Cairo M Design Document

Clément Walter <clement@kakarot.org>

2025-09-26

Contents

1 Introduction

The motivation behind building a new *Zero-Knowledge Virtual Machine (zkVM)* comes from our experience of building a non-provable EVM client in Cairo Zero [1], the provable language (zkDSL) of Starkware, targeting the Cairo VM [2]. How can a program written in a zkDSL eventually not be provable? Not because of any logic issues, but because of scaling issues! The Cairo VM was just not designed to prove billions-long traces, nor to leverage parallel proving with recursion.

Facing this hard limitation made us re-evaluate its design. Actually, some decisions may be relevant when considering a given order of magnitude of execution length (around 10^5 steps at most), but become irrelevant when considering a much larger one (10^8 steps at least). Furthermore, though being supposed to be a general-purpose VM, it has been designed mainly with Starknet [3] in mind, i.e., with a focus on (small) transaction processing, rather than general-purpose computation.

The original Cairo architecture, as described in the seminal paper *Cairo – a Turing-complete STARK-friendly CPU architecture* [2] defines a general framework for building a ZK-friendly CPU, denominated as a "Cpu AIR — CAIR-o" and known as a zero-knowledge Virtual Machine (zkVM) nowadays. This framework is general both in terms of underlying proving scheme and base operating prime field. However, some design decisions (like the instruction encoding) require a base prime number larger than 2^{64}, while modern STARK provers favor smaller prime numbers like Babybear ($2^{31} - 2^{27} + 1$) or Mersenne31 ($2^{31} - 1$). Consequently, even the recent stwo-cairo prover [4] emulates the original prime number chosen 5 years ago: $2^{251} + 17 \cdot 2^{192} + 1$ [5]. This emulation makes the prover up to 28x less efficient as each native field element from the original Cairo VM is now up to 28 M31s [6], depending on the actual values used in the program and some optimizations.

Furthermore, the Cairo VM features a non-deterministic read-only memory model with relocation, which creates two severe limitations:

1. a program can only make a limited number of writes to memory, so the VM cannot run arbitrary long (meaningful) programs;
2. the final relocation step prevents from streaming the generated trace to start proving chunks in parallel while the program is still running (a technique called *continuation* [7]) as final memory addresses are only known after the program has exited.

Cairo M has been designed to overcome these limitations:

- Leverage small-field provers: STARK provers using small prime fields (e.g., M31 or Babybear);

- Continuation: Arbitrarily long program runs provable out-of-the-box
- Recursion: Direct proof verification within the prover framework;
- Low host memory usage: Efficient memory consumption on consumer devices.

The following sections of this document assume Mersenne31 (M31: $2^{31} - 1$) as the prime number. The design can be adapted for other primes with minor modifications.

This document does not describe the current state of the Cairo M codebase [8] nor the v0.1.0 release [9] but focuses on the decision framework and trade-offs considered when building a zkVM. The `v0` design is actually more a hybrid of the original Cairo VM and the proposed zkVM design from this document. It however tries to maintain a developer-friendly narrative and focus, rather than evolving into a general whitepaper about zkVMs. For this latter purpose, the OpenVM whitepaper [10] is a good reference as we eventually share the same views.

The design of a virtual machine mainly encompasses four decisions: the memory model, the number of registers, the opcodes and the addressing scheme. The remaining of this document addresses each of these questions in turn. An Appendix section provides a succinct background about some part of STARK provers (especially the Stwo framework [11] used in our implementation) as well as a detailed description of the implementation of a minimal zkVM.

2 Memory

In the context of a Virtual Machine, the memory is the main data structure that stores the program and the data. It is typically organized as a collection of linear arrays of addressable units (bytes, words, or field elements), where each location has a unique address. The VM's processor reads instructions from memory, loads/stores data values, and manages the execution state through memory operations.

Since memory segments are 1D-addressable arrays indexed by field elements, their maximum length is determined by the prover's prime field.

The following of this section mainly describes the implementation of one memory segment. There is no reason to stick to a single memory segment for the whole zkVM though (see also Section 4.4.1 and the OpenVM whitepaper [10]).

