Design and Implementation of a Toy STARK-Based Zero-Knowledge Virtual Machine

BSc Bogdan Rudnevkii

MSc thesis
January 2026

MATEMATIIKAN JA TILASTOTIETEEN LAITOS

Reviewers:
Prof. Jarkko Kari
Doc. Ville Junnila

The originality of this thesis has been checked in accordance with the University of Turku quality assurance system using the Turnitin OriginalityCheck service

UNIVERSITY OF TURKU, Department of Mathematics and Statistics

MSc Thesis
Subject: Mathematics
Author: Bogdan Rudnevkii
Title: Design and Implementation of a Toy STARK-Based Zero-Knowledge Virtual Machine
Supervisor: Prof. Jarkko Kari
Pages: 33 pages + 27 appendix pages
Month and year: January 2026

With recent advancements in zero-knowledge proofs, zero-knowledge virtual machines (zkVMs) have emerged as a practical way to verify program correctness while keeping inputs private. A zkVM allows a prover to generate a succinct proof of computational integrity. Verification of such proofs is significantly cheaper than recomputing the original program. This thesis presents the design and implementation of a toy zkVM using the Scalable Transparent ARgument of Knowledge (STARK)-style approach. The system models execution in the algebraic intermediate representation (AIR): an execution trace together with boundary and transition constraints that restrict valid computations. These AIR constraints are proven using a cryptographic backend, based on commitment schemes and low-degree testing. Implementation is demonstrated through progressively enhanced examples that prove and verify the computation. The thesis discusses key design choices, limitations, and directions for future work toward a full virtual machine with richer instruction sets and memory.

Keywords: zero-knowledge proofs, zkVM, STARK, FRI, polynomial IOP, Rust.

Contents

1 Introduction **1**
1.1 Arithmetic Circuits 1
1.2 SNARKs parameters 2
1.3 Zero-knowledge virtual machines 2

2 Background **3**
2.1 Radix-2 evaluation domains 4
2.2 Polynomials and evaluation domains 4
2.3 NTT/INTT and the evaluation representation 6
2.4 Commitments and Merkle trees 6
2.5 Fiat–Shamir transcript and challenges. 7
2.6 FRI low-degree testing and the STARK IOP 8

3 Arithmetic Intermediate Representation **9**
3.1 Constructing the quotient polynomial in evaluation form 10

4 Cryptographic Backend **11**
4.1 Merkle-tree commitment 11
4.2 Fiat–Shamir transcript 11
4.3 FRI low-degree test 12
4.4 High-level protocol flow 12

5 System Design and Implementation **13**
5.1 Project structure 14
5.2 Example overview 15
5.3 Trace implementation 15
5.4 AIR and constraints 15
5.5 Prover and verifier pipeline 16
5.6 Basic Fibonacci Example 17
5.7 Padded Fibonacci Example 18
5.8 Virtual Machine 20
5.9 VM AIR integration 25
5.9.1 Structural Constraints 25
5.9.2 Initialization Constraints 26
5.9.3 Transition Constraints 27
5.9.4 Preservation Constraints 28
5.9.5 Branching Constraints 29
5.9.6 Halt Semantics Constraints 30
5.10 Complete zkVM pipeline 30

6 Conclusion and Future Work **31**

List of Abbreviations **34**

A Rust Merkle Tree Implementation **35**

B Rust Transcript Implementation **38**

C Rust FRI Implementation **40**

D Rust AIR Implementation **45**

E Rust ZKVM Public Params **46**

F Rust ZKVM Prover **48**

G Rust ZKVM Verifier **50**

H Rust Basic Fibonacci Constraints **54**

I Rust Padded Fibonacci Constraints **56**

1 Introduction

Zero-knowledge proofs (ZKPs) allow a prover to convince a verifier that a statement is true without revealing the secret witness underlying the statement. A big class of ZKPs are zero-knowledge succinct non-interactive arguments of knowledge (zk-SNARKS), which combine zero-knowledge, succinct proofs that are cheap to verify, and non-interactivity in the random-oracle model.

Succinctness means that proof size and verification time grow sublinearly in the computation size, often polylogarithmically, or may even remain constant [2]. An argument of knowledge provides a knowledge guarantee: a prover that succeeds in convincing the verifier must “know” a corresponding witness, under standard cryptographic assumptions.

Succinct verification is particularly important in blockchain applications. For example, in zero-knowledge rollups, transaction execution is performed off-chain, and a succinct proof is generated to show that the resulting state transition is correct. This proof is then verified on-chain at a cost substantially lower than that of re-executing the full computation, enabling scalability while preserving integrity [3].

The work presented in this thesis was inspired by the approach introduced in [1] and was carried out in collaboration with Adapt Framework Solutions Ltd. In particular, discussions with collaborators associated with the original work helped shape the implementation direction of this thesis. Nevertheless, the design decisions, implementation, experimental evaluation, and conclusions presented here are the author’s own, and any errors or omissions remain the author’s responsibility.

The ChatGPT AI has been used to enhance the language of the thesis.

1.1 Arithmetic Circuits

Many SNARK systems express computations as arithmetic circuits (or equivalent constraint systems) over a finite field. An arithmetic circuit is a directed acyclic graph (DAG) whose internal nodes compute additions and multiplications, and whose inputs consist of public inputs (the instance) and private inputs (the witness).

For example, proving knowledge of an x such that $x^3 = 27$ can be expressed with two multiplication gates computing x^2 and x^3, while keeping x private and the output public (See 1). Although proving the cube of a number is trivial, arithmetic circuits are a scalable tool for modeling general computation. For instance, Kosba et al. present an optimized circuit for the Secure Hash Algorithm 256 (SHA-256) containing 26,155 gates [4].

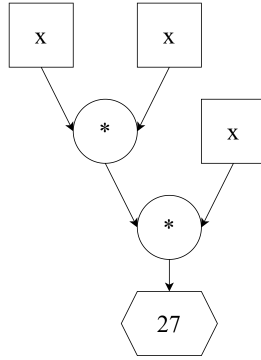


Figure 1: Arithmetic circuit for computing $y = x^3$ using two multiplication gates. The input x is private (witness) and the output y is public (instance).

1.2 SNARKs parameters

SNARK constructions vary by several parameters, including whether they require a trusted setup, proof size, prover time, verifier time, and underlying cryptographic assumptions (including post-quantum considerations). For instance, Groth16 requires a circuit-specific structured reference string, whereas Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge (PLONK) with Kate–Zaverucha–Goldberg (KZG) commitments supports a universal and updatable structured reference string that can be reused across circuits. [5][6][7]. In contrast, zk-STARKs rely on hash-based commitments and typically require no trusted setup ("transparent setup"), at the cost of larger proof sizes [8].

1.3 Zero-knowledge virtual machines

Zero-knowledge virtual machines (zkVMs) aim to provide proofs of correct execution for general-purpose programs. Instead of expressing an application directly as a hand-designed arithmetic circuit, a zkVM proves the execution of a program written for a fixed instruction set, for instance, Reduced Instruction Set Computer (RISC-V), allowing developers to write code in high-level languages such as C or Rust and then prove properties about its execution. Widely used examples include RISC Zero and SP1 [9][10].

This approach enables the reuse of existing tooling and language ecosystems. However, this generality introduces overhead compared to specialized circuit-first designs, which may be more efficient for narrowly targeted statements. zkVMs, therefore, represent a practical trade-off between expressiveness and performance, especially in applications where proving arbitrary program execution is desirable.

The goal of this thesis is to study the components required to implement a zkVM from scratch, document the key design choices, and demonstrate the system across several examples. The thesis also discusses the limitations of the current prototype and outlines directions for future improvements.

The remainder of the thesis is organized as follows. Section 2 reviews the necessary background on finite fields, polynomials, and proof-system primitives. Section 3 introduces AIR and the construction of execution traces and constraints. Section 4 describes the proving backend and verification procedure. Section 5 discusses implementation details, and Section 6 summarizes the progress and outlines future improvements.

2 Background

As discussed in the introduction, a ZKP allows a prover to convince a verifier of the truth of a statement without revealing the underlying witness. In a zkVM setting, the witness is typically an *execution trace* T that records the machine state across time. Correctness of the trace is enforced by a collection of *boundary constraints*, which fix designated rows, such as the initial and final states, and by *transition constraints*, which enforce that each step follows the VM's transition function. The prover and verifier also agree on the *public* part of the execution, including the program, the public input, and the claimed output (or termination value).

Many proof systems used in practice are based on interactive protocols. For this reason, we recall standard notions of completeness and soundness for interactive proofs and arguments following Thaler [11, Sec. 3.1]. Let $f : \{0,1\}^n \to R$ be a function with finite range R. An interactive proof for f consists of a probabilistic polynomial-time verifier V and a prescribed honest prover P, who are given a common input $x \in \{0,1\}^n$. At the start of the protocol, the prover sends a claimed value y intended to equal $f(x)$. The parties then exchange k messages according to the protocol rules, and V outputs accept or reject.

Definition 1 (Completeness and soundness [11])**.** An interactive proof system (V, P) has completeness error δ_c and soundness error δ_s if:

1. **(Completeness)** For every $x \in \{0,1\}^n$, an honest interaction causes V to accept with probability at least $1 - \delta_c$ (over V's randomness).

2. **(Soundness)** For every $x \in \{0,1\}^n$ and every (possibly cheating) prover strategy P', if P' claims a value $y \neq f(x)$, then V accepts with probability at most δ_s (over V's randomness).

The system is called *valid* if $\delta_c, \delta_s \leq 1/3$.

Informally, completeness guarantees that correct claims are accepted with high probability when the prover is honest, while soundness guarantees that false claims are accepted only with small probability. In this thesis, we model zkVM proofs as *argument systems*: completeness is required for honest executions, and soundness is required only against polynomial-time adversaries (computational soundness) under standard cryptographic assumptions [11, Def. 3.2].

Notation. Let $\mathbb{F}$ be a prime field, and let $\mathbb{F}^* = \mathbb{F} \setminus \{0\}$ denote its multiplicative group of nonzero elements. N denotes the trace length, and we assume $N = 2^\ell$

for some $\ell \in \mathbb{N}$. We assume that $\mathbb{F}$ contains a multiplicative subgroup of size N, equivalently that $2^\ell \mid (|\mathbb{F}| - 1)$. Let $H \subset \mathbb{F}^*$ be the multiplicative subgroup of size N generated by an element $\omega \in \mathbb{F}^*$ of order N, i.e.,

$$H = \langle \omega \rangle = \{1, \omega, \omega^2, \dots, \omega^{N-1}\},$$

where $\omega^N = 1$ and $\omega^i \neq 1$ for all $0 < i < N$. This subgroup serves as the trace evaluation domain.

Let $b \geq 1$ denote the blowup factor. We write $H_{\text{lde}} \subset \mathbb{F}^*$ for a multiplicative subgroup of size $|H_{\text{lde}}| = bN$, generated by an element $\omega_{\text{lde}} \in \mathbb{F}^*$ of order bN, so that

$$H_{\text{lde}} = \langle \omega_{\text{lde}} \rangle.$$

This subgroup serves as the low-degree extension evaluation domain. We write $s \cdot H_{\text{lde}}$ for a shifted (multiplicative) coset, where $s \in \mathbb{F}^*$ and $s \notin H_{\text{lde}}$ is chosen such that $s \cdot H_{\text{lde}} \cap H = \emptyset$. The shifted coset $s \cdot H_{\text{lde}}$ is used as an alternative evaluation domain when quotient expressions must be evaluated away from the zeros of the vanishing polynomial.

Low-degree extension. Given values of a function on the trace domain (equivalently, evaluations of a low-degree polynomial over H), the low-degree extension refers to evaluating the corresponding polynomial over a larger domain such as H_{lde} or $s \cdot H_{\text{lde}}$.

2.1 Radix-2 evaluation domains

In this thesis, we primarily work with multiplicative subgroups whose size is a power of two. This choice is defined by two core components used throughout the prover and verifier: the Fast Reed–Solomon interactive oracle proof of proximity (FRI) [12] and the number-theoretic transform (NTT) [13]. FRI is used to test that an oracle of evaluations is consistent with a polynomial of low degree over a prescribed domain, while the NTT provides an efficient way to convert between a polynomial's coefficient representation and its evaluations on a radix-2 domain (and vice versa via the inverse NTT).

2.2 Polynomials and evaluation domains

Let $\mathbb{F}[X]$ denote the ring of univariate polynomials over $\mathbb{F}$. Since $|H| = N$, every assignment of values to the points of H determines a unique interpolating polynomial of degree less than N. Thus, each trace column may be viewed as the evaluation vector of a polynomial on H.

For the trace domain $H \subset \mathbb{F}^*$, define its vanishing polynomial by

$$Z_H(X) := \prod_{x \in H} (X - x).$$

By construction, $Z_H(x) = 0$ for every $x \in H$. Since $H = \langle \omega \rangle$ is a multiplicative subgroup of size N, its elements are exactly the N-th roots of unity in $\mathbb{F}$, and

therefore

$$Z_H(X) = \prod_{i=0}^{N-1} (X - \omega^i) = X^N - 1.$$

Using this algebraic view, relations between trace values can be expressed as polynomial identities. As a simple example, suppose we want to enforce that the pointwise difference between two polynomials $g(X)$ and $f(X)$ is the constant 50 on the trace domain H. Define

$$P(X) := g(X) - f(X) - 50.$$

Then this condition is equivalent to requiring that P vanish on H, i.e.,

$$P(x) = 0 \qquad \text{for all } x \in H.$$

Equivalently, P is divisible by the vanishing polynomial $Z_H(X)$, so there exists a polynomial $Q(X) \in \mathbb{F}[X]$ such that

$$P(X) = Q(X)\, Z_H(X).$$

Suppose we are given three vectors

$$A = (a_0, \dots, a_{N-1}), \qquad B = (b_0, \dots, b_{N-1}), \qquad C = (c_0, \dots, c_{N-1})$$

indexed by the trace domain points

$$x_i = \omega^i \in H, \qquad i = 0, \dots, N-1.$$

Let $f(X), g(X), C(X) \in \mathbb{F}[X]$ denote the unique interpolating polynomials of degree $< N$ such that

$$f(x_i) = a_i, \qquad g(x_i) = b_i, \qquad C(x_i) = c_i$$

for all $x_i \in H$. To enforce the pointwise relation

$$a_i + b_i = c_i \qquad \text{for all } i = 0, \dots, N-1,$$

define

$$P(X) := f(X) + g(X) - C(X).$$

If the relation holds on H, then $P(x) = 0$ for all $x \in H$, and therefore $Z_H(X)$ divides $P(X)$.

Once a polynomial $f(X) \in \mathbb{F}[X]$ is fixed, it can be evaluated at any field element $x \in \mathbb{F}$. In particular, one may consider its evaluations on different subsets of $\mathbb{F}$, such as the trace domain H, the larger subgroup H_{lde}, or the shifted coset $s \cdot H_{\text{lde}}$. Passing from H to one of these larger sets does not change the polynomial itself, it only changes the set of points at which its values are sampled. Therefore, the algebraic degree of the polynomial remains the same, even though its evaluation vector becomes longer.

Another common technique is to enforce constraints at specific points by introducing a linear factor. Let $x_0 \in H$ be a designated evaluation point, for example $x_0 = \omega^i$, and let $a \in \mathbb{F}$ be a target value. Consider

$$P(X) := f(X) - a.$$

The condition $f(x_0) = a$ is equivalent to $P(x_0) = 0$, that is, to x_0 being a root of P. Therefore, $(X - x_0)$ divides $P(X)$, so there exists a polynomial $Q(X) \in \mathbb{F}[X]$ such that

$$P(X) = (X - x_0)\, Q(X).$$

This gives a simple algebraic interpretation of boundary constraints: requiring a polynomial to take a prescribed value at a designated point is equivalent to requiring the corresponding difference polynomial to have that point as a root.

