OPENVM WHITEPAPER

OPENVM CONTRIBUTORS

Abstract. OpenVM is a performant and modular zkVM framework built for customization and extensibility. OpenVM introduces a new "no-CPU" zkVM design paradigm where the execution trace is not materialized in any centralized chip and is instead managed collectively by all instruction executor chips. This no-CPU design allows OpenVM to support all functionality through VM extensions, which are groups of instructions interoperating through common memory spaces. Developers can add custom VM support to OpenVM using custom VM extensions without forking or modifying the core OpenVM libraries, and all default functionality in OpenVM, including RISC-V and recursion support, is implemented through a set of default VM extensions.

This paper describes the core OpenVM architecture, including the arithmetization and zkVM design, modular ISA design via VM extensions, support for unbounded programs and on-chain verification via proof recursion and continuations, and Rust toolchain support targeting the 32-bit RISC-V ISA.

Contents

1. Introduction

Recent advancements in zero-knowledge (ZK) proofs have led to the introduction of new tools to enable developers to use ZK. Systems such as Circom [ide25], halo2 [EP25], Gnark [Con23], and plonky3 [Pol25] provide tools to encode computation in an intermediate representation designed specifically for ZK. This approach provides the most flexibility and theoretically best performance by allowing developers to directly access the inputs to a ZK backend. However, writing circuits directly has high developer overhead and requires new security work for each ZK circuit.

A more recent class of ZK virtual machines (zkVMs) addresses these developer experience concerns by introducing specialized ZK circuits verifying the execution of programs targeting an instruction set architecture (ISA). These tools allow developers to use standard imperative

Date: March 31, 2025.
Contact: info@openvm.dev.

programming models while enabling verifiability in ZK, but do so at the expense of developer overhead to use a ZK-native ISA (Cairo [GPR21], Valida [Val24]) or performance overhead to support a standard ISA such as RISC-V (Risc0 [RIS25], SP1 [Suc25]).

This paper introduces OpenVM, a zkVM framework which balances flexibility and performance with a modular, developer-defined ISA that enables customization of zkVMs to specific use cases. By introducing a novel "no-CPU" zkVM design that decouples opcode implementations, OpenVM enables developers to do this by building custom VM extensions without forking or modifying the core OpenVM libraries. We describe the core OpenVM architecture for a modular zkVM, how it can be used to support proving unbounded program execution via recursion and continuations, and Rust toolchain support via the RISC-V target.

1.1. **Background and Motivation.** zkVMs have emerged as the most ergonomic means for developers to write verifiable software. However, existing zkVMs have struggled with trade-offs between performance and developer experience depending on the computer architecture they wish to emulate. Those designed for custom ZK-native ISAs require developers to learn new programming models, while others must make a fixed choice of traditional ISA to emulate and incur performance overheads.

OpenVM addresses these limitations by introducing a framework that:

- enables familiar development workflows using standard languages like Rust,
- provides an extensible ISA designed for custom instruction sets across diverse use cases,
- facilitates seamless composition of VM extensions,
- embraces a modular architecture to accommodate future proof system advancements.

1.2. **Design Principles.** The architecture and features of OpenVM are guided by a set of core design principles:

(1) **Modularity and Composability:** The framework is designed so that the ISA and zkVM circuit can be extended independently of the core system. We minimize the reliance on centralized components and ensure that new components can be added to the zkVM without requiring changes to the core system. This principle is embodied in the "no-CPU" design and the concept of VM extensions.
(2) **Performance:** We believe that developers should not have to choose between modularity and performance. On the contrary, we believe that modularity, when incorporated correctly, can provide more opportunities for performance optimization.
(3) **Developer Experience:** We ensure that customization of the zkVM integrates with existing developer workflows.
(4) **Future Proof:** By maintaining modular abstraction layers, we ensure that different parts of the system can be updated independently. Notably, we choose intermediate representations that enable the underlying ZK proof system to be upgraded without completely rewriting the zkVM circuit logic.

1.3. **OpenVM Architecture Overview.** The OpenVM architecture is comprised of the following main components. The circuit arithmetization framework and ZK backend (§2) provides the interface to represent circuit logic. We keep the circuit logic abstracted away from the implementation of the proof system, allowing the latter to be upgraded without affecting the former. The instruction set architecture (§3) defines the virtual machine execution model and the foundations to support VM extensions. The zkVM design (§4) explains our circuit architecture which enables zkVMs to be extended to support custom instruction set extensions

without core system changes. In §5, we explain how we prove unbounded program execution using continuations and proof recursion (see §5 for definitions). The framework is designed so that custom extensions have first-class support in the Rust toolchain (§6), which is achieved by synchronizing OpenVM extensions with those in the RISC-V ISA.

Acknowledgments. OpenVM is a new zkVM design framework. In the process of building it, we studied and learned from the designs and implementations of many projects. We would like to especially thank StarkWare [Sta21] for their pioneering work on STARKs, Plonky3 [Pol25] for their modular design of polynomial IOPs which is currently used by our ZK backend, Valida [Val24] for new designs around circuit arithmetization and ZK-native ISAs, RISC Zero [RIS25] for their foundational work on Rust toolchain integration for zkVMs, and SP1 [Suc25] for their precompile-centric design and zkVM based recursion programs.

2. Arithmetization Framework and ZK Backend

We describe the arithmetization framework that we use to design ZK circuits. This arithmetization framework represents a *ZK frontend* which we maintain as an abstraction boundary between the representation of the circuit and the proof generation process. We then discuss how our *ZK backend* provides a general framework for generating proofs from circuits designed using this arithmetization. Our design allows the ZK backend to be upgraded for newer proof systems without changing the frontend.

2.1. Notation.

For the rest of this paper, we fix the base field $\mathbb{F}$ used in the arithmetization and the proof system. We assume that $\mathbb{F}$ is a prime field of characteristic p where p has 31 bits. Our present implementation uses the BabyBear prime $p = 15 \cdot 2^{27} + 1 = 2^{31} - 2^{27} + 1$, but this choice is not fundamental to the design and may be changed in the future.

The proof system also requires a choice of extension field $\mathbb{F}_{\text{ext}}$ over $\mathbb{F}$. The extension field is used to boost the security of the proof system. The extension field is chosen so that the order $|\mathbb{F}_{\text{ext}}|$ is at least 2^{120}. Presently, the quartic extension field $\mathbb{F}[x]/(x^4 - 11)$ of BabyBear is used.

2.2. AIRs with interactions.

Modern ZK systems use *arithmetization* to represent circuits, which may constrain arbitrary computation, in an intermediate representation that is more readily processed by the proof system. The choice of arithmetization is crucial as it determines how circuits will be designed and represented in the system.

Our arithmetization framework is an extension of the *algebraic intermediate representation* (AIR) framework, with extensions tailored for newer proving techniques such as LogUp [Hab22a, PH23]. We refer to this framework as **AIRs with Interactions**.

AIRs were first introduced by [BBHR18b] and have been prevalent in STARK proof systems since. The precise definition of an AIR varies slightly in the literature, so we provide a definition below to clarify our terminology.