The remaining of this section is organized as follows: it first describes how to create a public commitment of a memory segment, then how to build actual memory words from base limbs made of field elements, and finally how to implement the read/write operations. The reader unfamiliar with lookup arguments may read first Section 6.1.

2.1 Commitment

Memory segments require efficient commitment for continuation, ensuring that the final memory state of stage `n` matches the initial state of stage `n + 1`.

Merkle trees provide a good memory commitment mechanism due to their:

- Challenge-independent commitment through initial and final root hashes;
- Natural sparse memory handling via partial tree pruning of unused intermediate nodes.

One shall pick a ZK-friendly hash function for these commitments. The current implementation uses Poseidon2, though alternative hash functions can be substituted as needed.

The natural memory address space spans $[0, P)$, but to avoid Merkle root padding requirements, we recommend to simply use the greatest power of 2 smaller than P, i.e., 2^{30}. Extension to P is possible but doesn't seem necessary, especially with continuation and for client-side proving scenarios where memory requirements remain modest compared to long-trace applications. Memory exceeding $2^{30} = 1{,}073{,}741{,}824$ is only required for extremely long runs, which would demand RAM capacity beyond typical consumer device specifications: client-side proving scenarios have maximum VM traces about 1M to 10M rows, ie 2^{20} to 2^{28} field elements.

Furthermore, the Merkle tree can optionally omit leaf hashing to minimize computational overhead. Although leaf hashing prevents sibling value disclosure in inclusion proofs, it provides no benefit for state root commitment. This approach requires all memory cells to contain values, resulting in an implicit zero-initialization of the entire memory segment. This default zero value has no practical impact.

The Merkle component is responsible for proving the 2^{30} leaves from a public (initial or final) root. It does this by iteratively consuming a node and emitting its two children leaves in the logup sum. The partial underlying Merkle tree is built during witness generation. The component only enforces via the lookup argument that the nodes and leaves actually derive from the root, using two relations: the `Merkle` relation that removes parent and emit children nodes, and the `Poseidon2` relation to prove the Poseidon2 hash computation of the children [12].

Eventually, the multiplicity at any given node can be set to 0 if the branch is actually not used, in which case the branch is pruned from the tree. If the memory is read-write and the multiplicity is guessed, then it needs to be constrained to be 0 or 1, otherwise several leaves can be emitted and the memory "forked". The whole local process can be summarized as follows:

$$
\begin{aligned}
&- \text{Merkle}(\texttt{index}, \texttt{depth}, \texttt{parent}, \texttt{root}) \\
&+ \{0,1\} \cdot \text{Merkle}(2 * \texttt{index}, \texttt{depth} + 1, \texttt{child\_left}, \texttt{root}) \\
&+ \{0,1\} \cdot \text{Merkle}(2 * \texttt{index} + 1, \texttt{depth} + 1, \texttt{child\_right}, \texttt{root}) \\
&+ \text{Poseidon2}(\texttt{child\_left}, \texttt{child\_right}) \\
&- \text{Poseidon2}(\texttt{parent})
\end{aligned}
$$

where `root` is either the initial or the final root hash, provided in the public data of the proof.

2.2 Range-checked memory segment

In the previous section, we described how to create a public commitment of a memory segment. This commitment lets derive a felt-based memory segment from a public root hash. In some contexts though (see Section 4.4.1), it may be desirable to have a range-checked memory segment instead.

A range-checked memory segment is a memory segment where each value is guaranteed to be within given boundaries, for example $[0, 2^8)$. If range-checking a given field element is always possible "at runtime" calling directly the `RangeCheck` component, keeping a dedicated range-checked segment is more efficient as only writes need to be range-checked.

The easiest and cheapest way to implement a range-checked memory segment is to initialize it full of zeros and commit the corresponding public root hash. Alternatively, one can add extra range-check lookup when initializing the memory segment from the `Merkle` relation.

2.3 Word size

If the Merkle root allows for committing to up to 2^{30} field elements, the VM doesn't need to use a single field element as the base word size. The `Memory` component is actually responsible for turning a list of M31 leaves into memory values. Hence, leaves can be grouped together as limbs of a single memory word.