2.3 NTT/INTT and the evaluation representation

A zkVM execution trace can be viewed as a two-dimensional matrix $T \in \mathbb{F}^{N \times m}$, where each row corresponds to a time step and each column records a register or an intermediate value. In algebraic intermediate representations (AIRs), it is convenient to treat each trace column as the evaluation of a polynomial over the trace domain $H = \langle \omega \rangle$ of size N. Consequently, we need an efficient method to convert between (i) the *evaluation form* of a column, i.e., the vector $(f(\omega^i))_{i=0}^{N-1}$, and (ii) the *coefficient form* of the corresponding polynomial $f(X) \in \mathbb{F}[X]$. Over radix-2 domains this conversion is performed using the finite-field FFT, commonly referred to as the number-theoretic transform (NTT), and its inverse (INTT).

A key property of the evaluation form is that pointwise operations correspond to the same operations on the underlying polynomials. For example, given evaluation vectors $(f(\omega^i))_{i=0}^{N-1}$ and $(g(\omega^i))_{i=0}^{N-1}$, the pointwise sum $(f(\omega^i) + g(\omega^i))_{i=0}^{N-1}$ is exactly the evaluation vector of the polynomial $f(X) + g(X)$ on H. This allows many constraint expressions to be computed efficiently by working directly with evaluation vectors.

Finally, quotienting by the vanishing polynomial must be handled carefully. If a constraint polynomial $P(X)$ is required to vanish on H, then $P(x) = 0$ for all $x \in H$, and the vanishing polynomial satisfies $Z_H(x) = 0$ for all $x \in H$ as well. Therefore, expressions such as $P(X)/Z_H(X)$ cannot be evaluated *on* H because they take the indeterminate form $0/0$ at every point of the domain. For this reason, quotient polynomials are typically evaluated on a larger low-degree extension domain, often on a shifted coset $s \cdot H_{\text{lde}}$ where $s \notin H_{\text{lde}}$, so that $Z_H(x) \neq 0$ on the evaluation points and the division is well-defined.

2.4 Commitments and Merkle trees

A Merkle tree provides a compact commitment to a vector of data items while allowing efficient openings of individual positions. In this thesis, the committed data is typically a vector of field elements, such as a column of evaluations of a polynomial f over the trace domain $H = \langle \omega \rangle$:

$$\mathbf{a} = (a_0, \ldots, a_{N-1}) \quad \text{where } a_i = f(\omega^i).$$

We require opening only a small number of positions while keeping the commitment size equal to a single hash digest.

Definition 2 (Merkle commitment[14])**.** Let $\mathcal{H}$ be a cryptographic hash function and let $\mathbf{a} = (a_0, \ldots, a_{N-1})$ be a vector of field elements, where $N = 2^\ell$. Arrange the elements of $\mathbf{a}$ as the leaves of a full binary tree of height ℓ, with a_i assigned to the i-th leaf.

Define the leaf hashes as

$$L_i := \mathcal{H}(\mathrm{enc}(a_i)),$$

where $\mathrm{enc}(\cdot)$ is a fixed-length encoding into bytes. Thus, the hash of the value stored at the i-th leaf is L_i.

For each internal node u, let $\mathrm{left}(u)$ and $\mathrm{right}(u)$ denote the left and right child of u, respectively. The value at each internal node is computed recursively as

$$N_u := \mathcal{H}\big(N_{\mathrm{left}(u)} \,\|\, N_{\mathrm{right}(u)}\big),$$

where, at the leaf level, the node values are the corresponding leaf hashes L_i. The value at the root, denoted root, is the Merkle commitment to $\mathbf{a}$.

Definition 3 (Authentication path)**.** For an index i, an authentication path is the sequence of sibling hashes along the path from the leaf corresponding to a_i to the root. Given (i, a_i) and the authentication path, the verifier first computes

$$L_i = \mathcal{H}(\mathrm{enc}(a_i)),$$

then recomputes the hashes on the path to the root in $O(\log N)$ hash evaluations, and accepts if the recomputed root equals the committed root.

Merkle commitments are efficient: the commitment consists of a single hash value, while an opening for one entry has size $O(\log N)$ and verification time $O(\log N)$. The binding property follows from collision resistance of $\mathcal{H}$: if an adversary can provide two different valid openings for the same index i under the same root, then one can derive a collision in $\mathcal{H}$.

2.5 Fiat–Shamir transcript and challenges.

Many zkVM proof systems are derived from interactive protocols in which the verifier supplies random challenges that the prover must answer. In practice, these challenges are generated non-interactively using the Fiat–Shamir transform [11, Sec 5.2], which replaces verifier randomness with hash outputs and is typically analyzed in the random-oracle model.

Concretely, the prover maintains a transcript (an ordered byte string) that absorbs public parameters, the statement being proved, and all commitments produced during the protocol. Whenever a verifier challenge is required, the prover derives it deterministically as a hash of the current transcript, e.g.,

$$\beta := \mathcal{H}(\mathsf{tr}),$$

and continues the protocol using β as the challenge. Under the random-oracle heuristic, the derived challenges are treated as unpredictable to any prover that has not yet fixed the preceding transcript contents.

2.6 FRI low-degree testing and the STARK IOP

We now provide a high-level overview of how STARK-style zkVMs verify correct execution. The prover executes the program and records a trace table $T \in F^{N\times m}$. An AIR specifies a collection of constraint polynomials that must vanish on the trace domain H when evaluated on the honest trace. The prover extends the relevant trace columns and constraint expressions to a larger low-degree extension domain and typically works over a shifted coset $s \cdot H_{\text{lde}}$ (with $s \notin H_{\text{lde}}$), which avoids obstacles caused by the vanishing polynomial Z_H and supports low-degree testing.

To reduce many constraints to a single check, the verifier samples random coefficients (via Fiat–Shamir) and the prover forms a *composition polynomial* $P(X)$ as a random linear combination of the individual constraint polynomials. The constraints are satisfied on H if and only if $P(x) = 0$ for all $x \in H$, which is equivalent to divisibility by the vanishing polynomial:

$$Z_H(X) \mid P(X).$$

Accordingly, one defines a quotient polynomial

$$Q(X) := \frac{P(X)}{Z_H(X)},$$

which is well-defined if the constraints hold. Moreover, under standard degree bounds on the AIR, $Q(X)$ has degree bounded by a value polynomial in N.

The STARK interactive oracle proof then reduces soundness to a *low-degree test*. The prover commits to the low-degree extension evaluations of the trace columns on the chosen low-degree Extension (LDE) domain. Precisely, each leaf of the trace commitment contains a *row* of LDE evaluations, for example

$$\big(f_1(x), \ldots, f_m(x)\big) \quad \text{for } x \in s \cdot H_{\text{lde}},$$

where f_j denotes the interpolant corresponding to the j-th trace column. The prover also commits to the LDE evaluations of the quotient polynomial $Q(X)$ (or, equivalently, to the composition polynomial together with the required normalization), evaluated on the same domain.

Using Fiat–Shamir, the verifier samples a small number of random query indices. For each queried position, the prover opens the corresponding leaves in the Merkle trees and provides the authentication paths. The verifier checks the Merkle openings and then locally recomputes the relevant constraint expressions at the queried points from the opened trace values and, for transition constraints, from the opened values at the next step.

These recomputed values are combined using the verifier's random coefficients into the expected evaluation of the composition polynomial and are checked to be consistent with the opened evaluation of Q via the relation $P(X) = Z_H(X)\, Q(X)$ at the queried points. Finally, the verifier runs the FRI protocol on the committed evaluations to test that Q (or the appropriate combined polynomial required by the protocol) has the claimed low degree.

Intuitively, if a cheating prover alters the trace so that the constraints fail on H, then the resulting composition polynomial $P(X)$ with high probability will not

be divisible by $Z_H(X)$. Consequently, no low-degree quotient $Q(X)$ can satisfy $P(X) = Z_H(X)\,Q(X)$ on the queried points except with small probability, and the low-degree test rejects with high probability.

3 Arithmetic Intermediate Representation

STARK-style zkVMs execute a program and record the resulting computation in an *execution trace* $T \in \mathbb{F}^{N\times m}$, where N is the number of steps (rows) and m is the number of trace columns (e.g., registers and auxiliary values). Each trace column is represented by a function (or polynomial) t_j evaluated on H, such that for each column index $j \in \{0, 1, \dots, m-1\}$.

$$t_j(\omega^i) = T[i, j] \qquad \text{for } i \in \{0, 1, \dots, N-1\}.$$

Correct execution is enforced by a collection of algebraic constraints over these column functions. Informally, the prover must show that the constraints *hold on* H equivalently, that the corresponding constraint polynomials *vanish on* H, which implies that the trace is consistent with the VM transition rules and boundary conditions.

Constraints are typically divided into *boundary constraints* and *transition constraints*. Boundary constraints restrict designated rows (e.g., the initial and final state), whereas transition constraints enforce that consecutive rows follow the VM transition function. Using the "previous-row" notation, transition constraints are expressed in terms of $t_j(X)$ and $t_j(\omega^{-1}X)$.

As a simple example, consider a time-step column t_{ts} that should start at 1 and increment by 1 at each step. We require the boundary condition $t_{\text{ts}}(\omega^0) = 1$ and the transition condition

$$t_{\text{ts}}(\omega^i) = t_{\text{ts}}(\omega^{i-1}) + 1 \qquad \text{for } i \geq 1,$$

which corresponds to the transition constraint polynomial

$$C_{\text{ts,trans}}(X) \;=\; t_{\text{ts}}(X) - t_{\text{ts}}(\omega^{-1}X) - 1.$$

This constraint is intended to apply only for rows $i \geq 1$, so it must be disabled on the first row. A common approach is to *gate* constraints using *control* (selector) columns. Let $s_{\text{trans}}(X)$ be a fixed binary selector such that $s_{\text{trans}}(\omega^0) = 0$ and $s_{\text{trans}}(h) = 1$ for all $h \in H \setminus \{\omega^0\}$. We then enforce the gated transition constraint

$$\widetilde{C}_{\text{ts,trans}}(X) \;=\; \big(t_{\text{ts}}(X) - t_{\text{ts}}(\omega^{-1}X) - 1\big) \cdot s_{\text{trans}}(X),$$

which vanishes on all of H if and only if the increment rule holds for all non-initial rows.

Similarly, boundary constraints can be expressed using a selector $s_{\text{init}}(X)$ that equals 1 at $X = \omega^0$ and 0 elsewhere on H. The condition $t_{\text{ts}}(\omega^0) = 1$ becomes

$$C_{\text{ts,init}}(X) \;=\; \big(t_{\text{ts}}(X) - 1\big) \cdot s_{\text{init}}(X).$$

Selectors are part of the AIR specification and are not chosen by the prover; otherwise, setting them to 0 would trivially satisfy the constraints.

Finally, the verifier combines all (gated) constraints into a *composition polynomial*

$$C(X) \;=\; \sum_i \beta_i \, C_i(X),$$

using random coefficients β_i. If all constraints vanish on H, then the vanishing polynomial $Z_H(X)$ divides $C(X)$, and the prover defines the *quotient polynomial*

$$Q(X) \;=\; \frac{C(X)}{Z_H(X)}.$$

3.1 Constructing the quotient polynomial in evaluation form

Recall that the prover forms the quotient polynomial

$$Q(X) = \frac{C(X)}{Z_H(X)},$$

where $C(X)$ is the random linear combination of all constraint polynomials. Since $Z_H(x) = 0$ for all $x \in H$, the quotient cannot be evaluated on H. Instead, all relevant polynomials are evaluated on the shifted low-degree extension domain $s \cdot H_{\text{lde}}$. For every point $z \in s \cdot H_{\text{lde}}$, one has $Z_H(z) \neq 0$, so the division is well-defined.

Here, $H_{\text{lde}} = \langle \omega_{\text{lde}} \rangle$ is the multiplicative subgroup of size bN, generated by an element $\omega_{\text{lde}} \in \mathbb{F}^*$ of order bN. We index the evaluation points of the shifted coset as

$$z_i := s \cdot \omega_{\text{lde}}^i \in s \cdot H_{\text{lde}}, \qquad i = 0, \dots, bN - 1.$$

Since the low-degree extension domain has blowup factor b, one original trace step corresponds to an index shift by b. Thus, under the previous-row convention, the value from the previous trace row at z_i is read at z_{i-b}, with indices taken modulo bN.

To obtain low-degree extension evaluations of a trace column t_j, the prover first recovers the coefficient representation of the unique degree $< N$ polynomial agreeing with t_j on H using the inverse number-theoretic transform. It then evaluates this polynomial on $s \cdot H_{\text{lde}}$ using the number-theoretic transform. This yields the extended trace table

$$T_{\text{lde}} \in \mathbb{F}^{(bN) \times m}.$$

Constraints are then evaluated pointwise on $s \cdot H_{\text{lde}}$. For the time-step example, the gated constraints are evaluated as

$$C_{\text{trans}}(z_i) = \big(t_{\text{ts}}(z_i) - t_{\text{ts}}(z_{i-b}) - 1\big) \cdot s_{\text{trans}}(z_i), \qquad C_{\text{init}}(z_i) = \big(t_{\text{ts}}(z_i) - 1\big) \cdot s_{\text{init}}(z_i),$$

and combined into

$$C(z_i) \;=\; \beta_0 \, C_{\text{trans}}(z_i) + \beta_1 \, C_{\text{init}}(z_i).$$

Finally, the prover computes the quotient evaluations

$$Q(z_i) \;=\; \frac{C(z_i)}{Z_H(z_i)} \qquad \text{for all } z_i \in s \cdot H_{\text{lde}},$$

commits to the evaluation vector of Q, and uses FRI to prove that these evaluations are consistent with a polynomial of the claimed low degree. Upon queries, the prover opens the required trace and quotient values, including both $t_{\text{ts}}(z_i)$ and $t_{\text{ts}}(z_{i-b})$ for transition constraints, and the verifier recomputes $C(z_i)$ and checks that

$$Q(z_i) = \frac{C(z_i)}{Z_H(z_i)}$$

at the queried points.

4 Cryptographic Backend

This section defines the core components of the cryptographic backend used throughout proof generation and verification: the Fiat–Shamir transcript, the Merkle-tree commitment scheme, and the FRI protocol. Together, these components provide the cryptographic interfaces required by the prover and verifier.

4.1 Merkle-tree commitment

A Merkle-tree commitment scheme provides algorithms to (i) commit to a vector of leaves and obtain a short commitment (the Merkle root), and (ii) open a leaf at a chosen index by producing an authentication path that can be verified against the root.

We instantiate the hash function $\mathcal{H}$ with BLAKE3 [15]. Leaves are computed by hashing a canonical byte encoding of the committed data. In our implementation, the committed object is typically a vector of leaves representing (i) a single column of evaluations, or (ii) a row containing multiple column values at the same evaluation point (depending on the protocol step). Concretely, for a leaf payload $m_i \in \{0,1\}^*$, the leaf digest is

$$L_i := \mathcal{H}(m_i),$$

and internal nodes are computed as in Definition 2.

The Merkle API exposes the following operations: (i) Commit, which builds the tree over $(m_0, \ldots, m_{N-1})$ and returns the root digest; (ii) Open, which returns an authentication path for a requested index i; and (iii) Verify, which checks an opening against the root digest. The corresponding code is provided in Appendix A.