Definition 2.2.1 (AIR)**.** An algebraic intermediate representation (AIR) is a set of pairs (C_i, S_i), where $C_i \colon \mathbb{F}[x_1, \ldots, x_w, y_1, \ldots, y_w]$ are *constraint polynomials* and w is a fixed width associated with the AIR. The S_i are special selectors, to be explained below, which can be one of All, First, Last, $\mathsf{Transition}$.

A *trace matrix* is a matrix with entries in $\mathbb{F}$ where the number of rows is a power of two. We define the height of the matrix to be the number of rows and the width to be the number of columns. We say that a trace matrix $\mathbf{T}$ satisfies the AIR $\{(C_i, \mathsf{S}_i)\}$ if the width of $\mathbf{T}$ equals

w and for each i the constraint polynomial $C_i(x_1, \ldots, x_w, y_1, \ldots, y_w)$ evaluates to zero on the following domain:

- If S_i is All, then this applies to all pairs of cyclically consecutive rows $(x_1, \ldots, x_w)$, $(y_1, \ldots, y_w)$ of $\mathbf{T}$.
- If S_i is First, then this applies to the first pair of cyclically consecutive rows of $\mathbf{T}$.
- If S_i is Last, then this applies to the pair (last row, first row) of $\mathbf{T}$.
- If S_i is $\mathsf{Transition}$, then this applies to all pairs of non-cyclically consecutive rows – that is, like All but except (last, first).

Note that in our definition, the height of the trace matrix is not specified by the AIR – trace matrices of different heights can satisfy the same AIR.

In our arithmetization, we also allow each AIR to specify a partition of the columns $\{1, \ldots, w\}$ of the trace matrix into parts of different types:

- Preprocessed
- Cached
- Common

This partitioning specifies to the ZK backend how data is supplied for different parts of the trace matrix. The *preprocessed trace* is data that shared and agreed upon ahead of time between the prover and verifier. The *cached trace* is data that is only available to the prover, but may be cached and reused across different proofs. Lastly the *common trace* is the remaining data only available to the prover. We note that if an AIR requires a preprocessed trace, then the height of the trace matrix is fixed.

For greater flexibility, we allow multiple AIRs in the arithmetization of a single circuit. In other words, a circuit is proved by providing multiple trace matrices. We extend the AIR arithmetization framework with an intermediate representation for constraining relations between different AIRs. This intermediate representation, known as *interactions*, was first introduced by [Val24], building on previous interfaces for lookup tables and permutation arguments.

Definition 2.2.2 (Interactions and buses)**.** An *interaction* of width w and message length ℓ on *bus* b is a triple (σ, m, b), where:

- $\sigma \in \mathbb{F}[x_1, \ldots, x_w, y_1, \ldots, y_w]^\ell$ is a sequence of ℓ polynomials defining the **message**.
- $m \in \mathbb{F}[x_1, \ldots, x_w, y_1, \ldots, y_w]$ is a polynomial that determines the **multiplicity** of the corresponding message.
- $b \in \mathbb{F} \setminus \{0\}$ is the **bus index** specifying the bus. It must be nonzero.

Given a trace matrix $\mathbf{T}$ of width w with entry $\mathbf{T}_{ij}$ on row i and column j, we say that an interaction (σ, m, b) defined on $\mathbf{T}$ *sends* over bus b, for each row i, the image

$$\sigma(\mathbf{T}_{i1}, \ldots, \mathbf{T}_{iw}, \mathbf{T}_{\mathrm{next}(i)1}, \ldots, \mathbf{T}_{\mathrm{next}(i)w}) \in \mathbb{F}^\ell$$

with multiplicity

$$m(\mathbf{T}_{i1}, \ldots, \mathbf{T}_{iw}, \mathbf{T}_{\mathrm{next}(i)1}, \ldots, \mathbf{T}_{\mathrm{next}(i)w}) \in \mathbb{F}.$$

where $\mathrm{next}(i)$ is the cyclic next row in the trace matrix, i.e., $\mathrm{next}(i) = i + 1$ if i is not the last row and next of last row is first row.

In our arithmetization, an AIR is augmented with a set of interactions, where the AIR width and interaction width coincide. An AIR may have multiple interactions, where each interaction may have a different message length and/or bus index.

Definition 2.2.3 (Circuit)**.** A *circuit* $\mathcal{C}$ is a collection of (A, I) where A is an AIR and I is a collection of interactions associated with A.

2.2.4. $\mathbb{F}$*-multiset balancing.* In order to define what it means for a collection of trace matrices to satisfy a circuit, we introduce a notion of $\mathbb{F}$-multiset balancing below.

Consider a circuit with t AIRs of widths $w_1, \ldots, w_t$, where the i-th AIR has k_i interactions $(\sigma_k^{(i)}, m_k^{(i)}, b_k^{(i)})$ of length ℓ_{ik} for $k \in \{1, \ldots, k_i\}$.

The set of possible messages is denoted $\mathbb{F}^+ = \bigsqcup_{i \geq 1} \mathbb{F}^i$ (disjoint union). An $\mathbb{F}$-*multiset* is a function

$$\mathscr{M} : \mathbb{F}^+ \to \mathbb{F}$$

that assigns an $\mathbb{F}$-valued "multiplicity" to each message $\mathbb{F}^+$.

Given a set of trace matrices $\mathbf{T}^{(1)}, \ldots, \mathbf{T}^{(t)}$ with respective heights $h_1, \ldots, h_t$, these traces together define a multiset $\mathscr{M}_b$ for each bus index b. To simplify notation, for $i \in \{1, \ldots, t\}$ (per AIR), $j \in \{1, \ldots, h_i\}$ (per trace row), and $k \in \{1, \ldots, k_i\}$ (per interaction), define $\hat{m}_{j,k}^{(i)} \in \mathbb{F}$ by:

$$\hat{m}_{j,k}^{(i)} = m_k^{(i)}(T_{j,1}^{(i)}, \ldots, T_{j,w}^{(i)}, T_{\mathrm{next}(j),1}^{(i)}, \ldots, T_{\mathrm{next}(j),w}^{(i)})$$

and define $\hat{\sigma}_{jk}^{(i)} \in \mathbb{F}^{\ell_{ik}}$ analogously. The multiset defined by the traces is then given by:

$$\mathscr{M}_b(\tau) = \sum_{i=1}^{t} \sum_{k=1}^{k_i} \sum_{j=1}^{h_i} \hat{m}_{jk}^{(i)} \mathbf{1}(b = b_k^{(i)} \wedge \tau = \sigma_{jk}^{(i)}), \quad \tau \in \mathbb{F}^+.$$

where $\mathbf{1}(b = b_k^{(i)} \wedge \tau = \sigma_{jk}^{(i)})$ is the indicator function that is 1 if $b = b_k^{(i)}$ and $\tau = \sigma_{jk}^{(i)}$ and 0 otherwise. Here we implicitly embed $\sigma_{jk}^{(i)}$ into $\mathbb{F}^+$.