For example, grouping 4 leaves together as a single memory word can be done as follows:

$$
\begin{aligned}
&- \text{Merkle}(a, 30, v_0, \texttt{root}) \\
&- \text{Merkle}(a + 1, 30, v_1, \texttt{root}) \\
&- \text{Merkle}(a + 2, 30, v_2, \texttt{root}) \\
&- \text{Merkle}(a + 3, 30, v_3, \texttt{root}) \\
&+ \text{Memory}(a, 0, v_0, v_1, v_2, v_3)
\end{aligned}
$$

where the 30 and 0 are hard-coded values, for `Merkle` as the tree height, and for `Memory` as the initial clock value.

In our first implementation, we actually used a fixed-size word built from 4 M31 elements to easily accommodate all field-element-based instructions in a single read. This effectively reduces however the memory size to $2^{30-2} = 2^{28} =$ 268,435,456.

However, there is no requirement to use such a fixed-size memory word. Given that memory consistency is enforced only with lookup arguments, each address can consume any number of field elements. The only requirement is that all the lookup terms, from the initially emitted ones derived from the Merkle commitment, to the last ones matching the final memory root, eventually sum up to zero.

Considering the raw memory as a 1D segment of 2^{30} field elements derived from the Merkle commitment, one can think of this as accessing slices at a given index, with the slice length depending on the index, instead of just a single value:

```
memory[address: address + len(address)]
```

instead of

```
memory[address]
```

Furthermore, since a logup term is computed as the weighted sum of the challenge coefficients with the provided values, trailing zeros have no impact on the result, i.e.:

$$\text{Memory}(a, \texttt{clock}, \texttt{value}, 0, \cdots, 0) = \text{Memory}(a, \texttt{clock}, \texttt{value})$$

In other words, a lookup argument with logup actually does not commit to a given number of values, which means that individual limbs cannot be associated directly with their own address at $a + i$. If this is needed, one can simply update the `Memory` relation with a `LEN` term so that address a is bound to use `LEN` values:

$$\text{Memory}(a, \texttt{LEN}, \texttt{clock}, \texttt{value}, \cdots)$$

Practically speaking, this "lazy" word size is especially useful when handling and casting multi-limbs types like `u32` (see also Section 4.4.1). Because any trailing zeros are ignored, the same address can be used in different opcodes with no extra cost.

For example, let us consider a range-checked memory with 16-bits limbs, and opcodes for the `u16`, `u32`, and `u64` types. If each type needs to have its own

component, say `U16StoreAdd`, `U32StoreAdd`, and `U64StoreAdd`, a value that was previously used and emitted as a `u16` with `+Memory(a, clock, value)` in `U16StoreAdd` can be read as is in a forthcoming `U32StoreAdd` with `-Memory(a, clock, value, 0)` since the two terms are actually equivalent.

2.4 Read/Write operations

To support arbitrarily long programs within a fixed-size memory segment, the design employs a read-write memory model for the RAM (Random Access Memory).

2.4.1 Lookup arguments for memory consistency

Read and write operations are implemented through lookup arguments (see also Section 6.1): each memory access is a lookup of a tuple $(\texttt{address}, \texttt{clock}, \texttt{value})$. The `clock` is a monotonic counter from 0, determined during witness generation and updated with the registers. It timestamps when `address` contained `value`.

To access a memory cell, one adds to the logup sum a term cancelling the previous access, and a new term for registering the new access. As there is no ordering in a global logup sum, the notion of "previous access" is enforced with a range-check argument on the clock difference: $\texttt{clock} - \texttt{prev\_clock} > 0$.

Altogether, using the notation defined in Section 6.1, a memory read or write operation is implemented as follows:

$$\begin{aligned} &- \text{Memory}(\texttt{address}, \texttt{prev\_clock}, \texttt{prev\_value}) \\ &+ \text{Memory}(\texttt{address}, \texttt{clock}, \texttt{value}) \\ &+ \text{RangeCheck20}(\texttt{clock} - \texttt{prev\_clock} - 1) \end{aligned}$$

with `address`, `prev_clock`, `prev_value`, `clock` and `value` being part of the main execution trace. Note that when the memory is only read, one has $\texttt{prev\_value} = \texttt{value}$ and this simplifies to:

$$\begin{aligned} &- \text{Memory}(\texttt{address}, \texttt{prev\_clock}, \texttt{value}) \\ &+ \text{Memory}(\texttt{address}, \texttt{clock}, \texttt{value}) \\ &+ \text{RangeCheck20}(\texttt{clock} - \texttt{prev\_clock} - 1) \end{aligned}$$

