Performance Improvements of ZK-Prover for rWasm: A Sound and Efficient AIR for 32-bit Division and Remainder

Suleyman Kardas, Mehmet Sabir Kiraz, Dmitry Savonin, Yao Wang, and Aliaksei Dziadziuk

`suleyman@fluentlabs.xyz`, `mehmet@fluentlabs.xyz`, `dmitry@fluentlabs.xyz`, `yao@fluentlabs.xyz`, `aliaksei@fluentlabs.xyz`

Fluent Labs

Abstract. Zero-knowledge virtual machines (zkVMs) must correctly model all edge cases of low-level machine instructions, including division and remainder, while keeping algebraic constraints simple enough for efficient STARK proving. This work presents a focused but meaningful redesign of the `DivRemChip` used in the SP1 zkVM (as adapted to rWasm) to implement WASM's 32-bit division and remainder instructions (`i32.div{u,s}`, `i32.rem{u,s}`) over the BabyBear field. Our chip remains local (single-row AIR) and is purpose-built so that all "small aggregate" expressions are strictly bounded by the BabyBear modulus, allowing us to use inverses securely without ever inverting zero. We replace heavier constant-equality logic and trap gadgets with:

(a) a lightweight small-aggregate zero-test for magnitudes that is sound in BabyBear,
(b) a simple, byte-level mechanism for detecting the special value `INT_MIN` using only degree-2 constraints,
(c) a low-degree test for $c = -1$ based on checking $|c| = 1$ together with the sign bit, and
(d) a constraint pattern ensuring that divide-by-zero and the overflow case `i32.div_s(INT_MIN, -1)` are *not provable*, matching WASM trap semantics.

The resulting AIR has maximal degree 3, removes all "invert-zero" failure modes in BabyBear, and enforces the correct semantics for every non-trapping instruction. A malicious prover test framework based on the SP1 CPU bus shows that any incorrect witness causes constraint failure, while honest execution traces for extensive boundary tests and mixed programs produce valid proofs over BabyBear. Our design also improves efficiency: the trace width drops from 102 to 74 columns, all `Mul`/`Add`/`Lt`/MSB cross-chip calls are removed, and each row requires only a single CPU-bus interaction. Prover benchmarks on unsigned, signed, and mixed workloads with up to 125,000 division/remainder operations show a consistent 4–6% reduction in single-threaded proving time compared to the original SP1 `DivRemChip` as adapted to rWasm, with a paired t-test across program sizes confirming that the improvement is statistically significant at the 95% confidence level.

1 Introduction

Zero-knowledge proof systems for virtual machines (zkVMs), such as SP1, must accurately encode low-level instruction semantics inside a polynomial constraint system. Among all instructions, integer division and remainder are particularly subtle: they mix signed and unsigned interpretations, have non-trivial corner cases, and include explicit traps in the WASM specification. In particular, for 32-bit words, WASM defines four opcodes: `i32.div_u`, `i32.rem_u`, `i32.div_s`, and `i32.rem_s`. For unsigned operations, semantics follow ordinary Euclidean division with a trap on division by zero. For signed operations, operands are interpreted in two's-complement, with the additional overflow trap

```
i32.div_s(INT_MIN, -1).
```

A zkVM must enforce these semantics even against a fully malicious prover who may attempt to mis-report quotients, remainders, or trap behaviour. The SP1 framework already includes a `DivRemChip` for these operations, but the original design used relatively heavy constant-equality gadgets and trap logic. In practice, this led to two undesirable properties:

(i) zero-detection gadgets whose witnesses could attempt to invert the zero field element, and
(ii) constraint expressions that exceeded the degree bound intended by the SP1 system. Both issues complicate implementation, and the first one can appear at runtime as a "tried to invert zero" failure.

1.1 Contributions

This paper focuses on a compact, self-contained improvement to the SP1 `DivRemChip`. Given that the chip is foundational for security and reused across many programs, even a modest redesign has non-trivial impact. Our main contributions are:

1. **A simplified AIR for 32-bit division and remainder.** We replace constant-equality gadgets based on large aggregated encodings with:
   - a reusable small aggregate equality test to detect the special magnitude 1 for C_{abs}, and
   - a lightweight detection of `INT_MIN` that only uses the low three bytes and a single MSB check. All equality and zero-detection logic is implemented with low-degree polynomials over the base field.
2. **Trap as non-provability.** We do not introduce a dedicated trap column. Instead, our constraints are constructed so that any trace containing a divide-by-zero event or the overflow case `i32.div_s(INT_MIN, -1)` admits *no* satisfying witness for the `DivRemChip`. This matches the WASM semantics without requiring additional global trap wiring inside the chip itself.
3. **Degree-3 and zero-safe gadgets.** Every constraint in our AIR has total degree at most 3, including those involving the special-case detection. All inverses are multiplied only by small aggregates that are strictly smaller than the field modulus, eliminating any possibility of "inverting zero" in a valid witness.
4. **Implementation and malicious prover tests.** We provide a concrete Rust implementation inside an SP1-style framework, together with tests that:
   (i) generate traces for random inputs and boundary values,
   (ii) prove and verify STARK proofs over these traces using the BabyBear field,
   (iii) attempt to maliciously modify the quotient/remainder and the CPU bus result, and
   (iv) confirm that divide-by-zero executions and signed-overflow divisions are non-provable.

   In all these experiments, honest executions verify and malicious executions are rejected.

These changes have negligible impact on the width of the chip and the number of constraints per row, and they preserve the local, single-row structure of the original SP1 design. In the following sections, we give a concise description of our algorithm, its security in the malicious model, and a brief performance discussion.

1.2 Roadmap

Section 2 reviews related work on AIR-based zkVMs, lookup-based front-ends, and division gadgets. Section 3 formalises our improved `DivRemChip` and its AIR constraints. Section 4 analyses soundness and completeness of the construction against a malicious prover, organised by attack class and supported by lemmas. Section 5 discusses width, constraint degree, and the impact on prover and verifier costs, including empirical benchmarks. Section 6 concludes and outlines possible future extensions.

2 Related Work

AIR-based proof systems and zkVM backends. Our work follows the line of research that models low-level machine semantics via Algebraic (or Arithmetic) Intermediate Representations (AIRs) for STARK-based proof systems. The original STARK constructions of Ben-Sasson et al. established AIR as a dominant abstraction for expressing CPU traces and state-transition rules in a transparent, post-quantum setting [1]. Cairo is a canonical example of a STARK-based zkVM whose arithmetization is directly expressed as an AIR over a CPU-like instruction set, together with dedicated machinery for public memory and boundary constraints [2, 3]. On the SNARK side, Plonky2 and its successor Plonky3 provide high-performance recursive proof backends that operate on AIR-like constraint systems and are explicitly designed to serve as proving engines for zkVMs [4, 5]. At a design level, these systems implement each machine step as a bounded-degree polynomial identity over the current and next row of the execution trace, and rely on FRI-style low-degree testing to obtain soundness against a malicious prover. Recent surveys of zero-knowledge proof (ZKP) frameworks and ZK applications highlight AIR-based arithmetizations and zkVMs such as Cairo, RISC Zero, and SP1 as central building blocks for scalable, general-purpose ZK systems [6–8].