We say that a bus b is *balanced* with respect to the trace matrices if the $\mathbb{F}$-multiset $\mathscr{M}_b$ satisfies $\mathscr{M}_b(\tau) = 0$ for all messages $\tau \in \mathbb{F}^+$.

Under certain conditions which prevent integer overflow of the field characteristic, $\mathbb{F}$-multiset balancing can be used to imply:

- Lookup table relations: a single AIR declares a set of messages and other AIRs may constrain that a given message belongs to this set.
- Permutation checks: AIRs may add messages with positive integer multiplicities to one of two multisets (a "send" and "receive" multiset), and the bus constrains that the two multisets are equal.

These constraints will play a key role in our zkVM design, cf. §4.

2.2.5. *Circuit satisfiability.* Given a circuit $\mathcal{C}$, we say that a collection of trace matrices $(\mathbf{T}_A)_A$, one per AIR A in $\mathcal{C}$, satisfies the circuit if each trace matrix satisfies the corresponding AIR and all interaction buses are balanced with respect to the trace matrices.

2.3. **ZK backend.** Our ZK backend provides a framework to separate the implementation of the proof system from the design of the circuit by using the AIRs with interactions arithmetization as an intermediate representation.

The goal of the ZK backend is to allow different proof system implementations, in different hardware environments, to take a common input format — a circuit described as AIRs with interactions, together with a collection of trace matrices — and produce a non-interactive proof of knowledge that the trace matrices satisfy the circuit.

The backend models the proof system as a non-interactive proof system derived from a polynomial IOP via the Fiat-Shamir transform. Notably the backend does not force a choice of PIOP. The backend framework organizes the proof system implementation into the following components, which can be customized based on the proof system and hardware environment:

- **Trace Commitment**: The trace matrices are committed to by the prover, and the verifier only receives a commitment. This is typically chosen for compatibility with a polynomial commitment scheme where each column of the trace matrix is interpolated into a polynomial. The backend supports mixed matrix commitment schemes, where multiple matrices of different heights can be committed to together.
- **Partial Proving with Verifier Randomness**: The prover may generate additional trace data or impose additional constraints by simulating verifier injected randomness via the Fiat-Shamir transform. The prover may commit to additional trace data that depends on this randomness. As an example, this component may be used to partially prove interactions using LogUp techniques [Hab22a].
- **Constraint Evaluation**: The prover may generate additional data or simulate further randomness to provide the data necessary for a verifier to check that constraints are satisfied.
- **Commitment Opening**: In a proof system derived from a polynomial IOP, the prover generates all necessary polynomial opening proofs. Analogously, the verifier must verify these opening proofs.

We have designed the above components to maximize flexibility and support for future proof system developments.

We describe the implementations of the above components at the time of writing of this paper to serve as an example of how the framework can be used. The proof system is based on FRI [BBHR18a, BCI+20, Hab22b] as a univariate polynomial commitment scheme. The trace commitment is a combination of low degree extension (LDE) of univariate polynomials via discrete Fourier transform (DFT) and coset DFT together with a modified Merkle tree commitment that commits to matrices of possibly differing heights. The partial proving step generates additional trace matrices and AIR constraints using simulated verifier randomness to constrain the computation of LogUp partial sums necessary to prove the balancing of interaction buses. For constraint evaluation, DEEP-ALI [BGKS19] is used to generate the univariate quotient polynomial. The commitment opening is a batch-FRI polynomial opening.

3. Instruction Set Architecture

We introduce the OpenVM instruction set architecture (ISA). The goal of the ISA is to provide a common open and extensible architecture that can be used for:

- Emulation of existing computer architectures (e.g., RISC-V, WASM, x86, aarch, etc.),
- Custom programs that desire lower-level integration with the underlying ZK circuit arithmetization,
- Hybrid combinations of the above within *the same* execution environment.

The ISA is intentionally designed for virtual software execution which can be provably verified. While traditional ISAs must fix certain properties of the ISA (e.g., register size) to optimize for performance on a fixed set of hardware, the OpenVM ISA is freed from many of these restrictions and provides a platform where these traditionally conflicting properties can all coexist within the same architecture.

The ISA globally depends on a prime field $\mathbb{F}$, which is the base field used in the circuit arithmetization. We identify the elements of this field with integers in the range $[0, p)$ where p is the characteristic of the field.

3.1. **Virtual machine execution environment.** Programs in the OpenVM ISA are executed in a virtual machine (VM). The VM is a state machine where program execution occurs within a *guest* environment that modifies a guest state. The VM is executed on a physical *host* machine, which maintains an external host state. The ISA provides certain interfaces that allow the guest program to request or modify the host state.

3.2. **Virtual machine state.** The state of the virtual machine consists of the following components:

Guest State	**Host State**
Program ROM (Read Only)	Input Stream
Program Counter pc	Hint Stream
Data Memory (Read/Write)	Hint Spaces
User Public Outputs	

We describe these components in more detail below.

3.2.1. *Program ROM.* OpenVM operates under the Harvard architecture, where program code is stored separately from data memory. The program code is loaded as read-only memory (ROM) in the VM state prior to execution, and it remains immutable throughout the execution. Program code is a partially defined map

$$[0, 2^{\texttt{PC\_BITS}}) \to \mathbb{F} \times \mathbb{F}^{\texttt{NUM\_OPERANDS}}$$

where $\texttt{PC\_BITS} = 30$ and the target is the space of instructions (see below). Instructions will typically only exist at a subset of the domain $[0, 2^{\texttt{PC\_BITS}})$.

3.2.2. *Instruction format.* Instructions are encoded as a global opcode (field element) followed by $\texttt{NUM\_OPERANDS} = 7$ operands (field elements):

$$\mathsf{Inst} = (\mathsf{opcode}, \text{operands}) \in \mathbb{F} \times \mathbb{F}^{\texttt{NUM\_OPERANDS}}.$$

The maximum number of operands may be increased in the future without affecting existing instructions.

3.2.3. *Program counter.* There is a single special purpose register $\mathsf{pc} \in \mathbb{F}$ for the program counter which stores the location of the instruction being executed.

3.2.4. *Data memory.* Data memory is a random access memory (RAM) which supports read and write operations. Memory is comprised of addressable cells which represent a single field element indexed by *address space* and *pointer*, where address space and pointer are each a field element. Memory may be viewed as a partially defined map

$$\mathbb{F} \times \mathbb{F} \to \mathbb{F} : (\texttt{addr\_space}, \texttt{ptr}) \mapsto [\texttt{ptr}]_{\texttt{addr\_space}}$$

VM instructions can access (read or write) a contiguous list of cells (called a *block*) in a single address space. The block size must be a power of two, with a configurable upper limit. In other words, instructions may independently specify different sizes for atomic memory accesses.

3.2.5. *Address spaces.* To support execution of programs with different data assumptions and properties within the same architecture, we separate data memory into different *address spaces*. While memory cells within each address space can always be viewed as field elements, the ISA may impose additional invariants on certain address spaces to optimize the performance of sub-classes of instructions.

