore small: leaf width multiplies every first-round query, and re-hashing those leaves is the recursion verifier’s dominant cost. Later rounds query a single folded polynomial and fold more variables at a time. Each committed round re-encodes the folded polynomial at its new rate; the opening of a Reth shard costs about 39 ms in the GPU profile, measured from the host while the round data remain device-resident, the round-0 re-encoding being the largest term. Under the core schedule grinding is spent only on the query phases and on the batching and LogUp-GKR challenges, not on folding challenges (Table 5).

Table 5 gives the two schedules; §9.4 describes how a schedule controls proof size. Each recommitment lowers the rate by three powers of two. The core schedule is tuned for prover time: the first oracle has rate 2^{-2} , and at log stacking height 21 the three folds consume 15 variables, the remaining six-variable polynomial being sent in the clear after its final query phase. The compress schedule, under which the root of the recursion tree proves, is tuned for the size of the published proof: it starts at 2^{-3} , the lowest rate the two-adicity of KoalaBear admits at this stacking height, counts queries under the Johnson bound, folds two variables first so that a first-round query opens half the leaf, and grinds every folding and batching challenge, which the list size of the Johnson-bound analysis requires. Each of its parameters is a setting, so a deployment with a different security requirement changes the schedule, not the prover.

8 Security

This section collects the security argument. It states the relation the block proof is an argument for (§8.1), the concrete soundness of each proof (§8.2), how those levels compose into a statement about the non-interactive

block proof (§8.3), how the verifying keys tie that proof to the advertised machine (§8.4), the one assumption that is not about hashing (§8.5), what makes the satisfying trace unique (§8.6), why it is the MIPS32 execution, with the determinism of every extracted chip machine-checked in Lean 4 (§8.7), and what changes against quantum adversaries (§8.8). Every hypothesis the end-to-end statement carries is named in this section, and the ones that remain open are listed together in §8.1.

8.1 The proved relation

The shard argument proves one shard; the statement of Section 4.4 is about an execution. Between the two sits the arithmetised relation the recursion tree is knowledge sound for, with each condition assigned to the component that enforces it:

$$\begin{aligned} \mathcal{R}_{\text{arith}} = \Big\{ & (\text{vk}, h) : \exists (\mathbf{T}_i, \text{pv}_i)_{i < N} \quad \forall i : \mathbf{T}_i \text{ is a valid shard for } \text{pv}_i \text{ and } \text{vk}, h_i = h \quad (\text{shard argument}) \\ & \wedge \forall i < N - 1 : \text{sh}_{i+1} = \text{sh}_i + 1, \text{exit}_i = \text{entry}_{i+1}, \quad (\text{compose nodes}) \\ & \quad \text{cur}_{\text{out}, i} = \text{cur}_{\text{in}, i+1} \\ & \wedge \text{entry}_0 = \text{start}(\text{vk}), \text{cur}_{\text{in}, 0} = 0 \quad (\text{first-shard leaf}) \\ & \wedge \text{sh}_0 = 1, \text{pc}(\text{exit}_{N-1}) = 0, \sum_i D_i = O, \quad (\text{root}) \\ & \quad \forall i : \text{code}_i = 0 \Big\} \end{aligned}$$

Validity is Definition 2.6; sh_i , h_i and code_i are fields of pv_i ; the halt row sets the exit program counter to zero; $\text{start}(\text{vk})$ is the start state the key fixes; the entry and exit states are program-counter pairs; O is the identity of the digest’s curve group; the cursors tile the address space; D_i is the shard’s cross-shard digest net of the offset D_{off} of §5.5. Theorem 8.2 (§8.3) is knowledge soundness for $\mathcal{R}_{\text{arith}}$, and Theorem 8.6 (§8.6) with the conformance evidence of §8.7 is what connects $\mathcal{R}_{\text{arith}}$ to $\mathcal{R}$, through Lemma 8.5, which joins the shards’ memory into one execution.

Summary of the proved statement. Theorem 8.2 is knowledge soundness for $\mathcal{R}_{\text{arith}}$ under the hypotheses below, the last of three layers after the functional IOP and the interactive commit-and-open argument (§8.3). Theorem 8.6 makes the rows a valid shard reaches unique up to permutation, given chip determinism (§8.7), well-formed shard boundary rows and real-row exhaustiveness; identifying them with an execution needs the conformance evidence of the same section. Neither statement depends on a particular chip set, field or proximity test.

Open hypotheses. The end-to-end statement holds under five hypotheses that this paper does not discharge:

1. round-by-round knowledge soundness of every node’s committed protocol, which carries function binding of the jagged WHIR commitment with an unquantified loss (§8.3);
2. that each node’s extractor succeeds on the transcripts its parent’s extractor produces (§8.3);
3. that the leaf and compose programs, which are not extracted, enforce the compose relation, the key binding included (§8.4);
4. Assumption 8.4, for which we give no numerical security bound (§8.5);
5. determinism of the two SHA-256 control chips, well-formed shard boundary rows (one per touched word, linked to actual accesses), and real-row exhaustiveness, the open premises of the memory and trace-uniqueness arguments (§8.6).

Conformance evidence (§8.7), not a proof, identifies the arithmetised machine with the MIPS32 instruction set.

Table 6: Soundness of the analysed configuration by component group, in bits ($-\log_2$ of the error) as computed by `soundcalc` [Eth26a], each component floored to whole bits. The query group includes the two shift phases and the final-query term, counted once. A group’s figure is the union over its members, and the last row the union over the stage. All figures are for the interactive protocol.

Component group	core	compress
Query phases	104.42	104.91
Folding rounds	100.93	101.69
LogUp-GKR	106.00	116.00
Batching	107.00	103.00
Zerocheck	109.00	114.00
Reduction to the dense polynomial	116.00	116.00
Out-of-domain samples	213.00	200.00
Union over the stage	100.74	101.10

8.2 Soundness accounting

How the level is computed. We compute soundness with `soundcalc` [Eth26a], the Ethereum Foundation’s calculator for hash-based proof systems, extended for jagged commitments over WHIR, from a configuration that records the field, the trace dimensions and the schedule of Table 5; a census program kept with the source emits its machine figures from the machine the prover builds.

We use the calculator to evaluate the component errors of the interactive protocol: the out-of-domain samples and shift queries of each WHIR round, the batching of the committed stripes in the opening (column batching is counted in the jagged reduction), each folding round, the final polynomial, the three terms of the jagged reduction (Section 6), the zerocheck and the LogUp-GKR argument, with grinding counted inside the term it protects. The calculator’s `fin` term accounts for the final query phase; we include it once and sum the component errors, whereas the calculator’s reported total takes their minimum.

Query soundness is counted in the unique-decoding regime for the core schedule and under the Johnson bound for the compress schedule [BSCH⁺25]; neither needs a proximity-gap conjecture. A proof’s level is $-\log_2$ of the *sum* of its component errors, a union bound rather than the smallest component, and a block’s level sums over every node of its recursion tree. The analysis follows WHIR’s [ACFY24b], adding the batching over several commitments and the jagged terms; levels computed by other calculators, such as RISC Zero’s [RIS24], are comparable only when they charge the same components.

Result. Table 6 gives the components. Each stage has more than 100 bits of interactive soundness; the union bound for one core-schedule proof and one compress-schedule proof gives 99.91 bits; the folding rounds are the largest error group in both schedules, and only the compress schedule grinds them (Table 5). We call $-\log_2 \sum_i \varepsilon_i$, summed over every node i of a block’s recursion tree, the block’s *composite security*; when all nodes have the same error bound, this is $\log_2$ of the node count below that common per-proof level; for different schedules we sum their individual errors. For an illustrative tree of 150 proofs, comprising 149 proofs under the core schedule and one root under the compress schedule, it is 93.51 bits. These figures bound the interactive protocol’s information-theoretic terms.

Applying the Q factor of Theorem 8.2 to these quantified interactive terms gives $93.51 - \log_2 Q$ bits, or 53.5 bits at $Q = 2^{40}$. This calculation excludes the unquantified function-binding loss of Layer 2 (§8.3), the digest term of Assumption 8.4, and the theorem’s separate hash-collision term. It therefore does not give a numerical bound on the full non-interactive proof’s security.

8.3 Security in three layers

The figures above bound an interactive protocol; the deployed proof is its Fiat–Shamir compilation. The commitment layer connects these constructions. We separate the argument into the three layers of the commit-and-open paradigm in its functional form [CGKY26]: an information-theoretic protocol whose verifier reads the prover’s messages only through evaluation queries; the interactive argument obtained by committing

to those messages; and the non-interactive argument obtained by Fiat–Shamir. Each layer has its own security statement, and the statements compose in that order.

Layer 1: a functional interactive oracle proof. Let Σ be an alphabet and $\ell \in \mathbb{N}$. A *query class* is a set $\mathcal{Q}$ of functions $\alpha : \Sigma^\ell \rightarrow D$; a *constraint* is a pair $(\alpha, \beta) \in \mathcal{Q} \times D$, satisfied by Π when $\alpha(\Pi) = \beta$.

Definition 8.1 (Functional IOP [CGKY26]). A k -round public-coin functional interactive oracle proof (FIOP) with query class $\mathcal{Q}$ is an interactive oracle proof in which, in round i , the prover sends a string $\Pi_i \in \Sigma^{\ell_i}$ and the verifier replies with fresh randomness ρ_i , and in which the verifier accesses each Π_i only through constraints $(\alpha, \alpha(\Pi_i))$ with $\alpha \in \mathcal{Q}$. Knowledge soundness with error $\varepsilon_{\text{FIOP}}$ means that an extractor, given one accepting transcript $(\Pi_1, \rho_1, \dots, \Pi_k, \rho_k)$, outputs a witness except with probability $\varepsilon_{\text{FIOP}}$.

The shard protocol of Section 3 is an FIOP for the *evaluation* query class $\mathcal{Q}_{\text{ev}}$ over $\Sigma = \mathbb{F}$: a committed table $\Pi \in \mathbb{F}^{2^n}$ is read only through $\alpha_z(\Pi) = \tilde{\Pi}(z) \in \mathbb{F}_{p^4}$ for points $z \in \mathbb{F}_{p^4}^n$, the claims (z, v) of Section 2. Its rounds are the committed traces, the **LogUp-GKR** layers, the zerocheck, the batching of column claims and the jagged reduction; a message the verifier reads in full, such as a sumcheck round polynomial, is a short string queried at every position. In Table 6, the rows for **LogUp-GKR**, the zerocheck and the reduction to the dense polynomial bound $\varepsilon_{\text{FIOP}}$, and the rows for query phases, folding rounds, the final polynomial and the out-of-domain samples belong to the opening of Layer 2; batching sits at the boundary, since it combines claims just before they are opened.

Layer 2: the interactive commit-and-open argument. The Funky compiler [CGKY26] replaces each oracle Π_i by a commitment under a functional commitment scheme for the same query class and, after the last round, opens every queried constraint. For ZIREN the functional commitment is the jagged commitment of Section 6 opened through **WHIR**: the jagged reduction turns the shard’s column claims into one evaluation claim on the dense polynomial, and the batched **WHIR** proof opens it. Its security requirement is *function binding*: no efficient prover opens one commitment to constraints that no single string satisfies.

Knowledge soundness of the interactive argument then reduces to the knowledge soundness of the FIOP, with error $\varepsilon_{\text{FIOP}}$, and to function binding of the commitment [CGKY26].

Assumptions of Layer 2. For the jagged **WHIR** commitment, function binding rests on the collision resistance of **H** and on the soundness of **WHIR**. We have not proved it in that form, and we do not quantify the loss of the reduction; function binding is a hypothesis of this layer, and Theorem 8.2 carries it inside its per-node round-by-round premise, which is stated for the committed protocol. No figure of Table 6 depends on it: those bound the information-theoretic terms.

Layer 3: Fiat–Shamir in the random-oracle model. The deployed proof is non-interactive. In the random-oracle model a Merkle commitment admits straightline extraction, which reads the committed strings from the adversary’s oracle queries without rewinding [CY24], so the random-oracle bound carries no rewinding loss. The Fiat–Shamir security of arguments built from functional commitments is studied by Chiesa et al. [CGKY26]; for ZIREN we state the compiled level through the per-round premise of the following theorem.

Theorem 8.2 (Conditional composition). *Work in the random-oracle model, and let an adversary make at most Q oracle queries. Let every proof in the tree, including each shard, leaf and compose proof, be the Fiat–Shamir compilation [BSCS16] of an interactive protocol that is round-by-round knowledge sound with error $\varepsilon_i = \sum_c \varepsilon_{i,c}$. Let a block proof contain n_π such proofs, each verified once by its parent, with the root checked by the external verifier, every leaf and compose key, the root’s included, an entry of the allowlist, and every shard proof verified against the guest key vk (§8.4). Assume further that each node extractor succeeds on transcripts produced by its parent extractor, not only on transcripts drawn from an honest interaction, and that the leaf and compose programs enforce the compose relation of Section 8.4, which no theorem of §8.7 covers. Then the block proof is an argument of knowledge for $\mathcal{R}_{\text{arith}}$ with knowledge error at most*

$$Q \sum_{i=1}^{n_\pi} \varepsilon_i + \varepsilon_{\text{hash}} + \varepsilon_{\text{digest}},$$

where $\varepsilon_{\text{digest}}$ is the error of Assumption 8.4 and $\varepsilon_{\text{hash}} = O(Q^2/2^d)$ is the probability of a collision in the d -bit output of H , which builds the transcript and the Merkle trees, with $d = 248$ in the measured instantiation, the term SWIRL states explicitly for its own compilation [Ope26b, Cor. 4.2.6].