Lookup-based front-ends: Jolt. On the SNARK side, Jolt [9] proposed a lookup-based front-end for zkVMs that shifts most instruction semantics into large lookup tables over a RISC-V-style instruction set. At the polynomial level, Jolt interpolates the trace columns $\mathsf{pc}, \mathsf{inst}, \mathsf{state}$ and the corresponding table columns into low-degree polynomials over two evaluation domains, one for the execution trace and one for the lookup table. A Lasso-style lookup argument then compresses each tuple $(\mathsf{pc}, \mathsf{inst}, \mathsf{state})$ into a single field element using random challenges and reduces the multiset containment “every executed instruction appears in the table of valid transitions” to a univariate polynomial identity between an accumulator over the trace domain and an accumulator over the table domain. This identity is finally checked via the sumcheck protocol, hence proving algebraically that every executed instruction tuple is reduced from the precomputed transition relation. For arithmetic instructions such as division and remainder, correctness is described in terms of table rows: the prover must show that (b, c, q, r) appears in a table satisfying

$$b = q \cdot c + r, \qquad 0 \le r < |c|,$$

with signedness handled by an appropriate encoding, while the low-degree constraints themselves only check table membership and consistency with the machine state [9]. This architecture results in small local constraint systems but requires very large tables and a trusted, or at least separately audited, table-generation procedure. Subsequent work formally verifies the Jolt constraint system, showing that its lookup relations match the intended RISC-V interpreter semantics and revealing subtle bugs in how table lookups interact with control flow and memory [10]. From an algebraic perspective, Jolt treats semantics as a set of allowed tuples: an instruction is correct exactly when its tuple is found in the corresponding relation. In contrast, our `DivRemChip` takes a more internal view: we give an explicit family of polynomial identities that describe 32-bit Euclidean division, two’s-complement encoding, WASM’s trap behaviour, and constant-equality detection, without relying on precomputed tables.

SP1 and AIR-driven zkVMs with precompiles. The SP1 zkVM developed by Succinct is a modern AIR-based system that targets programs compiled to a RISC-V-like ISA and uses the Plonky3 stack as its proving backend [11–13]. SP1 uses a Rust-friendly programming model and relies heavily on precompiles which are specialised chips for heavy operations such as hashing, elliptic curve arithmetic, and certain big-integer functions. Succinct’s public benchmarks and technical write-ups show that this architecture can outperform Jolt on several workloads at comparable proof sizes and cycle counts, largely by collapsing long instruction traces for complex primitives into constant-factor overheads inside dedicated chips [14]. At the AIR level, SP1's precompiles play the same role as our `DivRemChip`: they factor out complex operations into reusable constraint fragments. However, public descriptions of SP1 focus on systems aspects (recursion, integration with the prover network, and on-chain verification) and treat precompile correctness in terms of Rust-level reference implementations rather than fully algebraic specifications.

To the best of our knowledge, there is currently no published algebraic specification for SP1's 32-bit division or remainder chip in the malicious model, including all two's-complement corner cases. Our contribution addresses this gap for 32-bit division in full: starting from an executable Rust specification, we derive a complete AIR formalization of the `DivRemChip` and demonstrate that every satisfying assignment corresponds to a unique, valid, non-trapping execution under the intended WASM semantics. In principle, the same algebraic construction can serve as a soundness-defining precompile within SP1, resulting in a reusable component with an explicit, white-box guarantee against malicious-prover behaviour.

Instruction-set design with dedicated division chips. Several recent zkVM instruction-set proposals include dedicated 32-bit division units at the ISA level, for example Valida's Div32 chip and Miden-style `U32Div` instructions used in ZK-friendly assembly backends for Move and other high-level languages [15,16]. The Valida ISA, for example, defines a modular collection of "chips" including a Div32 chip that introduces integer-division opcodes alongside chips for multiplication, shifting, comparisons, and range checking [15]. The Valida design document makes explicit that these chips are intended to be implemented as AIR fragments inside a Plonky3-based STARK stack, and that Div32 in particular is structurally more complex than other ALU operations [17,18]. Algorithmically, such designs typically follow the textbook Euclidean pattern: introduce a quotient witness Q and remainder witness R, check

$$B = Q \cdot C + R$$

in the base field, and use an inequality gadget to enforce $0 \le R < |C|$ for the appropriate notion of signedness. This cleanly separates algebraic constraints (multiplication and addition) from ordering constraints (range and comparison), but public specifications often leave important details underspecified: unsigned versus signed division, handling of $\text{INT\_MIN}/-1$ overflow, and the internal soundness of the less-than gadget itself.

Our `DivRemChip` can be viewed as a strengthened, fully algebraic variant of a Div32-style chip in the SP1 setting. Rather than relying on an external comparison primitive, we encode the inequality reasoning using a "difference" word $diff$ and carry constraints of the form $R_{\text{abs}} + diff + 1 = C_{\text{abs}}$ over base-256 decompositions, enforced locally inside the chip.

Integer division gadgets and constraint encodings. Beyond zkVMs, integer division has been studied as a gadget in generic R1CS and polynomial-constraint systems, often in the context of verifiable ML or general-purpose arithmetic circuits. A standard pattern is to encode Euclidean division via the relation $B = Q \cdot C + R$ together with range and inequality constraints enforcing $0 \le R < C$, sometimes using non-deterministic witnesses and specialised comparison subcircuits [19]. In many of these works, division appears as an auxiliary primitive whose main role is to support higher-level functionality; correctness is therefore argued over an abstract finite field and typically focuses on asymptotic constraint counts rather than a complete case analysis of machine-level semantics (word size, two's-complement encoding, and explicit trap behaviour).

Our design follows the same high-level algebraic template but specialises it to 32-bit two's-complement arithmetic embedded into a concrete prime field. We combine this with byte-wise base-256 decompositions and local carry constraints, allowing inequalities to be enforced internally within the AIR without relying on external gadgets.

Formal verification of ZK circuits. There has been significant study on the formal verification of ZK circuits and zkVM implementations. Coglio et al. develop a compositional framework for proving that a circuit or constraint system implements a given high-level specification, and apply it to a range of arithmetic and ZK-specific constructions [20]. This line of work models circuit descriptions as first-class objects inside an interactive theorem prover and focuses on machine-checked proofs of semantic correctness. More recently, the formal verification of Jolt's lookup semantics illustrates this trend in the zkVM setting: the goal is to show that a complex lookup-based constraint system correctly enforces the RISC-V semantics encoded by its tables [10].

3 Our Improved Algorithm

In this section we describe our improved `DivRemChip`. The chip is specified in three layers:

- the layout of the trace and the columns used by the chip,
- the deterministic row-generation algorithm that fills these columns from a given ALU event, and
- the AIR constraints that enforce the 32-bit WASM div/rem semantics against a malicious prover.