The address spaces below are reserved and used by existing OpenVM instructions: The

Address Space	Name	Description
1	Registers	Elements are constrained to lie in $[0, 2^8)$
2	User Memory	Elements are constrained to lie in $[0, 2^8)$
3	User IO	Used to expose user outputs
4	Native	Elements are native field elements.

address space 0 is reserved and not used by data memory[1].

3.2.6. *Inputs and hints.* To enable user input and non-determinism in OpenVM programs, the host state must maintain an input stream, hint stream, and hint space. The input stream is a non-interactive queue of data that the host is instantiated with prior to program execution. The hint stream and space provide the only way for the guest program to request or modify data from the host.

The ISA allows the addition of custom instructions that can request modification of the host state by either moving data from the input stream to the hint stream/space or by specifying preferred operations that the host should perform to mutate its state (e.g., the host may perform a square root of a value in guest memory). The guest may also copy data from the host state to guest memory via explicit instruction calls.

3.2.7. *User public outputs.* To make program outputs public, OpenVM allows the guest program to specify a list of field elements to make public. This list can be modified by certain custom instructions.

3.3. **VM execution model.** The VM is a state machine, and we view VM execution as a series of state transitions

$$\mathsf{State}_0 \xrightarrow{\mathsf{Step}} \mathsf{State}_1 \xrightarrow{\mathsf{Step}} \cdots \xrightarrow{\mathsf{Step}} \mathsf{State}_{\text{final}}$$

We define the conditions for a state transition to be valid below. Program execution in the VM is declared *successful* with respect to the initial state if all state transitions are valid and the instruction at the program counter $\mathsf{pc}_{\text{final}}$ in the final state is a special terminating instruction `TERMINATE` with exit code 0 (success).

Program execution may *fail* for the following reasons:

(1) The sequence of state transitions are valid and the final program counter yields the `TERMINATE` instruction with a non-zero exit code. This is a requested trap by the guest program to exit unsuccessfully.
(2) The sequence of state transitions results in an invalid state transition. This indicates that the VM state no longer satisfies the required properties of either the architecture or the specific instruction being executed.

[1]Address space 0 is used as an operand flag to specify immediate values in instructions.

3.3.1. *Guest program execution.* We define *guest program execution* to be the subset of VM execution that only mutates the guest state:

- program counter
- data memory
- user public outputs.

Guest program execution can be modeled as a series of state transitions on the guest state, assuming oracle access to a non-deterministic host state.

3.3.2. *Initial state.* The initial state of the VM consists of:

- Program ROM – immutable throughout VM execution
- pc_0 – starting program counter
- Initial data memory
- Input stream

The user public outputs, hint stream, and hint spaces are empty.

3.3.3. *State transition.* State transition is a function

$$\mathsf{Step} : \mathsf{State}_{\text{from}} \to \mathsf{State}_{\text{to}}.$$

consisting of the following stages:

Instruction Fetch: The instruction is fetched from the program ROM based on the current program counter $\mathsf{pc}_{\text{from}} \in \mathsf{State}_{\text{to}}$.

Instruction Routing: The opcode is parsed from the instruction and used to decide how instruction execution will be handled.

Instruction Execution: Execution of the instruction is custom according to specification of the instruction. Instruction execution is not required to be atomic and has mutable access to the global VM state:

- Program Counter
- Data Memory
- User Public Outputs
- Input Stream
- Hint Stream
- Hint Spaces

Local state may be maintained during execution of a single instruction, but the only state that is persisted and shared outside of the lifecycle of execution of a single instruction is the VM state above.

Instruction execution is expected to end with an update of the program counter to pc_{to}. Instructions that do not pertain to control flow will advance $\mathsf{pc}_{\text{to}} = \mathsf{pc}_{\text{from}} + \texttt{DEFAULT\_PC\_STEP}$ where $\texttt{DEFAULT\_PC\_STEP} = 4$ is set for compatibility with RISC-V conventions.

The state transition is valid if a valid instruction is fetched from the program ROM and the instruction's execution maintains the required invariants of the VM state. In particular, memory access bounds and required properties of memory address spaces must be respected. The pc_{to} program counter must also be set to a valid instruction.

3.3.4. *Phantom sub-instructions.* To facilitate hinting and debugging on the host, the ISA supports the notion of *phantom instructions*. These are instructions which are identical to a no-op at the level of the guest state, but which may be used to request mutations in the host state. Use cases of phantom instructions include interacting with the input or hint streams or displaying debug information on the host machine.

3.4. **VM extensions.** The ISA is designed to be maximally extensible. Outside of a few system instructions, there is no required "base" instruction set. Instead, instructions are grouped into composable **VM extensions** which are instruction set extensions to the ISA meant to provide the highest degree of customization and performance for VM execution.

A VM extension consists of a collection of instructions, which must adhere to the ISA invariants previously described in this section, and the specification for each instruction of:

(1) The guest instruction execution,
(2) The preferred host behavior during execution,
(3) The implementation of the instruction within the circuit architecture.

We discuss item (3) in more detail in §4.

VM extensions allow new instructions to be easily introduced to accelerate custom workloads as composable "add-ons" to existing instruction sets. This takes full advantage of the virtual nature of the execution environment to dramatically lower the barrier to customizing computer architecture.

4. zkVM Design

We describe the design of our ZK circuit architecture that enables creation of zkVMs for the OpenVM ISA defined in §3. Since the OpenVM ISA is not a single instruction set but a framework to define custom instruction sets from composable VM extensions, we clarify our definition of a zkVM:

Definition 4.0.1 (zkVM)**.** Given a fixed set of instructions within the OpenVM ISA, a ZK virtual machine (zkVM) is the collection of:

(1) A host execution environment capable of executing every instruction in the instruction set.
(2) A circuit $\mathcal{C}$ as defined in Definition 2.2.3 that constrains a finite sequence of guest state transitions are valid with respect to a public commitment to an initial guest state. The circuit exposes a commitment of the final guest state as public output. The state transitions are presented in the form of trace matrices.
(3) Methods to generate the trace matrices from the guest state transitions necessary to generate a proof for the circuit $\mathcal{C}$.

We note that the zkVM circuit only supports a *finite* sequence of state transitions, whereas program execution is *a priori* unbounded. We describe how we prove successful guest execution of unbounded programs using continuations in §5.2.

Further note that the zkVM circuit only constrains validity of guest state transitions. The circuit *does not* constrain host behavior. As such, the guest program always operate under the assumption that the host environment is *untrusted*. The guest program may use control flow to handle preferred host behavior, but it must ensure that program execution cannot be tampered with by a dishonest host.

In the rest of this section, we explain the framework which enables the creation of zkVMs for *any* instruction set defined within the OpenVM ISA. In particular, the framework enables zkVM circuits to be extended to support new VM extensions via composition.

4.1. **Overview.** At a high level, we organize the zkVM into *chips*, where a chip contains a single AIR and the methods to generate the trace matrix for that AIR. Our zkVM consists of a collection of system chips required by the architecture together with an *inventory* of chips defined by VM extensions.