Proof sketch. Compiling a round-by-round knowledge sound protocol with Fiat–Shamir costs a factor in the number of oracle queries: an adversary may re-query the oracle on modified prefixes, and the standard analysis charges Q attempts against the per-round error [BSCS16, CY24]. Within the tree, the extractor of the compiled argument runs the extractor of each verified proof in turn: a compose node’s extractor yields its children’s transcripts, and so on down to the shard proofs, whose extractor yields the witness rows. The allowlist hypothesis makes each step an extraction from the intended verifier; without it a node could verify a program other than the compose or leaf program. The hypothesis about parent-produced transcripts is not discharged here: a child’s extractor is invoked on a transcript the parent’s extractor constructed, which is not the distribution its guarantee is stated against, and a compose node checks its children inside its own circuit rather than as further rounds of one protocol. Discharging this hypothesis would remove the extractor-compatibility premise; the theorem’s other hypotheses would still apply. We leave that premise open. Given it, the failure events combine by the union bound. $\square$

8.4 Verifying-key binding

The compose relation. Write a node’s public values as

$$\text{pv} = (\text{vk}, \text{sh}_{\text{in}}, \text{sh}_{\text{out}}, \text{entry}, \text{exit}, \text{cur}_{\text{in}}, \text{cur}_{\text{out}}, D, h, \text{code}, \text{root}),$$

the guest’s verifying key, the index of the node’s first shard and the index after its last, the entry and exit machine states, the memory-address cursors the node opens and closes with, the accumulated cross-shard digest, the digest of committed output, the exit code, and the root of the allowlist of admissible recursion keys; a completion flag **complete**, set only by the root, accompanies the tuple. With a delay slot the entry and exit states are *pairs* $(\text{pc}, \text{next_pc})$, so adjacency must pin both components. A compose node of arity k proves $\mathcal{R}_{\text{comp}}(\text{pv}; \pi_0, \dots, \pi_{k-1})$, which holds when every check of Algorithm 2 passes: each π_i verifies against a key the allowlist at **root** contains, the children share vk , h , **code** and **root**, their ranges are adjacent and their cursors tile, and pv spans the children with $D = \sum_i D_i$. Digests are stored net of the offset D_{off} of §5.5, so $D = O$ below means the stored value equals D_{off} .

Leaves and the root. A leaf proves the same relation for its one child, except that the child is a core shard proof verified against the guest’s vk ; it is the leaf’s own key that the allowlist must contain, checked by its parent. A leaf sets sh_{in} and sh_{out} from the shard indices it verifies. The leaf of the first shard, index 1, also requires its entry state to be **start**(vk), the start state the guest key fixes, and its starting cursors to be zero. The root’s additional checks in Algorithm 2 are $\text{sh}_{\text{in}} = 1$, so that its leftmost leaf is the first-shard leaf; an exit program counter of zero, which the halt row sets (§5.10.4); $D = O$, which closes the memory argument and the cross-shard digest over the execution; and the exit code zero that the relation of Section 4.4 admits. It then sets **complete**.

The statement of Section 4.4 concerns the advertised machine only if every leaf verified a *core* proof of that machine, every compose node ran the compose program, and the external verifier checked the root under a key of that machine. Three checks give this. A compose node checks that its children’s recursion keys belong to a Merkle allowlist (Algorithm 2, line 2); inside the tree that root is a prover-supplied witness, so it is exported as a public value of the root proof. A leaf verifies each shard proof against the guest key vk its public values carry, and every node carries that vk upward unchanged. The external verifier then checks the root proof under the recursion key the proof names, requires that key to be in the published allowlist and the exported root to equal the published root, requires the completion flag **complete** that only the root sets (§3.3), and compares the exported vk with the key of the guest program it expects. Every node’s key, the root’s included, is thus an allowlist entry or the expected guest key.

The recursion machine is outside the extraction of §8.7, so this binding is part of hypothesis 3 of §8.1. A prover configured to skip the in-circuit check produces a different compose key, which the allowlist does not contain, and is rejected. The allowlist is enumerated for leaf as well as compose programs (§3.3). The shard-to-leaf step is uniform per shard shape but not across shapes, and the allowlist lets one compose program accept the non-uniform set of leaves by committing to their keys in advance.

Algorithm 2: The compose node of arity k , and what the root additionally requires. Line 2 is the binding of §8.4: without it the node would verify *some* proof rather than a proof of this machine.

Input: child proofs $\pi_0, \dots, \pi_{k-1}$ with public values $\mathbf{pv}_0, \dots, \mathbf{pv}_{k-1}$, recursion keys $\mathbf{vk}_0^{\text{rec}}, \dots, \mathbf{vk}_{k-1}^{\text{rec}}$ and membership paths $\text{path}_0, \dots, \text{path}_{k-1}$; a claimed allowlist root root

Output: the node's public values $\mathbf{pv}$, or $\perp$

```

1 for  $i \leftarrow 0$  to  $k - 1$  do
2   if  $\neg \text{MerkleVerify}(\text{root}, \mathbf{vk}_i^{\text{rec}}, \text{path}_i)$  then return  $\perp$            // the child's key is admissible
3   if  $\neg \text{Verify}(\mathbf{vk}_i^{\text{rec}}, \mathbf{pv}_i, \pi_i)$  then return  $\perp$ 
4   if  $\exists i : \text{root}_i \neq \text{root}$  then return  $\perp$                                // children share the root
5   if  $\mathbf{vk}_i, h_i, \text{code}_i$  not all equal over  $i$  then return  $\perp$ 
6   for  $i \leftarrow 0$  to  $k - 2$  do
7     if  $\text{sh}_{\text{out},i} \neq \text{sh}_{\text{in},i+1} \vee \text{exit}_i \neq \text{entry}_{i+1}$  then return  $\perp$            // ranges adjacent
8     if  $\text{cur}_{\text{out},i} \neq \text{cur}_{\text{in},i+1}$  then return  $\perp$                                // cursors tile
9    $D \leftarrow \sum_i D_i$                                                    // digests accumulate
10   $\mathbf{pv} \leftarrow (\mathbf{vk}, \text{sh}_{\text{in},0}, \text{sh}_{\text{out},k-1}, \text{entry}_0, \text{exit}_{k-1}, \text{cur}_{\text{in},0}, \text{cur}_{\text{out},k-1}, D, h, \text{code}, \text{root})$ 
11  if this node is the root then
12    if  $\text{sh}_{\text{in}} \neq 1 \vee \text{pc}(\text{exit}) \neq 0$  then return  $\perp$            // first shard; exit pc zero
13    if  $D \neq O \vee \text{code} \neq 0$  then return  $\perp$ 
14    set complete
15 return  $\mathbf{pv}$ 

```

8.5 Security of the cross-shard digest

This subsection states the curve-addition constraints, message encoding and computational binding assumption for the elliptic-curve option of §5.5.

The addition row. The chord identities

$$C_x = (x_1 + x_2 + x_3)(x_2 - x_1)^2 - (y_2 - y_1)^2, \quad C_y = (y_1 + y_3)(x_2 - x_1) - (y_2 - y_1)(x_1 - x_3)$$

both carry the factor $(x_2 - x_1)$. At $P_2 = -P_1$, $C_x = -4y_1^2 \neq 0$ whenever $y_1 \neq 0$; at $P_2 = P_1$ both hold for every P_3 , so the row witnesses $(x_2 - x_1)^{-1}$ and constrains $(x_2 - x_1) \cdot (x_2 - x_1)^{-1} = 1$, which excludes both cases for points on the curve. The offset point D_{off} of §5.5 has $\pm D_{\text{off}}$ outside the image of the map, so a running sum coinciding with the next interaction's point is unprovable rather than unconstrained, and Assumption 8.4 bounds the probability of such a coincidence by $\varepsilon_{\text{digest}}$.

The lift. SP1 derives the x -coordinate from a Poseidon2 permutation constrained inside the chip, with an 8-bit tweak hashed in and the sign of y fixed by range checks, so its points are random-oracle outputs; its tweak is uncanonicalised as ours is, which under that model is harmless [RR24, Suc26a]. Here the message is the x -coordinate, which changes the security argument: the lift is a map on *pairs*, $\text{enc}(\mathbf{m}, \text{offset})$. The chip constrains the offset only to be a byte in the low positions of x_6 ; the honest prover takes the smallest admissible one, but minimality is not an AIR constraint. Write $\mathcal{P}$ for the *reachable pairs* $(\mathbf{m}, \text{offset})$ an interaction of a satisfying trace can carry, given the sending chips' range facts and the byte bound on the offset. $\mathcal{P}$ is far smaller than $\mathbb{F}^7 \times [0, 256)$, since every limb of a memory or syscall tuple is a byte, a bounded clock limb or a bounded address. A send and a receive cancel only if they agree on the offset, so the free offset costs completeness, not soundness; but an adversary seeking a vanishing combination searches over $\mathcal{P}$.

Lemma 8.3 (Injectivity of the lift). *Assume the message limbs satisfy the range facts the sending chips enforce, so that $256 m_6 + \text{offset} < p$. Then enc is injective: limbs $x_0, \dots, x_5$ are $m_0, \dots, m_5$; x_6 is m_6 shifted above a low byte holding the offset, which the bound makes separable; and the two sign bands are disjoint. So $\mathbf{m}$ and the offset are recoverable from the point.*

If a claimed execution has send and receive multisets $S \neq R$ over $\mathcal{P}$ with equal digests, then

$$\sum_i c_i \text{enc}(\mathbf{m}_i, \text{offset}_i) = O, \quad c = \text{mult}_S - \text{mult}_R \neq 0;$$

and by Lemma 8.3, equal encoded multisets give equal message multisets. Soundness is therefore exactly:

Assumption 8.4 (No reachable vanishing combination). Let λ be the security parameter, let N_{int} bound the interactions a block proof may contain and B the per-interaction multiplicity bound, and let $\mathcal{P}$ be the reachable pairs above. For every adversary $\mathcal{A}$ running in time at most $T(\lambda)$ and making at most $q(\lambda)$ queries to the hash functions of the protocol,

$$\Pr [\mathcal{E}(\mathcal{A})] \leq \varepsilon_{\text{digest}}(\lambda, T, q, N_{\text{int}}, B),$$

where $\mathcal{E}(\mathcal{A})$ is the event that $\mathcal{A}$ outputs reachable pairs $\{(\mathbf{m}_i, \text{offset}_i)\}_{i \leq N_{\text{int}}} \subseteq \mathcal{P}$ together with a non-zero integer vector $c \in \mathbb{Z}^{N_{\text{int}}}$ satisfying $\|c\|_\infty \leq B$ and $\sum_i c_i \text{enc}(\mathbf{m}_i, \text{offset}_i) = O$ in $E(\mathbb{F}_{p^7})$.

Theorem 8.2 carries $\varepsilon_{\text{digest}}$ as an additive term of its bound. With SP1’s hashed lift a discrete-logarithm reduction would apply. Here the prover chooses points from a structured set, reading a message off a target x -coordinate by inverting the limb layout, and the offset widens the set by a factor of at most 256. The sparsity of $\mathcal{P}$ blocks the obvious attack of reading messages off the x -coordinates of scalars that sum to zero, since almost no x has a reachable limb shape; this depends on every sending chip bounding every limb, an obligation not yet collected in one certificate. Sparsity gives no bit estimate: the generalised-birthday cost falls with the number of lists and must be bounded by the interactions a trace can hold, not only by $|E(\mathbb{F}_{p^7})| \approx 2^{217}$.

That analysis is open, as is a parameter analysis of the curve: SP1’s memo cites a best known discrete-logarithm attack of about 2^{137} for its curve, which depends only on the base field and extension degree and so transfers [RR24], but this curve’s group order, twist, embedding degree and discriminant are not analysed here. OpenVM avoids the question by carrying memory across segments as Merkle roots over touched addresses, checked for adjacency during aggregation, which rests on collision resistance at the price of in-circuit Merkle paths [Ope26a]. The digest is also the one component of the argument that is not post-quantum; §8.8 treats quantum adversaries, including when fast proving limits such an attack (Remark 8.7).

8.6 Trace uniqueness

We first link memory accesses across shard boundaries, then derive trace uniqueness within each shard.