Throughout we work over a prime field (BabyBear in the implementation) and represent 32-bit words in base 256 as four bytes $x = x_0 + 256x_1 + 256^2x_2 + 256^3x_3$ with $x_i \in \{0, \ldots, 255\}$.

3.1 Trace Layout

Each row of the `DivRemChip` trace corresponds to a single 32-bit WASM operation among

$$\mathsf{I32DivU},\ \mathsf{I32DivS},\ \mathsf{I32RemU},\ \mathsf{I32RemS}.$$

For a given ALU event $(\mathsf{pc}, \mathsf{op}, b, c, a)$ the row contains:

Instruction and operands.

- Program counter $\mathsf{pc} \in$.
- Input words $B, C \in^4$ and output word $A \in^4$, each stored as four base-256 bytes.
- Opcode selector bits

$$\mathsf{is\_div\_u},\ \mathsf{is\_div\_s},\ \mathsf{is\_rem\_u},\ \mathsf{is\_rem\_s} \in \{0, 1\}$$

  indicating which of the four operations is active in the row.

Special-value helpers and INT_MIN detection.

- A flag $\mathsf{c\_abs\_is\_one}$ and inverse accumulator $\mathsf{c\_abs\_one\_inv}$ implementing a small aggregate test for $C_{\text{abs}} = 1$.
- A derived flag $\mathsf{is\_c\_neg\_one}$ indicating $c = -1$ in signed mode (detected as $\mathsf{c\_sign} \wedge (C_{\text{abs}} = 1)$).
- A helper flag $\mathsf{is\_divs\_intmin}$ encoding $\mathsf{is\_div\_s} \wedge \mathsf{is\_b\_int\_min}$, used to express the overflow exclusion as a degree-3 constraint.
- Two helper bits

$$\mathsf{is\_b\_int\_min},\ \mathsf{is\_c\_int\_min} \in \{0, 1\}$$

  signalling that the magnitudes B_{abs} or C_{abs} equal INT_MIN $= 0x80000000$, encoded as $[0, 0, 0, 128]$ in little-endian.
- MSB magnitude check scalars $\mathsf{b\_check\_msb}$, $\mathsf{c\_check\_msb}$ which, together with u8 range checks, ensure that the most significant byte of a signed magnitude is at most 127 unless the corresponding $\mathsf{is\_\_int\_min}$ flag is set.

Signs and magnitudes.

– Sign bits
$$\mathsf{b\_sign},\ \mathsf{c\_sign},\ \mathsf{q\_sign},\ \mathsf{r\_sign},\ \mathsf{sign\_xor} \in \{0,1\}$$
where b_sign (resp. c_sign) indicates whether b (resp. c) is negative in signed mode and $\mathsf{sign\_xor} = \mathsf{b\_sign} \oplus \mathsf{c\_sign}$.

– Absolute values (magnitudes)
$$B_{\text{abs}}, C_{\text{abs}}, Q_{\text{abs}}, R_{\text{abs}} \in^4$$
representing $|b|, |c|, |q|, |r|$ as base-256 byte vectors.

– Boolean flags
$$\mathsf{is\_q\_zero},\ \mathsf{is\_r\_zero},\ \mathsf{is\_q\_nz\_signed},\ \mathsf{is\_r\_nz\_signed} \in \{0,1\}$$
indicating whether Q_{abs} and R_{abs} are zero and whether they are non-zero in signed mode.

Core div/rem arithmetic and inequality.

– A carry array carry[0..3] encoding the base-256 relation
$$B_{\text{abs}} = Q_{\text{abs}} \cdot C_{\text{abs}} + R_{\text{abs}},$$
computed byte-wise.

– A "difference" word diff[0..3] and carries diff_carry[0..2] encoding
$$R_{\text{abs}} + \mathsf{diff} + 1 = C_{\text{abs}}$$
in base 256 without overflow, enforcing $0 \le R_{\text{abs}} < C_{\text{abs}}$ whenever $C_{\text{abs}} \ne 0$.

Two's-complement reconstruction.

– Three arrays of carries
$$\mathsf{b\_tc\_carry}[0..3], \quad \mathsf{c\_tc\_carry}[0..3], \quad \mathsf{a\_tc\_carry}[0..3]$$
used to reconstruct the signed representations of b, c, and a from their magnitudes. These enforce the 2's-complement relation $x + |x| \equiv 2^{32}$ when the sign bit of x is 1.

3.2 Deterministic Row Generation

Given an ALU event $(\mathsf{pc}, \mathsf{op}, b, c, a)$, the `event_to_row` procedure deterministically fills all columns:

1. Embed pc, b, c, a into and split words into four bytes in base 256.
2. Set the opcode selector bits according to op and a boolean *signed* flag for I32DivS/I32RemS.
3. In signed mode compute magnitudes $B_{\text{abs}}, C_{\text{abs}}$ by conditionally negating negative inputs using 32-bit wrapping, and set b_sign and c_sign accordingly.
4. Compute
$$(Q_{\text{abs}}, R_{\text{abs}}) = \begin{cases} (0, B_{\text{abs}}) & \text{if } C_{\text{abs}} = 0, \\ \big(\lfloor B_{\text{abs}}/C_{\text{abs}} \rfloor,\ B_{\text{abs}} \bmod C_{\text{abs}}\big) & \text{otherwise,} \end{cases}$$
and fill the carry arrays so that the base-256 equalities hold.

5. Set q and r signs according to WASM semantics: signed division returns $\mathrm{sgn}(q) = \mathsf{b\_sign} \oplus \mathsf{c\_sign}$ when $q \neq 0$, and signed remainder returns $\mathrm{sgn}(r) = \mathsf{b\_sign}$ when $r \neq 0$. The output word A is the CPU result, i.e. q or r in 2's-complement.
6. Set the zero and nonzero flags, the INT_MIN flags $\mathsf{is\_b\_int\_min}, \mathsf{is\_c\_int\_min}$ from the pattern $[0, 0, 0, 128]$, and the MSB helpers $\mathsf{b\_check\_msb}$, $\mathsf{c\_check\_msb}$.
7. Implement the $C_{\text{abs}} = 1$ gadget by computing

$$\text{agg}_1 = (c_0 - 1) + 2c_1 + 4c_2 + 8c_3,$$

set $\mathsf{c\_abs\_is\_one}$ to $(C_{\text{abs}} == 1)$, and $\mathsf{c\_abs\_one\_inv}$ to agg_1^{-1} when $C_{\text{abs}} \neq 1$ (and 0 otherwise). The derived flag $\mathsf{is\_c\_neg\_one}$ is set to $\texttt{is\_signed} \wedge (c = -1)$, which is compatible with the AIR constraint $\mathsf{is\_c\_neg\_one} = \mathsf{c\_sign} \cdot \mathsf{c\_abs\_is\_one}$. The helper flag $\mathsf{is\_divs\_intmin}$ is set to $(\mathsf{opcode} = \mathsf{I32DivS}) \wedge \mathsf{is\_b\_int\_min}$.
8. Populate the two's-complement carry chains encoding $b + B_{\text{abs}}$, $c + C_{\text{abs}}$, and, in signed modes, $a + Q_{\text{abs}}$ or $a + R_{\text{abs}}$.
9. Fill the inequality witness $diff$ and $\mathsf{diff\_carry}$ with $diff = C_{\text{abs}} - R_{\text{abs}} - 1$ when $C_{\text{abs}} \neq 0$, and zero otherwise.