4.2 Fiat–Shamir transcript

The Fiat–Shamir transcript deterministically derives verifier challenges from the sequence of messages exchanged in the corresponding interactive protocol. Concretely, the prover absorbs all commitments and public values into the transcript state and then derives challenge values by applying a cryptographic hash function to the current transcript state, interpreting the resulting digest as pseudorandom bytes or field elements. The verifier recomputes the same challenges by replaying the same absorption order on the received proof data.

In our implementation, the transcript is instantiated using BLAKE3. We use explicit domain separation for initialization, absorption, and challenge derivation, and we maintain a counter to ensure that successive challenges derived under the same label are distinct. The application programming interface (API) provides methods to absorb byte strings, field elements via canonical serialization, and fixed-size digests, as well as methods to derive challenges as byte strings, field elements, or indices in $[0, n)$. The implementation is given in Appendix B.

4.3 FRI low-degree test

We implement the FRI protocol as the low-degree testing component of the STARK backend. FRI allows the verifier to check, with high probability, that a committed evaluation vector is consistent with a polynomial of degree at most d, while reading only a small number of positions. This is achieved by iteratively *folding* the evaluation vector, committing to each folded layer with a Merkle tree, and verifying a small number of Merkle openings per layer.

Our implementation follows the standard commit query structure. In each round, the prover Merkle-commits to the current layer of evaluations and the transcript derives a random folding challenge β_i. The prover then folds pairs of evaluations into a new layer of half the size. After repeating this process until the claimed degree bound becomes sufficiently small, the prover sends the coefficients of the final polynomial of degree at most r, where r is the remainder bound.

During verification, the verifier uses the transcript to recompute the same challenges, checks the Merkle openings for the queried indices in every layer, and checks that the folded values are consistent across layers and with the evaluations of the final polynomial on the final domain.

For readability, Appendix C includes the core `prove`, `verify`, and `fold_once` routines. The full implementation, including transcript integration, Merkle utilities, and tests, is available in the accompanying repository [16].

4.4 High-level protocol flow

We summarize the high-level flow of the implemented STARK-style protocol. The prover and verifier first agree on public parameters: the trace domain $H = \langle \omega \rangle$ of size N, an LDE domain H_{lde} of size $N_{\text{lde}} = bN$ (for some blowup factor b), a coset shift $s \in \mathbb{F}^*$ such that sH_{lde} is disjoint from H, and FRI parameters consisting of the claimed maximum degree d, the remainder bound r, and the number of queries q. They also fix the AIR constraints that define validity of an execution trace $T \in \mathbb{F}^{N \times m}$.

Prover.

1. Initialize the Fiat–Shamir transcript with the public parameters.
2. Execute the program and construct the trace table T over H.
3. Compute the low-degree extension of the trace over the shifted coset sH_{lde}, obtaining an evaluation table $T_{\text{lde}} \in \mathbb{F}^{N_{\text{lde}} \times m}$. Commit to T_{lde} with a Merkle

tree whose leaves are row encodings, and obtain the root digest r_0. Send r_0 to the verifier and absorb r_0 into the transcript.

4. Derive random challenges (denoted $\beta_1, \ldots, \beta_k$) from the transcript and form the composition polynomial $C(X)$ as the specified random linear combination of constraint polynomials.

5. Evaluate C over sH_{lde} and compute the quotient evaluations

$$Q(x) := \frac{C(x)}{Z_H(x)} \qquad \text{for } x \in sH_{\text{lde}}$$

Commit to the evaluation vector of Q (if committed separately) send the commitment root r_1 to verifier and absorb r_1 into the transcript.

6. Run FRI on the committed evaluations of Q to prove that Q is of low degree. Produce a FRI proof together with the required Merkle openings.

7. For each queried index selected by FRI, provide Merkle authentication paths for the corresponding rows of T_{lde} needed to evaluate the constraints, current and previous-step rows, depending on the transition constraints.

Verifier.

1. Initialize the transcript with the same public parameters and absorb the received r_0.

2. Recompute the challenges $\beta_1, \ldots, \beta_k$ from the transcript.

3. Verify all provided Merkle openings against the corresponding root r_0, recovering the queried rows of T_{lde}.

4. Using the opened trace values, recompute the constraint evaluations and the corresponding composition value $C(x)$ at each queried point $x \in sH_{\text{lde}}$, and check consistency with the quotient relation

$$C(x) = Q(x)\, Z_H(x) \qquad \text{at the queried points.}$$

5. Verify the FRI proof to ensure the opened evaluations are consistent with a polynomial of degree at most the claimed bound.

5 System Design and Implementation

This chapter describes the design decisions behind the `toy-zkVM` implementation. It motivates the choice of programming language and external dependencies, briefly outlines plausible alternatives, and summarizes the project structure by explaining the role of each module. The chapter concludes with two example programs that demonstrate the integrity of the Fibonacci computation.

Programming language. As indicated in the title, the implementation is written in *Rust*. Rust is widely used in modern zero-knowledge engineering, and its ecosystem provides mature libraries for cryptographic development, including finite-field arithmetic, polynomials, and NTTs. In addition, Rust is a systems programming language that enables low-level control and high performance without sacrificing memory safety. Several prominent zkVM projects, such as RISC Zero, SP1, and Jolt, are implemented in Rust as well.[9, 10, 17]

Alternative languages. The same project could be implemented in several other languages. C and C++ are long-standing choices for performance-critical cryptography due to their predictable and low-level control. Go also provides practical tooling for cryptographic implementations; for example, the `gnark` ecosystem supports zero-knowledge proof development and offers solid performance.[18] For an educational prototype, higher-level languages such as JavaScript or Python can reduce development friction, although typically at the cost of performance and stricter control over memory and arithmetic.

Crate selection. The implementation relies on external crates to streamline the development of core primitives. In particular, the `arkworks` ecosystem provides reusable abstractions for finite fields, radix-2 evaluation domains, and NTT-based polynomial operations, as well as concrete field instantiations (e.g., BN254) suitable for testing.[19, 22] As an alternative design point, the `winterfell` library offers a STARK-oriented backend that is also suitable for educational implementations.[20] Finally, `Merlin` provides a Fiat–Shamir transcript implementation that can serve as a drop-in replacement for a custom transcript.[21]

5.1 Project structure

The codebase is organized into the following modules: `air`, `backend`, `zkvm`, `examples`, and `test_utils`. This separation isolates the AIR specification, the proving backend, and the high-level zkVM API, while keeping reusable utilities and demonstrations in dedicated locations.

The `air` module defines the abstractions and data structures required to specify an AIR instance, including the `AIR` trait, the `Constraint` trait, and the execution trace representation (the trace table), together with helper functions for constraint evaluation.

The `backend` module implements the proving and verification procedures described in Section 4. It provides the low-level components needed by the zkVM, and exposes a backend interface consumed by the higher-level `zkvm` module.

The `test_utils` module contains shared testing helpers used across the test suite, such as `pick_coset_shift` and `pick_domain`. Finally, the `zkvm` module composes the AIR definition and the backend into a *prover* and a *verifier*.

5.2 Example overview

To illustrate the end-to-end workflow, we consider an AIR that models the Fibonacci recurrence. The execution trace consists of three columns: a time-step column t, and two value columns A and B. At row i, the columns encode

$$A(i) = a_{i-1}, \qquad B(i) = a_i,$$

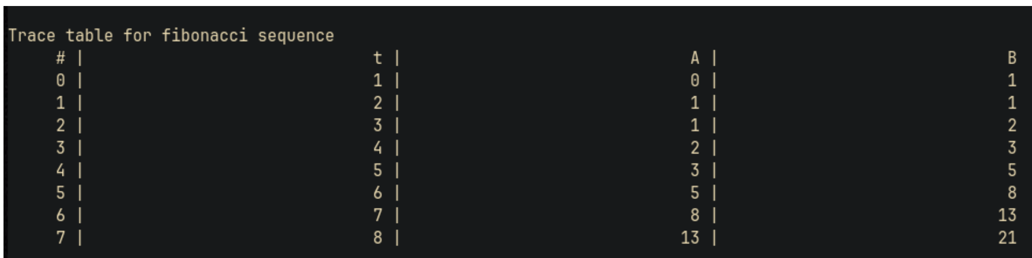
where $(a_i)_{i\geq 0}$ is the Fibonacci sequence determined by the chosen initial values. The time-step column t is initialized to 1 and increments by one at each step.

Although this example is simple, it is well-suited for exposition because it uses both boundary constraints and transition constraints. By the end of this section, we demonstrate computational integrity for this AIR by showing that a prover can convince a verifier that the committed trace satisfies all constraints, without revealing any additional information beyond the public statement.

5.3 Trace implementation

As introduced earlier, the execution trace is a matrix $T \in \mathbb{F}^{N\times m}$. In the implementation, the trace is represented by a lightweight wrapper around a `Vec<Vec<F>>`, i.e., a vector of column vectors. Optionally, the trace stores column names to improve debug output.

The `Trace` constructor enforces basic structural invariants: all columns must have the same length m, and the length is required to be a power of two, this restriction simplifies the use of radix-2 evaluation domains in the backend. For the Fibonacci example, the trace has three columns and can be printed in a tabular form for inspection (see 2).



Trace table for fibonacci sequence

#	t	A	B
0	1	0	1
1	2	1	1
2	3	1	2
3	4	2	3
4	5	3	5
5	6	5	8
6	7	8	13
7	8	13	21

Figure 2: Trace table for the Fibonacci sequence length 8.

5.4 AIR and constraints

The AIR is expressed as a Rust trait that specifies (i) the metadata of an AIR instance (e.g., number and names of constraints), and (ii) the row-wise evaluation procedure used by the prover during quotient construction.

To decouple constraint logic from the concrete trace representation, we introduce a `RowAccess` trait that provides a uniform API for reading the values required by constraints at a given row index (and, when needed, previous row). The prover implements `RowAccess` over the full trace, whereas the verifier implements `RowAccess`

over the *opened* values returned by the prover, the values revealed at queried positions, which enables local recomputation of constraint evaluations at those positions.

Each individual constraint is represented by the `Constraint` trait, which defines a name and an `eval` method. In a concrete AIR implementation, constraints are stored as a list (e.g., `Vec<Box<dyn Constraint<F>>>`). During quotient construction, the prover iterates over all row indices i in the evaluation domain, evaluates every constraint on row i via the `RowAccess` API, and combines these evaluations according to the constraint composition rule. This yields the per-row evaluations of the quotient polynomial. A complete example is provided in Appendix D.

5.5 Prover and verifier pipeline

The `zkvm` module defines the shared public parameters, the prover, and the proof format used by the verifier. Public parameters are stored in a `ZkvmPublicParameters` structure and include the trace evaluation domain H, the low-degree extension (LDE) domain H_{lde}, the multiplicative shift s defining the disjoint coset $s \cdot H_{\text{lde}}$, and the FRI options. In addition, the module defines transcript labels that domain-separate all absorbed values and challenges. A dedicated method `seed_tx` deterministically absorbs the public parameters into the transcript to bind the proof to these choices. For concrete implementation see Appendix E.

Prover. The prover implementation consists of: (i) a `RowAccess` view over the shifted LDE trace (`ProverRowLdeView`), (ii) a dedicated error type (`ZkvmProveError`), (iii) the proof object (`ZkvmProof`), and (iv) helper routines for committing to the trace and generating openings.

Given a concrete trace table T and an AIR instance, the `prove` function proceeds as follows.

1. **Validate inputs.** The prover checks that the trace length matches the size of the trace domain and that the LDE domain size is a multiple of the trace-domain size (defining the blow-up factor).

2. **Seed the transcript.** The prover absorbs the public parameters into the transcript via `seed_tx`.

3. **Compute the shifted LDE trace and commit.** Each trace column is interpolated over H, its coefficients are scaled so that the resulting polynomial is evaluated on the shifted coset $s \cdot H_{\text{lde}}$, and the column is then evaluated over H_{lde}. The prover hashes each row of the resulting LDE table into a Merkle leaf and commits to the table by publishing the Merkle root r_0, which is absorbed into the transcript.

4. **Sample mixing challenges.** The prover derives a sequence of random field elements $\beta_1, \ldots, \beta_k$ (one per constraint) from the transcript.

5. **Compute the composition evaluations.** For each point $x \in s \cdot H_{\text{lde}}$, the prover constructs a `RowAccess` view that exposes the current row, the previous

step row, and auxiliary values such as $Z_H(x)^{-1}$. It then evaluates the AIR composition function to obtain a vector of values $Q(x)$ over $s \cdot H_{\text{lde}}$.

6. **Run FRI and generate openings.** The prover invokes FRI on the committed evaluations $Q(x)$ to prove that they are consistent with a low-degree polynomial. FRI determines a set of query indices; the prover opens the corresponding rows of the committed LDE trace and also opens the matching "previous step" rows required for transition constraints. These openings include the row values and their Merkle authentication paths.

The prover outputs a `ZkvmProof` object containing the trace commitment $\texttt{root}_T$, the commitment(s) used by the FRI protocol for $Q(x)$, the FRI proof itself, and the trace openings required for verification. For readability we provide implementation of the core `prove` routine and the `ZkvmProof` in Appendix F, remaining codebase can be found in the project repository [16].

Verifier The verifier implementation consists of: (i) a `RowAccess` view over the opened trace rows (`VerifierRowLdeView`), (ii) an error type describing failure modes (`ZkvmVerifyError`), and (iii) the core verification routine `verify`.

The verifier begins by replaying the transcript: it absorbs the public parameters and the trace commitment root $\texttt{root}_0$ from the proof, and then re-derives the mixing challenges $\beta_1, \ldots, \beta_k$ via the Fiat–Shamir transform. It additionally checks basic proof consistency (e.g., that the number of trace openings matches the number of FRI queries).

For each query index i determined by FRI, the verifier performs two checks. First, it validates the Merkle authentication paths for the opened trace rows: the current row at index i and the corresponding previous-step row. Second, using only the opened row values, the verifier reconstructs the composition evaluation at the queried point $x = s \cdot \omega_{\text{lde}}^i$ by invoking `eval_composition` through the `RowAccess` interface. It then checks that this locally recomputed value matches the value opened by the FRI proof at the same index.

Finally, the verifier runs the FRI verification procedure. It accepts the proof if (i) all Merkle openings are valid, (ii) all queried composition values match the FRI openings, and (iii) the FRI proof verifies successfully. A complete implementation is given in Appendix G.

5.6 Basic Fibonacci Example

We now demonstrate the basic Fibonacci sequence proving process using the described STARKish protocol, which utilizes linear factorization for boundary constraints. As mentioned earlier, we prove the correct Fibonacci recurrence over two columns, A and B, and the time-step column t. The recurrence is represented using two columns and a one-step time shift to enable referencing two consecutive steps and satisfy the recurrence relation: $B(i) = A(i) + A(i-1)$.

Thus, the constraints required to prove computational integrity are presented below (see Table 1). Constraints are designed as structs implementing the `Constraint` trait; in this way, internal fields store the indices of referenced columns and constant

values used during computation, for example, the final sequence value. The full constraint implementation for sequence length 128 can be found in Appendix H. A complete proving and verification procedure can be found in the thesis's repository.