Lemma 8.5 (Cross-shard memory). *Let $(\mathbf{T}_i, \mathbf{pv}_i)_{i < N}$ satisfy $\mathcal{R}_{\text{arith}}$ with at most N_{int} interactions in all, and suppose no reachable vanishing combination of Assumption 8.4 occurs. Assume further that each shard has at most one real boundary row per word, and that such a row supplies the previous tuple of the first actual access to that word in the shard and consumes the tuple of its last actual access. Consider a shard and a word it touches. If an earlier shard touched the word, the current shard’s initial boundary tuple equals the final tuple of the most recent earlier shard that touched it. If no earlier shard touched the word, the initial tuple comes from the program image for an image word and from the memory-initialisation table for any other word. The image tuples are bound by the digest in $\mathbf{vk}$, which the first shard’s leaf adds (§5.4). The per-shard chains of Lemma 5.4 therefore join into one chain per word, ordered by (shard, clk).*

Proof sketch. By Lemma 8.3 and the hypothesis, $\sum_i D_i = O$, including the image digest added by the first shard’s leaf, makes the sent and received multisets of memory tuples equal. Fix an address a . Let I_a count its memory-initialisation rows, P_a its entries in the fixed program image, and F_a its memory-finalisation rows. Every boundary row contributes one tuple to each side at the same address. Counting multiplicities at a therefore gives

$$I_a + P_a = F_a \leq 1.$$

The inequality follows from the strictly increasing finalisation addresses and the public cursors that chain their tables across shards. Thus the combined initial multiset has at most one tuple per address. In particular, if $P_a = 1$, then $I_a = 0$: disjointness of image entries and initialisation rows follows from tuple balance. Their initial timestamp is $(0, 0)$.

Under the boundary-row premise, Lemma 5.4 makes each shard’s accesses to a one path from its boundary row’s initial tuple to its final tuple. Every actual access strictly increases $(\text{shard}, \text{clk})$, so each nonempty path does too. The boundary paths form a finite directed graph with balanced internal tuple multiplicities, at most one initial tuple and at most one finalisation tuple. Strictly increasing timestamps exclude cycles; the unit initial multiplicity and balance then force one path. Each shard contributes at most one segment of that path, so consecutive segments link the last access of one touching shard to the first access of the next. This is offline memory checking [BEG⁺94] across shards. $\square$

The boundary-row premise is not proved by this argument or by the extracted determinism theorems. The honest executor emits one boundary row per touched word, but the current `MemoryLocal` AIR does not enforce uniqueness or a nonempty access segment: a row with identical initial and final tuples contributes cancelling interactions to both the local memory bus and the cross-shard digest. Establishing that every satisfying trace has the required boundary rows remains an open obligation of item 5 in §8.1.

Theorem 8.6 (From chip determinism to trace uniqueness). *Fix a program image, public values and a valid shard with the well-formed boundary rows assumed by Lemma 8.5. If every chip of the shard is deterministic in the sense of Definition 2.7, then the rows reachable from the entry state along the state and memory buses are determined by the program image, the public values, the initial values of the words the shard touches (the hint words of the memory-initialisation table and, in a shard after the first, the initial tuples of its boundary rows), and the declared free values of Definition 2.7 (Table 7): any two valid shards that agree on these agree on those rows up to permutation.*

Proof. By Lemma 5.2 the instruction rows of either shard form one chain from the entry state; write $r_1, \dots, r_N$ for the chain of the first of the two shards compared, and let a step i be r_i together with the precompile rows it invokes, which access memory at r_i ’s clock. We show by induction on i that the rows of step i have the same inputs and outputs in both shards. The inputs of r_i are its received state tuple s_{i-1} , its fetched instruction, and the previous values of the addresses it accesses. s_0 is public and s_{i-1} is the output of r_{i-1} by induction. The fetched instruction is a function of $\text{pc}(s_{i-1})$ because the program bus is received only by the preprocessed program table, which is committed. The previous value of an address is, by Lemma 5.4, the value written by the last earlier access to that address, or the word’s initial value, which is fixed: an image word is bound to the program image through the image digest of vk , and a hint word or boundary tuple is one of the initial values the statement fixes. Earlier accesses belong to steps $j < i$, whose outputs agree by induction. So the inputs of r_i agree, and determinism of its chip gives that its outputs agree; a precompile row of the step then receives its arguments from r_i ’s syscall tuple and its previous values from earlier accesses, so the same argument applies to it, in the order of its accesses.

Rows of non-instruction chips (the memory-initialisation and finalisation tables, the shadow reads, the digest rows, the tables) are determined in the same way from the bus tuples they receive, which are outputs of instruction rows or public. Rows that participate in no bus reachable from the entry state are not covered by this argument; the arithmetisation admits none, because every real row carries a frame or a memory access, but that is a property of the chip set rather than of the buses, and we do not prove it here. $\square$

None of the declared free values makes the execution ambiguous. The address and value of a memory-initialisation row are the initial value of a touched word, already in the list; the length of a hint read is part of the untrusted hint, as the hint words are; and the lift offset of a digest row changes only the curve point that row adds to the digest, which no instruction row reads.

For a fixed sequence of shard public values, applied to the shards in index order, the theorem extends to the whole execution: by Lemma 8.5 the initial tuples of a shard’s boundary rows are final tuples of earlier shards, which the induction over shards has already fixed.

The theorem is stated, not machine-checked; chip determinism, its main premise, is machine-checked for every extracted chip (§8.7.2), and the premises still open are item 5 of §8.1.

8.7 From constraints to MIPS32: conformance and determinism

The preceding subsections show, under the hypotheses of Theorem 8.2, that an accepting proof implies a trace satisfying every constraint, and Theorem 8.6 makes that trace unique given chip determinism. This subsection argues that such a trace is the MIPS32 execution, by checking two properties: agreement with

the ISA on the tested executions, supported by conformance evidence, and uniqueness of chip outputs given their inputs, proved in Lean 4. The constraints must compute the instruction set’s function: a chip that computes the wrong one is sound for the wrong machine (§8.7.1). And they must admit only that function: a chip with two outputs for one input lets a cheating prover choose, which honest traces never expose (§8.7.2). Conformance suites carried through proving and an independent executable model provide the evidence; the proof is in Lean 4, chip by chip.

Correctness requirements. Soundness rests only on the verifier and what it reads: every chip’s constraint description, the preprocessed tables and program commitment in the verifying key, the recursion programs, and the allowlist root. The executor, the just-in-time compiler, trace generation and the GPU kernels bear only on completeness: an error there yields a proof that fails to verify, not one that verifies falsely. For the chips, the determinism theorems show at most one behaviour per input, where an omission would be silent, and the conformance vectors are the evidence that this behaviour is the ISA’s. They cover the core chips; the recursion programs are a hypothesis of Theorem 8.2.

8.7.1 ISA conformance

Vectors from the specification. The encoding tables of the MIPS32 manual were transcribed into a machine-readable table of 77 instructions, reproduced with the chip proving each in Appendix B. For each, a generator emits ten programs with pseudo-random operands, boundary values and the delay-slot placements the instruction admits, assembled with `llvm-mc`. The reference is the Unicorn/QEMU MIPS32 emulator [QV15], which shares no code with ZIREN’s; each vector compares the general registers, HI/LO and touched memory. All 770 vectors agree, on both the interpreter and the JIT.

Shared instruction suites. Fifteen hand-written suites, 144 cases, cover instructions with difficult semantics: signed and unsigned division, multiply-accumulate, bit-field and byte-order instructions, rotates, and leading-one and leading-zero counts. One crate supplies case data to both emulator and prover tests, as it does for the generated vectors, so a regression in either appears as a disagreement.

The Cannon suite. Optimism’s Cannon ships hand-written MIPS test programs [Opt23] from the fault-proof project whose semantics a hybrid fault and validity proof must match, exercising the machine against another team’s reading of the manual. A harness byte-swaps them to little-endian and adapts the halt convention; all 49 programs applicable to a little-endian guest pass.

From vectors to proofs. 224 of the generated programs and 48 of the 49 Cannon programs were also proved and verified end to end. The exception halts with a non-zero exit code, which the emulator accepts and the arithmetisation does not admit. Elsewhere the arithmetisation accepts every behaviour the reference exhibits. The next subsection gives part of the converse: the arithmetisation admits at most one behaviour per input; that this behaviour is the ISA’s rests on the vectors above.

Executable formal model. A Lean 4 model of the ISA was written independently of ZIREN’s emulator: a decoder transcribed from the encoding tables, an interpreter with the delay-slot program-counter pair and little-endian byte memory, and an ALU. Two checks, both by `native_decide`, connect it to the implementation: its decoder agrees with the emulator’s on all 2 275 distinct instruction words in the vectors, and it reproduces the reference results on all 770 vectors. Digests of the emitted event records also agree between repeated runs and between interpreter and JIT, which is evidence of reproducibility rather than a proof of determinism.

8.7.2 Determinism of the chips

Statement. The statement for a chip U is Definition 2.7 for its constraint system C_U , with byte-table lookups lowered to range constraints or to an abstract table relation T . A theorem that uses such a relation takes $\forall i, o, o'. T(i, o) \wedge T(i, o') \Rightarrow o = o'$ as a hypothesis; 32 of the 104 do, for the AND, XOR, OR, NOR and shift-carry operations of the `Byte` table, which is a function because the verifying key fixes it. Each theorem

also takes the equality of a list of values assumed deterministic, which is empty for every chip. The inputs I_U include the declared free values, which Table 7 records. These theorems are the premises of Theorem 8.6.

Extraction. The Rust constraint description that the prover evaluates and the verifier checks is evaluated once with symbolic columns; the resulting constraints, range facts and bus interactions are lowered to a first-order statement, and a generator emits one Lean 4 file per chip with a row record, the constraints, the input and output projections, and the theorem. Selector-specialised chips get one theorem per opcode selector, all restricted to real rows. The extraction covers every core chip of Appendix A except the three preprocessed tables, which carry no statement, and the two SHA-256 control chips, which it currently models as opaque helpers: 104 theorems over 57 chips. The symbolic extraction and derivation replay use a shared constraint interface. Extending coverage to those two chips or to the recursion machine requires modelling their helpers and supplying the corresponding gadget proofs within that pipeline.

Proof automation. As in Picus [PCW⁺23], a propagation analysis over the extracted constraints derives every output column of w' from the inputs as a sequence of steps: linear and inverse solving, bit and one-hot decoding, integer bounds, carry roots, and *gadget summaries* for recurring sub-circuits (a field operation with its carry rows, a byte comparison, a leading-one detector, a canonical word), with case splits on selector bits both witnesses share. The generator replays the derivation in Lean 4. Each step becomes a lemma stated over only the columns and conjuncts it uses, closed by a small tactic that lifts its statement from $\mathbb{F}$ to the integers with each column’s range and decides it with `omega`, `linear_combination` or `grind`. Not everything is generated: the gadget library is written by hand, the accelerator, division and `Global` gadget proofs come from per-family generator scripts, and the selector-flag postconditions of ten chips are proved by hand where the automation runs out of budget.

Large chips. The gadget summaries are library theorems proved once; an accelerator whose outputs come from chained field operations or permutation rounds gets a generated proof that its output columns agree (`gadget_det`), split into a chain of segment lemmas of a few hundred steps, each taking the facts live at its start and ending by applying the next. The chip’s theorem never splits w and w' into their thousands of columns: it applies the step lemmas to projections of the two constraint sets, each step lemma restating its conjuncts in their original form and substituting what earlier steps proved inside its own small context. A theorem elaborated as one declaration in the full context of a precompile chip took hours; in this form every module of the check takes minutes.

Findings. A determinism statement is false when a row admits two satisfying witnesses with the same inputs and different outputs. For each statement the automation left open, we inspected the constraints for such a counterexample. Table 7 lists what that reading found. Three are soundness gaps in the constraints, two exploitable and one (`Global`) latent while every sender sets a flag, each closed by new constraints; the others are prover freedom the protocol allows by design, which the statements now name as inputs rather than hide.

Coverage. All 104 theorems are closed: 23 for the integer ALU chips, 8 for the shifts, 9 for control flow, 14 for the memory instructions, 5 for the memory argument, 15 for the miscellaneous and syscall chips, and 30 for the accelerators, one per chip or per opcode-selector specialisation of it. For each, `#print axioms` lists only `propxt`, `Classical.choice` and `Quot.sound`, so no proof rests on `sorry` or an added axiom. The current check covers 9 854 modules in the 57 chip files with statements, split by dependency and built in parallel on the evaluation host of §9.1, with a 30-minute cap per module. Checker receipts and build-identity records bind the checked source hashes, theorem inventories and Lean library inputs. The matching archive² contains the checked Lean sources, a fresh extraction from revision `cf0b3742`, per-module diagnostics, fresh public-theorem axiom audits and scripts for rebuilding every proof. Its validator compares the constraints, gadget relations, input and output projections and public theorem types with the fresh extraction; proof-generation hints may differ without changing those statements. The external snippet generators and recipes are included and

²<https://zk-toolchain.s3.amazonaws.com/fv/ziren-fv-lean-20261014.tar.gz>.

pinned by the archive manifest. Changes to the AIR require regeneration and a new check of the affected proofs.