Note that for divide-by-zero we deliberately choose a "trap-like" witness $(Q_{\text{abs}}, R_{\text{abs}}) = (0, B_{\text{abs}})$. As explained below, this witness cannot satisfy the AIR constraints, so any proof containing such a row is rejected.

3.3 AIR Constraints and Degree Bound

The `eval` method of the chip implements a collection of local AIR constraints, all of which are multivariate polynomials of degree at most 3 in the row variables. The main groups of constraints are:

Selector and sign constraints.

- Exactly one opcode selector bit is active:

$$\mathsf{is\_div\_u} + \mathsf{is\_div\_s} + \mathsf{is\_rem\_u} + \mathsf{is\_rem\_s} \in \{0, 1\}$$

together with booleanity constraints on all selector and sign bits.
- Unsigned ops enforce zero signs: $\mathsf{b\_sign} = \mathsf{c\_sign} = \mathsf{q\_sign} = \mathsf{r\_sign} = 0$.
- The XOR relation is enforced in signed mode by

$$\mathsf{sign\_xor} \ = \ \mathsf{b\_sign} + \mathsf{c\_sign} - 2\,\mathsf{b\_sign}\,\mathsf{c\_sign},$$

a quadratic polynomial.

Core div/rem arithmetic.

- For each byte position i we enforce

$$B_{\text{abs}}[i] + 256 \cdot \mathsf{carry}[i] \ = \ R_{\text{abs}}[i] + \mathsf{carry}[i-1] + \sum_{u+v=i} Q_{\text{abs}}[u]\, C_{\text{abs}}[v],$$

where $\mathsf{carry}[-1]$ is taken to be 0. This expresses $B_{\text{abs}} = Q_{\text{abs}} C_{\text{abs}} + R_{\text{abs}}$ in base 256. The sum of products is quadratic; after gating by the opcode selectors this results in degree at most 3.
- The inequality $0 \leq R_{\text{abs}} < C_{\text{abs}}$ (when $C_{\text{abs}} \neq 0$) is encoded via the byte-wise equality

$$R_{\text{abs}} + \mathsf{diff} + 1 = C_{\text{abs}}$$

with a final carry constrained to 0. Only additions are involved, so the degree remains at most 2 (or 3 after gating).

Zero-testing gadgets for Q_{abs} and R_{abs}. We use a low-degree "small aggregation"

$$\text{agg}(x) = x_0 + 2x_1 + 4x_2 + 8x_3$$

which is always strictly less than the field modulus for BabyBear. The AIR enforces

$$\text{agg}(Q_{\text{abs}}) = 0 \iff \mathsf{is\_q\_zero} = 1, \qquad \text{agg}(R_{\text{abs}}) = 0 \iff \mathsf{is\_r\_zero} = 1$$

via the pattern

$$\text{agg}(Q_{\text{abs}}) \cdot \mathsf{q\_inv} = 1 - \mathsf{is\_q\_zero},$$

and similarly for R_{abs}. These are quadratic in the row variables (degree 2, or 3 when gated).

Equality gadget for $C_{\text{abs}} = 1$ and detection of $c = -1$. To detect the special constant $c = -1$ we first detect the magnitude $C_{\text{abs}} = 1$ and then combine this with the sign bit:

- Define the aggregate
$$\text{agg}_1(C_{\text{abs}}) = (c_0 - 1) + 2c_1 + 4c_2 + 8c_3,$$
where c_i are the bytes of C_{abs}. This aggregate lies in $[-1, 3824]$, which embeds injectively into the BabyBear field.
- The AIR enforces
$$\mathsf{c\_abs\_is\_one}^2 = \mathsf{c\_abs\_is\_one}, \quad \mathsf{c\_abs\_is\_one} \cdot \text{agg}_1 = 0,$$
and
$$\text{agg}_1 \cdot \mathsf{c\_abs\_one\_inv} = 1 - \mathsf{c\_abs\_is\_one},$$
which together force $\text{agg}_1 = 0$ if and only if $\mathsf{c\_abs\_is\_one} = 1$, and never require inverting zero.
- Finally, we encode
$$\mathsf{is\_c\_neg\_one} = \mathsf{c\_sign} \cdot \mathsf{c\_abs\_is\_one}$$
for real rows, expressing that $c = -1$ in signed mode if and only if c is negative and $|c| = 1$.

INT_MIN detection and overflow exclusion. For INT_MIN we use a lightweight pattern check:

- When $\mathsf{is\_b\_int\_min} = 1$ we enforce that the low three bytes of B_{abs} aggregate to zero and the MSB is 128; similarly for C_{abs} and $\mathsf{is\_c\_int\_min}$. The reverse direction (only the vector $[0, 0, 0, 128]$ can satisfy this with all bytes range-checked) follows from the MSB helper constraints and u8 range checks.
- We introduce a degree-2 helper flag $\mathsf{is\_divs\_intmin} = \mathsf{is\_div\_s} \cdot \mathsf{is\_b\_int\_min}$, and enforce the degree-3 overflow exclusion
$$\mathsf{is\_divs\_intmin} \cdot \mathsf{is\_c\_neg\_one} = 0,$$
which rules out any witness for `i32.div_s(INT_MIN, -1)`.

Sign logic and two's-complement reconstruction. We enforce:

- If $\mathsf{is\_q\_nz\_signed} = 1$ then $\mathsf{q\_sign} = \mathsf{sign\_xor}$; if $\mathsf{is\_r\_nz\_signed} = 1$ then $\mathsf{r\_sign} = \mathsf{b\_sign}$.
- If $\mathsf{q\_sign} = 0$ then $A = Q_{\text{abs}}$, and if $\mathsf{q\_sign} = 1$ then $A + Q_{\text{abs}} \equiv 2^{32}$ via the $\mathsf{a\_tc\_carry}$ chain (and analogously for $\mathsf{I32RemS}$ with R_{abs}).
- For b and c we assert that sign 0 implies $X_{\text{abs}} = X$ and sign 1 implies $X + X_{\text{abs}} \equiv 2^{32}$ with final carry 1.

All these are at most quadratic equalities, gated by selector and sign bits.

Instruction bus coupling. Finally, the chip exposes the standard SP1 instruction bus:

$$(\mathsf{pc},\ \mathsf{pc} + \mathit{\Delta},\ \mathsf{opcode},\ A,\ B,\ C),$$

where `opcode` is reconstructed as a linear combination of the selector bits and opcode constants. This ties the DivRem trace to the CPU trace and prevents inconsistencies between the ALU and the execution record.