Table 1: Constraints for the basic Fibonacci example

Constraint Name	Polynomial Relation	Purpose
TimeStepBoundary	$\frac{t_i - 1}{x - \omega_0}$	Ensure the time-step column starts from 1 at the first row ($t_0 = 1$).
TimeStepTransition	$\frac{(x - \omega_0)\,(t_i - t_{i-1} - 1)}{z_H(x)}$	Enforce the time-step transition: increment by one between consecutive rows ($t_i = t_{i-1} + 1$).
ATransition	$\frac{(x - \omega_0)\,(A_i - B_{i-1})}{z_H(x)}$	Enforce the shift relation for the Fibonacci state: current A equals previous B ($A_i = B_{i-1}$).
BTransition	$\frac{(x - \omega_0)\,(B_i - B_{i-1} - A_{i-1})}{z_H(x)}$	Enforce the Fibonacci recurrence in the state transition ($B_i = B_{i-1} + A_{i-1}$).
FinalValueBoundary	$\frac{B_i - F_{\text{final}}}{x - \omega_{\text{last}}}$	Enforce the final Fibonacci value in column B at the last row.

As mentioned earlier, using linear factorization for boundary enforcement instead of selector columns introduces inconveniences in computation. Turning off a boundary constraint at a specific index requires multiplication by $(x - \omega_i)$ to nullify the constraint at index i. Additionally, if not nullified, all traces and thus all sequences are enforced to have radix-two length. This affects the scalability of the zkVM and the range of statements it can prove.

5.7 Padded Fibonacci Example

To address the limitations observed in the previous example, we introduce selector columns. These columns are populated with binary values (0 or 1), depending on the statement being proved. Thus, the boundary selector column, and hence the selector polynomial $s_{\text{b}}(x)$ used to enforce boundary constraints, is 1 at ω_0 and 0 at all other indices. In contrast, the transition selector column and $s_{\text{tr}}(x)$ are 1 during the actual computation steps and 0 at ω_0 and during computation-unrelated steps such as padding.

The constraints for this example are listed below (see Table 2). In addition to the previous example, we add `AInit`, `BInit`, and `TerminationValue` to enforce initialization and termination values. Additionally, we enforce booleanity of the selector columns using a booleanity constraint applied to multiple columns. The corresponding code implementation is provided in Appendix I.

In addition to transition enforcement, the polynomial $s_{\text{tr}}(x)$ enables proving statements of arbitrary length. For example, generating a proof for a Fibonacci

Constraint Name	**Polynomial Relation**	**Purpose**
TimeStepBoundary	$\frac{(t_i - 1)\, s_{\mathrm{b}}(x)}{z_H(x)}$	Enforce the initial time-step value at the row selected by the boundary selector (initial row): $t_i = 1$.
TimeStepTransition	$\frac{(t_i - t_{i-1} - 1)\, s_{\mathrm{tr}}(x)}{z_H(x)}$	Enforce the time-step transition on transition rows: increment by one between consecutive rows ($t_i = t_{i-1} + 1$).
ATransition	$\frac{(A_i - B_{i-1})\, s_{\mathrm{tr}}(x)}{z_H(x)}$	Enforce the Fibonacci state shift on transition rows: current A equals previous B ($A_i = B_{i-1}$).
BTransition	$\frac{(B_i - B_{i-1} - A_{i-1})\, s_{\mathrm{tr}}(x)}{z_H(x)}$	Enforce the Fibonacci recurrence on transition rows: $B_i = B_{i-1} + A_{i-1}$.
TerminationValue	$\frac{(B_i - F_{\mathrm{final}})\, s_{\mathrm{term}}(x)}{z_H(x)}$	Enforce the final Fibonacci value in column B at the row selected by the termination selector.
AInit	$\frac{(A_i - A_{\mathrm{init}})\, s_{\mathrm{b}}(x)}{z_H(x)}$	Enforce the initial value of column A at the row selected by the boundary selector.
BInit	$\frac{(B_i - B_{\mathrm{init}})\, s_{\mathrm{b}}(x)}{z_H(x)}$	Enforce the initial value of column B at the row selected by the boundary selector.
Booleanity (transition)	$\frac{s_{\mathrm{tr}}(x)\, \big(s_{\mathrm{tr}}(x) - 1\big)}{z_H(x)}$	Enforce that the transition selector is boolean.
Booleanity (boundary)	$\frac{s_{\mathrm{b}}(x)\, \big(s_{\mathrm{b}}(x) - 1\big)}{z_H(x)}$	Enforce that the boundary selector is boolean.
Booleanity (termination)	$\frac{s_{\mathrm{term}}(x)\, \big(s_{\mathrm{term}}(x) - 1\big)}{z_H(x)}$	Enforce that the termination selector is boolean.

Table 2: Constraints for the Fibonacci example with selector (control) columns

sequence of length 17 is possible by embedding the sequence into the next power-of-two-sized trace (32 in this case) and padding the remaining 15 cells with arbitrary values, since the AIR is still satisfied when $s_{\text{tr}}(x) = 0$ for all trace-unrelated steps (see 3).

#	t	A	B	transition control	init control a	termination control
0	1	70492524767089125814114	114059301025943970552219	0	1	0
1	2	114059301025943970552219	184551825793033096366333	1	0	0
2	3	184551825793033096366333	298611126818977066918552	1	0	0
3	4	298611126818977066918552	483162952612010163284885	1	0	0
4	5	483162952612010163284885	781774079430987230203437	1	0	0
5	6	781774079430987230203437	1264937032042997393488322	1	0	0
6	7	1264937032042997393488322	2046711111473984623691759	1	0	0
7	8	2046711111473984623691759	3311648143516982017180081	1	0	0
8	9	3311648143516982017180081	5358359254990966640871840	1	0	0
9	10	5358359254990966640871840	8670007398507948658051921	1	0	0
10	11	8670007398507948658051921	14028366653498915298923761	1	0	0
11	12	14028366653498915298923761	22698374052006863956975682	1	0	0
12	13	22698374052006863956975682	36726740705505779255899443	1	0	0
13	14	36726740705505779255899443	59425114757512643212875125	1	0	0
14	15	59425114757512643212875125	96151855463018422468774568	1	0	0
15	16	96151855463018422468774568	155576970220531065681649693	1	0	0
16	17	155576970220531065681649693	251728825683549488150424261	1	0	1
17	85338813629915906997079131501…	85338813629915906997079131501…	85338813629915906997079131501…	0	0	0
18	74617303987960748722165010410…	74617303987960748722165010410…	74617303987960748722165010410…	0	0	0
19	11706452300691740521341308352…	11706452300691740521341308352…	11706452300691740521341308352…	0	0	0
20	11668528978892137012047407078…	11668528978892137012047407078…	11668528978892137012047407078…	0	0	0
21	99614193461916621524339369345…	99614193461916621524339369345…	99614193461916621524339369345…	0	0	0
22	85508777937519128710701422069…	85508777937519128710701422069…	85508777937519128710701422069…	0	0	0
23	73507152327266553422058026395…	73507152327266553422058026395…	73507152327266553422058026395…	0	0	0
24	15577284980634572502532475721…	15577284980634572502532475721…	15577284980634572502532475721…	0	0	0
25	42260312176577134158776050076…	42260312176577134158776050076…	42260312176577134158776050076…	0	0	0
26	19551772625763793777903626610…	19551772625763793777903626610…	19551772625763793777903626610…	0	0	0
27	32381559824127031707627633993…	32381559824127031707627633993…	32381559824127031707627633993…	0	0	0
28	20043158708903312753577701884…	20043158708903312753577701884…	20043158708903312753577701884…	0	0	0
29	17252037973732332567609385166…	17252037973732332567609385166…	17252037973732332567609385166…	0	0	0
30	11588281163203992989423867077…	11588281163203992989423867077…	11588281163203992989423867077…	0	0	0
31	67513299285853416753968210039…	67513299285853416753968210039…	67513299285853416753968210039…	0	0	0

Figure 3: Padded trace table for the Fibonacci sequence length 17.

Finally, it is important to note that control columns should not be part of the witness; instead, they should be public or derived in a deterministic way. If control columns are included in the witness, the prover can trivially satisfy the AIR by setting all selector values to zero.

At this stage, the approach allows proving the integrity of specific computations by manually designing AIR constraints that encode the desired computation. While this is sufficient for demonstrating the core proving workflow, it does not scale well, since each new computation requires a custom trace design and custom constraint definitions. A more scalable approach is to introduce a domain-specific language (DSL) layer that compiles programs into execution traces and automatically generates constraints from a fixed instruction set. This would make it possible to prove the integrity of arbitrary computations expressible in the DSL, rather than only a small set of hand-crafted examples.

5.8 Virtual Machine

As discussed earlier, proving computations expressed in a general-purpose DSL allows engineers to abstract away the manual construction of AIR. This is achieved by defining constraints that enforce instruction semantics, control flow, and valid state transitions. Such an approach makes it possible to prove general-purpose computation in a way that is more accessible to engineers.

A virtual machine provides an isolated environment for program execution. Its state consists of data registers together with metadata, such as the program counter PC and a flag indicating whether execution has halted. Instructions transform this state step by step in order to carry out a computation.

A common approach in zkVM design is to implement constraints for an existing central processing unit (CPU) instruction set, such as RISC-V. In this way, systems such as RISC Zero [9] can verify the execution of programs written in languages

such as Rust or C. However, designing AIR constraints for a full existing instruction set is a substantial task. For the purposes of this thesis, we therefore introduce a toy DSL instead.

DSL and environment. The execution environment consists of four registers, r_0, r_1, r_2, and r_3, each storing a field element. The machine state further includes a program counter *pc* and a halted flag. Instructions operate by reading from and writing to the registers, while control-flow instructions modify *pc* according to their semantics. The halted flag indicates that the machine has terminated execution.

The DSL is expressive enough to represent simple programs, including the computation of the Fibonacci sequence. Its instruction set therefore includes constant initialisation, arithmetic operations, jump instructions, and a halt instruction. The full instruction set is listed in Table 3, while the grammar of the language is shown in extended Backus–Naur form (EBNF) form (see 4).

Table 3: Instruction set of the toy VM DSL

Instruction	**Semantics**
`const rX, c`	Set register $r_X := c$ and advance the program counter, $pc := pc + 1$.
`mov rX, rY`	Copy the value of register r_Y into register r_X and advance the program counter, $pc := pc + 1$.
`add rX, rY`	Update register $r_X := r_X + r_Y$ and advance the program counter, $pc := pc + 1$.
`sub rX, rY`	Update register $r_X := r_X - r_Y$ and advance the program counter, $pc := pc + 1$.
`jmp L`	Unconditionally jump to label L by setting $pc := \text{addr}(L)$.
`jnz rX, L`	If $r_X \neq 0$, jump to label L by setting $pc := \text{addr}(L)$; otherwise advance the program counter, $pc := pc + 1$.
`halt`	Set the halted flag to 1, indicating termination of execution.

program ::= { *line* } `EOF`

line ::= { *newline* }
| *statement newline*
| *statement* `EOF`

statement ::= *label*
| *instruction*

label ::= *identifier* :

instruction ::= *const_instr*
| *mov_instr*
| *add_instr*
| *sub_instr*
| *jmp_instr*
| *jnz_instr*
| *halt_instr*

const_instr ::= `const` *reg* , *number*
mov_instr ::= `mov` *reg* , *reg*
add_instr ::= `add` *reg* , *reg*
sub_instr ::= `sub` *reg* , *reg*
jmp_instr ::= `jmp` *identifier*
jnz_instr ::= `jnz` *reg* , *identifier*
halt_instr ::= `halt`

reg ::= `r0` | `r1` | `r2` | `r3`

identifier ::= *ident_start* { *ident_continue* }
ident_start ::= `_` | `A..Z` | `a..z`
ident_continue ::= *ident_start* | `0..9`

number ::= *digit* { *digit* }
digit ::= `0..9`

newline ::= `\n`

Figure 4: Grammar of the toy VM DSL in EBNF form

Using this instruction set, figure (see 5) presents a simple program that computes $3 + 7$. The registers r_0 and r_1 are initialised with the operands, and the `add` instruction updates r_0 with their sum. Execution is then terminated by the `halt` instruction.

```
const r0, 3
const r1, 7
add r0, r1
halt
```

Figure 5: A simple DSL program computing $3 + 7$

Figure 6 also presents a program for computing the Fibonacci sequence of length 100. The registers r_0 and r_1 hold the initial sequence values, r_2 serves as tempo-

rary storage, and r_3 acts as a loop counter. After each iteration, the counter is decremented, and execution continues while the `jnz` condition holds.

```
const r0, 0
const r1, 1
const r3, 100

loop:
mov r2, r1
add r1, r0
mov r0, r2
const r2, 1
sub r3, r2
jnz r3, loop
halt
```

Figure 6: A DSL program computing the Fibonacci sequence up to length 100

DSL pipeline. The integration of the DSL requires a language-processing pipeline that transforms source code into a form executable by the virtual machine. During execution, the runtime environment records state transitions and converts them into an execution trace, which is then used for proof generation.

Accordingly, the DSL pipeline consists of standard compiler components. First, the lexer transforms the source code into a stream of tokens according to the language delimiters and grammar. Next, the parser groups these tokens into instructions according to the syntactic structure of the language. In addition to these standard stages, the pipeline includes a resolver, whose role is to resolve labels referenced by jump instructions. Thus, passing the source code through the pipeline

$$\texttt{Lexer} \to \texttt{Parser} \to \texttt{Resolver}$$

produces a sequence of instructions ready for execution.

Each instruction transforms the VM state according to its semantics. Execution therefore proceeds step by step, with the program counter determining which instruction is applied at each stage. For ordinary instructions, the program counter advances to the next instruction, whereas control-flow instructions update it according to their defined semantics. Besides executing the program, the runtime must also record the sequence of state transitions together with auxiliary metadata, such as the instruction selector values associated with each row transition.

Execution trace. State transitions alone are not sufficient to prove computational integrity. It is also necessary to record auxiliary metadata, such as instruction selectors, operands, jump targets, and additional columns required for conditional jumps. These values populate the execution-row structure shown in Table 4. Each execution step produces one execution row, and every entry is represented as an element of the field F, which simplifies the conversion of recorded execution into a trace suitable for proof generation.

Column	Description
`pc`	Program counter of the current instruction.
`regs[0]`, `regs[1]`, `regs[2]`, `regs[3]`	Values of registers r_0, r_1, r_2, and r_3 in the current machine state.
`halted`	Halt flag indicating whether execution has terminated.
`s_const`	Selector column for the `const` instruction.
`s_mov`	Selector column for the `mov` instruction.
`s_add`	Selector column for the `add` instruction.
`s_sub`	Selector column for the `sub` instruction.
`s_jmp`	Selector column for the `jmp` instruction.
`s_jnz`	Selector column for the `jnz` instruction.
`s_halt`	Selector column for the `halt` instruction.
`a`	Encoded register index of the first operand, used as destination register or conditional register depending on the instruction.
`b`	Encoded register index of the second operand, used as the source register where applicable.
`imm`	Immediate value used by instructions such as `const`.
`target`	Encoded jump target program counter used by `jmp` and `jnz`.
`jnz_taken`	Boolean flag indicating whether the conditional jump in `jnz` is taken.
`jnz_inv`	Auxiliary inverse value used to constrain the non-zero condition in `jnz`.

Table 4: Columns of the execution row used to construct the execution trace.

Columns prefixed with `s`, such as `s_add`, form a one-hot encoding of the instruction applied at a given step. Therefore, for every valid execution row,

$$s_{\texttt{const}} + s_{\texttt{mov}} + s_{\texttt{add}} + s_{\texttt{sub}} + s_{\texttt{jmp}} + s_{\texttt{jnz}} + s_{\texttt{halt}} = 1.$$

Depending on the active instruction, the additional fields `a`, `b`, `imm`, and `target` are constrained accordingly.