8.7.3 A negative control for the height claims

The verifying key depends on the chip set (§3.3) and the committed geometry is quantised from declared heights (§6.6), so the argument must reject a shard whose declared shape is false. A harness proves real shard proofs for an arithmetic and a hash-heavy guest and tampers with them in eighteen ways: over- and under-claimed heights, false chip activity, and tampered column and row counts, each injected into the degree mask, the Fiat–Shamir prologue, or both. Every case is rejected: height and activity lies in the degree mask by the **LogUp-GKR** last-layer reconstruction, lies confined to the transcript by the grinding witness (the prologue observes the declared heights), a tampered column count by the jagged verifier, and a tampered row count by the preprocessed round, which reads it from the verifying key.

8.8 Post-quantum security

Every component of the shard and recursion arguments except the elliptic-curve multiset hash is hash-based: the Merkle commitments under H , the **WHIR** (or **BaseFold**) proximity test, the sumchecks and **LogUp-GKR**. The Fiat–Shamir compilation of such interactive oracle proofs remains sound against quantum adversaries in the quantum random-oracle model [CMS19], with weaker concrete bounds: a quantum collision search on the 248-bit digest of the measured instantiation costs about 2^{83} rather than 2^{124} evaluations [BHT98], so the levels of §8.2 are classical. The digest is the exception. Shor’s algorithm computes discrete logarithms in $E(\mathbb{F}_{p^r})$ [Sho97], which reduces Assumption 8.4 to finding a short integer relation among known scalars, and at the interaction counts of a block such a relation should be assumed easy to find. Against a quantum adversary the options of §5.5 therefore differ: the global challenge and Merkle roots rest on hashes, and a lattice multiset hash on a problem not known to be easy for quantum adversaries, while the elliptic-curve multiset hash does not survive. Table 3 gives their costs; this paper claims post-quantum security for no configuration.

Of the two hash-based options, Merkle roots perform best: on one GPU, where the card is the bottleneck, their smaller committed area makes them faster; on four GPUs their proving work is smaller but arrives in execution order, and the cards idle while the last leaves and compositions of a block drain, a tail the elliptic-curve multiset hash fills with its independent precompile and memory shards.

Remark 8.7 (Real-time quantum adversaries). Proving a block in time close to linear in its trace does not by itself protect the elliptic-curve digest. The lift applies no salt: a point is a fixed function of its message, so a quantum adversary can compute a vanishing combination once, offline, and reuse it in a false proof for any later block, and the speed of honest proving does not bound that attack.

A time-bounded claim holds under two further conditions, neither of which the configuration of this paper meets. (i) *Salting*: the lift mixes a per-block value σ that is unknown before the block is fixed, such as the block hash or a challenge drawn after the shards’ trace commitments, constrained in the chip and checked by the verifier, so that a relation among one block’s points is useless for another. (ii) *Timeliness*: a proof for block b has value only if it is delivered within a window ω after b appears, as in real-time proving where a late proof is not consumed. Under (i) and (ii) an adversary must find its relation for block b ’s salted points within ω , while the honest proof, and its verification, completes in about the honest proving time; if solving the discrete logarithms such a relation needs takes longer than ω , the forgery arrives after the proof has lost its value, and fast proving lets ω shrink toward the honest proving time, 10 to 31 s per block on four GPUs for the twelve blocks of Table 3. This is a threat-model argument, not a theorem. It needs (i), which changes the digest and the verifying key; a lower bound on quantum discrete-logarithm time, which we do not have; and a deployment rule that enforces ω .

It gives no protection after ω , and none for proofs that keep their value later, such as archived or bridged proofs.

Table 7: What the determinism statements exposed. Each row is a chip whose statement, as first extracted, did not hold; the last column is how the constraints or the statement were changed. The three new constraint sets change the verifying key.

Chip	What the statement exposed	Consequence	Resolution
*DoubleAssign (secp256k1, secp256r1, BN254, BLS12-381)	The slope gadget proves $s \cdot 2y = 3x^2 + a$ and nothing about $2y$. For an input with $y \equiv 0$, such as $(0, 0)$ on a curve with $a = 0$, every slope satisfies the chip, and the row writes $(s^2, -s^3)$.	Soundness gap: the written point is the prover's choice. An AIR-level test forges it for three slopes.	An inverse check proves $\text{is_real}/2y$ exists, and the executor refuses $y \equiv 0$.
Global (flags)	is_send and is_receive choose the sign of the lifted point's y and were neither boolean nor tied to is_real : with both zero, (x, y) and $(x, -y)$ both satisfy the row and give different digests.	Soundness rests on the senders: every current sender sets exactly one flag, so a $(0, 0)$ row cannot balance the bus today. SP1 v6 constrains the flags the same way.	Both flags boolean and is_send + is_receive = is_real .
KeccakSponge- Control	Of a multi-block sponge, only the first block's input and output addresses were bound to the syscall; later blocks' addresses were free, and nothing tied the number of blocks to the input length.	Soundness gap: a prover can absorb later blocks from any address, stop early or late, and write the digest anywhere.	Each block carries the input address, output address and length on the block bus; the first block is block 0 and binds the length read from memory, its top byte range-checked; the final block has $(b + 1) \cdot 36 = \text{len}$ (36 words, the chip's 1152-bit rate).
Global	The lift_x offset is a byte the prover picks among those that lift the message to a curve point.	None: the encoding stays injective because the top message field is below 2^{16} ; SP1 is the same.	The offset is an input of the statement.
SyscallInstrs	The length of a hint read is chosen by the host.	None: hints are untrusted input by definition.	The hint length is an input of the statement.
MemoryGlobalInit	The address and value of an initial-memory row are the prover's choice.	None: the global memory argument fixes them against the image and the first access.	Address and value are inputs of the statement.

9 Evaluation

We evaluate three quantities: *instruction efficiency*, the committed area and bus interactions an executed instruction costs, which is a property of the arithmetisation (§9.2); *proving efficiency*, the rate at which an implementation turns that area into a proof on named hardware (§9.3); and *proof size*, the bytes of the compressed proof a verifier reads, with the parameters that control it (§9.4). One primary workload, one GPU model and one host are used, with interleaved repetitions, so that differences between configurations are attributable. No cross-system comparison is attempted: other systems’ figures are self-reported on different guests and mostly on clusters, and nothing here is a same-hardware baseline.

9.1 Experimental setup

Workload. Ethereum mainnet block 25 907 955, executed by a MIPS32 build of the Reth execution client with an arena-represented witness trie; the guest hashes trie nodes, recomputes a Merkle root over state it does not hold and verifies signatures. A word-wise `memcmp`, a zero-copy Keccak wrapper and the arena trie reduce it from 379 to 288 million cycles. The single-GPU timing of *Memory traffic and shard size* (§9.3) uses that cached record; the multi-GPU measurements of Table 10 use three blocks of Table 11 with the Reth guest, 26 138 428, 26 138 416 and 26 138 421 (206, 348 and 508 M cycles), and the instruction-efficiency census of §9.2 uses block 25 955 640 (495.6 M cycles) in the analysed configuration.

Hardware. One NVIDIA RTX 5090 (32 GB) in a virtual machine with eight such GPUs, 124 vCPUs of AMD EPYC 9355 and 925 GB of memory; the runs of Table 10 use one, two and four, and those of Table 11 one and all eight (driver 570.153.02, CUDA 12.8, Ubuntu 22.04). The GPU server for Tables 10 and 11 runs in a CUDA 12.8.1 container based on Ubuntu 24.04. The GPU backend is a separate CUDA implementation of the same protocol and is not open source.

Configurations. Sections 5–8 describe the *analysed configuration*. Most timings below, and the proof-size comparison of §9.3, were taken on an earlier *measured configuration*, which differs in five respects: the Poseidon2 permutation used 13 partial rounds rather than the corrected 20; the lookup challenge was not ground; the syscall bus carried the sum of two 16-bit half-words rather than the half-words themselves (§5.1); the set of three soundness fixes of Table 7 was not applied; and the system-call tables did not enumerate the numbers they accept (Appendix B). Each of the last two adds constraints. Comparisons between measured-configuration runs are therefore internal to that configuration, and the soundness figures of §8.2 are for the analysed one. Unless a caption or the text says otherwise, every timing below is of the measured configuration and every area of the analysed one; the exceptions are the timings of Tables 10 and 11 and the compressed-proof size of §9.4, which are of the analysed configuration.

Metrics and protocol. *Trace area* (§2.4) is taken from the emulator’s shard-close census; Table 9 takes its kernel shares from the measured configuration. *Proving time*, used by Tables 10 and 11, runs from the prove request at the prover to the compressed proof, over shard proving and recursion, and excludes the client’s execution and verification; the measured-configuration timings are *wall-clock*, from reading the cached input to writing the compressed proof, and include 2–3 s of host verification. *Proving throughput* is guest cycles divided by the time, reported in MHz (10^6 guest instructions/s). *Proof size* is the serialised compressed proof. A ten-shard `nsys` sample attributes kernel time. Table 10 reports the mean of three consecutive warm proofs per configuration, warm meaning after a first proof has loaded the keys and compiled the programs. Every proof contributing to Tables 10 and 11 is checked by a native verifier. No other proving ran on the host during the runs of Tables 10 and 11. The per-run timings and build and runtime metadata of the 11 October 2026 rerun are archived separately.

9.2 Instruction efficiency

We first estimate what one executed instruction costs, from the chip widths and interaction counts of §§5.9 and 5.10, in committed cells and bus interactions (§1.3). Table 8 counts an instruction’s own row only; the

Table 8: Committed cells and bus interactions per executed instruction, for the most frequent chips. The frame includes the program-counter pair; interactions count the program fetch, the state bus, the memory bus and table lookups.

Chip	cells			interactions		
	frame	own	row	frame	own	row
AddSubImm	29	4	33	16	0	16
AddSub	32	4	36	20	0	20
Bitwise	32	5	37	20	4	24
Lt	32	18	50	20	3	23
ShiftLeftImm	26	19	45	16	4	20
ShiftRight	32	57	89	20	25	45
Branch	29	27	56	16	≤ 9	≤ 25
LoadWord	29	19	48	16	9	25
StoreWord	29	23	52	16	9	25
LoadNarrow	29	30	59	16	10	26
Mul	32	42	74	20	22	42
DivRem	32	131	163	20	44	64

memory-argument rows for each shard’s first and last access to a word, and the tables’ multiplicity columns, are paid per shard and are measured below.

Per instruction. The frame is most of every common row: an addition or bitwise operation pays 29–32 cells and 16–20 interactions of frame for at most five cells and four interactions of its own, so the per-instruction floor is set by the register memory check. Expensive instructions are expensive in columns, not rows: a division is one row of 163 cells where Jolt’s sequence takes eight steps [AST24]. Per-opcode chips are what keep the common rows narrow: one table carrying every opcode’s columns would pay the widest row on every instruction, over 400 cells (`MiscInstrs` alone has 432 columns) where the common instructions pay 33 to 37; SP1 makes the same choice [Suc24].

Per block. Block 25 955 640, 495.6 million cycles, commits 29.2 G cells and raises 12.8 G (row, interaction) pairs, 58.9 cells and 25.8 pairs per executed instruction (Table 9, Figure 7a). This is larger than the single-row costs of Table 8 mainly because of the per-shard memory-argument rows: the `Global` chip holds 15% of the area, 8.8 cells per instruction, because every word a shard touches costs two rows of it, which is the cost of sharding. The figure depends on the workload, not only on the instruction set. A precompile call is one guest cycle and many rows of an accelerator chip, and since a shard closes at the area budget (§4.2), the precompile share sets both the cells per cycle and the shard count: two Ethereum mainnet blocks of 949 and 943 million cycles proved in production as 702 and 213 shards. These are cells of a 31-bit field and do not convert into Jolt’s bit-weighted count.

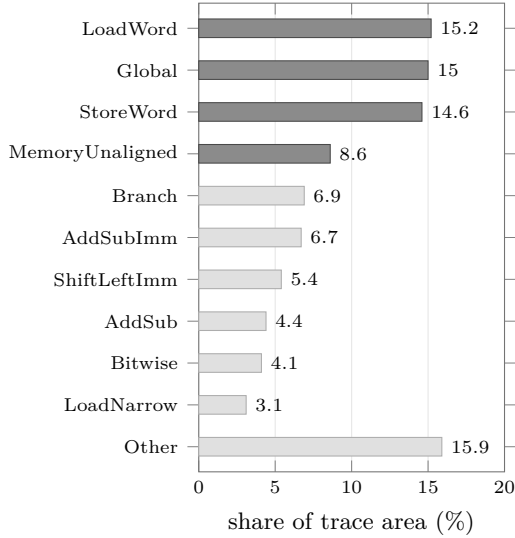
Effect of reducing area. The census identifies two cost measures to reduce: trace cells for the commitment and constraint argument, and (row, interaction) pairs for the bus argument. Two changes follow from it, both mechanisms rather than tuning: replacing a boolean sign decomposition in the cross-shard digest chip with three byte lookups, and dropping byte checks on memory-instruction accesses that another chip already enforces. Both change the constraint system, and with it the chip keys and the recursion-key allowlist; neither changes the statement proved.