Chips come in two types:

- Instruction executor
- Periphery

An instruction executor chip owns the execution methods for a subset of the instructions in the instruction set. Each instruction must be handled by exactly one chip, but the same chip may handle multiple instructions. Periphery chips provide auxiliary functionality such as lookup tables that may be shared by other chips. Each chip may expose interfaces for communicating with other chips, even those defined by other VM extensions.

Instruction execution is distributed across multiple chips. In the host execution environment, instruction parsing, routing, and the global VM state are managed by a centralized controller. However in the zkVM circuit, there is *no centralized entity*: the constraints on valid state transitions are enforced entirely through interaction buses, which we explain below.

We will focus on the design of the circuit architecture, as that dictates the other aspects of the zkVM implementation.

4.2. **Temporal execution in circuit.** We revisit the VM execution model introduced in §3.3. Our VM execution implicitly incorporates temporal logic – execution is treated as a serial sequence of state transitions continually progressing forward in time. On the other hand, the circuit arithmetization does not inherently provide a representation of temporal constraints.

In order to represent constraints of temporal logic within the zkVM circuit, we discretize time and materialize a global timestamp t as a separate variable in circuit constraints. The execution trace must associate a timestamp t_i with each state State_i that appears in a state transition. Given state transitions

$$\mathsf{State}_0 \xrightarrow{\mathsf{Step}} \mathsf{State}_1 \xrightarrow{\mathsf{Step}} \cdots \xrightarrow{\mathsf{Step}} \mathsf{State}_{\text{final}}$$

where each Step represents execution of a single instruction, the circuit must constrain:

(1) **Continuity between instructions**: The timestamps must be monotonically increasing:
$$t_0 < t_1 < t_2 < \cdots < t_{\text{final}}.$$

(2) **Continuity within an instruction**: Execution of a single instruction consists of a finite sequence of atomic accesses to the guest state. The execution trace must associate an intermediate timestamp $t_{i,j}$ to each guest state access such that
$$t_i < t_{i,1} < t_{i,2} < \cdots < t_{i,k} < t_{i+1},$$
and these discrete timestamps represent the only points at which the guest state is accessed. A state access includes any read or write.

Recall that the guest state consists of the program ROM, pc, data memory, and user public outputs. To simplify the design, we require that the program ROM is only read at the boundaries t_i, t_{i+1} of a $\mathsf{Step} : \mathsf{State}_i \to \mathsf{State}_{i+1}$. Likewise, the program counter

is only accessed and updated at timestamps t_i, t_{i+1}. Given this, the only state accesses at intermediate timestamps $t_{i,j}$ are of the data memory and user public outputs. To summarize, a single instruction execution can be decomposed into a sequence of smaller state transitions

$$\overset{\curvearrowright\ \mathsf{Step}\ \curvearrowright}{(\mathsf{State}_i, t_i) \xrightarrow{f_{i,0}} (\mathsf{State}_{i,1}, t_{i,1}) \xrightarrow{f_{i,1}} \cdots \xrightarrow{f_{i,k}} (\mathsf{State}_{i+1}, t_{i+1})}$$

where each intermediate $t_{i,j}$ marks an atomic access to data memory or user public outputs, and the transitions functions $f_{i,j}$ contain no other state accesses.

4.3. **System buses.** The circuit architecture relies on three system buses to constrain valid state transitions during program execution:

- Program bus
- Execution bus
- Memory bus

We explain each bus's message format and how it is used within the architecture below.

4.4. **Program bus.** The program bus ensures the correctness of instruction fetching during execution. A message on the program bus has the form:

$$(\mathsf{pc}, \mathsf{opcode}, \text{operands})$$

where pc is the program counter and $\mathsf{opcode}, \text{operands}$ represents an instruction.

The program bus is used as a lookup bus for instruction executor AIRs to constrain that the instruction they are executing indeed exists in the program ROM at the given program counter. This is enabled by a *program chip*, whose trace matrix contains the program ROM as a cached trace with one instruction per row. The cached trace is committed to separately from the rest of the zkVM trace so that its commitment is a commitment of the program ROM. In this way, the program ROM is *not* a fixed part of the circuit constraints. Instead the prover can load different program ROMs, hence initializing the VM with different states, to generate proofs of execution for *the same* circuit.

4.5. **Execution bus: the no-CPU design.** The execution bus constrains the continuity of program execution between instructions and also constrains state accesses to the program counter. A message on the execution bus has the form:

$$(\mathsf{pc}, t)$$

Chips that interact with the execution bus must maintain the invariant that a message (pc, t) appears in the bus if and only if the program counter equals pc in the guest state at timestamp t during program execution.

The execution bus is used to perform a permutation check between a "send" and "receive" set of messages. The architecture requires that every instruction executor AIR *must* constrain that it adds a message $(\mathsf{pc}_{\text{from}}, t_{\text{from}})$ to the receive set and a message $(\mathsf{pc}_{\text{to}}, t_{\text{to}})$ to the send set exactly once for each instruction that appears in the AIR trace. The AIR must also constrain that $t_{\text{from}} < t_{\text{to}}$.

A *connector chip* adds a message $(\mathsf{pc}_0, 1)$ to the send set and a message $(\mathsf{pc}_{\text{final}}, t_{\text{final}})$ to the receive set and exposes $(\mathsf{pc}_0, \mathsf{pc}_{\text{final}})$ as public outputs.

Assuming that the instruction executor AIRs enforce the constraints above, the send and receive sets are equal if and only if the read and write access to the program counter is consistent across all instructions executed. In other words, it constrains that the routing of instructions and the transition between instructions is consistent.

The execution bus allows the transcript of program execution to be distributed across the traces of multiple chips, and there does not need to be a single central chip that materializes the full transcript. Prior zkVM designs have traditionally relied on a single central processing unit (CPU) chip to materialize this complete transcript. The existence of a CPU chip leads to a trace matrix with rows that grow with the total number of clock cycles in program execution, which creates a performance bottleneck. Our use of the execution bus avoids this bottleneck, which is why we colloquially refer to our design as the "no-CPU" architecture.

4.6. **Memory bus.** The memory bus is an adaptation of the offline memory checking argument of [BEG+94]. The main modification we make is that we allow the memory bus to contain messages of *different* lengths. A message on the memory bus has the form:

$$(\texttt{addr\_space}, \texttt{ptr}, \texttt{data}, t)$$

where $\texttt{addr\_space}, \texttt{ptr}, t \in \mathbb{F}$ and $\texttt{data} \in \mathbb{F}^N$ where N is the block size of the memory access. The bus allows N to be a variable power of two, up to a maximum block size specified by the instruction set. Chips that interact with the memory bus must maintain the invariant that a message $(\texttt{addr\_space}, \texttt{ptr}, \texttt{data}, t)$ appears in the bus if and only if at timestamp t the data memory had values $\texttt{data}$ in address space $\texttt{addr\_space}$ at pointers $[\texttt{ptr}, \texttt{ptr} + N)$ during program execution.

4.6.1. *Offline Memory Checking: Handling Read and Write Operations.* The memory bus is used to perform a permutation check between a "send" and "receive" multiset of messages. Each instruction executor chip in the system independently maintains memory consistency by interacting with the send and receive multisets during both read and write operations.