The columns `jnz_taken` and `jnz_inv` are auxiliary fields introduced specifically for the implementation of conditional jumps. In order to encode and constrain conditional branching algebraically, the trace must record both whether the jump was taken and an inverse associated with the condition register. Their precise role is discussed in more detail in the subsection on instruction constraints.

As discussed previously, the trace length must be a power of two. After the VM executes a `halt` instruction, the halted flag is set to one, and the trace is padded by repeating the same halted state until the next power of two is reached. This behaviour is later enforced by constraints as well. It allows programs of arbitrary length to be executed while still producing a trace of valid length for the protocol.

Thus, by consecutively executing instructions and mutating the VM state, the VM produces a sequence of execution rows containing all necessary data. Based on this sequence, the trace table is then populated. A correctly populated trace for the program shown in Figure 5 is presented in Table 5.

#	t	pc	r_0	r_1	r_2	r_3	halted	s_{const}	s_{mov}	s_{add}	s_{sub}	s_{jmp}	s_{jnz}	s_{halt}	a	b	imm	target	jnz_taken	jnz_inv
0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	3	0	0	0
1	2	1	3	0	0	0	0	1	0	0	0	0	0	0	1	0	7	0	0	0
2	3	2	3	7	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0
3	4	3	10	7	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
4	5	3	10	7	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0
5	6	3	10	7	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0
6	7	3	10	7	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0
7	8	3	10	7	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0

Table 5: Execution trace for the addition program shown in Figure 5.

5.9 VM AIR integration

As introduced in Section 3, an AIR consists of a trace table together with constraints that enforce correctness of execution by expressing it through polynomial relations. For example, to enforce that the trace column t increases by 1 at each step, one may define

$$C_t(\omega_i) = \big(t(\omega_i) - t(\omega_{i-1}) - 1\big)\, s_{\text{tr}}(\omega_i) = 0.$$

Equivalently, this may be written as

$$C_t(\omega_i) = \big(t(\omega_i) = t(\omega_{i-1}) + 1\big)\, s_{\text{tr}}(\omega_i),$$

although the polynomial form above is the one used in the AIR. Here, ω_i denotes a point in the evaluation domain. To simplify the notation in further, the symbol $\#t(X)$ is used to denote the value of column t in the previous row.

To ensure computational integrity, the constraints are grouped into the following categories. Structural constraints enforce general properties of columns, such as booleanity. Initialization constraints enforce correct initial values. Transition constraints enforce correct state evolution across consecutive rows. Preservation constraints ensure that registers not involved in a given instruction remain unchanged. Branching constraints enforce correct transitions under conditional jumps. Finally, halt constraints enforce correct padding behaviour after the VM has halted.

5.9.1 Structural Constraints

Structural constraints enforce that trace values satisfy the required correct properties. In particular, the instruction selector columns, the `halted` flag, and `jnz_taken` must be constrained to be Boolean. The one-hot opcode constraint ensures that exactly one instruction is active at each execution step. Register-index validity constraints ensure that, for instructions involving registers, the operands `a` and `b` contain valid register indices.

Unused-operand constraints ensure that operands not used by a given instruction are set to zero. For example, during a `const` instruction, `b` and `target` must both be zero. The auxiliary columns `jnz_taken` and `jnz_taken_inv` are constrained to

be zero on non-`jnz` rows, ensuring that conditional-jump-specific metadata appears only when the `jnz` instruction is active. All structural constraints are listed in Table 6.

Constraint	Formula
Booleanity of opcode selectors	$s_{\text{op}}(s_{\text{op}} - 1) = 0$, for each $s_{\text{op}} \in \{s_{\text{const}}, s_{\text{mov}}, s_{\text{add}}, s_{\text{sub}}, s_{\text{jmp}}, s_{\text{jnz}}, s_{\text{halt}}\}$
Booleanity of the halted flag	$\text{halted}(\text{halted} - 1) = 0$
Booleanity of the branch-taken flag	$\text{jnz_taken}(\text{jnz_taken} - 1) = 0$
One-hot opcode constraint	$s_{\text{const}} + s_{\text{mov}} + s_{\text{add}} + s_{\text{sub}} + s_{\text{jmp}} + s_{\text{jnz}} + s_{\text{halt}} - 1 = 0$
Register-index validity for `a`	$(s_{\text{const}} + s_{\text{mov}} + s_{\text{add}} + s_{\text{sub}} + s_{\text{jnz}}) \cdot a(a-1)(a-2)(a-3) = 0$
Register-index validity for `b`	$(s_{\text{mov}} + s_{\text{add}} + s_{\text{sub}}) \cdot b(b-1)(b-2)(b-3) = 0$
Unused-operand zeroing for `const`	$s_{\text{const}} \cdot b = 0, \quad s_{\text{const}} \cdot \text{target} = 0$
Unused-operand zeroing for `mov`	$s_{\text{mov}} \cdot \text{imm} = 0, \quad s_{\text{mov}} \cdot \text{target} = 0$
Unused-operand zeroing for `add`	$s_{\text{add}} \cdot \text{imm} = 0, \quad s_{\text{add}} \cdot \text{target} = 0$
Unused-operand zeroing for `sub`	$s_{\text{sub}} \cdot \text{imm} = 0, \quad s_{\text{sub}} \cdot \text{target} = 0$
Unused-operand zeroing for `jmp`	$s_{\text{jmp}} \cdot a = 0, \quad s_{\text{jmp}} \cdot b = 0, \quad s_{\text{jmp}} \cdot \text{imm} = 0$
Unused-operand zeroing for `jnz`	$s_{\text{jnz}} \cdot b = 0, \quad s_{\text{jnz}} \cdot \text{imm} = 0$
Unused-operand zeroing for `halt`	$s_{\text{halt}} \cdot a = 0, \quad s_{\text{halt}} \cdot b = 0, \quad s_{\text{halt}} \cdot \text{imm} = 0, \quad s_{\text{halt}} \cdot \text{target} = 0$
`jnz_taken` zero on non-`jnz` rows	$(1 - s_{\text{jnz}}) \cdot \text{jnz_taken} = 0$
`jnz_inv` zero on non-`jnz` rows	$(1 - s_{\text{jnz}}) \cdot \text{jnz_inv} = 0$

Table 6: Structural constraints of the VM AIR.

5.9.2 Initialization Constraints

Initialization constraints ensure that the first row of the trace contains the correct initial values. In particular, the program counter, all registers, and the halted flag must be zero at the beginning of execution. These conditions are enforced using the boundary selector polynomial s_{b} applied to the relevant trace columns. All initialization constraints used in the toy zkVM are shown in Table 7.

Constraint	Formula	Purpose
Initial t	$s_{\mathrm{b}} \cdot (t-1) = 0$	Ensures execution starts at t = 1
Initial program counter	$s_{\mathrm{b}} \cdot pc = 0$	Ensures execution starts at instruction address 0
Initial register r_0	$s_{\mathrm{b}} \cdot r_0 = 0$	Ensures register r_0 is initially zero
Initial register r_1	$s_{\mathrm{b}} \cdot r_1 = 0$	Ensures register r_1 is initially zero
Initial register r_2	$s_{\mathrm{b}} \cdot r_2 = 0$	Ensures register r_2 is initially zero
Initial register r_3	$s_{\mathrm{b}} \cdot r_3 = 0$	Ensures register r_3 is initially zero
Initial halted flag	$s_{\mathrm{b}} \cdot \mathrm{halted} = 0$	Ensures the machine is not halted at the beginning of execution

Table 7: Initialization constraints of the toy zkVM AIR

As discussed in Section 5.7, the selector polynomials s_{b}, s_{tr}, and s_{term} should not be included in the witness. Otherwise, a malicious prover could set them to zero and trivially satisfy the AIR constraints using the zero polynomial. Instead, both the prover and the verifier derive these selector polynomials independently from public parameters, such as the original domain size. Given the domain size, both parties can construct these polynomials over the shifted domain using the chosen interpolation procedure.

In contrast, instruction selector columns such as s_{const}, s_{add}, and the others may be included in the witness, since they are constrained by the one-hot opcode condition and are therefore tied to the computational semantics of the trace.

5.9.3 Transition Constraints

State transition constraints enforce correct evolution of the VM state across consecutive rows of the trace. For arithmetic and data-movement instructions, such as `const`, `mov`, `add`, and `sub`, correctness is expressed by relating the values in the current row to those in the previous row. For example, the instruction `add r0, r1` is enforced by checking that the updated value of register r_0 satisfies

$$\#r_0 + \#r_1 - r_0 = 0.$$

The program-counter transition constraint enforces correct changes of the instruction address. For non-branching instructions, namely `const`, `mov`, `add`, and `sub`, the program counter is incremented by one. The `jmp` instruction sets the program counter to the target address, whereas the `halt` instruction leaves the program counter unchanged.

The `jnz` transition requires additional handling. It uses the auxiliary column `jnz_taken`, which is equal to 1 when the branch is taken and 0 otherwise. Using this value, the expected program counter can be expressed as

$$pc_{\text{expected}} = \#\text{jnz_taken} \cdot \#\text{target} + \big(1 - \#\text{jnz_taken}\big) \cdot (\#pc + 1),$$

and the constraint then checks that the current value of pc matches this expected value. In addition, computational integrity requires constraining `jnz_taken` using the auxiliary inverse column associated with the branch condition. This is discussed in more detail in the subsection on branching constraints. All state transition constraints are listed in Table 8.

Constraint	**Formula**
Program counter transition	$s_{\text{tr}} \cdot \big((\#s_{\text{const}} + \#s_{\text{mov}} + \#s_{\text{add}} + \#s_{\text{sub}}) \cdot (pc - \#pc - 1) + \#s_{\text{jmp}} \cdot (pc - \#\text{target}) + \#s_{\text{halt}} \cdot (pc - \#pc)\big) = 0$
Conditional program counter transition for `jnz`	$s_{\text{tr}} \cdot \#s_{\text{jnz}} \cdot \Big(pc - (\#\text{jnz_taken} \cdot \#\text{target} + (1 - \#\text{jnz_taken}) \cdot (\#pc + 1))\Big) = 0$
Register transition for `const`	$s_{\text{tr}} \cdot \#s_{\text{const}} \cdot (r_a - \#\text{imm}) = 0$
Register transition for `mov`	$s_{\text{tr}} \cdot \#s_{\text{mov}} \cdot (r_a - \#r_b) = 0$
Register transition for `add`	$s_{\text{tr}} \cdot \#s_{\text{add}} \cdot (r_a - (\#r_a + \#r_b)) = 0$
Register transition for `sub`	$s_{\text{tr}} \cdot \#s_{\text{sub}} \cdot (r_a - (\#r_a - \#r_b)) = 0$

Table 8: State transition constraints of the toy zkVM AIR.

5.9.4 Preservation Constraints

Preservation constraints ensure that registers not affected by the active instruction remain unchanged across a state transition. For read-only instructions, this means that all registers must retain their previous values. For instructions that write to a destination register, the preservation constraint must be relaxed for that particular register. This is achieved using Lagrange selector polynomials $l_i(a)$, which indicate whether the destination operand a is equal to register index i. For the four-register machine, these selectors are defined as

$$l_0(a) = \frac{(a-1)(a-2)(a-3)}{(0-1)(0-2)(0-3)}, \quad l_1(a) = \frac{a(a-2)(a-3)}{(1-0)(1-2)(1-3)},$$

$$l_2(a) = \frac{a(a-1)(a-3)}{(2-0)(2-1)(2-3)}, \quad l_3(a) = \frac{a(a-1)(a-2)}{(3-0)(3-1)(3-2)}.$$

Accordingly, for writing instructions, the factor $1 - \#l_i$ disables the preservation constraint exactly when the destination register in the previous row is r_i. For read-only instructions, all registers are required to remain unchanged. One preservation constraint is introduced for each register.

In addition, the halted flag must be preserved on all transitions except the one on which the `halt` instruction is executed. Once the halted flag is set to 1, it remains equal to 1 throughout the padding rows. All preservation constraints are listed in Table 9.

Constraint	**Formula**
Preservation of register r_0	$s_{\text{tr}} \cdot \big((\#s_{\text{const}} + \#s_{\text{mov}} + \#s_{\text{add}} + \#s_{\text{sub}}) \cdot (1 - \#l_0) \cdot (r_0 - \#r_0) + (\#s_{\text{jmp}} + \#s_{\text{jnz}} + \#s_{\text{halt}}) \cdot (r_0 - \#r_0)\big) = 0$
Preservation of register r_1	$s_{\text{tr}} \cdot \big((\#s_{\text{const}} + \#s_{\text{mov}} + \#s_{\text{add}} + \#s_{\text{sub}}) \cdot (1 - \#l_1) \cdot (r_1 - \#r_1) + (\#s_{\text{jmp}} + \#s_{\text{jnz}} + \#s_{\text{halt}}) \cdot (r_1 - \#r_1)\big) = 0$
Preservation of register r_2	$s_{\text{tr}} \cdot \big((\#s_{\text{const}} + \#s_{\text{mov}} + \#s_{\text{add}} + \#s_{\text{sub}}) \cdot (1 - \#l_2) \cdot (r_2 - \#r_2) + (\#s_{\text{jmp}} + \#s_{\text{jnz}} + \#s_{\text{halt}}) \cdot (r_2 - \#r_2)\big) = 0$
Preservation of register r_3	$s_{\text{tr}} \cdot \big((\#s_{\text{const}} + \#s_{\text{mov}} + \#s_{\text{add}} + \#s_{\text{sub}}) \cdot (1 - \#l_3) \cdot (r_3 - \#r_3) + (\#s_{\text{jmp}} + \#s_{\text{jnz}} + \#s_{\text{halt}}) \cdot (r_3 - \#r_3)\big) = 0$
Preservation of the halted flag	$s_{\text{tr}} \cdot (1 - \#s_{\text{halt}}) \cdot (\text{halted} - \#\text{halted}) = 0$

Table 9: Preservation constraints of the toy zkVM AIR.

5.9.5 Branching Constraints

Branching constraints enforce the algebraic relations needed to implement the conditional jump instruction `jnz`. In particular, they constrain the auxiliary column `jnz_inv`, which stores the inverse of the condition-register value when that value is non-zero. Consequently, whenever the branch is taken, the column `jnz_taken` must be equal to 1.

To express this condition algebraically, the value of the condition register selected by operand a is reconstructed using the Lagrange selector polynomials:

$$\#r_a = \#r_0 \cdot \#l_0 + \#r_1 \cdot \#l_1 + \#r_2 \cdot \#l_2 + \#r_3 \cdot \#l_3.$$

Using this value together with the auxiliary inverse column, the branching logic is enforced by the relation

$$\#r_a \cdot \#\text{jnz_inv} - \#\text{jnz_taken} = 0.$$

This guarantees that if the condition value is non-zero, then `jnz_taken` must be equal to 1, since the inverse exists only in that case.

In addition, the constraint

$$\#r_a \cdot (1 - \#\text{jnz_taken}) = 0$$

ensures that whenever the condition register is non-zero, the branch is marked as taken. All branching constraints are listed in Table 10.

Constraint	**Formula**
Nonzero condition implies taken for `jnz`	$s_{\text{tr}} \cdot \#s_{\text{jnz}} \cdot \#r_a \cdot (1 - \#\text{jnz_taken}) = 0$
Inverse relation for `jnz`	$s_{\text{tr}} \cdot \#s_{\text{jnz}} \cdot (\#r_a \cdot \#\text{jnz_taken_inv} - \#\text{jnz_taken}) = 0$

Table 10: Branching constraints of the toy zkVM AIR.