9.3 Proving efficiency

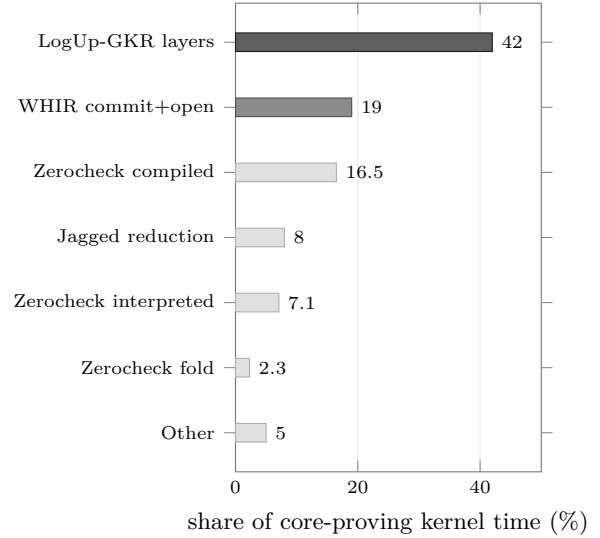
Time breakdown. Kernels account for about 340 ms of the 500 ms of GPU time per shard in the measured configuration; launch gaps and host synchronisation account for the remaining time (Table 9, Figure 7b). The lookup argument is the largest consumer, and its cost scales with (row, interaction) pairs rather than with columns: memory-instruction chips carry 25–26 interactions per row and 47% of all pairs, while the `Global`

Table 9: GPU kernel time by kernel group, and trace area by chip. Kernel shares are from a ten-shard **nsys** sample of the measured configuration; area is from the analysed configuration’s census of block 25 955 640 (29.2 G cells).

Kernel group	share	Chip	area share
LogUp-GKR layers (fold-and-sum, transitions, first layer)	42%	LoadWord (92.6 M rows $\times$ 48)	15.2%
Zerocheck, compiled constraint kernels	16.5%	Global (67.2 M $\times$ 65)	15.0%
Zerocheck, interpreted (wide precompile chips)	7.1%	StoreWord (81.8 M $\times$ 52)	14.6%
Zerocheck fold	2.3%	MemoryUnaligned (42.7 M $\times$ 59)	8.6%
WHIR commit and open (Merkle absorb, DFT)	19%	Branch (36.1 M $\times$ 56)	6.9%
Jagged reduction	8%	AddSubImm (59.4 M $\times$ 33)	6.7%
Other	5%	ShiftLeftImm (34.8 M $\times$ 45)	5.4%



(a) Trace area by chip; **Global** and the three largest memory-instruction chips hold 53%.



(b) Kernel time by kernel group.

Figure 7: Trace area by chip in the analysed configuration and kernel time by group in the measured configuration, for block 25 955 640 (Table 9).

chip holds 15% of the area but 3.7% of the pairs. The cost model is therefore area for the commitment and zerocheck and pairs for the lookup, with one (row, interaction) pair weighted as about two cells in this profile.

Memory traffic and shard size. A byte-level profile of the **LogUp-GKR** circuit showed that its first layer paired fractions across the most-significant index bit, so short chips were carried through every layer of the tallest and the traffic summed over layers was $5.2\times$ that of the first. The prover therefore pairs on the least-significant bit, so each chip folds out after $\log_2$ of its own height, and the memory this frees admits shards of twice the size. The two changes are enabled together: the old-pairing, doubled-shard configuration does not fit in memory, so their effects are not separated. With both, the cached 288 M-cycle block 25 907 955 proves in 48.7s on one GPU (5.9 MHz), against 67.8s for the revision without either at the largest shard size it could run (-28%), with the compressed proof unchanged in size.

Scaling across devices. Table 10 and Figure 8 report proving time and throughput on one to four GPUs for three blocks of Table 11 proved with the **Reth** guest, of 206, 348 and 508 M cycles. Two GPUs prove each block $1.9\text{--}2.0\times$ faster than one and four $3.3\text{--}3.5\times$, reaching 16.5–23.6 MHz. The gap to linear scaling is the serial recursion tail and the start-up interval before the emulator has produced a shard for every GPU; both are fixed per block, so the larger the block, the closer the scaling is to linear.

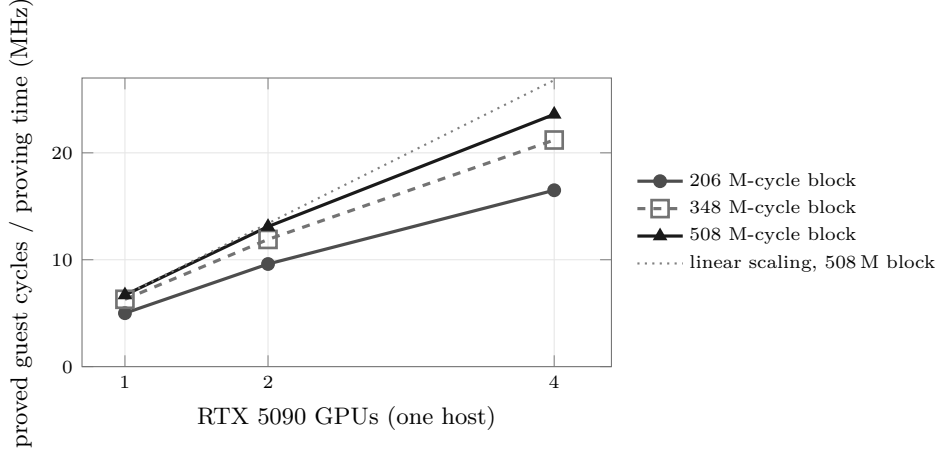


Figure 8: Throughput on one to four GPUs for three blocks proved with the Reth guest. The dotted line is linear scaling from one GPU on the 508 M-cycle block.

Table 10: Proving time and throughput on one, two and four GPUs for three Ethereum mainnet blocks of Table 11 proved with the Reth guest, analysed configuration; mean of three consecutive warm proofs.

Block (guest cycles)	proving time (s)			throughput (MHz)		
	1 GPU	2	4	1 GPU	2	4
26 138 428 (206 M)	41.4	21.5	12.5	5.0	9.6	16.5
26 138 416 (348 M)	55.1	29.2	16.4	6.3	11.9	21.2
26 138 421 (508 M)	75.8	38.9	21.5	6.7	13.1	23.6

Execution clients. ZIREN proves Ethereum blocks with more than one execution client compiled to MIPS32 from different languages: Reth, written in Rust, and the block keeper of Geth, written in Go, whose runtime issues the no-op Linux calls of Appendix B while starting. Both guests call the precompiles for Keccak, SHA-256 and the elliptic curves, and the Geth guest was tuned for 32-bit MIPS in its BN254 backend, hashing, jump-destination analysis, signature recovery and trie handling, which removes 43% of its cycles on these blocks. Table 11 shows that a block costs in proportion to its shards rather than its cycles: a shard takes about 0.47s on one GPU under either guest, and the Reth guest, which also runs the EVM’s 256-bit arithmetic on a precompile, closes its shards after 2.3 million cycles on average against 5.1 million for Geth, so it runs at the lower rate and still proves every block in less time. Its blocks are also smaller, so on eight GPUs the serial tail after the last shard (§3.3) is a larger share of each.

The host. On one core of an AMD EPYC 9355, the just-in-time compiler executes block 25 955 640 (496 M cycles) at 198 MHz once compiled (166 MHz including its single compilation pass), and the interpreter at 40.6 MHz without the event record and 14.6 MHz with it; the coordinator’s whole producer job takes 6.1 s for a 420 M-cycle block, about 69 MHz. On one GPU, emulation does not limit proving throughput. It can limit throughput when one host feeds several GPUs, because each shard is re-emulated by the worker that proves it, at about one second of host time per shard against half a second of GPU time in the measured configuration, so the event record carries only what a column witnesses and the coordinator never holds the whole execution (§3.1).

GPU implementation. Three kinds of engineering act on different costs. *Fused kernels* cut memory traffic: the jagged fold and the round polynomial it feeds are one kernel, as are the layer transition and the round over the mixed round-0 slab. *A cross-stage pipeline* cuts idleness (§3.3). *Occupancy tuning* caps the register budget for the measured card. Field arithmetic rests on the inline-PTX Montgomery primitives of *sppark* [Sup25], which we use and did not write.

Table 11: The same 15 consecutive Ethereum mainnet blocks, 26 138 416 to 26 138 430, proved with two execution clients on the same GPUs, analysed configuration. The initial block 26 138 415 is excluded to avoid startup effects, as are the separately flagged warm-up proofs. Proving time runs from the prove request at the prover to the compressed proof, over shard proving and recursion. For each client, the range row gives the minimum and maximum per-block values. In the total (15) row, guest cycles, shards and proving time are sums across the 15 blocks; throughput is total guest cycles divided by total proving time.

Client	Summary	Guest cycles	Shards	proving time (s)		throughput (MHz)	
				1 GPU	8 GPUs	1 GPU	8 GPUs
Reth (Rust)	range	43–508 M	33–164	16.4–76.6	6.3–14.3	2.6–6.6	6.8–35.5
	total (15)	3.30 G	1 409	661	153	5.0	21.5
Geth (Go)	range	0.25–2.99 G	64–537	33.9–257	10.3–39.4	7.5–12.0	24.6–75.8
	total (15)	21.65 G	4 251	2 020	358	10.7	60.6

Prover cost of the soundness target. The level of §8.2 is paid for in queries and in proof-of-work. We measured the grinds separately, on four GPUs, at a build that grinds the lookup challenge, with the arms interleaved on one cached block. The 22/22/14-bit query, **LogUp-GKR** and batching grinds took 33.4s against 33.9s for 16/16/8, a difference within run-to-run spread, while buying the same bits with queries (133, 95 and 91 queries in the three **WHIR** rounds at a 16-bit grind) cost 8.6%. That holds only because the search runs on the device: with the grinds on the host the same schedule took 166s, 123s of it the **LogUp-GKR** grind.

9.4 Controlling proof size

The shard proofs of a block total on the order of 10^8 bytes before recursion; the published object is the compressed proof at the root of the tree, whose size does not grow with the execution. It is dominated by the openings of the first **WHIR** oracle, about four fifths of it in the instrumented proofs, because each query authenticates one coset row of every stripe; the later **WHIR** rounds, the **LogUp-GKR** layer proofs and the other sumchecks share the rest. To first order the size is

$$t_0 \cdot (\text{stripes} \cdot 2^{k_0} + \text{roots} \cdot \log_2(\text{leaves}) \cdot |\text{digest}|) + \sum_{i \geq 1} t_i \cdot (\text{folded leaf} + \text{path}),$$

with t_i the queries of round i and k_0 the first folding factor. Each term is controlled by a parameter of the schedule (§7), fixed per stage, and each trades against something else at a fixed soundness level:

- *Queries per round.* A query at rate ρ yields $-\log_2 \frac{1+\rho}{2}$ bits under unique decoding and close to $\frac{1}{2} \log_2 \frac{1}{\rho}$ under the Johnson bound, which is also proven but adds proximity-gap terms paid for by the grinding of §7.2. Lowering ρ therefore cuts every round’s queries, most of all under the Johnson-bound analysis; the first round’s rate is bounded by the field, since the stacking height times $1/\rho$ must not exceed the two-adic subgroup (2^{24} for **KoalaBear**), and the prover’s encoding work and memory grow as $1/\rho$.
- *Grinding.* Each bit of query grinding removes about one bit’s worth of queries, at a doubling of the search, which on the device is cheap (§9.3); the size of a grinding witness, one field element, bounds how far this goes.
- *Leaf width.* A smaller first folding factor k_0 narrows every first-round leaf and moves the variables to later rounds, whose queries are fewer.
- *Stripes.* The number of stripes a query opens follows the committed area of the stage, which for a compress node is set by the recursion verifier program rather than by the guest.

The parameters that make the proof small need to apply only to the published proof: the core shards and every recursion node below the root keep the schedule tuned for prover time, since each of those proofs is consumed by its parent and never leaves the prover, and the root alone proves under the compress schedule of Table 5. The root is a compose of arity one over the node that covers the execution (§3.3), so its verifier program, and with it the committed area that sets the number of stripes, is the same small one for every tree,

and the root commits under a fixed class of 32 stripes per round. The two schedules are two instances of one ring type, so the field, the hash and the transcript are shared and only the schedule differs; a verifier takes the schedule from the stage it expects, never from the proof.