The key point of this design based on lookup arguments is that one needs to remove from the logup sum a term added at a point in time strictly before the current point in time, and that one adds terms with multiplicity 1 only. Adding terms with multiplicity greater than 1 would make it possible for the prover to "fork" the memory at some point, accessing a value already normally updated during the execution. The boundary conditions (initial and final memory) are handled by the memory commitment (see Section 2.1) and the public memory of the proof.

2.4.2 Clock Update Component

The clock update component is responsible for updating the clock value when the clock difference exceeds the capacity `RC_LIMIT` of the biggest available range-check component. It is not part of the VM specification but of the prover implementation. During witness generation, the prover checks which clock updates are required. If it encounters a clock difference exceeding its capacity, it performs a clock update, which essentially consists in mimicking a read operation:

$$
\begin{aligned}
&- \text{Memory}(\texttt{address}, \texttt{prev\_clock}, \texttt{prev\_value}) \\
&+ \text{Memory}(\texttt{address}, \texttt{prev\_clock} + \texttt{RC\_LIMIT}, \texttt{prev\_value})
\end{aligned}
$$

without the need to range-check the hard-coded clock update. It then adds as many clock updates as needed to cover the clock difference. For example, let us denote by δ the required clock difference. The number of clock updates required is $\lfloor \delta / \texttt{RC\_LIMIT} \rfloor$.

2.4.3 Memory Model Cost Analysis

Let us denote by T a regular trace column and by L a lookup operation.

Read-write memory (per access):

- Main trace: 4 to 5 columns (`address`, `prev_clock`, `clock`, `prev_value`, `value`)
- Lookup: 3
- Total: up to $5T + 3L$

Read-only memory (per access):

- Main trace: 2 columns (`address`, `value`)
- Lookup: 1 ($-$Memory(`address`, `value`))
- Total: $2T + 1L$

Since lookup columns are defined over the secure field, which is QM31 in stwo [13] (i.e., 4 M31s), each lookup column is actually 4 trace columns.

- Overhead per access: $(5T + 3L) - (2T + 1L) = 3T + 2L = 3 + 8 = 11$ base columns
- STORE operation example (dst = op0 + op1): up to 31 additional columns (2 reads and 1 write)

This overhead can be mitigated using opcodes that write in place (e.g., `x += y`). It can also be limited by grouping the logup columns by two, precomputing the logup sums in pairs when the maximum constraint degree remains low (see the pre-sum optimization in Section 6.1.2). If we consequently count only 2 columns per lookup, the memory access overhead becomes $3 + 2 * 2 = 7$. If we further consider an in-place operation, then it becomes $(5T + 3L) + (4T + 3L) = 9 + 12 = 21$

columns for the read-write memory and $3 \times (2T + 1L/2) = 6 + 8 = 14$ for the read-only memory.

Since the read-write memory model allows for much easier control flow, reduces the need to copy memory values with new frames, and is much easier to reason about when developing software, this overhead is deemed worthwhile.

On the other hand, not all parts of the memory need to be writable. In particular, the program with its embedded constant values can remain read-only.

The most direct way to make sure that an address space is read-only is to range-check (or lookup) the write address in all the `STORE` opcodes to prevent from writing to it. This would however not save any columns, just enforce that some regions are unchanged. To save on columns, one should instead add dedicated opcodes that would only "consume" the memory value with a constant clock set to 0. These values would be added from the commitment or the public memory with the required multiplicity.

However, range-checking the write address can become inefficient when the address range is large. In fact, the Cairo M design embeds a `RangeCheck20` (i.e., 20-bit range-check) as the largest single range-check component (i.e., with no limb splitting). This means that any value greater or equal than 2^{20} would require splitting (see also Section 2.4.2). As a consequence, this approach becomes inefficient when the address range is large.