5.9.6 Halt Semantics Constraints

Halt semantics constraints ensure that, once the `halt` instruction has been executed, the halted flag is set and the machine state remains unchanged in all subsequent rows. In particular, the `halt` instruction must set the `halted` flag to 1, and every row following a halted state must preserve the values of the program counter and all registers. In this way, the halted state is repeated during trace padding. All halt semantics constraints are listed in Table 11.

Constraint	**Formula**
Halt entry	$s_{\text{tr}} \cdot \#s_{\text{halt}} \cdot (\text{halted} - 1) = 0$
Halted-state freeze	$s_{\text{tr}} \cdot \big(\#\text{halted} \cdot (\text{halted} - 1) + \#\text{halted} \cdot (pc - \#pc) + \#\text{halted} \cdot (r_0 - \#r_0) + \#\text{halted} \cdot (r_1 - \#r_1) + \#\text{halted} \cdot (r_2 - \#r_2) + \#\text{halted} \cdot (r_3 - \#r_3)\big) = 0$

Table 11: Halt semantics constraints of the toy zkVM AIR.

Together, these 36 constraints restrict the trace so that all state transitions conform to the specified instruction semantics. As a result, they are sufficient to enforce the computational integrity of the program and can be incorporated directly into the protocol described in Section 5.5.

5.10 Complete zkVM pipeline

The complete zkVM pipeline combines the DSL frontend, the VM execution engine, the AIR, and the proving system into a single end-to-end workflow. First, a source program written in the toy DSL is processed by the lexer, parser, and resolver, producing a sequence of instructions with resolved jump targets. This instruction sequence, together with the initial VM state, is then executed by the virtual machine. During execution, the VM records each state transition together with the

required auxiliary metadata in the form of execution rows, which are subsequently transformed into the execution trace.

Next, the execution trace is combined with the VM AIR, whose constraints enforce correct initialization, state transitions, register preservation, branching behaviour, and halt semantics. Using the public parameters, the AIR instance, and the Fiat–Shamir transcript, the prover executes the proving algorithm and produces a proof attesting that the trace satisfies all constraints. Finally, the verifier uses the same public parameters together with the proof to check that the claimed execution is valid. In this way, the system provides an end-to-end method for proving the correct execution of programs written in the toy DSL without requiring the verifier to re-execute the program.

6 Conclusion and Future Work

This thesis studied the design and implementation of a STARK-based zero-knowledge virtual machine in Rust. The main goal was not to build a production-ready zkVM, but to understand and demonstrate the core components required to prove computational integrity in a STARK-style setting. The prototype presented in this thesis models computation using AIR, constructs quotient evaluations over a shifted low-degree extension domain, and verifies low-degree consistency using the FRI protocol. In addition, it uses Merkle commitments and a Fiat–Shamir transcript to replace verifier interaction.

The thesis then extends this foundation by introducing a DSL for expressing simple programs and examining the design decisions of the frontend compilation pipeline. Then, it develops a virtual machine execution environment, describes the execution process of programs, and explains the construction of the execution trace. Finally, it designs AIR constraints that enforce the correctness of the execution trace.

The main outcome of this work is an end-to-end zkVM pipeline that combines the DSL, the AIR layer, and the cryptographic backend into a single zero-knowledge system. The implementation demonstrates how the components of the proving and verification pipeline interact in practice. Two Fibonacci examples motivate key design decisions and illustrate the construction of constraints for proving computational integrity.

Despite the limitations of the demonstrated proving system, the prototype provides a useful foundation for understanding STARK-style zkVM constructions from first principles. In particular, it demonstrates a modular and decoupled structure with a clear separation between the DSL frontend, AIR definitions, backend primitives, constraint logic, and a higher-level prover/verifier API.

There are several natural directions for future work. First, the constraints could be optimized to reduce their degree and thereby improve the efficiency of the proving and verification pipeline. Second, adding a memory model, including read and write operations together with memory-consistency checks, would increase expressiveness and move the system closer to practical zkVM designs. Third, the implementation could be complemented with systematic benchmarking in order to better evaluate the trade-offs of the chosen primitives. Finally, the zero-knowledge properties of

the prototype could be strengthened by introducing blinding, together with a more formal analysis of soundness and implementation-level security.

Overall, this thesis shows that a toy STARK-based zkVM can be implemented in a modular and understandable way, and that the prototype presented here makes a step toward a deeper study of modern zero-knowledge virtual machines and proof systems.

References

[1] Tim Dokchitser and Alexandr Bulkin. *Zero Knowledge Virtual Machine Step by Step*. *IACR Cryptology ePrint Archive*, Paper 2023/1032, 2023. Available at: `https://eprint.iacr.org/2023/1032`. Accessed 25 Jan 2026.

[2] Mohammed El-Hajj and Bjorn Oude Roelink. Evaluating the Efficiency of zk-SNARK, zk-STARK, and Bulletproof in Real-World Scenarios: A Benchmark Study. *Information*, 15(8):463, 2024. doi:10.3390/info15080463.

[3] Jan Gorzny and Martin Derka. *A Rollup Comparison Framework*. *arXiv preprint* arXiv:2404.16150, 2024. Available at: `https://arxiv.org/abs/2404.16150`. Accessed 3 Apr 2026.

[4] Ahmed E. Kosba, Charalampos Papamanthou, and Elaine Shi. xJsnark: A Framework for Efficient Verifiable Computation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 944–961, 2018. doi:10.1109/SP.2018.00018.

[5] Jens Groth. On the Size of Pairing-Based Non-interactive Arguments. In *Advances in Cryptology – EUROCRYPT 2016 (Part II)*, Lecture Notes in Computer Science, vol. 9666, pp. 305–326. Springer, 2016. doi:10.1007/978-3-662-49896-5_11.

[6] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive Arguments of Knowledge. *IACR Cryptology ePrint Archive*, Report 2019/953, 2019. Available at: `https://eprint.iacr.org/2019/953`. Accessed 25 Jan 2026.

[7] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-Size Commitments to Polynomials and Their Applications. In *Advances in Cryptology – ASIACRYPT 2010*, Lecture Notes in Computer Science, vol. 6477, pp. 177–194. Springer, 2010. doi:10.1007/978-3-642-17373-8_11.

[8] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *IACR Cryptology ePrint Archive*, Report 2018/046, 2018. Available at: `https://eprint.iacr.org/2018/046`. Accessed 25 Jan 2026.

[9] Jeremy Bruestle, Paul Gafni, and the RISC Zero Team. *RISC Zero zkVM: Scalable, Transparent Arguments of RISC-V Integrity*. Technical report (draft), August 11, 2023. Available at: `https://dev.risczero.com/proof-system-in-detail.pdf`. Accessed 25 Jan 2026.

[10] Succinct Labs. *SP1 Documentation: Introduction.* Online documentation. Available at: `https://docs.succinct.xyz/docs/sp1/introduction`. Accessed 25 Jan 2026.

[11] Justin Thaler. Proofs, Arguments, and Zero-Knowledge. *Foundations and Trends in Privacy and Security*, 4(2–4):117–660, 2022. doi:10.1561/3300000030.

[12] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Fast Reed-Solomon Interactive Oracle Proofs of Proximity. In *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*, LIPIcs, pages 14:1–14:17, 2018. doi:10.4230/LIPIcs.ICALP.2018.14.

[13] J. M. Pollard. The Fast Fourier Transform in a Finite Field. *Mathematics of Computation*, 25(114):365–374, 1971. doi:10.1090/S0025-5718-1971-0301966-0.

[14] Ralph C. Merkle. *Secrecy, Authentication, and Public Key Systems*. PhD thesis, Stanford University, 1979.

[15] Jean-Philippe Aumasson, Samuel Neves, Jack O'Connor, and Zooko Wilcox-O'Hearn. The BLAKE3 Hashing Framework. Internet-Draft, IETF, draft-aumasson-blake3-00, July 20, 2024. Available at: `https://www.ietf.org/archive/id/draft-aumasson-blake3-00.html`. Accessed 2 Feb 2026.

[16] Bogdan Rudnevskii (brudnevskij). *toy-zkvm*: A toy zero-knowledge virtual machine implementation. GitHub repository. Available at: `https://github.com/brudnevskij/toy-zkvm`. Accessed 28 Jan 2026.

[17] Arasu Arun, Srinath T. V. Setty, and Justin Thaler. Jolt: SNARKs for Virtual Machines via Lookups. *IACR Cryptology ePrint Archive*, Report 2023/1217, 2023. Available at: `https://eprint.iacr.org/2023/1217`. Accessed 2 Feb 2026.

[18] ConsenSys. *gnark*. Online documentation (*pkg.go.dev* / project repository). Available at: `https://github.com/Consensys/gnark`. Accessed 2 Feb 2026.

[19] arkworks contributors. *arkworks: Rust ecosystem for zkSNARK programming*. Online documentation. Available at: `https://arkworks.rs/`. Accessed 2 Feb 2026.

[20] Winterfell contributors. *winterfell: STARK prover and verifier (crate documentation)*. Online documentation. Available at: `https://docs.rs/crate/winterfell/latest`. Accessed 2 Feb 2026.

[21] Henry de Valence and Oleg Andreev. *merlin: Composable proof transcripts for public-coin arguments (crate documentation)*. Online documentation. Available at: `https://docs.rs/crate/merlin/latest`. Accessed 2 Feb 2026.

[22] arkworks contributors. *ark-bn254: The BN254 pairing-friendly elliptic curve (crate documentation)*. Online documentation (docs.rs), version 0.5.0. Available at: `https://docs.rs/ark-bn254/latest/ark_bn254/`. Accessed 2 Feb 2026.

List of Abbreviations

Abbreviation	Meaning
AI	Artificial Intelligence
AIR	Algebraic Intermediate Representation
API	Application Programming Interface
BLAKE3	BLAKE3 cryptographic hash function
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DSL	Domain-Specific Language
EBNF	Extended Backus–Naur Form
EOF	End of File
FFT	Fast Fourier Transform
FRI	Fast Reed–Solomon Interactive Oracle Proof of Proximity
INTT	Inverse Number-Theoretic Transform
IOP	Interactive Oracle Proof
KZG	Kate–Zaverucha–Goldberg polynomial commitment scheme
LDE	Low-Degree Extension
NTT	Number-Theoretic Transform
PC	Program Counter
PLONK	Permutations over Lagrange-bases for Oecumenical Non-interactive arguments of Knowledge
RISC-V	Reduced Instruction Set Computer V
SHA-256	Secure Hash Algorithm 256
SNARK	Succinct Non-interactive Argument of Knowledge
STARK	Scalable Transparent Argument of Knowledge
VM	Virtual Machine
ZKP	Zero-Knowledge Proof
zk-SNARK	Zero-Knowledge Succinct Non-interactive Argument of Knowledge
zk-STARK	Zero-Knowledge Scalable Transparent Argument of Knowledge
zkVM	Zero-Knowledge Virtual Machine

Table 12: Abbreviations used in the thesis.

A Rust Merkle Tree Implementation

```
use std::marker::PhantomData;
use thiserror::Error;

pub type Digest = [u8; 32];

#[derive(Debug, Clone)]
pub struct AuthPath {
    pub nodes: Vec<Digest>,
    pub index: usize,
}

#[derive(Error, Debug)]
pub enum MerkleError {
    #[error("empty input")]
    Empty,
    #[error("index out of range")]
    IndexOutOfRange,
}

pub trait Hasher {
    fn hash_leaf(data: &[u8]) -> Digest;
    fn hash_node(left_node: &Digest, right_node: &Digest) -> Digest;
}

#[derive(Debug)]
pub struct Blake3Hasher;

impl Hasher for Blake3Hasher {
    fn hash_leaf(data: &[u8]) -> Digest {
        let mut hasher = blake3::Hasher::new();
        hasher.update(&[0x00]);
        hasher.update(data);
        *hasher.finalize().as_bytes()
    }

    fn hash_node(left: &Digest, right: &Digest) -> Digest {
        let mut hasher = blake3::Hasher::new();
        hasher.update(&[0x01]);
        hasher.update(left);
        hasher.update(right);
        *hasher.finalize().as_bytes()
    }
}

#[derive(Debug)]
pub struct MerkleTree<H: Hasher = Blake3Hasher> {
```

```
    nodes: Vec<Digest>,
    leaf_count: usize,
    leaf_cap: usize,
    _h: PhantomData<H>,
}

impl<H: Hasher> MerkleTree<H> {
    pub fn from_rows(rows: &[&[u8]]) -> Result<Self, MerkleError> {
        if rows.is_empty() {
            return Err(MerkleError::Empty);
        }
        let leaves: Vec<Digest> = rows.iter().map(|x| H::hash_leaf(x)).
            collect();
        Ok(Self::from_leaf_digests(&leaves)?)
    }

    pub fn from_leaf_digests(leaves: &[Digest]) -> Result<Self,
        MerkleError> {
        if leaves.is_empty() {
            return Err(MerkleError::Empty);
        }
        let leaf_count = leaves.len();
        let cap = next_pow2(leaf_count);

        // since number of leaves is power of 2, cap also includes, next
            floors which are also powers of two
        // making the sum = 2^leaf_count + (2^leaf_count - 1) + 1 (last is
            unused 0th index) = 2 * 2^leaf_count
        let mut nodes = vec![[0u8; 32]; cap * 2];

        for (i, leaf) in leaves.iter().enumerate() {
            nodes[cap + i] = *leaf;
        }

        for i in leaf_count..cap {
            nodes[cap + i] = nodes[cap + leaf_count - 1];
        }

        for i in (1..cap).rev() {
            let l = nodes[2 * i];
            let r = nodes[2 * i + 1];
            nodes[i] = H::hash_node(&l, &r);
        }

        Ok(Self {
            nodes,
            leaf_count,
            leaf_cap: cap,
            _h: PhantomData,
```

```
        })
    }

    // [0, 1..powe^2-1]
    pub fn open(&self, index: usize) -> Result<AuthPath, MerkleError> {
        if index >= self.leaf_count {
            return Err(MerkleError::IndexOutOfRange);
        }

        let mut pos = self.leaf_cap + index;
        let mut nodes = Vec::with_capacity(self.height());
        while pos > 1 {
            let sib = if pos & 1 == 0 { pos + 1 } else { pos - 1 };
            nodes.push(self.nodes[sib]);
            pos >>= 1;
        }
        Ok(AuthPath { nodes, index })
    }

    pub fn root(&self) -> &Digest {
        &self.nodes[1]
    }

    /// Tree height in levels from leaves to root (log2(cap)).
    fn height(&self) -> usize {
        // For cap == 1, height == 0 (single leaf, path is empty).
        usize::BITS as usize - (self.leaf_cap.leading_zeros() as usize) - 1
    }
}

pub fn verify_leaf<H: Hasher>(root: &Digest, leaf: &Digest, auth_path: &
    AuthPath) -> bool {
    let mut acc = *leaf;
    let mut idx = auth_path.index;

    for sib in auth_path.nodes.iter() {
        if idx & 1 == 0 {
            acc = H::hash_node(&acc, sib);
        } else {
            acc = H::hash_node(sib, &acc);
        }
        idx >>= 1;
    }
    root == &acc
}

pub fn verify_row<H: Hasher>(root: &Digest, row: &[u8], auth_path: &
    AuthPath) -> bool {
    let leaf = H::hash_leaf(row);
```

```
    verify_leaf::<H>(root, &leaf, auth_path)
}

fn next_pow2(n: usize) -> usize {
    // treat n >= 1
    let mut x = n - 1;
    x |= x >> 1;
    x |= x >> 2;
    x |= x >> 4;
    x |= x >> 8;
    x |= x >> 16;
    if usize::BITS > 32 {
        x |= x >> 32;
    }
    x + 1
}
```