Taken together, the Johnson-bound schedule at a first rate of 2^{-3} with $k_0 = 2$ opens $58 \cdot 2^2$ values per stripe in the first oracle against $124 \cdot 2^3$ for the unique-decoding schedule at rate 2^{-2} with $k_0 = 3$, under a quarter, with both stage union bounds above 100 bits (Table 6), for 27 bits of grinding on the folding and batching challenges, paid once per proof at the root, which is the prover-time cost of the smaller proof. Measured on 14 October 2026 at revision `cf0b3742` with the Fibonacci guest ($n = 500$), the compressed proof is 290 643 bytes (284 KiB). Both the prover’s in-circuit child-key allowlist checks and the native verifier’s root-key check are enabled, using the regenerated key map of that revision. Native verification takes 25.2 ms (median of five warm runs, 24.8–25.3 ms) on the host of §9.1, excluding setup, deserialisation and one warm-up verification. The serialised object is the compressed proof together with its recursion key; the SDK container adds 38 bytes for the variant, public outputs and version. For this fixed root program and commitment shape, proof size is independent of the guest’s execution length.

9.5 Threats to validity

Internal. The spread of the interleaved repetitions is small relative to the effects reported, though another process shared the host (§9.1). *Construct.* The timings of Table 9 and of the ablations predate the Poseidon2 correction, which adds seven partial rounds to commitments, transcripts and recursion and changes absolute performance by an unmeasured amount; they are evidence about mechanisms to the extent the correction leaves relative costs unchanged. The timings of Tables 10 and 11 and the compressed-proof size postdate it (§9.1). Trace area estimates work from committed cell counts; the kernel profile measures elapsed device time. The pipelining of §3.3 is not ablated: no run defers recursion until every shard is proved, so its effect on wall-clock time is not separated from the rest. *External.* The primary workload is one Ethereum block, and other guests have different instruction mixes. The per-run timings for Tables 10 and 11 are archived separately; the earlier raw per-run timings are not retained, and the retained area census is for the analysed configuration rather than the measured one.

10 Conclusion

We have presented ZIREN, a succinct argument for MIPS32 execution. Its architecture is one proving stack whose compressed proof a native verifier checks, with SNARK wrapping and garbled verification as routes to EVM-compatible rollups, private computation and Bitcoin rollups (Section 3). One row of one chip per executed instruction, carried by an instruction frame, replaces the CPU table; one lookup argument, one zerocheck and one jagged WHIR opening prove each shard; and a recursion tree with an enumerated key allowlist composes the shards into one proof. Conformance suites and an executable Lean 4 model test the emulator. Determinism is extracted from the constraint description the prover evaluates; all 104 extracted theorems are closed in Lean 4, and the statements exposed three soundness gaps that the constraints now close.

The cost model expresses instruction efficiency in trace cells and bus interactions per guest instruction, separately from hardware-dependent proving throughput (§1.3). It accounts for instruction-frame overhead, shard-boundary memory rows and precompile expansion, which explain why guest-instruction counts alone do not measure prover work.

The performance study identifies trace area, lookup interactions, layer order and shard size as distinct factors in proving cost. The two largest costs it identifies are the lookup argument, 42% of kernel time, and the cross-shard digest, 15% of the block’s area, which a digest of per-shard deltas or a larger shard would reduce. The open proof obligations are the five hypotheses of §8.1; future work should measure the corrected Poseidon2 schedule against one configuration, soundness report, key set and proof.

Table 12: Columns of the R-type instruction frame. The I-type frame replaces the `op_c` index and access (7 columns) by a word immediate (4), 27 columns in all; the shift-amount frame replaces them by a scalar immediate (1) and drops the access, 24 columns. The program-counter pair is owned by the chip and passed to the frame.

Element	Purpose	Columns
<code>shard</code>	shard index, the high part of every timestamp; 16-bit range check	1
<code>clk limbs</code>	the instruction’s clock as a 16-bit and a 10-bit limb, both range-checked	2
<code>opcode, op_a, op_b, op_c</code>	the decoded instruction, sent on the program bus with <code>pc</code>	4
<code>op_a_0</code>	set when the destination is register 0, which pins the written word to zero	1
<code>op_a</code> access	previous word, written word, previous timestamp and one limb of the timestamp difference	10
<code>op_b</code> access	word read, previous timestamp, one difference limb	6
<code>op_c</code> access	word read, previous timestamp, one difference limb	6
Frame		30
<code>pc, next_pc</code>	the counter pair received on the state bus; owned by the chip	2

Acknowledgements

The Plonky3 [Pol24] toolkit supplies the field, the Poseidon2 permutation [GKS23], the constraint interfaces and the transforms this implementation is written against. We thank the developers of SP1 [Suc24, Suc25a] and OpenVM [Ope26a] for publishing their systems openly; the components of theirs that this paper uses are cited where they are used.

A The frame and the chips in detail

This appendix itemises the columns and interactions of the instruction frame (Table 12) and the width of every chip (Table 13).

A.1 The instruction frame

The R-type frame raises 20 interactions per row: one program fetch; a receive of (`shard`, `clk`, `pc`, `next_pc`) and a send of (`shard`, `clk + 5 +  $\delta$` , `next_pc`, `next_next_pc`) on the state bus, where δ is zero except for a system call; a send of the previous and a receive of the new tuple on the memory bus for each of its three accesses; and eleven table lookups, one 16-bit check of the shard, two for the clock limbs, two per access for the timestamp difference, and two for the four bytes of the written word, which the byte table checks in pairs. The I-type and shift-amount frames have one access fewer, and raise 16.

A.2 The chips

Table 13 lists all 62 chips with their widths in the analysed configuration. `Global` differs most from the measured configuration: it has 65 columns here, against 100 in the measured baseline and 58 after the change of §9.2, because the analysed configuration keeps seven chord-witness columns that change removed.

Table 13: The 62 chips of the core machine with their main-trace widths (columns over $\mathbb{F}$), in the analysed configuration; a second figure is a preprocessed width. The immediate (Imm) variants take the second operand from the instruction word. Appendix B gives the instruction-to-chip map.

Family	Chips (width)
Integer ALU	AddSub (36), AddSubImm (33), Bitwise (37), BitwiseImm (33), Mul (74), DivRem (163), Lt (50), LtImm (47), CloClz (93)
Shifts	ShiftLeft (62), ShiftLeftImm (45), ShiftRight (89), ShiftRightImm (83)
Control flow	Branch (56), Jump (57)
Memory instructions	LoadWord (48), LoadNarrow (59), StoreWord (52), StoreNarrow (55), MemoryUnaligned (59)
Misc. / syscall	MiscInstrs (432), MovCond (49), SyscallInstrs (79), SyscallCore (11), SyscallPrecompile (11), SysLinux (135)
Memory argument	MemoryLocal (80), MemoryBump (12), MemoryGlobalInit (152), MemoryGlobalFinalize (152), Global (65)
Tables	Program (1, +14), Byte (10, +12), Range (1, +2)
Hash precompiles	ShaExtend (166) and control (4), ShaCompress (249) and control (69), KeccakSponge (2637) and control (1095), Poseidon2Permute (571)
Curve precompiles	add (1685) and double (1666) for Secp256k1, Secp256r1 and Bn254, decompress (1040) for the first two; Bls12381 add (2533), double (2506), decompress (1613); Ed25519 add (1433), decompress (1166)
Field precompiles	Bls12381Fp (512), Bls12381Fp2AddSub (1014), Bls12381Fp2Mul (1773), Bn254Fp (344), Bn254Fp2AddSub (678), Bn254Fp2Mul (1181), Uint256MulMod (417), U256XU2048Mul (2627)

B The guest instruction set

This appendix lists the MIPS32 instructions a guest may execute, which is the set the relation of §4.4 quantifies over. It is also the table from which the conformance corpus of §8.7.1 is generated. The last column names the chip that arithmetises the instruction, so the table fixes which rows of Appendix A a program can occupy, and with it the chip set that Section 3.3 enumerates over.

Three conventions apply. The four moves to and from LO and HI are additions of an immediate zero (§4.1). SYNC, SYNCI and PREF are accepted and retire as no-ops on AddSubImm, as an addition of zero. SYSCALL is absent from the table because it is the boundary of the machine rather than a computation within it; it is arithmetised by SyscallInstrs and described in §5.10.4. MIPS32r2 user-mode integer instructions outside the table (the conditional traps other than teq, the linking conditional branches bgezal and bltzal, the branch-likely forms, break and rdhwr) are not supported: the decoder maps them to an unimplemented opcode and execution stops with an error.

System calls the machine does not implement. The behaviour is fixed in every case; none is left unconstrained. A call number outside the machine’s table (neither an accelerator call nor a listed Linux number) stops the emulator with an error before any state changes: the relation rejects, and no proof is produced by the honest prover. Seven Linux calls, under eight numbers, have their semantics (brk, read, write, exit_group, mmap/mmap2, clone, fcntl). Seventeen more, the ones a standard runtime issues while starting (open, openat, close, fstat64, clock_gettime, nanosleep, futex, prctl, uname, prlimit64, sigaltstack, rt_sigaction, rt_sigprocmask, madvise, munmap, gettid, sched_getaffinity), are accepted as no-ops: they touch no memory and return 0 in \$v0 with the error flag \$a3 clear. That is deterministic, since the machine’s memory is fully defined, but it is not the ABI’s meaning: a program that reads a time, a file or an affinity mask through one of them gets no value, and must not depend on one. The circuit pins the same outcomes and accepts the same numbers. The SysLinux table decodes each of the eight implemented and seventeen no-op numbers and requires a real row to carry one of them; the instruction table requires a call it does not send to a table to be one of the ten local calls it implements (halt, commit, hint and proof-verification calls). A row carrying any other number has no satisfying assignment, so a proof states only an execution the emulator performs. We checked this by construction: a guest issuing call number

4999 is refused by the emulator, and a prover modified to handle the number as a no-op obtained a proof the unmodified verifier accepted before the tables enumerated their numbers, and obtains none since. The enumeration costs the two chips 11 and 35 columns. The accelerator calls are listed in Appendix A.

Table 14: The 77 guest instructions, their primary encoding fields and the chip that proves each. *Op* is bits 31–26 and *func* bits 5–0; a dash marks a field the format does not use, and the immediate forms are distinguished from the register forms by *Op* alone. Rows that share both fields are told apart by another field: *bgez*, *bltz*, *bal* and *synci* by *rt*, *rotr* and *rotrv* from *srl* and *srlv* by one bit of *rs* or *sa*, and *wsbh*, *seb* and *seh* by *sa*. Semantics are as transcribed from the architecture manual into the specification table that generates the conformance corpus; on rows marked [†] the implementation fixes a choice (§4.3), taking the region bits (PC + 4) [31:28] as zero and letting *sc* always succeed. Bit ranges *x*[*h*:1] include both ends.

Mnemonic	Op	func	Family	Semantics	Chip
<i>add</i>	000000	100000	Integer arithmetic	$rd = rs + rt$	AddSub
<i>addu</i>	000000	100001		$rd = rs + rt$	AddSub
<i>addi</i>	001000	–		$rt = rs + sext(imm)$	AddSubImm
<i>addiu</i>	001001	–		$rt = rs + sext(imm)$	AddSubImm
<i>sub</i>	000000	100010		$rd = rs - rt$	AddSub
<i>subu</i>	000000	100011		$rd = rs - rt$	AddSub
<i>mul</i>	011100	000010		$rd = rs * rt$	Mul
<i>mult</i>	000000	011000		$(hi, lo) = rs * rt$	Mul
<i>multu</i>	000000	011001		$(hi, lo) = rs * rt$	Mul
<i>div</i>	000000	011010		$(hi, lo) = (rs \div rt, rs \div rt)$, signed	DivRem
<i>divu</i>	000000	011011		$(hi, lo) = (rs \div rt, rs \div rt)$, unsigned	DivRem
<i>madd</i>	011100	000000		$(hi, lo) = (hi, lo) + rs * rt$ (signed)	MiscInstrs
<i>maddu</i>	011100	000001		$(hi, lo) = (hi, lo) + rs * rt$ (unsigned)	MiscInstrs
<i>msub</i>	011100	000100		$(hi, lo) = (hi, lo) - rs * rt$ (signed)	MiscInstrs
<i>msubu</i>	011100	000101		$(hi, lo) = (hi, lo) - rs * rt$ (unsigned)	MiscInstrs
<i>clo</i>	011100	100001	Comparison and logic	$rd = count_leading_ones(rs)$	CloClz
<i>clz</i>	011100	100000		$rd = count_leading_zeros(rs)$	CloClz
<i>slt</i>	000000	101010		$rd = rs < rt$	Lt
<i>sltu</i>	000000	101011		$rd = rs < rt$ (unsigned)	Lt
<i>slti</i>	001010	–	Shifts and bit fields	$rt = rs < sext(imm)$	LtImm
<i>sltiu</i>	001011	–		$rt = rs < sext(imm)$ (unsigned)	LtImm
<i>and</i>	000000	100100		$rd = rs \& rt$	Bitwise
<i>andi</i>	001100	–		$rt = rs \& zext(imm)$	BitwiseImm
<i>or</i>	000000	100101		$rd = rs \mid rt$	Bitwise
<i>ori</i>	001101	–		$rt = rs \mid zext(imm)$	BitwiseImm
<i>xor</i>	000000	100110		$rd = rs \wedge rt$	Bitwise
<i>xori</i>	001110	–		$rt = rs \wedge zext(imm)$	BitwiseImm
<i>nor</i>	000000	100111		$rd = \sim(rs \mid rt)$	Bitwise
<i>lui</i>	001111	–		$rt = imm \ll 16$	AddSubImm
<i>sll</i>	000000	000000		$rd = rt \ll sa$	ShiftLeftImm
<i>sllv</i>	000000	000100		$rd = rt \ll rs[4:0]$	ShiftLeft
<i>srl</i>	000000	000010		$rd = rt \gg sa$	ShiftRightImm
<i>srlv</i>	000000	000110		$rd = rt \gg rs[4:0]$	ShiftRight
<i>sra</i>	000000	000011		$rd = rt \gg sa$ (arithmetic)	ShiftRightImm
<i>srav</i>	000000	000111		$rd = rt \gg rs[4:0]$ (arithmetic)	ShiftRight
<i>rotr</i>	000000	000010		$rd = rotate_right(rt, sa)$	ShiftRightImm
<i>rotrv</i>	000000	000110		$rd = rotate_right(rt, rs[4:0])$	ShiftRight
<i>ext</i>	011111	000000		$rt = rs[msb+lsb:lsb]$	MiscInstrs
<i>ins</i>	011111	000100		$rt = rt[31:msb+1] \mid\mid rs[msb-lsb:0] \mid\mid rt[lsb-1:0]$	MiscInstrs
<i>wsbh</i>	011111	100000		$rd = swaphalf(rt)$	MovCond