Read Operation: To constrain a read operation, an AIR must

(1) add a message $(\texttt{addr\_space}, \texttt{ptr}, \texttt{data}, t_{\text{prev}})$ to the receive set, and
(2) add a message $(\texttt{addr\_space}, \texttt{ptr}, \texttt{data}, t)$ to the send set.

The AIR must constrain that $t_{\text{prev}} < t$. The value of t_{prev} should be the maximum over the timestamps of the last accesses to $(\texttt{addr\_space}, \texttt{ptr} + i)$ for $i \in [0, N)$ prior to timestamp t.

Write Operation: To constrain a write operation, an AIR must

(1) add a message $(\texttt{addr\_space}, \texttt{ptr}, \texttt{data}_{\text{prev}}, t_{\text{prev}})$ to the receive set, and
(2) add a message $(\texttt{addr\_space}, \texttt{ptr}, \texttt{data}, t)$ to the send set.

The AIR must constrain that $t_{\text{prev}} < t$. The value of t_{prev} should be the maximum over the timestamps of the last accesses to $(\texttt{addr\_space}, \texttt{ptr} + i)$ for $i \in [0, N)$ prior to timestamp t, and the value of $\texttt{data}_{\text{prev}}$ should be the last values of those cells in data memory.

4.6.2. *Initial and Final Memory States.* A memory *boundary chip* is used to add messages to the send multiset at timestamp 0 and to the receive multiset at timestamp t_{final}. In order for the bus to balance, the messages at timestamp 0 (resp. t_{final}) must correspond to the initial (resp. final) memory state.

We have two different boundary chips, which are used in different scenarios:

- **Volatile**: The volatile boundary chip sets the initial memory state to an arbitrary state specified by the host. The initial memory state is *not* exposed as a public value or commitment, and the guest program should assume that all memory cells are dirty. The final memory state is also not exposed, so no part of the memory state is persisted outside of the program execution. The volatile boundary chip is used in recursion circuits, where continuations are disabled (see §5).
- **Persistent**: The persistent boundary chip exposes a Merkle root of the initial memory state as a public value and constrains the messages at timestamp 0 to be consistent with the Merkle root via Merkle proofs, which are constrained by an auxiliary chip. The persistent boundary chip also exposes a Merkle root of the final memory state and constrains that the messages at timestamp t_{final} are consistent with this Merkle root. The persistent boundary chip is used when continuations are enabled (see §5).

4.6.3. *Memory access adapter chips.* Since we allow messages of different lengths on the memory bus, we need to introduce *access adapter chips* to ensure that the bus remains balanced. The read and write operations described above operate on a fixed-size memory block. Meanwhile the ISA allows memory access of different block sizes. The idea behind access adapter chips is that they provide "just-in-time" conversions between messages of different block sizes to keep the memory bus balanced.

Since block sizes are powers of two, the only necessary conversions are:

- Splitting a block into two blocks of half the size.
- Merging two consecutive blocks of the same size into a block of double the size.

4.7. **Public values.** The public values of a zkVM circuit are as follows:

(1) The initial and final program counters $\mathsf{pc}_0, \mathsf{pc}_{\text{final}}$.
(2) (Continuations enabled) Merkle roots of the initial and final memory states.
(3) (Continuations disabled) User public outputs.

The way that user public values are handled differs depending on whether continuations are enabled. When continuations are enabled, the user public outputs are stored within the data memory in a dedicated address space – they are not public values of the zkVM circuit because they will be extracted from the final memory Merkle root during the aggregation process for continuations. When continuations are disabled, a special `PUBLISH` instruction may be used in the guest program to set a fixed number of circuit public values.

4.8. **Hinting and phantom sub-instructions.** The system provides a *phantom chip* to handle phantom sub-instructions. The phantom chip's AIR interacts only with the program and execution bus and advances the program counter by the default step size and performs no other state accesses. However the phantom chip's host execution can register custom behavior for phantom sub-instructions. The host execution of these sub-instructions can read both guest and host state and modify the hint stream and hint space in the host state.

Special instructions may write to the guest data memory based on values read from the host state. The AIRs associated to these instructions must constrain the memory writes via the memory bus as required for all guest state accesses.

4.9. **VM extension support.** The zkVM design is centered around support for custom VM extensions. An existing zkVM circuit can be extended to support the instructions from a new VM extension (assuming they are compatible with existing instructions) by adding new chips – both instruction executor chips and periphery chips – to handle execution of the new

instructions. The VM extension may specify new buses for cross-chip communication and also use existing buses from other extensions. The chips are provided with interfaces to access the full global VM state during instruction execution. The circuit architecture requires that the chips' AIRs must interact with the three system buses as described above. Chip design is fully modular because the architecture gives each chip full control of the state transition for the instructions that it executes.

5. Recursion and Continuations

We describe how our zkVM design supports proofs of execution of unbounded programs using continuations. Our high-level continuations framework follows previous designs [RIS25], but introduces novel enhancements enabled by our modular zkVM design.

5.1. **Overview.**

As noted in Definition 4.0.1, the zkVM circuit only supports a finite sequence of state transitions. To prove unbounded program execution, the overall execution of a program is broken into *segments*, where each segment is a finite sequence of state transitions. Each segment generates trace matrices for its own state transitions and the same zkVM circuit is used to independently prove the validity of each segment's state transitions.

The proofs of an unbounded number of segments are merged into a single proof using the technique of *recursive proof aggregation*. We recursively verify ZK proofs by creating a special zkVM using a *native* VM extension optimized for this purpose.

Although VM program execution is inherently serial, our design aims to maximize parallelism in proof generation to support distributed proving environments.

5.2. **Continuations via Merkleized memory.**

To optimize for parallel proof generation, we minimize communication during proof generation between different segments. Therefore the proof for each segment must be self-contained, relying only on data that can be generated from the state transitions within the segment.

In order for proof aggregation to check consistency of boundary states between segments, the segment proof must expose commitments of the boundary guest states:

(1) The program ROM does not change between segments and is recorded as a separate cached trace commitment in the proof (cf. §4.4).
(2) The initial and final program counters of the segment are exposed as public values by the connector chip (cf. §4.5). The connector chip also exposes an *exit code* public value, indicating whether the segment's final state corresponds to program termination or an intermediate boundary state.
(3) As described in §4.6.2, when continuations are enabled the persistent memory boundary chip exposes as public values the Merkle roots of the initial and final data memory states.

To expand on (3), we store the data memory state as a binary Merkle trie, where a path in the trie corresponds to the binary encoding of a $(\texttt{addr\_space}, \texttt{ptr})$ memory address. The consistency of memory accesses within state transitions is constrained by the memory bus, while the persistent boundary chip must add messages to the memory bus corresponding to the correct initial and final memory states. These messages must be consistent with the public Merkle roots, where consistency is constrained by Merkle proof verifications in circuit.

Our design includes the following novel optimization: We observe that the boundary chip only needs to add messages for addresses that were accessed in the segment, and all Merkle proofs for the initial (resp. final) memory state can be combined into a single multi-proof that

proves inclusion of multiple leaf nodes using shared witness data. When memory accesses can be grouped into large continuous ranges, which is often the case for programs using standard memory allocators, the multi-proof provides significant performance benefits.