B Rust Transcript Implementation

```
use crate::backend::Digest;
use ark_ff::{Field, PrimeField};
use ark_serialize::CanonicalSerialize;

pub struct Transcript {
    state: [u8; 32],
    counter: u64,
    label: Vec<u8>,
}

impl Transcript {
    pub fn new(label: &[u8], seed: &[u8]) -> Self {
        let mut h = blake3::Hasher::new();
        h.update(b"transcript:init");
        h.update(label);
        h.update(seed);
        let state = *h.finalize().as_bytes();
        Self {
            state,
            counter: 0,
            label: label.to_vec(),
        }
    }

    pub fn absorb_bytes(&mut self, label: &str, data: &[u8]) {
        let mut h = blake3::Hasher::new();
        h.update(b"transcript:absorb");
```

```
        h.update(&self.state);
        h.update(&self.label);
        h.update(label.as_bytes());
        h.update(data);
        self.state = *h.finalize().as_bytes();
    }

    pub fn absorb_field<F: CanonicalSerialize>(&mut self, label: &str, x:
        &F) {
        let mut buf = Vec::new();
        x.serialize_compressed(&mut buf).expect("serialize field");
        self.absorb_bytes(label, &buf);
    }

    pub fn absorb_digest(&mut self, label: &str, d: &Digest) {
        self.absorb_bytes(label, d);
    }

    /// Derive `n` pseudo-random bytes from the transcript (domain-
        separated by label + counter).
    pub fn challenge_bytes(&mut self, label: &str, n: usize) -> Vec<u8> {
        let mut base = blake3::Hasher::new();
        base.update(b"transcript:challenge");
        base.update(&self.state);
        base.update(&self.label);
        base.update(label.as_bytes());
        base.update(&self.counter.to_le_bytes());
        self.counter = self.counter.wrapping_add(1);

        // Use XOF to produce as many bytes as needed.
        let mut xof = base.finalize_xof();
        let mut out = vec![0u8; n];
        xof.fill(&mut out);

        // absorb challenge
        self.absorb_bytes("output", &out);
        out
    }

    pub fn challenge_u64(&mut self, label: &str) -> u64 {
        let b = self.challenge_bytes(label, 8);
        u64::from_le_bytes(b.try_into().expect("len 8"))
    }

    /// Derive a field element challenge via canonical mod-p reduction.
    pub fn challenge_field<F: PrimeField>(&mut self, label: &str) -> F {
        let bytes = self.challenge_bytes(label, 64);
        F::from_le_bytes_mod_order(&bytes)
    }
```

```
    /// Sample an unbiased index in [0, n) using rejection sampling.
    pub fn challenge_index(&mut self, label: &str, n: u64) -> u64 {
        assert!(n > 0, "n must be > 0");
  let limit = u64::MAX / n * n; // largest multiple of n <= u64::MAX
        loop {
            let x = self.challenge_u64(label);
            if x < limit {
                return x % n;
            }
        }
    }
}
```

C Rust FRI Implementation

```
pub fn prove<F: PrimeField + FftField>(
    evals: &[F],
    initial_domain: &Radix2EvaluationDomain<F>,
    options: &FriOptions<F>,
    tx: &mut Transcript,
) -> Result<FriProof<F>, ProofError> {
    let initial_domain_size = initial_domain.size();
    assert!(initial_domain_size > 0, "empty domain");
    assert!(
        options.max_remainder_degree > 0,
        "remainder degree must be greater than 0"
    );
    assert!(
        options.max_remainder_degree < options.max_degree,
        " need r < d"
    );
    assert!(
        options.max_degree < initial_domain_size,
        "need d < N (rate < 1)"
    );
    if evals.is_empty() {
        return Err(ProofError::EmptyEvaluations);
    }
    if evals.len() != initial_domain_size {
        return Err(ProofError::BadEvaluationsLength {
            got: evals.len(),
            expected: initial_domain_size,
        });
    }
```

```
options.seed_tx(tx);

let mut evaluations_layers = vec![];
let mut roots = vec![];
let mut trees = vec![];
let mut g = initial_domain.group_gen();
let mut shift = options.shift;
let mut evals_i = evals.to_vec();
let mut current_max_degree = options.max_degree;

while current_max_degree > options.max_remainder_degree {
    let tree = commit_evals(&evals_i)?;
    let root = tree.root();
    tx.absorb_digest(ROOT, root);

    roots.push(*root);
    trees.push(tree);
    evaluations_layers.push(evals_i);

    // get challenge
    let beta_i: F = tx.challenge_field(BETA_I);
    // fold
    let last = evaluations_layers.last().unwrap();
    evals_i = fold_once(last, g, beta_i, shift);
    // advance
    g = g.square();
    shift = shift.square();

    // halving upper bound
    current_max_degree = current_max_degree.div_ceil(2);
}

// interpolating final poly
let final_domain_size = initial_domain_size >> roots.len(); // N / 2^
    n_folds
assert_eq!(final_domain_size, evals_i.len());
assert!(final_domain_size > options.max_remainder_degree);

let final_domain = Radix2EvaluationDomain::new(final_domain_size).
    unwrap();
if g != final_domain.group_gen() {
    return Err(ProofError::FinalDomainGeneratorMismatch);
}

let mut final_poly = final_domain.ifft(&evals_i);
trim_trailing_zeroes(&mut final_poly);

if final_poly.len() - 1 > options.max_remainder_degree {
```

```
        return Err(ProofError::
            FinalPolynomialDegreeExceedMaxRemainderDegree {
            got: final_poly.len() - 1,
            expected: options.max_remainder_degree,
        });
    }

    let final_tree = commit_evals(&final_poly)?;
    tx.absorb_digest(FINAL_POLY, final_tree.root());

    // query phase
    let mut queries = Vec::with_capacity(options.query_number);
    for _ in 0..options.query_number {
        let half = initial_domain_size >> 1;
        let idx = tx.challenge_index(QUERY_I, half as u64) as usize;
        let q = produce_query(idx, initial_domain_size, &evaluations_layers
            , &trees)?;
        queries.push(q);
    }

    Ok(FriProof {
        roots,
        queries,
        final_poly,
    })
}


fn fold_once<F: PrimeField + FftField>(evals: &[F], g: F, beta: F, shift:
    F) -> Vec<F> {
    let n = evals.len();
    let half = n / 2;

    let inv2 = F::from(2u64).inverse().expect("inverse");
    let ginv = g.inverse().expect("inverse");

    // 1/x where x = shift g^i => invx = shift^{-1} * g^{-1}
    // starting with shift because x0 = 1
    let mut invx = shift.inverse().expect("shift inverse");
    let mut out = Vec::with_capacity(half);

    for i in 0..half {
        let j = i + half;

        // f(x)
        let fx = evals[i];
        // f(-x)
        let fnegx = evals[j];
```

```
        let f_even = (fx + fnegx) * inv2;
        let f_odd = (fx - fnegx) * inv2 * invx;

        out.push(f_even + beta * f_odd);

        invx *= ginv
    }
    out
}


pub fn verify<F: PrimeField + FftField>(
    proof: &FriProof<F>,
    domain: &Radix2EvaluationDomain<F>,
    options: &FriOptions<F>,
    tx: &mut Transcript,
) -> Result<(), VerificationError> {
    let n0 = domain.size();
    if n0 == 0 || proof.roots.is_empty() || proof.queries.is_empty() {
        return Err(VerificationError::BadProof);
    }

    options.seed_tx(tx);
    let inv2 = F::from(2u64).inverse().expect("inverse");
    let mut betas = Vec::with_capacity(proof.roots.len());
    for root in &proof.roots {
        tx.absorb_digest(ROOT, root);
        let beta: F = tx.challenge_field(BETA_I);
        betas.push(beta);
    }

    // finall poly degree assertions
    let mut final_poly = proof.final_poly.clone();
    trim_trailing_zeroes(&mut final_poly);

    if final_poly.is_empty() {
        return Err(VerificationError::BadProof);
    }

    let deg = final_poly.len() - 1;

    if deg > options.max_remainder_degree {
        return Err(
            VerificationError::
                FinalPolynomialDegreeExceedMaxRemainderDegree {
                got: deg,
                expected: options.max_remainder_degree,
            },
        );
```

```
}

let folded_degree = degree_after_folds(options.max_degree, proof.roots
    .len());
if deg > folded_degree {
    return Err(
        VerificationError::
            FinalPolynomialDegreeExceedsClaimedDegreeAfterFolds {
            got: deg,
            expected: folded_degree,
            folds: proof.roots.len(),
        },
    );
}

let final_tree = commit_evals(&final_poly).unwrap();
tx.absorb_digest(FINAL_POLY, final_tree.root());

let final_domain_size = n0 >> proof.roots.len(); // n0 / 2^k
if final_domain_size < options.max_remainder_degree + 1 {
    return Err(VerificationError::BadProof);
}

let final_domain =
    Radix2EvaluationDomain::<F>::new(final_domain_size).ok_or(
        VerificationError::BadProof)?;
let final_evals = final_domain.fft(&final_poly);

for (q_i, query) in proof.queries.iter().enumerate() {
    let mut idx = tx.challenge_index(QUERY_I, (n0 / 2) as u64) as usize
        ;
    let mut domain_size = n0;
    let mut g = domain.group_gen();
    let mut shift = options.shift;

    if query.rounds.len() != proof.roots.len() {
        return Err(VerificationError::RootsNEtoQueryRounds);
    }
    let rounds = &query.rounds;
    for (layer_i, round) in rounds.iter().enumerate() {
        let half = domain_size / 2;
        let (left_value, right_value) = verify_round_openings(
            &proof.roots[layer_i],
            idx,
            q_i,
            layer_i,
            domain_size,
            round,
        )?;
```

```
            // fold
            let x = shift * g.pow([idx as u64]);
            let y = fold_evaluation_pair(
                left_value,
                right_value,
                x.inverse().unwrap(),
                betas[layer_i],
                inv2,
            );

            if layer_i + 1 < query.rounds.len() {
                if y != query.rounds[layer_i + 1].left.value {
                    return Err(VerificationError::VerificationFailed);
                }
            } else if y != final_evals[idx % half] {
                return Err(VerificationError::VerificationFailed);
            }
            // advance
            idx %= half;
            domain_size = half;
            g = g.square();
            shift = shift.square();
        }
    }
    Ok(())
}
```

D Rust AIR Implementation

```
use ark_ff::PrimeField;

pub mod trace;
pub use trace::TraceTable;

pub trait RowAccess<F: PrimeField> {
    fn current_step_column_value(&self, column: usize) -> F;
    fn previous_step_column_value(&self, column: usize) -> F;

    fn x(&self) -> F;
    fn x0(&self) -> F;
    fn x_last(&self) -> F;
    fn z_h_inverse(&self) -> F;
}

pub trait Constraint<F: PrimeField>: Send + Sync {
```

```rust
    fn name(&self) -> &'static str;
    fn eval(&self, row: &dyn RowAccess<F>) -> F;
}

pub trait Air<F: PrimeField> {
    fn width(&self) -> usize;

    fn column_name(&self, _col: usize) -> &'static str {
        "col"
    }

    fn num_constraints(&self) -> usize;

    fn eval_constraint(&self, i: usize, row: &dyn RowAccess<F>) -> F;

    fn constraint_name(&self, _i: usize) -> &'static str {
        "constraint"
    }
}

pub fn eval_composition<F: PrimeField>(
    air: &impl Air<F>,
    row: &dyn RowAccess<F>,
    alphas: &[F],
) -> F {
    assert_eq!(alphas.len(), air.num_constraints());
    let mut acc = F::zero();
    for i in 0..air.num_constraints() {
        acc += alphas[i] * air.eval_constraint(i, row);
    }
    acc
}
```

E Rust ZKVM Public Params

```rust
use crate::backend::{FriOptions, Transcript};
use ark_ff::PrimeField;
use ark_poly::{EvaluationDomain, Radix2EvaluationDomain};

pub struct TranscriptLabels;

impl TranscriptLabels {
    // --- public params ---
    pub const TRACE_DOMAIN_SIZE: &'static str = "trace_domain_size";
    pub const LDE_DOMAIN_SIZE: &'static str = "lde_domain_size";
    pub const SHIFT: &'static str = "shift";
    pub const FRI_MAX_DEGREE: &'static str = "fri_max_degree";
```

```
    pub const FRI_MAX_REMAINDER_DEGREE: &'static str = "
        fri_max_remainder_degree";
    pub const FRI_NUM_QUERIES: &'static str = "fri_num_queries";

    // --- commitments / roots ---
    pub const TRACE_ROOT: &'static str = "trace_root";
    pub const COMPOSITION_ROOT: &'static str = "composition_root"; // if
        you have one
    pub const TRACE_ROW_PREFIX: &'static [u8] = b"trace_row";

    // --- challenges ---
    pub const AIR_ALPHA_PREFIX: &'static str = "air/alpha/";

    #[inline]
    pub fn air_alpha(i: usize) -> String {
        format!("{}{i}", Self::AIR_ALPHA_PREFIX)
    }
}

#[derive(Debug, Clone, Copy)]
pub struct ZkvmPublicParameters<F: PrimeField> {
    pub trace_domain: Radix2EvaluationDomain<F>,
    pub lde_domain: Radix2EvaluationDomain<F>,
    pub shift: F,

    pub fri_options: FriOptions<F>,
}

impl<F: PrimeField> ZkvmPublicParameters<F> {
    pub fn seed_tx(&self, tx: &mut Transcript) {
        tx.absorb_bytes(
            TranscriptLabels::TRACE_DOMAIN_SIZE,
            &self.trace_domain.size().to_le_bytes(),
        );
        tx.absorb_bytes(
            TranscriptLabels::LDE_DOMAIN_SIZE,
            &self.lde_domain.size().to_le_bytes(),
        );
        tx.absorb_field(TranscriptLabels::SHIFT, &self.shift);
        tx.absorb_bytes(
            TranscriptLabels::FRI_MAX_DEGREE,
            &self.fri_options.max_degree.to_le_bytes(),
        );
        tx.absorb_bytes(
            TranscriptLabels::FRI_MAX_REMAINDER_DEGREE,
            &self.fri_options.max_remainder_degree.to_le_bytes(),
        );
        tx.absorb_bytes(
            TranscriptLabels::FRI_NUM_QUERIES,
```

```
            &self.fri_options.query_number.to_le_bytes(),
        );
    }
}
```