continued on the next page

Table 14, continued from the previous page

Mnemonic	Op	func	Family	Semantics	Chip
seb	011111	100000	Control transfer	$rd = \text{signExtend}(rt[7:0])$	MiscInstrs
seh	011111	100000		$rd = \text{signExtend}(rt[15:0])$	MiscInstrs
beq	000100	—		$PC = PC + 4 + \text{sext}(\text{offset} \ll 2),$ if $rs == rt$	Branch
bne	000101	—		$PC = PC + 4 + \text{sext}(\text{offset} \ll 2),$ if $rs != rt$	Branch
bgez	000001	—		$PC = PC + 4 + \text{sext}(\text{offset} \ll 2),$ if $rs \geq 0$	Branch
bgtz	000111	—		$PC = PC + 4 + \text{sext}(\text{offset} \ll 2),$ if $rs > 0$	Branch
blez	000110	—		$PC = PC + 4 + \text{sext}(\text{offset} \ll 2),$ if $rs \leq 0$	Branch
bltz	000001	—		$PC = PC + 4 + \text{sext}(\text{offset} \ll 2),$ if $rs < 0$	Branch
j	000010	—		$PC = (PC + 4)[31:28] \parallel$ $\text{instr_index} \parallel 00^\dagger$	Jump
jal	000011	—		$r31 = PC + 8, PC = (PC +$ $4)[31:28] \parallel \text{instr_index} \parallel$ $00^\dagger$	Jump
jr	000000	001000	Aligned memory	$PC = rs$	Jump
jalr	000000	001001		$rd = PC + 8, PC = rs$	Jump
bal	000001	—		$r31 = PC + 8, PC = PC + 4 +$ $\text{sext}(\text{offset} \ll 2)$	Jump
teq	000000	110100		trap, if $rs == rt$	MiscInstrs
lb	100000	—		$rt = \text{sext}(\text{mem_byte}(\text{base} +$ $\text{offset}))$	LoadNarrow
lbu	100100	—		$rt = \text{zext}(\text{mem_byte}(\text{base} +$ $\text{offset}))$	LoadNarrow
lh	100001	—		$rt = \text{sext}(\text{mem_halfword}(\text{base} +$ $\text{offset}))$	LoadNarrow
lhu	100101	—		$rt = \text{zext}(\text{mem_halfword}(\text{base} +$ $\text{offset}))$	LoadNarrow
lw	100011	—		$rt = \text{mem_word}(\text{base} + \text{offset})$	LoadWord
ll	110000	—		$rt = \text{mem_word}(\text{base} + \text{offset})$	LoadWord
sb	101000	—	Unaligned memory	$\text{mem_byte}(\text{base} + \text{offset}) = rt$	StoreNarrow
sh	101001	—		$\text{mem_halfword}(\text{base} + \text{offset}) =$ rt	StoreNarrow
sw	101011	—		$\text{mem_word}(\text{base} + \text{offset}) = rt$	StoreWord
sc	111000	—		$\text{mem_word}(\text{base} + \text{offset}) = rt,$ $rt = 1$, if atomic update, else $rt = 0^\dagger$	StoreWord
lwl	100010	—		$rt = rt$ merge most significant part of $\text{mem}(\text{base} + \text{offset})$	MemoryUnaligned
lwr	100110	—		$rt = rt$ merge least significant part of $\text{mem}(\text{base} + \text{offset})$	MemoryUnaligned
swl	101010	—		store most significant part of rt	MemoryUnaligned
swr	101110	—		store least significant part of rt	MemoryUnaligned
mfhi	000000	010000	Moves and no-ops	$rd = hi$	AddSubImm
mflo	000000	010010		$rd = lo$	AddSubImm
mthi	000000	010001		$hi = rs$	AddSubImm
mtlo	000000	010011		$lo = rs$	AddSubImm
movn	000000	001011		$rd = rs$, if $rt != 0$	MovCond
movz	000000	001010		$rd = rs$, if $rt == 0$	MovCond
sync	000000	001111		sync (nop)	AddSubImm
synci	000001	—		sync (nop)	AddSubImm
pref	110011	—		prefetch(nop)	AddSubImm

Table 15: Results imported from prior work.

Result	Source	Used in
Basic jagged reduction, soundness $2m/ \mathbb{F}_{p^4} $	[HJR ⁺ 25, Thm. 1.4]	Theorem 6.1
Jagged assist; batch evaluation	[HJR ⁺ 25, Thm. 1.5, Lemma 5.1]	Theorem 6.2
Indicator as a sum over columns	[HJR ⁺ 25, Claim 3.2.1, Eq. (5)]	§6.3
Width-four branching program for g	[HJR ⁺ 25, Claim 3.2.2]	§6.3
Evaluating a read-once branching program’s extension	[HJR ⁺ 25, Lemma 4.2], after Holm-gren and Rothblum [HR18]	§6.3
Sumcheck for a product of multilinear	[HJR ⁺ 25, Lemma 2.3]	§6.2
Round-by-round soundness of WHIR	[ACFY24b]	§8.2
Round-by-round soundness of sumcheck; Fiat–Shamir compilation	[BSCS16, CY24]	Theorem 8.2
Hash-collision term of the compilation	[Ope26b, Cor. 4.2.6]	Theorem 8.2
Integer reading of balanced LogUp-GKR buses	[Ope25, Cor. 3.6]	Definition 2.5
Limbs determine integers	[GPR21, Thm. 1]	Lemma 5.1
Offline memory checking	[BEG ⁺ 94, SAGL18]	Lemma 5.4
Functional IOPs and the commit-and-open compiler	[CGKY26]	Definition 8.1, §8.3
Fiat–Shamir against quantum adversaries; quantum collision search	[CMS19, BHT98]	§8.8
Multiset hashing; on an elliptic curve	[CDvD ⁺ 03, BM97, MSTA17]	§5.5; the security argument of §8.5 is not inherited

C Results used from prior work

Table 15 lists each result proved elsewhere that this paper uses, with its number in the source; where a result is specialised, the section using it says how.

References

- [ACFY24a] Gal Arnon, Alessandro Chiesa, Giacomo Fenzi, and Eylon Yogev. STIR: Reed–Solomon Proximity Testing with Fewer Queries. Cryptology ePrint Archive, Paper 2024/390, 2024.
- [ACFY24b] Gal Arnon, Alessandro Chiesa, Giacomo Fenzi, and Eylon Yogev. WHIR: Reed–Solomon Proximity Testing with Super-Fast Verification. Cryptology ePrint Archive, Paper 2024/1586, 2024.
- [AHIV17] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligerio: Lightweight sublinear arguments without a trusted setup. In *ACM CCS*, 2017.
- [AST24] Arasu Arun, Srinath Setty, and Justin Thaler. Jolt: SNARKs for virtual machines via lookups. EUROCRYPT 2024; Cryptology ePrint Archive, Paper 2023/1217, 2024.
- [BEG⁺94] Manuel Blum, Will Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. Checking the Correctness of Memories. *Algorithmica*, 12(2–3):225–244, 1994.
- [BGt23] Jeremy Bruestle, Paul Gafni, and the RISC Zero Team. RISC Zero zkVM: Scalable, transparent arguments of RISC-V integrity. <https://dev.risczero.com/proof-system-in-detail.pdf>, 2023. Draft.
- [BHT98] Gilles Brassard, Peter Høyer, and Alain Tapp. Quantum cryptanalysis of hash and claw-free functions. In *LATIN’98: Theoretical Informatics*, volume 1380 of *LNCs*, pages 163–169, 1998.
- [Bit26] BitVM Contributors. BitVM Bridge: A trust-minimized Bitcoin bridge. <https://github.com/BitVM/BitVM/tree/7d1ca3660cac08aab62e76f3aa4daec0d7403ecc>, 2026. Source snapshot of the BitVM2 Groth16 verifier, accessed 2026-10-13.
- [BL24] Marc Beunardeau and O(1) Labs. Research meets ecosystem: The applied cryptography team roadmap. <https://www.o1labs.org/blog/applied-cryptography-team-roadmap>, December 2024.
- [BM97] Mihir Bellare and Daniele Micciancio. A new paradigm for collision-free hashing: Incrementality at reduced cost. In *Advances in Cryptology – EUROCRYPT*, 1997.
- [BSBHR18] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Fast Reed–Solomon Interactive Oracle Proofs of Proximity. In *Proceedings of the 45th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 14:1–14:17, 2018.
- [BSCH⁺25] Eli Ben-Sasson, Dan Carmon, Ulrich Haböck, Swastik Kopparty, and Shubhangi Saraf. On proximity gaps for Reed–Solomon codes. Electronic Colloquium on Computational Complexity, Report TR25-169, 2025. November 7, 2025.
- [BSCI⁺20] Eli Ben-Sasson, Dan Carmon, Yuval Ishai, Swastik Kopparty, and Shubhangi Saraf. Proximity gaps for Reed–Solomon codes. In *FOCS*, 2020.
- [BSCS16] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. Interactive Oracle Proofs. In *Theory of Cryptography — TCC 2016-B*, pages 31–60, 2016.
- [BSGKS20] Eli Ben-Sasson, Lior Goldberg, Swastik Kopparty, and Shubhangi Saraf. DEEP-FRI: Sampling outside the box improves soundness. In *ITCS*, 2020.
- [CDvD⁺03] Dwaine Clarke, Srinivas Devadas, Marten van Dijk, Blaise Gassend, and G. Edward Suh. Incremental multiset hash functions and their application to memory integrity checking. In *Advances in Cryptology – ASIACRYPT*, 2003.
- [CGKY26] Alessandro Chiesa, Ziyi Guan, Christian Knabenhans, and Zihan Yu. On the Fiat–Shamir Security of Succinct Arguments from Functional Commitments. In *Advances in Cryptology — CRYPTO 2026*, pages 189–213, 2026.