4 Security Analysis of Our Algorithm in the Malicious Model

We now discuss that our new `DivRemChip` is sound and complete with respect to the 32-bit WASM semantics, in the presence of a malicious prover. All constraints are local (single-row) and of degree at most 3, which matches the global design goals of the SP1 system.

4.1 Model and Goal

Each row of the `DivRemChip` corresponds to a single ALU event carrying an opcode

`i32.div_u`, `i32.rem_u`, `i32.div_s`, `i32.rem_s`

together with input words $b, c \in \{0, \ldots, 2^{32} - 1\}$ and an output word a. The chip exposes magnitudes $B_{\text{abs}}, C_{\text{abs}}, Q_{\text{abs}}, R_{\text{abs}}$, sign bits, and supporting helper columns in its trace, and it is linked to the CPU AIR via the instruction bus. An honest prover constructs a witness row from the actual ALU event via our deterministic `event_to_row` procedure. A malicious prover may instead attempt to provide arbitrary field elements in these columns and arbitrary CPU events, subject only to the global SP1 constraints. We want to show that:

– For every non-trapping WASM execution, there exists a satisfying assignment to the chip trace (completeness).
– For every row that satisfies the AIR constraints, the corresponding opcode and (a, b, c) triple must match the WASM semantics (soundness).

We now formalise the main building blocks as lemmas and then state a row-level soundness theorem.

4.2 Base-256 Facts and Core Arithmetic Lemmas

Lemma 1 (Base-256 Reconstruction). *Let $x_0, \ldots, x_3 \in \{0, \ldots, 255\}$ and set*

$$X := x_0 + 256x_1 + 256^2x_2 + 256^3x_3.$$

Then $X \in \{0, \ldots, 2^{32} - 1\}$, and the map

$$(x_0, x_1, x_2, x_3) \longmapsto X$$

is injective.

Proof. This is the uniqueness of base-256 expansion of integers in $\{0, \ldots, 2^{32} - 1\}$.

In the AIR, $B_{\text{abs}}, C_{\text{abs}}, Q_{\text{abs}}, R_{\text{abs}}$ are all stored as such 4-byte vectors; Lemma 1 lets us reason about them as honest 32-bit integers.

Lemma 2 (Bytewise Multiply-Add Correctness). *Let $q_i, c_i, r_i \in \{0, \dots, 255\}$ and $k_i \in \mathbb{Z}_{\geq 0}$ for $i = 0, 1, 2, 3$, and define*

$$M_k := \sum_{i+j=k} q_i c_j, \quad k = 0, 1, 2, 3.$$

Assume that for $k = 0, 1, 2, 3$ we have

$$M_k + r_k + k_{k-1} = b_k + 256\, k_k,$$

with $k_{-1} := 0$, $b_k \in \{0, \dots, 255\}$, and $k_3 = 0$. Let

$$Q := \sum_{i=0}^{3} q_i 256^i, \quad C := \sum_{i=0}^{3} c_i 256^i, \quad R := \sum_{i=0}^{3} r_i 256^i, \quad B := \sum_{i=0}^{3} b_i 256^i.$$

Then

$$B = Q \cdot C + R \quad \text{in } \mathbb{Z}, \qquad 0 \leq B, Q, C, R < 2^{32}.$$

Proof. The equations are exactly the schoolbook base-256 multiplication and addition of $Q \cdot C$ and R with carries. Summing the equalities $M_k + r_k + k_{k-1} = b_k + 256 k_k$ over k and multiplying by the appropriate powers of 256 results in $B = QC + R$ in $\mathbb{Z}$. The bound $< 2^{32}$ follows from the per-byte bounds and $k_3 = 0$.

4.3 Small-Aggregate Zero Gadgets

To test whether a magnitude word x_{abs} is zero, we associate to its bytes $x_0, \dots, x_3 \in \{0, \dots, 255\}$ the *small aggregate*

$$A := x_0 + 2x_1 + 4x_2 + 8x_3.$$

Because $A \in [0, 3825]$, it is strictly smaller than the BabyBear modulus and therefore embeds injectively into the field. For $x \in \{Q, R\}$, the AIR enforces the relations

$$\mathsf{is\_x\_zero}^2 = \mathsf{is\_x\_zero}, \qquad \mathsf{is\_x\_zero} \cdot A = 0, \qquad A \cdot \mathsf{x\_inv} = 1 - \mathsf{is\_x\_zero},$$

where $\mathsf{is\_x\_zero}$ is a boolean selector and $\mathsf{x\_inv}$ is an auxiliary inverse accumulator.

Lemma 3 (Soundness of the Zero-Test Gadget and Safety Against Zero Inversion). *Let the field modulus p satisfy $p > A_{\max} = 3825$ (as holds for BabyBear). Given bytes $x_0, \dots, x_3 \in \{0, \dots, 255\}$, let A be defined as above, and suppose*

$$\mathsf{is\_x\_zero}^2 = \mathsf{is\_x\_zero}, \quad \mathsf{is\_x\_zero} \cdot A = 0, \quad A \cdot \mathsf{x\_inv} = 1 - \mathsf{is\_x\_zero}$$

holds in the field. Then:

1. *$A = 0$ if and only if $\mathsf{is\_x\_zero} = 1$.*
2. *If $A = 0$ then necessarily $\mathsf{x\_inv} = 0$.*
3. *If $A \neq 0$ then $\mathsf{is\_x\_zero} = 0$ and $\mathsf{x\_inv} = A^{-1}$ in the field.*

Consequently, the gadget never requires inverting the zero element.

Proof. If $A = 0$, the last constraint implies $0 = 1 - \mathsf{is\_x\_zero}$, so $\mathsf{is\_x\_zero} = 1$. The relation $\mathsf{is\_x\_zero} \cdot A = 0$ then forces $\mathsf{x\_inv} = 0$. If $A \neq 0$, the value A is invertible in the field, so the last constraint yields $\mathsf{x\_inv} = A^{-1}$ and the middle constraint forces $\mathsf{is\_x\_zero} = 0$.

4.4 Internal Inequality Gadget

The inequality $0 \leq R_{\text{abs}} < C_{\text{abs}}$ is encoded via the relation

$$R_{\text{abs}} + \mathsf{diff} + 1 = C_{\text{abs}}$$

in base 256, using a stored word diff and carry vector diff_carry.

Lemma 4 (Inequality Encoding). *Let $R, C \in \{0, \ldots, 2^{32} - 1\}$ and suppose there exist $\mathsf{diff} \in \{0, \ldots, 2^{32}-1\}$ and $\kappa_0, \ldots, \kappa_3 \in \{0, \ldots, 255\}$ such that, writing r_i, d_i, c_i for the bytes of R, diff, C, we have*

$$\begin{aligned} r_0 + d_0 + 1 &= c_0 + 256\kappa_0, \\ r_i + d_i + \kappa_{i-1} &= c_i + 256\kappa_i, \quad i = 1, 2, 3, \end{aligned}$$

with $\kappa_3 = 0$. Let R, C, diff be reconstructed from their bytes as in Lemma 1. Then

$$R + \mathsf{diff} + 1 = C, \qquad 0 \leq R < C < 2^{32}.$$

Conversely, if $0 \leq R < C < 2^{32}$, then there exist unique $\mathsf{diff} \in \{0, \ldots, 2^{32} - 1\}$ and $\kappa_0, \ldots, \kappa_3 \in \{0, \ldots, 255\}$ with $\kappa_3 = 0$ satisfying the equations above. In particular, there is no such solution when $C = 0$.