Another solution is to use a dedicated read-only memory segment with its own commitment and lookup challenges. This effectively doubles the available address space, with half being read-only and the other half read-write. Furthermore, if the read-only memory is used only for the program, its commitment effectively becomes the program hash that can be used to identify the program in the proof, without requiring the read-write memory to be initialized with zeros.

3 Registers

The original Cairo VM uses 3 registers:

- `pc`: the program counter, is the address of the current instruction;
- `fp`: the frame pointer, is a pointer to the current frame initial address;
- `ap`: the application pointer, is the free-memory pointer.

In the context of a read-write memory, the free-memory pointer becomes unnecessary, and we can drop `ap`, leaving the VM initially with only two registers, similar to Valida [14], for example. In addition to these two registers, we add a third one, `clock`, which is a monotonic counter from 0, determined during witness generation and updated with the registers. Its purpose is to constrain the proper sequence of memory accesses since logup does not enforce any ordering.

Regular instruction set architectures also leverage registers to store temporary values used across multiple opcodes, acting as a fast buffer to avoid memory

accesses.

From the prover point of view:

- each register requires its own trace cell, i.e., it adds 1 column per component;
- a list of registers is like a memory slice (see Section 2.3) without address nor clock: each component updates the registers' state by removing the current state vector and adding the constrained update to the logup sum, much like a stack: you can access the last state and push a new one.

Ultimately, there is a trade-off between adding registers to the "main register stack" (i.e., together with `pc`, `fp` and `clock`) — which adds one column per component even if the value is not used, and is free otherwise — and adding registers to a "secondary register stack", or even a tertiary (etc.), in order to save on unused columns, but at the cost of performing 2 more lookups (4 columns) when they are needed. The limiting case is to simply use one relation per register, as if they were regular memory values with no address nor clock. Depending on the total number of components and how the compiler leverages registers, the optimal case may vary.

Adding registers to the main register stack may be temporarily better but is less maintainable, as the number of components may vary drastically over time and usage (see Section 4.3 and Section 4.4). The current design doesn't use any secondary register stack for the sake of simplicity. However, it would most likely be beneficial to leverage a secondary register stack with 2 values when several arithmetic operations are performed consecutively, or to easily factorize arithmetic operations over read-write and read-only memory.

4 Opcodes

The initial Cairo M design was heavily inspired by the Cairo VM. The currently implemented opcodes reflect this origin but would not be kept as-is if the project were started today.

4.1 AIR basics

An AIR (Algebraic Intermediate Representation) represents a computation as a collection of algebraic relationships that must be satisfied for the computation to be considered valid.

For simplicity, let us describe an AIR as a dataframe, with columns representing variables used in the defined constraint system (circuit) and rows representing circuit instantiations. All constraints are eventually described as a polynomial combination of the columns. For example, given a dataframe `df` with columns `a`, `b`, `c`, the computation $c = a + b$ is enforced with something like:

```
assert df[c] - df[a] - df[b] = 0
```

and applies to all rows of the dataframe.

During proof generation, each column is interpreted as values of a given polynomial over a base set $\{x^i\}_{i=0}^n$. This polynomial is interpolated and then evaluated over a larger domain. The prover commits to each column (each polynomial) and then generates Merkle inclusion proofs for some evaluations of these polynomials at random points. This means that the proof size and the verifier complexity are directly related to the number of columns in the AIR: the more columns, the more commitments in the proof and the greater the verifier complexity.

The Stwo framework allows for defining the whole AIR of the state transition of the virtual machine in several smaller such dataframes, called *components* [15] (other frameworks may call them *chips* [10]). Eventually, they are all concatenated by the column axis to form the whole AIR.

4.2 Design principles

Generally speaking, a reduced instruction set will generate more cycles, i.e., more rows, for a given operation than a complex instruction set (see also this RISC-V versus Cairo ISA comparison [16]). On the other hand, a complex instruction set will require more columns, i.e., more commitments, for a given operation than a reduced instruction set. In short, one can think of a reduced instruction set as a long and thin dataframe, as opposed to a complex instruction set as a short and wide one.