- [CJSC23] Zhigang Chen, Yuting Jiang, Xinxia Song, and Liquan Chen. A survey on zero-knowledge authentication for Internet of Things. *Electronics*, 12(5):1145, 2023.
- [CMS19] Alessandro Chiesa, Peter Manohar, and Nicholas Spooner. Succinct arguments in the quantum random oracle model. In *Theory of Cryptography Conference (TCC)*, 2019. Cryptology ePrint Archive, Paper 2019/834.
- [CY24] Alessandro Chiesa and Eylon Yogev. Building Cryptographic Proofs from Hash Functions. <https://github.com/hash-based-snargs-book/hash-based-snargs-book>, 2024.
- [DDGM23] Heini Bergsson Debes, Edlira Dushku, Thanassis Giannetsos, and Ali Marandi. ZEKRA: Zero-knowledge control-flow attestation. In *ACM ASIA Conference on Computer and Communications Security (ASIA CCS)*, 2023.
- [DP23] Benjamin E. Diamond and Jim Posen. Succinct arguments over towers of binary fields. Cryptology ePrint Archive, Paper 2023/1784, 2023.
- [DP24] Benjamin E. Diamond and Jim Posen. Polylogarithmic proofs for multilinear over binary towers. Cryptology ePrint Archive, Paper 2024/504, 2024.
- [Eag25] Liam Eagen. Glock: Garbled locks for Bitcoin. Cryptology ePrint Archive, Paper 2025/1485, 2025.
- [EFG22] Liam Eagen, Dario Fiore, and Ariel Gabizon. cq: Cached quotients for fast lookups. Cryptology ePrint Archive, Paper 2022/1763, 2022.
- [EH24] Shahriar Ebrahimi and Parisa Hassanizadeh. From interaction to independence: zkSNARKs for transparent and non-interactive remote attestation. In *Network and Distributed System Security Symposium (NDSS)*, 2024. IACR ePrint 2024/1068.
- [EL26] Liam Eagen and Ying Tong Lai. Argo MAC: Garbling with elliptic curve MACs. Cryptology ePrint Archive, Paper 2026/049, 2026.
- [Eth26a] Ethereum Foundation. soundcalc: security calculator for hash-based proof systems and zkVM soundness reports. <https://github.com/ethereum/soundcalc>, 2026. Airbender and ZisK v0.16.1 reports at revision 2af5e7d, <https://github.com/ethereum/soundcalc/tree/2af5e7dfc01e4aef1bc0e544a3b75c1940b1dded/reports>; Ziren analysis uses the jagged-over-WHIR extension in PR 91 at revision d683f29. Accessed 2026-10-14.
- [Eth26b] Ethproofs. Ethproofs: real-time proving of Ethereum blocks. <https://ethproofs.org>, 2026.
- [FS87] Amos Fiat and Adi Shamir. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. In *Advances in Cryptology — CRYPTO '86*, pages 186–194, 1987.
- [GKR08] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating Computation: Interactive Proofs for Muggles. In *Proceedings of the 40th ACM Symposium on Theory of Computing (STOC)*, pages 113–122, 2008.
- [GKS23] Lorenzo Grassi, Dmitry Khovratovich, and Markus Schofnegger. Poseidon2: A faster version of the Poseidon hash function. In *AFRICACRYPT*, 2023.
- [GKS⁺26] Sanjam Garg, Dimitris Kolonelos, Mikhail Sergeevitch, Srivatsan Sridhar, and David Tse. BABE: Verifying proofs on Bitcoin made 1000x cheaper. Cryptology ePrint Archive, Paper 2026/065, 2026.
- [GLS⁺21] Alexander Golovnev, Jonathan Lee, Srinath Setty, Justin Thaler, and Riad S. Wahby. Brake-down: Linear-time and field-agnostic SNARKs for R1CS. Cryptology ePrint Archive, Paper 2021/1043, 2021.

- [GPR21] Lior Goldberg, Shahar Papini, and Michael Riabzev. Cairo – a Turing-complete STARK-friendly CPU architecture. Cryptology ePrint Archive, Paper 2021/1063, 2021. Revision of February 2025.
- [Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. In *Advances in Cryptology – EUROCRYPT 2016*, pages 305–326, 2016.
- [GW20] Ariel Gabizon and Zachary J. Williamson. plookup: A simplified polynomial protocol for lookup tables. Cryptology ePrint Archive, Paper 2020/315, 2020. <https://eprint.iacr.org/2020/315>.
- [GWC19] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: Permutations over Lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Paper 2019/953, 2019.
- [Hab22] Ulrich Haböck. Multivariate lookups based on logarithmic derivatives. Cryptology ePrint Archive, Paper 2022/1530, 2022.
- [Har25] David Harold. MIPS at 40. Jon Peddie Research, <https://www.jonpeddie.com/news/mips-at-40/>, January 2025.
- [HJR⁺25] Tamir Hemo, Kevin Jue, Eugene Rabinovich, Gyumin Roh, and Ron D. Rothblum. Jagged Polynomial Commitments (or: How to Stack Multilinears). Cryptology ePrint Archive, Paper 2025/917, 2025. Version of April 10, 2026.
- [HLP24] Ulrich Haböck, David Levit, and Shahar Papini. Circle STARKs. Cryptology ePrint Archive, Paper 2024/278, 2024.
- [HR18] Justin Holmgren and Ron Rothblum. Delegating Computations with (almost) Minimal Time and Space Overhead. Electronic Colloquium on Computational Complexity, Report TR18-161, 2018.
- [KS08] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In *ICALP*, volume 5126 of *LNCS*, pages 486–498, 2008.
- [KTA⁺26] Nakul Khambhati, Mukesh Tiwari, Azz, Sapin Bajracharya, Manish Bista, Liam Eagen, Christian Lewe, and Aaron Feickert. Mosaic: Practical malicious security for garbled circuits on Bitcoin. Cryptology ePrint Archive, Paper 2026/812, 2026.
- [LFKN92] Carsten Lund, Lance Fortnow, Howard Karloff, and Noam Nisan. Algebraic Methods for Interactive Proof Systems. *Journal of the ACM*, 39(4):859–868, 1992.
- [Lin23] Robin Linus. BitVM: Compute anything on Bitcoin. <https://bitvm.org/bitvm.pdf>, 2023.
- [Lit24] Lita Foundation. Valida ISA specification, version 1.0: A zk-optimized instruction set architecture. <https://github.com/lita-xyz/valida-releases>, 2024.
- [LKMW19] Kevin Lewi, Wonho Kim, Ilya Maykov, and Stephen Weis. Securing update propagation with homomorphic hashing. Cryptology ePrint Archive, Paper 2019/227, 2019.
- [LZZ⁺24] Tianyi Liu, Zhenfei Zhang, Yuncong Zhang, Wenqing Hu, and Ye Zhang. Ceno: Non-uniform, segment and parallel zero-knowledge virtual machine. Cryptology ePrint Archive, Paper 2024/387, 2024.
- [Mas25] Hector Massip. Secure challenge derivation in ZisK. ZisK blog, October 2025.
- [Mat25] Matter Labs. ZKsync Airbender: A RISC-V zkVM over Mersenne-31. <https://github.com/matter-labs/zksync-airbender/tree/v0.5.2>, 2025. Version 0.5.2; architecture and subargument documentation, accessed 2026-10-13.

- [MIP10] MIPS Technologies. MIPS32 architecture for programmers volume II-A: The MIPS32 instruction set. Document MD00086, 2010.
- [MIP26] MIPS Tech LLC. IP processors. <https://mips.com/products/hardware/>, 2026. Accessed September 2026.
- [MSTA17] Jeremy Maitin-Shepard, Mehdi Tibouchi, and Diego F. Aranha. Elliptic curve multiset hash. *The Computer Journal*, 60(4):476–490, 2017.
- [MT21] Dimitris Mouris and Nektarios Georgios Tsoutsos. Zilch: A framework for deploying transparent zero-knowledge proofs. *IEEE Transactions on Information Forensics and Security*, 2021. IACR ePrint 2020/1155, <https://eprint.iacr.org/2020/1155>.
- [Nex24] Nexus Labs. Nexus 1.0: Enabling verifiable computation. https://nexus-xyz.github.io/assets/nexus_whitepaper.pdf, 2024. January 2024.
- [Nex25] Nexus Labs. Nexus zkVM 3.0 specification. <https://specification.nexus.xyz/>, 2025. March 2025.
- [OANWO20] Jack O’Connor, Jean-Philippe Aumasson, Samuel Neves, and Zooko Wilcox-O’Hearn. BLAKE3: one function, fast everywhere. <https://github.com/BLAKE3-team/BLAKE3-specs/blob/master/blake3.pdf>, 2020.
- [OP 24] OP Labs. Kona: A Rust implementation of the OP Stack fault-proof program. <https://github.com/op-rs/kona>, 2024.
- [Ope25] OpenVM Contributors. On the soundness of interactions via LogUp. https://github.com/openvm-org/stark-backend/blob/main/docs/Soundness_of_Interactions_via_LogUp.pdf, 2025. March 13, 2025.
- [Ope26a] OpenVM Contributors. OpenVM whitepaper. <https://openvm.dev/whitepaper.pdf>, 2026. Version of July 21, 2026.
- [Ope26b] OpenVM Contributors. SWIRL: Stacked WHIR with interaction reductions via LogUp. <https://openvm.dev/swirl.pdf>, 2026. March 13, 2026.
- [Ope26c] OpenVM Project. Formal verification of the OpenVM RISC-V, Keccak, and SHA-2 extensions. <https://github.com/openvm-org/openvm-fv>, 2026.
- [Opt23] Optimism Foundation. Cannon: A MIPS interpreter for Optimism fault proofs. <https://github.com/ethereum-optimism/optimism>, 2023.
- [PCW⁺23] Shankara Pailoor, Yanju Chen, Franklyn Wang, Clara Rodríguez, Jacob Van Geffen, Jason Morton, Michael Chu, Brian Gu, Yu Feng, and Işıl Dillig. Automated detection of under-constrained circuits in zero-knowledge proofs. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1510–1532, 2023.
- [PH23] Shahar Papini and Ulrich Haböck. Improving logarithmic derivative lookups using GKR. Cryptology ePrint Archive, Paper 2023/1284, 2023.
- [Pol24] Polygon Zero and contributors. Plonky3: A toolkit for STARKs over small fields. <https://github.com/Plonky3/Plonky3>, 2024.
- [QV15] Nguyen Anh Quynh and Dang Hoang Vu. Unicorn: The ultimate CPU emulator. Black Hat USA; <https://www.unicorn-engine.org/>, 2015.
- [RIS24] RISC Zero. STARK soundness calculator. <https://github.com/risc0/risc0/blob/main/risc0/zkp/src/prove/soundness.rs>, 2024.

- [RR24] Gyumin Roh and Ron D. Rothblum. SP1 V4 Turbo: Memory argument via elliptic curve based multiset hashing. Succinct Labs technical memo, https://docs.succinct.xyz/assets/files/SP1_Turbo_Memory_Argument-b042ba18b58c4add20a8370f4802f077.pdf, 2024. December 2024.
- [SAGL18] Srinath Setty, Sebastian Angel, Trinabh Gupta, and Jonathan Lee. Proving the correct execution of concurrent services in zero-knowledge. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.
- [Scr26] Scroll. Ceno. <https://github.com/scroll-tech/ceno>, 2026. Revision 7c58fdd4, retrieved 2026-09-24.
- [SD21] Xavier Salleras and Vanesa Daza. ZPiE: Zero-knowledge proofs in embedded systems. *Mathematics*, 9(20):2569, 2021. IACR ePrint 2021/1382.
- [Sho97] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, 1997.
- [ST25] Srinath Setty and Justin Thaler. Twist and Shout: faster memory checking arguments via one-hot addressing and increments. Cryptology ePrint Archive, Paper 2025/105, 2025.
- [Sta23] StarkWare Team. ethSTARK documentation—version 1.2. Cryptology ePrint Archive, Paper 2021/582, 2023. July 2023 revision.
- [STW24] Srinath Setty, Justin Thaler, and Riad S. Wahby. Unlocking the lookup singularity with Lasso. EUROCRYPT 2024; Cryptology ePrint Archive, Paper 2023/1216, 2024.
- [Suc24] Succinct Labs. SP1: A performant, open-source, contributor-friendly zkVM. <https://github.com/succinctlabs/sp1>, 2024. Accessed 2026.
- [Suc25a] Succinct Labs. SP1 Hypercube: Proving Ethereum in real-time. <https://blog.succinct.xyz/sp1-hypercube/>, 2025. Announcement, May 20, 2025.
- [Suc25b] Succinct Labs. sp1-jit: Just-in-time RISC-V executor for SP1. <https://github.com/succinctlabs/sp1/tree/main/crates/core/jit>, 2025.
- [Suc26a] Succinct Labs. SP1 proof system. <https://docs.succinct.xyz>, 2026.
- [Suc26b] Succinct Labs. SP1 security model. <https://docs.succinct.xyz/docs/sp1/security/security-model>, 2026. Retrieved 2026-09-24.
- [Sup25] Supranational. sppark: Zero-knowledge template library for accelerating SNARK provers. <https://github.com/supranational/sppark>, 2025. Retrieved 2026-09-25.
- [Tur21] Jim Turley. Wait, what? MIPS becomes RISC-V. *EE Journal*, March 2021. <https://www.eejournal.com/article/wait-what-mips-becomes-risc-v/>.
- [WAA⁺26] Robin Linus Woll, Ioannis Alexopoulos, Lukas Aumayr, Zeta Avarikioti, Matteo Maffei, and David Tse. BitVM3: Efficient Bitcoin bridges via garbled circuits. Cryptology ePrint Archive, Paper 2026/933, 2026.
- [Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 162–167, 1986.
- [ZCF23] Hadas Zeilberger, Binyi Chen, and Ben Fisch. BaseFold: Efficient Field-Agnostic Polynomial Commitment Schemes from Foldable Codes. Cryptology ePrint Archive, Paper 2023/1705, 2023.
- [ZGK⁺18] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. vRAM: Faster verifiable RAM with program-independent preprocessing. In *IEEE Symposium on Security and Privacy (S&P)*, 2018.

- [Zis25] ZisK Project. ZisK: A high-performance zkVM. <https://github.com/0xPolygonHermes/zisk>, 2025.
- [ZRE15] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole: Reducing data transfer in garbled circuits using half gates. In *EUROCRYPT*, volume 9057 of *LNCS*, pages 220–250, 2015.