Proof. The equations are exactly the base-256 addition of $R + \mathsf{diff} + 1$ with carries κ_i, and the condition $\kappa_3 = 0$ ensures no overflow beyond $2^{32} - 1$. Reconstructing R, diff, C from their bytes as in Lemma 1, we obtain $R + \mathsf{diff} + 1 = C$ and $0 \leq C < 2^{32}$. Since $\mathsf{diff} \geq 0$ and we add 1, necessarily $C > R$. The converse direction is by standard base-256 subtraction; $C = 0$ results in a contradiction.

4.5 Detecting `INT_MIN` and $c = -1$

We now formalise the correctness of the INT_MIN gadget and the $c = -1$ detection.

INT_MIN detection. Recall that INT_MIN $= 2^{31}$ has bytes $[0, 0, 0, 128]$ in little-endian.

Lemma 5 (INT_MIN Detection Soundness). *Let B_{abs} be a 4-byte word with bytes in $\{0, \ldots, 255\}$, and let $\mathsf{is\_b\_int\_min}, \mathsf{b\_check\_msb}$ satisfy the constraints in the AIR. Then:*

1. *If $\mathsf{is\_b\_int\_min} = 1$, then B_{abs} has bytes $[0, 0, 0, 128]$, i.e. $B_{\text{abs}} = 2^{31}$.*
2. *If $B_{\text{abs}} = 2^{31}$, then necessarily $\mathsf{is\_b\_int\_min} = 1$ in any satisfying assignment.*

An analogous statement holds for C_{abs} and is_c_int_min.

Proof. When $\mathsf{is\_b\_int\_min} = 1$, the low three bytes aggregate to zero and the MSB is constrained to 128, so the bytes are exactly $[0, 0, 0, 128]$. Conversely, if $B_{\text{abs}} = [0, 0, 0, 128]$, the MSB gadget and its range check force $\mathsf{is\_b\_int\_min} = 1$, otherwise the expression defining b_check_msb would violate the 8-bit bound.

Detection of $C_{\text{abs}} = 1$ *and* $c = -1$. We reuse the small aggregate pattern with a constant shift.

Lemma 6 (Detection of $C_{\text{abs}} = 1$). *Let* C_{abs} *have bytes* $c_0, \ldots, c_3 \in \{0, \ldots, 255\}$ *and define*

$$A_1 = (c_0 - 1) + 2c_1 + 4c_2 + 8c_3.$$

Let $\mathsf{c\_abs\_is\_one}, \mathsf{c\_abs\_one\_inv}$ *satisfy*

$$\mathsf{c\_abs\_is\_one}^2 = \mathsf{c\_abs\_is\_one}, \quad \mathsf{c\_abs\_is\_one} \cdot A_1 = 0, \quad A_1 \cdot \mathsf{c\_abs\_one\_inv} = 1 - \mathsf{c\_abs\_is\_one}.$$

Then $A_1 = 0$ *if and only if* $C_{\text{abs}} = 1$*, and in this case* $\mathsf{c\_abs\_is\_one} = 1$*; otherwise* $A_1 \neq 0$ *and* $\mathsf{c\_abs\_is\_one} = 0$*. In particular,* $\mathsf{c\_abs\_one\_inv}$ *never inverts zero.*

Proof. The value A_1 lies in $[-1, 3824]$, hence is embedded injectively into BabyBear. If $C_{\text{abs}} = 1$ then $c_0 = 1$ and $c_1 = c_2 = c_3 = 0$, so $A_1 = 0$. Conversely, if $A_1 = 0$, then as integers

$$c_0 - 1 + 2c_1 + 4c_2 + 8c_3 = 0.$$

Since $c_1, c_2, c_3 \geq 0$, we must have $c_1 = c_2 = c_3 = 0$ and $c_0 - 1 = 0$, hence $C_{\text{abs}} = 1$. The equivalence between $A_1 = 0$ and $\mathsf{c\_abs\_is\_one} = 1$ then follows exactly as in Lemma 3.

Combining Lemma 6 with the AIR constraint

$$\mathsf{is\_c\_neg\_one} = \mathsf{c\_sign} \cdot \mathsf{c\_abs\_is\_one}$$

results in a purely algebraic characterisation of the signed constant $c = -1$.

4.6 Trap Behaviour as Non-Provability

WASM specifies that division by zero and the signed overflow `i32.div_s(INT_MIN, -1)` must trap. We encode this property by ensuring that such events admit no satisfying witness for the `DivRemChip`.

- For $C = 0$, the inequality gadget $R_{\text{abs}} + \mathsf{diff} + 1 = C_{\text{abs}}$ and its carry constraints become impossible to satisfy under our deterministic `event_to_row` witness generation, since $C_{\text{abs}} = 0$ and $R_{\text{abs}} = B_{\text{abs}}$ cannot fulfil the low-byte relation. Any attempted modification of diff or R_{abs} that fixes this will break the main division identity or the small aggregate zero-tests.
- For the overflow case, we combine the `INT_MIN` and -1 detectors and assert that no row with opcode `i32.div_s`, $\mathsf{is\_b\_int\_min} = 1$ and $\mathsf{is\_c\_neg\_one} = 1$ may appear. This is expressed as the degree-3 constraint $\mathsf{is\_divs\_intmin} \cdot \mathsf{is\_c\_neg\_one} = 0$ and rules out any witness for `i32.div_s(INT_MIN, -1)`.

The global CPU AIR is responsible for handling traps at the control-flow level. Our contribution is that any trace containing a trapping event is *unprovable* by the `DivRemChip`, which matches the intended semantics and is confirmed by the `test_divrem_divide_by_zero_trap_compliance` test in the implementation.

4.7 CPU Bus Binding and malicious prover Tests

Finally, we bind the chip to the CPU via the standard SP1 instruction bus: each row encodes the opcode, inputs b, c and result a, and the CPU AIR enforces consistency with the execution trace. The `test_malicious_divrem` test case explicitly constructs a program and then corrupts both:

(i) the `DivRemChip` ALU event (by incrementing the result), and
(ii) the CPU event that records the result in memory. In all such cases the prover fails to generate a valid proof, showing that our constraints rule out these natural attack attempts.

4.8 Row Soundness Theorem

We now summarise the effect of all constraints in a single soundness statement.