5.3. **Native extension.** In order to support proof aggregation, we must construct circuits which constrain the correct verification of one or more ZK proofs generated by the ZK backend. To support flexibility in the proof aggregation logic and to support recursive proof aggregation (i.e., a circuit which verifies proofs of the same circuit), we take a zkVM approach to proof aggregation. More specifically, we create a custom *native* VM extension tailored for proof verification. Our zkVM design produces a zkVM circuit for the instruction set corresponding to this native extension, and we write the proof verification logic as a program in this instruction set. Proof aggregation then consists of executing a program that verifies a collection of ZK proofs and generating a single new proof of successful program execution for the zkVM circuit.

Our modular design allows the aforementioned instruction set and zkVM to be designed within the same OpenVM ISA and circuit architecture used to design other zkVMs with orthogonal use cases.

The native VM extension contains a minimal set of instructions designed to express proof verification logic:

- Memory load and store operations which operate directly on field elements
- Basic control flow (jump, branch)
- Instructions to store data from the host hint stream into guest memory
- Custom instructions specific to the proof verification logic

5.4. **Aggregation for proving unbounded programs.** We now describe how we aggregate proofs from all segments of an unbounded program execution into a single proof of the complete program execution. We organize proof aggregation into a tree (cf. Figure 1) where nodes consist of a zkVM and an associated program. Edges are drawn such that a node's program must verify the proofs of execution of the programs of its children.

We call the zkVM and associated instruction set for the unbounded program the "App" VM. The App VM has continuations enabled and uses persistent memory. The tree has the following other types of nodes:

Leaf Verifier: The zkVM is the minimal one supporting the native extension (we call this the "Agg" VM), and the program verifies a variable number of proofs of execution segments from the App VM. The Agg VM has continuations disabled and uses volatile memory.

When verifying multiple segments, the program must check that the boundary state is consistent. It does so by extracting the program ROM cached trace commitment from each proof and asserting they are equal. It checks that the program counter and memory Merkle root public values are consistent between consecutive segments.

The program re-exposes boundary state commitments as public values for subsequent aggregation purposes. In addition, recall that the App VM stores user public outputs within data memory. The leaf verifier program verifies a Merkle proof to extract the Merkle subroot of user public outputs from the final memory Merkle tree. It exposes this subroot as a public value.

Internal Verifier: The zkVM is the Agg VM. The program verifies a variable number of proofs of execution of independent programs, where each program can be either the leaf verifier program *or* the current program itself (a recursive verification). The program identifies the program via the program ROM commitment. It checks consistency of boundary states from

the App VM via the re-exposed public values in the leaf/internal verifier proof and again re-exposes the boundary state commitments for subsequent aggregation.

Proof aggregation is intended to continue until there is a single proof of the internal verifier program representing the complete program execution.

Root Verifier: The zkVM is the Agg VM configured with the appropriate number of public values to include all user public outputs (see below). The program verifies a single proof of correct execution of the internal verifier program. The internal verifier proof only has a single public value for the Merkle root of all user public outputs. The root verifier decommits the Merkle tree and exposes all leaf values of the Merkle tree as public values of the root verifier proof. We highlight that the user public outputs specified in the App VM program do not become true public values of a ZK proof until the root verifier proof is generated.

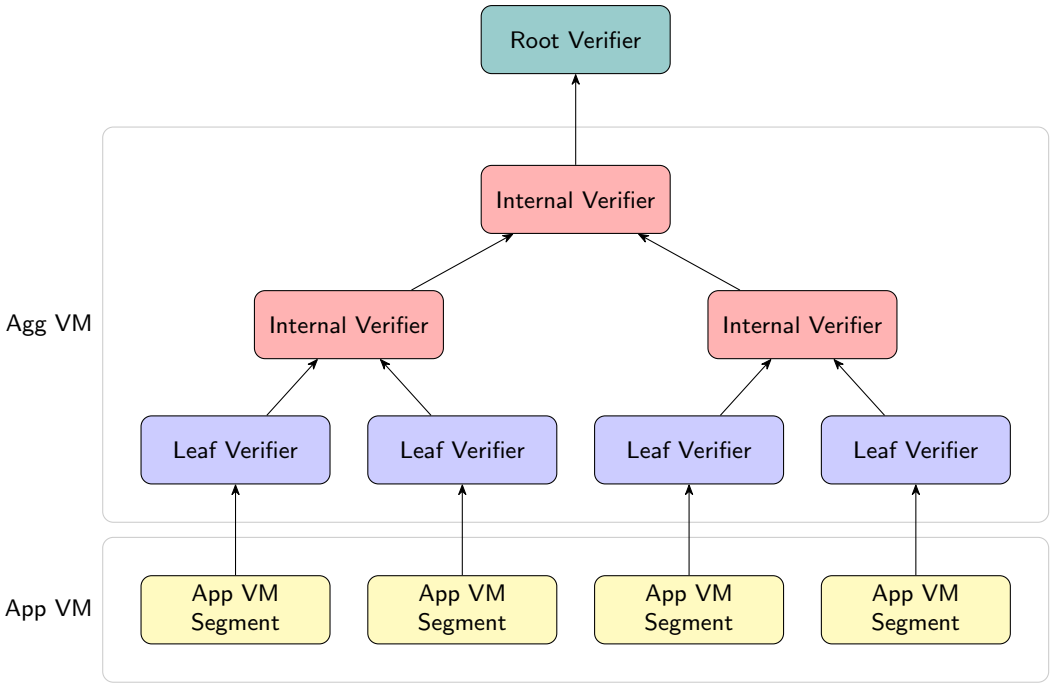

FIGURE 1. Proof aggregation tree for continuations. The figure shows 2-to-1 aggregation, but other configurations are also supported.

While the three types of nodes above all have zkVMs for the same instruction set, they are configured with different numbers of public values. Their proofs may also be configured to use different proof system parameters in the ZK backend.

5.5. **Onchain verification.** We have so far described how to generate a single proof of successful execution of an unbounded App VM program in the form of a root verifier proof. We describe how this proof can be verified on a blockchain. We focus for concreteness on blockchains that support the Ethereum Virtual Machine (EVM), though the approach is more broadly applicable.

The root verifier proof is typically generated by a ZK backend that produces proofs with size and verification cost that are still too large for data and compute-constrained environments like blockchains. As such, the root verifier proof must be further compressed into a proof with constant size and verification cost. This is done via *outer recursion*, where a proof in one proof system is verified in a circuit proven in a different proof system.

At the time of writing, we generate a *static verifier circuit* that constrains proof verification for a fixed zkVM circuit associated to the root verifier. The static verifier circuit is proven[2] using the Halo2 [EP25] proof system. The final SNARK proof proves the successful execution of the App VM program and exposes a hash commitment to the initial App program state – program ROM, starting pc, and initial memory state – as a public value. It additionally exposes as public values a commitment to the leaf program ROM and all user defined public outputs.