F Rust ZKVM Prover

```
#[derive(Clone, Debug)]
pub struct ZkvmProof<F: PrimeField> {
    /// Commitments
    pub trace_root: Digest, // Merkle root for LDE evaluations
    pub composition_root: Digest, // Merkle root of V_evals
    /// fri proof
    pub fri_proof: FriProof<F>,
    // openings to bind V to trace
    pub trace_queries: Vec<TraceQuery<F>>,
}

pub fn prove<A, F>(
    trace: &TraceTable<F>,
    air: &A,
    tx: &mut Transcript,
    public_params: &ZkvmPublicParameters<F>,
) -> Result<ZkvmProof<F>, ZkvmProveError>
where
    A: Air<F>,
    F: PrimeField + FftField + CanonicalSerialize,
{
    let ZkvmPublicParameters {
        trace_domain,
        lde_domain,
        shift: _,
        fri_options,
    } = public_params;

    let shift = public_params.shift;
    let trace_domain_size = trace_domain.size();
    let lde_domain_size = lde_domain.size();
    let trace_len = trace.n();
    let num_columns = trace.num_cols();

    if trace_domain_size != trace_len {
        return Err(ZkvmProveError::TraceLengthMismatch {
            trace_n: trace_len,
            domain_n: trace_domain_size,
        });
```

```
}

if lde_domain_size % trace_domain_size != 0 {
    return Err(ZkvmProveError::BadLdeDomains {
        n: trace_domain_size,
        m: lde_domain_size,
    });
}
let blowup_factor = lde_domain_size / trace_domain_size;
public_params.seed_tx(tx);

// build shifted LDE trace and commit it
let mut disguised_evaluations: Vec<Vec<F>> = Vec::with_capacity(
    num_columns);

for column in trace.columns.iter() {
    disguised_evaluations.push(lde_extend_column(column, trace_domain,
        lde_domain, shift));
}

let trace_tree = generate_trace_tree(lde_domain_size, &
    disguised_evaluations)?;
let trace_root = trace_tree.root();
tx.absorb_digest(TranscriptLabels::TRACE_ROOT, trace_root);

// generating alphas for future mixing
let alphas: Vec<F> = generate_mixing_challenges(air.num_constraints(),
     tx);

let verification_evaluations = build_verification_evaluations(
    air,
    shift,
    trace_domain,
    lde_domain,
    &alphas,
    &disguised_evaluations,
)?;

let fri_proof = fri_prove(&verification_evaluations, lde_domain,
    fri_options, tx)?;

// open same indexes as in fri
let trace_queries = generate_trace_queries(
    lde_domain_size,
    blowup_factor,
    &fri_proof.queries,
    &disguised_evaluations,
    &trace_tree,
)?;
```

```
    Ok(ZkvmProof {
        trace_root: *trace_root,
        composition_root: fri_proof.roots[0],
        fri_proof,
        trace_queries,
    })
}
```

G Rust ZKVM Verifier

```
use crate::{
    air::{Air, RowAccess, eval_composition},
    backend::{Blake3Hasher, FriVerificationError, Transcript, fri_verify,
        verify_leaf},
    zkvm::{
        TraceQuery, TranscriptLabels, ZkvmProof, ZkvmPublicParameters,
            generate_mixing_challenges,
        hash_trace_row_iter,
    },
};

use ark_ff::PrimeField;
use ark_poly::EvaluationDomain;
use thiserror::Error;

#[derive(Debug, Error)]
pub enum ZkvmVerifyError {
    #[error("verification failed")]
    VerificationFailed,

    #[error("bad fri proof")]
    BadFriProof,

    #[error(
        "merkle verification failed: current row verificattion: {
            current_row}, previous row verification {previous_row}"
    )]
    MerkleVerificationFailed {
        current_row: bool,
        previous_row: bool,
    },

    #[error("fri verification error: {0}")]
    FriVerificationError(#[from] FriVerificationError),
```

```
    #[error("vanishing polynomial Z_H(x)=x^n-1 is zero at i={i} (cannot
        invert)")]
    VanishingPolyNotInvertible { i: usize },
}

#[derive(Clone, Debug)]
struct VerifierRowLdeView<'a, F: PrimeField> {
    pub i: usize,
    pub previous_i: usize,
    pub x: F, // x in the shifted LDE cosset
    pub x0: F, // first x in the n domain
    pub x_last: F, // last x in the n domain
    pub z_h_inverse: F, // (x^n - 1)^-1
    pub current_row: &'a [F],
    pub previous_row: &'a [F],
}

impl<'a, F: PrimeField> RowAccess<F> for VerifierRowLdeView<'a, F> {
    fn current_step_column_value(&self, column: usize) -> F {
        self.current_row[column]
    }

    fn previous_step_column_value(&self, column: usize) -> F {
        self.previous_row[column]
    }

    fn x(&self) -> F {
        self.x
    }

    fn x0(&self) -> F {
        self.x0
    }

    fn x_last(&self) -> F {
        self.x_last
    }

    fn z_h_inverse(&self) -> F {
        self.z_h_inverse
    }
}

pub fn verify<A, F>(
    proof: &ZkvmProof<F>,
    air: &A,
    tx: &mut Transcript,
    public_params: &ZkvmPublicParameters<F>,
) -> Result<(), ZkvmVerifyError>
```

```
where
    A: Air<F>,
    F: PrimeField,
{
    public_params.seed_tx(tx);
    tx.absorb_digest(TranscriptLabels::TRACE_ROOT, &proof.trace_root);
    let challenges = generate_mixing_challenges::<F>(air.num_constraints()
        , tx);

    // asserting length of fri and zkvm queries, as well as length of
        rounds > 0
    let ok = proof.trace_queries.len() == proof.fri_proof.queries.len();
    if !ok {
        return Err(ZkvmVerifyError::BadFriProof);
    }

    let trace_domain_size = public_params.trace_domain.size();
    let x0 = public_params.trace_domain.element(0);
    let x_last = public_params.trace_domain.element(trace_domain_size - 1)
        ;
    for (trace_query, fri_query) in proof
        .trace_queries
        .iter()
        .zip(proof.fri_proof.queries.iter())
    {
        let TraceQuery {
            i: _,
            current_row,
            current_row_path,
            previous_row,
            previous_row_path,
        } = trace_query;

        let i = trace_query.i;
        let first_round = fri_query
            .rounds
            .first()
            .ok_or(ZkvmVerifyError::BadFriProof)?;
        if i != first_round.left.path.index {
            return Err(ZkvmVerifyError::BadFriProof);
        }

        // merkle verification
        let lde_domain_size = public_params.lde_domain.size();
        let blowup_factor = lde_domain_size / trace_domain_size;
        let previous_i = (i + lde_domain_size - blowup_factor) %
            lde_domain_size;
```

```
        let current_row_digest = hash_trace_row_iter(i, current_row.iter())
            ;
        let current_row_merkle_verificattion =
            verify_leaf::<Blake3Hasher>(&proof.trace_root, &
                current_row_digest, current_row_path);
        let previous_row_digest = hash_trace_row_iter(previous_i,
            previous_row.iter());
        let previous_row_merkle_verification =
            verify_leaf::<Blake3Hasher>(&proof.trace_root, &
                previous_row_digest, previous_row_path);

        if !(current_row_merkle_verificattion &&
            previous_row_merkle_verification) {
            return Err(ZkvmVerifyError::MerkleVerificationFailed {
                current_row: current_row_merkle_verificattion,
                previous_row: previous_row_merkle_verification,
            });
        }

        // compute verification polynomial evaluations for a given i
        let x = public_params.shift * public_params.lde_domain.element(i);
        let z_h = x.pow([trace_domain_size as u64]) - F::one();
        let z_h_inverse = z_h
            .inverse()
            .ok_or(ZkvmVerifyError::VanishingPolyNotInvertible { i })?;

        let row = VerifierRowLdeView {
            i,
            previous_i,
            x,
            x0,
            x_last,
            z_h_inverse,
            current_row,
            previous_row,
        };

        if first_round.left.value != eval_composition(air, &row, &
            challenges) {
            return Err(ZkvmVerifyError::VerificationFailed);
        }
    }

    fri_verify(
        &proof.fri_proof,
        &public_params.lde_domain,
        &public_params.fri_options,
        tx,
    )?;
```

```
    Ok(())
}
```

H Rust Basic Fibonacci Constraints

```
enum FibonacciColumns {
    TimeStep,
    SequenceValuesA, // a_{i-1}
    SequenceValuesB, // a_i
}

impl FibonacciColumns {
    pub const fn idx(self) -> usize {
        match self {
            Self::TimeStep => 0,
            Self::SequenceValuesA => 1,
            Self::SequenceValuesB => 2,
        }
    }
}

// n 128
const FIBONACCI_SEQUENCE_FINAL_VALUE: u128 = 251728825683549488150424261;

// ---------------- Constraints ----------------

struct TimeStepBoundary {
    time_col: usize,
}

impl<F: PrimeField> Constraint<F> for TimeStepBoundary {
    fn name(&self) -> &'static str {
        "time_step_boundary"
    }

    fn eval(&self, row: &dyn RowAccess<F>) -> F {
        let num = row.current_step_column_value(self.time_col) - F::one();
        let denom = row.x() - row.x0();
        num * denom.inverse().unwrap()
    }
}

struct TimeStepTransition {
    time_col: usize,
}
```

```
impl<F: PrimeField> Constraint<F> for TimeStepTransition {
    fn name(&self) -> &'static str {
        "time_step_transition"
    }

    fn eval(&self, row: &dyn RowAccess<F>) -> F {
        let t_cur = row.current_step_column_value(self.time_col);
        let t_prev = row.previous_step_column_value(self.time_col);
        (row.x() - row.x0()) * (t_cur - t_prev - F::one()) * row.
            z_h_inverse()
    }
}

struct ATransition {
    a_col: usize,
    b_col: usize,
}

impl<F: PrimeField> Constraint<F> for ATransition {
    fn name(&self) -> &'static str {
        "a_transition"
    }

    fn eval(&self, row: &dyn RowAccess<F>) -> F {
        let cur_a = row.current_step_column_value(self.a_col);
        let prev_b = row.previous_step_column_value(self.b_col);
        (row.x() - row.x0()) * (cur_a - prev_b) * row.z_h_inverse()
    }
}

struct BTransition {
    a_col: usize,
    b_col: usize,
}

impl<F: PrimeField> Constraint<F> for BTransition {
    fn name(&self) -> &'static str {
        "b_transition"
    }

    fn eval(&self, row: &dyn RowAccess<F>) -> F {
        let b_cur = row.current_step_column_value(self.b_col);
        let b_prev = row.previous_step_column_value(self.b_col);
        let a_prev = row.previous_step_column_value(self.a_col);
        (row.x() - row.x0()) * (b_cur - b_prev - a_prev) * row.z_h_inverse
            ()
    }
}
```

```
struct FinalValueBoundary {
    b_col: usize,
    final_value: u128,
}

impl<F: PrimeField> Constraint<F> for FinalValueBoundary {
    fn name(&self) -> &'static str {
        "final_value_boundary"
    }

    fn eval(&self, row: &dyn RowAccess<F>) -> F {
        let num = row.current_step_column_value(self.b_col) - F::from(self.
            final_value);
        let denom = row.x() - row.x_last();
        num * denom.inverse().unwrap()
    }
}
```

I Rust Padded Fibonacci Constraints

```
enum FibonacciColumns {
    TimeStep,
    SequenceValuesA, // a_{i-1}
    SequenceValuesB, // a_i
    TransitionControl,
    InitControl,
    TerminationControl,
}

impl FibonacciColumns {
    pub const fn idx(self) -> usize {
        match self {
            Self::TimeStep => 0,
            Self::SequenceValuesA => 1,
            Self::SequenceValuesB => 2,
            Self::TransitionControl => 3,
            Self::InitControl => 4,
            Self::TerminationControl => 5,
        }
    }

    pub fn name(self) -> String {
        match self {
            TimeStep => "t".to_string(),
```

```
            SequenceValuesA => "A".to_string(),
            SequenceValuesB => "B".to_string(),
            TransitionControl => "transition control".to_string(),
            InitControl => "init control a".to_string(),
            TerminationControl => "termination control".to_string(),
        }
    }
}

// n = 64
const FIBONACCI_SEQUENCE_FINAL_VALUE: u128 = 251728825683549488150424261;
const FIBONACCI_SEQUENCE_INIT_VALUE_A: u128 = 70492524767089125814114;
const FIBONACCI_SEQUENCE_INIT_VALUE_B: u128 = 114059301025943970552219;

// ---------------- Constraints ----------------
struct TimeStepBoundary {
    time_col: usize,
    init_ctrl_col: usize,
}

impl<F: PrimeField> Constraint<F> for TimeStepBoundary {
    fn name(&self) -> &'static str {
        "time_step_boundary"
    }

    fn eval(&self, row: &dyn RowAccess<F>) -> F {
        let boundary = row.current_step_column_value(self.time_col) - F::
            one();
        boundary * row.current_step_column_value(self.init_ctrl_col) * row.
            z_h_inverse()
    }
}

struct TimeStepTransition {
    time_col: usize,
    transition_ctrl_col: usize,
}

impl<F: PrimeField> Constraint<F> for TimeStepTransition {
    fn name(&self) -> &'static str {
        "time_step_transition"
    }

    fn eval(&self, row: &dyn RowAccess<F>) -> F {
        let transition = row.current_step_column_value(self.time_col)
            - row.previous_step_column_value(self.time_col)
            - F::one();
        transition * row.current_step_column_value(self.transition_ctrl_col
            ) * row.z_h_inverse()
```

```
    }
}

struct ATransition {
    a_col: usize,
    b_col: usize,
    transition_ctrl_col: usize,
}

impl<F: PrimeField> Constraint<F> for ATransition {
    fn name(&self) -> &'static str {
        "a_transition"
    }

    fn eval(&self, row: &dyn RowAccess<F>) -> F {
        let current_a = row.current_step_column_value(self.a_col);
        let previous_b = row.previous_step_column_value(self.b_col);
        let ctrl = row.current_step_column_value(self.transition_ctrl_col);
        (current_a - previous_b) * ctrl * row.z_h_inverse()
    }
}

struct BTransition {
    a_col: usize,
    b_col: usize,
    transition_ctrl_col: usize,
}

impl<F: PrimeField> Constraint<F> for BTransition {
    fn name(&self) -> &'static str {
        "b_transition"
    }

    fn eval(&self, row: &dyn RowAccess<F>) -> F {
        let current_b = row.current_step_column_value(self.b_col);
        let previous_b = row.previous_step_column_value(self.b_col);
        let previous_a = row.previous_step_column_value(self.a_col);
        let ctrl = row.current_step_column_value(self.transition_ctrl_col);
        (current_b - previous_b - previous_a) * ctrl * row.z_h_inverse()
    }
}

struct TerminationValue {
    b_col: usize,
    termination_ctrl_col: usize,
    final_value: u128,
}

impl<F: PrimeField> Constraint<F> for TerminationValue {
```

```
    fn name(&self) -> &'static str {
        "termination_value"
    }

    fn eval(&self, row: &dyn RowAccess<F>) -> F {
        let current_b = row.current_step_column_value(self.b_col);
        let ctrl = row.current_step_column_value(self.termination_ctrl_col)
            ;
        (current_b - F::from(self.final_value)) * ctrl * row.z_h_inverse()
    }
}

struct AInit {
    a_col: usize,
    init_ctrl_col: usize,
    init_value: u128,
}

impl<F: PrimeField> Constraint<F> for AInit {
    fn name(&self) -> &'static str {
        "a_init"
    }

    fn eval(&self, row: &dyn RowAccess<F>) -> F {
        let current_a = row.current_step_column_value(self.a_col);
        let ctrl = row.current_step_column_value(self.init_ctrl_col);
        (current_a - F::from(self.init_value)) * ctrl * row.z_h_inverse()
    }
}

struct BInit {
    b_col: usize,
    init_ctrl_col: usize,
    init_value: u128,
}

impl<F: PrimeField> Constraint<F> for BInit {
    fn name(&self) -> &'static str {
        "b_init"
    }

    fn eval(&self, row: &dyn RowAccess<F>) -> F {
        let current_b = row.current_step_column_value(self.b_col);
        let ctrl = row.current_step_column_value(self.init_ctrl_col);
        (current_b - F::from(self.init_value)) * ctrl * row.z_h_inverse()
    }
}

struct Booleanity {
```

```
    col: usize,
}

impl<F: PrimeField> Constraint<F> for Booleanity {
    fn name(&self) -> &'static str {
        "booleanity"
    }

    fn eval(&self, row: &dyn RowAccess<F>) -> F {
        let v = row.current_step_column_value(self.col);
        (v * (v - F::one())) * row.z_h_inverse()
    }
}
```