Theorem 1 (Row Soundness Under Malicious Prover). *Consider any row of the* `DivRemChip` *trace with* $\mathsf{is\_div\_u} + \mathsf{is\_div\_s} + \mathsf{is\_rem\_u} + \mathsf{is\_rem\_s} = 1$ *such that:*

- *all local AIR constraints of the chip (as described in Section 3 and in this section) are satisfied, and*
- *the instruction-bus constraint tying* $(\mathsf{pc}, \mathsf{opcode}, a, b, c)$ *to the CPU trace holds.*

Then:

1. $c \neq 0$ *and we do not have the overflow pattern* $\mathsf{op} =$ `i32.div_s`, $b = \mathrm{INT\_MIN}$, $c = -1$.
2. *If* $\mathsf{op} \in \{$`i32.div_u`, `i32.rem_u`$\}$, *then interpreting* b, c *as unsigned integers,*
   - *for* `i32.div_u` *we have* $a = \lfloor b/c \rfloor$,
   - *for* `i32.rem_u` *we have* $a = b \bmod c$,

   and $0 \le a < c$ *in the remainder case.*
3. *If* $\mathsf{op} \in \{$`i32.div_s`, `i32.rem_s`$\}$, *then interpreting* $b_s = \mathrm{tc}(b)$ *and* $c_s = \mathrm{tc}(c)$ *as signed two's-complement integers,*
   - *for* `i32.div_s` *we have* $a_s = \mathrm{tc}(a) = \mathrm{sgn}(b_s c_s)\,\lfloor |b_s|/|c_s| \rfloor$,
   - *for* `i32.rem_s` *we have* $a_s = \mathrm{tc}(a)$ *equal to the signed remainder, with* $|a_s| < |c_s|$ *and* $\mathrm{sign}(a_s) = \mathrm{sign}(b_s)$.

In other words, every satisfying row with a real opcode corresponds to a non-trapping WASM div/rem step with correct semantics.

Proof (Proof sketch). By Lemma 2, the core multiplication constraints enforce the Euclidean relation $B_{\mathrm{abs}} = Q_{\mathrm{abs}} C_{\mathrm{abs}} + R_{\mathrm{abs}}$ on magnitudes. By Lemma 4, the internal inequality gadget enforces $0 \le R_{\mathrm{abs}} < C_{\mathrm{abs}}$ and in particular $C_{\mathrm{abs}} \neq 0$. Combined with the signed reconstruction constraints and the MSB/INT_MIN gadget (Lemma 5), this results in the correct bounds for the signed remainder and ensures that signed magnitudes are in the intended range, with the correct handling of INT_MIN.

The zero-test gadgets (Lemma 3) ensure that the $\mathsf{is\_q\_zero}$, $\mathsf{is\_r\_zero}$ flags are consistent with Q_{abs} and R_{abs}; the sign constraints then fix $\mathsf{q\_sign}$ and $\mathsf{r\_sign}$ as required by the WASM rules. Lemma 6, together with the explicit constraint $\mathsf{is\_c\_neg\_one} = \mathsf{c\_sign} \cdot \mathsf{c\_abs\_is\_one}$, algebraically characterises the signed constant $c = -1$, and the overflow exclusion constraint $\mathsf{is\_divs\_intmin} \cdot \mathsf{is\_c\_neg\_one} = 0$ rules out the forbidden pattern `i32.div_s`(INT_MIN, -1). The output mux and two's-complement carry constraints tie a to either q or r with the appropriate sign, depending on the opcode. Finally, the instruction bus ensures that the opcode and (a, b, c) are exactly those seen by the CPU trace. Any deviation from the WASM semantics would fall into one of the attack classes considered above and would violate at least one of the corresponding constraint families.

5 Performance Analysis

Our modifications are intentionally minimal from a performance perspective. We keep the `DivRemChip` local (no cross-row dependencies), and we only introduce a handful of extra columns compared to the original SP1 design.

5.1 Width and Constraint Degree

The total number of columns in the chip is fixed by the Rust layout `DivRemCols<T>`, and the main differences relative to the original SP1 `DivRemChip` are:

– the small aggregate gadget ($\mathsf{c\_abs\_is\_one}, \mathsf{c\_abs\_one\_inv}$) for detecting $C_{\text{abs}} = 1$,
– additional boolean flags is_b_int_min, is_c_int_min, is_c_neg_one and is_divs_intmin, and
– the MSB helper columns b_check_msb and c_check_msb.

All other columns are either already present in SP1 (magnitudes, signs, carries, and CPU bus fields) or are simple re-arrangements of existing information. Every constraint in our AIR has total degree at most 3. The degree-2 constraints come from products such as $Q_{\text{abs}} \cdot C_{\text{abs}}$, sign XORs, the small aggregate zero-tests, and the $C_{\text{abs}} = 1$ gadget. The degree-3 constraints arise from gating these products by selector bits and from the overflow exclusion

$$\mathsf{is\_divs\_intmin} \cdot \mathsf{is\_c\_neg\_one} = 0,$$

which matches the degree bound used by SP1's composition strategy.

5.2 Impact on Prover and Verifier

Since the chip remains single-row and the per-row constraint set is only marginally larger, the asymptotic prover and verifier costs are unchanged. The additional columns contribute a constant-factor increase to the width but do not introduce any new lookups, cross-table arguments, or multi-row relations. In our implementation, the new chip passes the existing SP1-style test suite: we can generate traces, produce proofs, and verify them for a variety of mixed div/rem programs and boundary cases (including INT_MIN, -1, and maximal unsigned values). The malicious prover tests show that this correctness is robust under adversarial modifications of both the chip witness and the CPU bus events. To assess the concrete performance impact, we also ran deterministic benchmark programs with varying numbers of div/rem operations.

5.3 Benchmarks

Table 1. Single-threaded prover runtimes (seconds) for the original and improved `DivRemChip` across three opcode pools.

	Unsigned pool		Signed pool		Mixed pool	
# ops	Original	Improved	Original	Improved	Original	Improved
10 000	10.712	10.433	10.741	10.373	10.682	10.267
25 000	15.244	14.797	15.475	14.964	15.418	14.970
50 000	24.203	22.612	24.349	22.848	24.121	22.996
75 000	40.162	38.836	42.039	38.768	40.434	38.910
100 000	42.537	40.583	43.405	40.655	43.377	40.750
125 000	45.290	42.849	45.963	42.959	45.629	43.539

Benchmark setup. We constructed deterministic benchmark programs with 10 000, 25 000, 50 000, 75 000, 100 000, and 125 000 division/remainder operations on fixed inputs. For each size we evaluated three opcode pools: (i) an unsigned pool with equal numbers of `I32DivU` and `I32RemU`; (ii) a signed pool with equal numbers of `I32DivS` and `I32RemS`; and (iii) a mixed pool with equal numbers of signed and unsigned operations. The prover was run with the original SP1 `DivRemChip` (as adapted to rWasm) and with the improved design on the same machine (MacBook Pro 2021, Apple M1 Max, 64 GB RAM, 4 TB SSD, macOS Tahoe 26.1), in a single-threaded configuration. Table 1 reports wall-clock runtimes in seconds.