Finally, we generate an EVM smart contract that verifies proofs of the static verifier circuit. The smart contract only depends on the Agg VM circuit and the number of user public outputs. It does not depend on the App VM circuit or the App VM program.

6. RISC-V Support and Rust Toolchain

We enable developers to write guest programs targeting the OpenVM ISA in a familiar environment by leveraging the Rust programming language's use of LLVM as a compiler backend. We specifically use LLVM's support for 32-bit RISC-V as a compilation target, together with Rust macro support for injecting inline RISC-V assembly into the LLVM assembler.

6.1. Transpilation of RISC-V ELF to OpenVM executable.

We take advantage of the RISC-V design's inherent support for custom ISA extensions to make it easy for OpenVM VM extensions to integrate with the RISC-V ISA. We do this by introducing a transpiler framework that converts a RISC-V Executable and Linkable Format (ELF) file into an *OpenVM executable*, which comprises the following components:

- Program ROM
- Initial program counter
- Initial data memory

This executable specifies the initial guest VM state, as discussed in §3.2.

We define a *RISC-V machine code block* as a contiguous, 32-bit aligned bit sequence in RISC-V program memory, whose length is a multiple of 32 bits. Such a block may include instructions from standard and non-standard RISC-V ISA extensions or arbitrary bit sequences.

The transpiler can be instantiated with any supported set of VM extensions. Each VM extension that wishes to integrate with RISC-V defines a set of RISC-V machine code blocks alongside rules mapping each block to sequences of potentially multiple OpenVM instructions. Custom RISC-V machine code for VM extensions is classified into two distinct categories:

Intrinsic Instruction: A single custom instruction conforming to the RISC-V ISA specification.

Kernel Code: A 32-bit aligned arbitrary binary sequence whose length is a multiple of 32 bits. Kernel code need not comply with any RISC-V specification and is used primarily to embed foreign OpenVM assembly statically within ELF binaries, bypassing custom linking processes. Kernel code execution on standard RISC-V hardware requires specialized transpiler support.

[2]To further reduce the proof size, we verify the Halo2 proof inside another Halo2 circuit.

We envision these two categories of code to be used as follows: VM extensions designed for compatibility with the RISC-V ISA, and in particular its register and memory architecture, can define custom RISC-V instructions for a tighter and more direct integration. Extensions that require specialized use of the OpenVM ISA, often for performance reasons, which are not compatible with the RISC-V architecture, may generate kernel code through external means (e.g., via a separate compilation toolchain) and statically link the kernel code into the ELF binary. Our design is inspired by existing GPU toolchains, where GPU kernels are written and compiled for a distinct machine architecture and then linked into the host application binary.

6.2. **RV32IM extension.** We provide zkVM support for the RV32I base and RV32M multiplication instruction sets by defining a VM extension for RV32IM in OpenVM. This is a set of instructions in the OpenVM ISA designed to support transpilation from the RV32IM instruction set. The combination of the transpiler and the VM extension provides a zkVM that fully supports execution of RISC-V ELF binaries using the RV32IM instruction set. The transpiler and zkVM constrains the program execution to fully conform to the RISC-V ISA specification.

6.2.1. *System call support.* While the overall transpiler and ISA framework does support the RISC-V `ecall` instruction, we do not use `ecall` in any existing VM extensions. This is an intentional design choice: the `ecall` instruction allows the guest program to delegate execution logic to the host operating system, which is intended for environments where the ISA cannot fully specify or control the implementation. However in the zkVM setting where the instruction set and underlying circuit implementation can be co-designed, it is more modular and efficient to extend the instruction set and zkVM circuit directly.

As an example, guest program inputs and outputs are supported via another RV32Io VM extension, where movement of data from the host hint stream to guest memory and the writing of public outputs are provided by custom intrinsic instructions.

6.3. **Rust toolchain.** The Rust compiler supports 32-bit RISC-V as a target architecture. A Rust `no_std` program written without system calls will compile to an ELF binary that can be transpiled into an OpenVM executable. This executable is supported by any zkVM with the RV32IM extension.

Additional VM extensions that specify RISC-V transpilation rules can call their instructions from Rust by making use of the Rust inline assembly `asm!` macro. The macro allows the programmer to provide custom RISC-V assembly to the LLVM assembler to be injected into the ELF. Using this feature, VM extension developers can write *guest libraries* which provide Rust function bindings for their instructions.

6.4. **Rust `std` library support.** Rust programs written with the Rust standard library are supported with some limitations. The standard library provides functionalities which are traditionally implemented via system calls to the operating system. We externally link these functions to our implementations which use custom intrinsic functions when applicable.

References

[BBHR18a] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Fast Reed-Solomon interactive oracle proofs of proximity. In *Proceedings of the 45th International Colloquium on Automata, Languages, and Programming (ICALP)*, 2018.

[BBHR18b] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Paper 2018/046, 2018.

[BCI+20] Eli Ben-Sasson, Dan Carmon, Yuval Ishai, Swastik Kopparty, and Shubhangi Saraf. Proximity gaps for reed-solomon codes. Cryptology ePrint Archive, Paper 2020/654, 2020.

[BEG+94] Manuel Blum, W. Evans, Peter Gemmell, Sampath Kannan, and M. Naor. Checking the correctness of memories. *Algorithmica*, 12:225–244, 09 1994.

[BGKS19] Eli Ben-Sasson, Lior Goldberg, Swastik Kopparty, and Shubhangi Saraf. DEEP-FRI: Sampling outside the box improves soundness. Cryptology ePrint Archive, Paper 2019/336, 2019.

[Con23] ConsenSys. gnark. https://github.com/ConsenSys/gnark, 2023. Version 0.12.0.

[EP25] Electric Coin Co. and Privacy Scaling Explorations. halo2. https://crates.io/crates/halo2-axiom, 2025.

[GPR21] Lior Goldberg, Shahar Papini, and Michael Riabzev. Cairo – a turing-complete STARK-friendly CPU architecture. Cryptology ePrint Archive, Paper 2021/1063, 2021.

[Hab22a] Ulrich Haböck. Multivariate lookups based on logarithmic derivatives. Cryptology ePrint Archive, Paper 2022/1530, 2022.

[Hab22b] Ulrich Haböck. A summary on the FRI low degree test. Cryptology ePrint Archive, Paper 2022/1216, 2022.

[ide25] iden3. circom. https://github.com/iden3/circom, 2025.

[PH23] Shahar Papini and Ulrich Haböck. Improving logarithmic derivative lookups using GKR. Cryptology ePrint Archive, Paper 2023/1284, 2023.

[Pol25] Polygon Zero. Plonky3. https://github.com/Plonky3/Plonky3, 2025.

[RIS25] RISC Zero Team. RISC Zero. https://github.com/risc0/risc0, 2025.

[Sta21] StarkWare. ethSTARK documentation. Cryptology ePrint Archive, Paper 2021/582, 2021.

[Suc25] Succinct Labs. SP1. https://github.com/succinctlabs/sp1, 2025.

[Val24] Valida Team. Valida. https://github.com/valida-xyz/valida, 2024.