Results. For each pool, we treated the six program sizes as paired observations of (original, improved) runtimes and applied a two-sided paired t-test at the 95% confidence level on the per-size differences (original minus improved).

For the unsigned, signed, and mixed pools, the mean runtime reductions were approximately 1.34 s (about 4.2%), 1.90 s (about 5.6%), and 1.37 s (about 4.3%), respectively. The corresponding test statistics were $t \approx 3.88$, $t \approx 3.62$, and $t \approx 3.78$ with 5 degrees of freedom, resulting in two-sided p-values $p < 0.02$ in all cases. Thus, the improved `DivRemChip` achieves a consistent 4–6% reduction in prover time across all three opcode pools, with the improvement statistically significant at the 95% level. Given that the modifications are local and degree-preserving, this is compatible with the expectation that simplified equality and zero-test gadgets improve constant factors without changing asymptotic complexity.

6 Conclusion

We have presented a refined AIR for a 32-bit division/remainder chip suitable for STARK-based zkVMs in the malicious model. Our construction:

- enforces the Euclidean relation $B_{\text{abs}} = Q_{\text{abs}} C_{\text{abs}} + R_{\text{abs}}$ using purely 32-bit, byte-wise base-256 multiplication with local carries;
- integrates the inequality $R_{\text{abs}} < C_{\text{abs}}$ via the relation $R_{\text{abs}} + \mathsf{diff} + 1 = C_{\text{abs}}$, thus avoiding any dependence on external less-than gadgets;
- uses small aggregate zero-tests and lightweight special-value detectors for INT_MIN and $C_{\text{abs}} = 1$, combined with the sign bit to identify the signed corner case $c = -1$, while keeping all constraints of degree at most 3;
- employs explicit range checks, MSB logic, and sign constraints to rule out phantom-magnitude attacks and to reconstruct all inputs and outputs as 32-bit signed values via two's-complement embedding.

Critically, divide-by-zero and the overflow case `i32.div_s(INT_MIN, -1)` are encoded as *non-provable* in the `DivRemChip`: no witness can satisfy the local AIR constraints for these events. When combined with the CPU AIR and the instruction bus, this results in a clean separation between algebraic enforcement of non-trapping semantics and control-flow handling of traps. Our Rust implementation within an SP1-style framework, together with malicious prover tests and empirical benchmarks, demonstrates that the modified chip is both sound and practically efficient. The changes incur negligible overhead in width and constraint count, and they eliminate all observed "invert-zero" failures. We view this `DivRemChip` as a template for designing and analysing other components that are critical to the system's soundness guarantees zkVM components: start from an executable semantics, derive a low-degree AIR with explicit gadgets for corner cases, prove soundness in the malicious model, and validate the design with targeted adversarial tests. Extending this methodology to more complex instructions (e.g. 64-bit division, floating-point operations, or modular arithmetic) is a natural direction for future work.

References

1. E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, "Scalable, transparent, and post-quantum secure computational integrity," *Cryptology ePrint Archive*, vol. 2018, no. 046, 2018.

2. StarkWare, “Architecture: The cairo programming language.” `https://www.starknet.io/cairo-book/ch201-architecture.html`, 2023. Accessed 30 November 2025.
3. zkSecurity Team, “Cairo public memory: Stark book.” `https://zksecurity.github.io/stark-book/cairo/memory.html`, 2023. Accessed 30 November 2025.
4. Polygon Labs, “Introducing plonky2.” `https://polygon.technology/blog/introducing-plonky2`, Jan. 2022. Blog post.
5. Polygon Labs, “Plonky2: A deep dive.” `https://polygon.technology/blog/plonky2-a-deep-dive`, Nov. 2022. Blog post.
6. R. Lavin, X. Liu, H. Mohanty, L. Norman, G. Zaarour, and B. Krishnamachari, “A survey on the applications of zero-knowledge proofs,” *arXiv preprint arXiv:2408.00243*, 2024.
7. N. Sheybani, A. Ahmed, M. Kinsy, and F. Koushanfar, “Zero-knowledge proof frameworks: A systematic survey,” *arXiv preprint arXiv:2502.07063*, 2025.
8. Lita Team, “Exploring zk-VM design trade-offs.” `https://www.lita.foundation/blog/a-zero-knowledge-paradigm-part-2--exploring-zk-vm-design-trade-offs`, June 2024. Blog post.
9. A. Arun, S. T. V. Setty, and J. Thaler, “Jolt: SNARKs for virtual machines via lookups,” in *Advances in Cryptology - EUROCRYPT 2024*, Lecture Notes in Computer Science, pp. 3–33, Springer, 2024.
10. C. Kwan, Q. Dao, and J. Thaler, “Verifying jolt zkVM lookup semantics,” in *Financial Cryptography and Data Security 2025 (pre-proceedings)*, 2025. Available as preproceedings paper at FC ’25.
11. Succinct Labs, “SP1 zkVM documentation: Introduction.” `https://docs.succinct.xyz/docs/sp1/introduction`, 2024. Accessed 30 November 2025.
12. Polygon Labs, “Succinct’s SP1, built with polygon plonky3, will help enable performant cross-chain interoperability for the agglayer.” `https://polygon.technology/blog/succincts-sp1-built-with-polygon-plonky3-will-help-enable-performant-cross-chain-interoperability-for-the-` May 2024. Blog post.
13. L2Beat, “SP1 zkVM catalog entry.” `https://l2beat.com/zk-catalog/sp1`, 2024. Accessed 29 November 2025.
14. U. Roy, “SP1 testnet launch: the fastest, feature-complete zkVM for developers.” `https://www.succinct.xyz/blog-articles/sp1-testnet-launch-the-fastest-feature-complete-zkvm-for-developers`, May 2024. Blog post.
15. M. Thomas, M. Ratsimbazafy, M. Bugaj, L. Revill, C. Modica, S. Schmidt, V. Tan, D. Lubarov, M. Gillett, and W. Dai, “Valida ISA spec, version 1.0: A zk-optimized instruction set architecture,” *arXiv preprint arXiv:2505.08114*, 2025.
16. Polygon Miden Team, “The miden virtual machine.” `https://0xmiden.github.io/miden-vm/`, 2024. Accessed: 2025-11-30.
17. M. Thomas, “Valida march review notes.” `https://www.lita.foundation/blog/valida-march-review-notes`, Mar. 2024. Blog post.
18. Lita Foundation, “Valida zk-VM architecture documentation.” `https://lita.gitbook.io/lita-documentation/architecture/valida-zk-vm`, 2024. Accessed 29 November 2025.
19. D. Kang, T. Hashimoto, I. Stoica, and Y. Sun, “Scaling up trustless DNN inference with zero-knowledge proofs,” *arXiv preprint arXiv:2210.08674*, 2022.
20. A. Coglio, E. McCarthy, E. Smith, C. Chin, P. Gaddamadugu, and M. Dellepere, “Compositional formal verification of zero-knowledge circuits,” *Cryptology ePrint Archive*, vol. 2023, no. 1278, 2023.