Notice, however, that given a component with shape (n, m) (n rows and m columns), one can always reshape it to $(n/k, m \cdot k)$, where k is an integer, duplicating the columns and their corresponding constraints. The other way around is not possible: one cannot keep only a partial circuit. In other words, a reduced instruction set trace can always be reshaped to "look like" a complex instruction set one, while the other way around is not possible. Hence, reduced instruction sets give more flexibility.

Furthermore, long traces can be proven in batch, and even in parallel when the program is still running (so-called *continuation* [7]), and aggregated later with recursion. This effectively boils down to splitting the dataframe into chunks with less rows. This reduces either the proving time or the memory usage of the host, which is directly proportional to the area (i.e., $width \times height$) of the AIR [17].

Consequently, when designing an AIR, one tries to limit the number of columns as much as possible. This can be done by both limiting the number of opcodes in the instruction set, and by factorizing as much as possible several opcodes into the same component.

The required degree of the constraints can also influence the number of columns.

Actually, the maximum degree of the constraints influences the size of the evaluation domain, and adding constraints with a higher degree will double its size. Hence it is always better to just add intermediate columns to reduce the degree of the constraints.

Overall, the goal is to use as few columns as possible, and to keep the degree of the constraints as low as possible, which can in turn require more columns.

4.3 Minimal instruction set

We now present in this section a minimal instruction set.

Control Flow:

- `CallRel`, `Ret`: Function call/return management

Branching:

- `JmpRel`, `JnzRel`: Intra-frame jumps

Arithmetic Operations:

- `StoreAdd`, `StoreSub`, `StoreMul`, `StoreDiv`: Field arithmetic with memory storage

Memory Operations:

- `MoveString`: Copy a string from one address to another
- `MoveStringIndirect`: Copy a string from one dereferenced address to another
- `MoveStringIndirectTo`: Copy a string from one address to a dereferenced address

This proposed instruction set fits in a total of $52T + 39L$ columns. See the Section 6.2 section for more details.

4.4 Extensions

If the proposed instruction set is enough to perform any kind of computation, one may want to extend it with more opcodes. The purpose of extensions is to make some complex operations native to the prover, i.e. to give them directly a circuit representation. Whether extensions actually make the whole proving steps faster depends on the context and the actual optimization they allow.

Among the most common extensions, we describe below the case of adding different "native" types to the instruction set and built-in hash functions.

4.4.1 Uint Types

At the prover level, the only native type is the field element. However, at the software level, the most common native types are `u32` or `u64`. While it is possible to emulate for example a `u256` at the software level [18], it may be more efficient

to instead manage it at the AIR level. For example, creating a `u32` with a struct holding two field elements would require two memory accesses per variable use instead of one (see also Section 2.4).

At the software level, the main difference between a uint of a given size and a felt lies mainly in the division operation. In fact, unchecked arithmetic is generally preferred in production code for performance reasons (see for example rust release mode [19]) and uints silently overflow and wrap around, behaving like a field element over 2^n. On the other hand, the division for field elements is always exact (every field element has an inverse), while the division for uints is the Euclidean division. At the AIR level, emulating a uint mainly requires emulating operations over the uint size, i.e., properly handling the carry, borrow, and range-checking the values used.

Given that the current prime is $2^{31} - 1$, any uint using fewer than 31 bits can easily be represented as a single field element. However, as mentioned previously, every single value needs to be range-checked to ensure that it stays within the correct boundaries. Consequently, the largest simple native uint type that can be represented without any limb decomposition depends on the size of the largest `RangeCheck` component added to the prover. Since a `RangeCheck` component is just a plain enumeration of all the allowed numbers (e.g., $[0, 2^{20})$ for a `RangeCheck20` component), this is directly related to the size of the trace itself and so to the host memory usage and overall performance of the prover. As a matter of fact, given some initial benchmarks with Stwo [20], we decided to keep `RangeCheck20` as the largest single `RangeCheck` component, consequently making `u20` the largest simple native uint type that could be represented without any limb decomposition.

In any case, keeping the same memory segment for both felt and uint creates a significant range-check overhead, as every read needs to be range-checked, not just writes. For this reason, it is better to use a dedicated memory segment for every such simple uint type, where only the write operation needs to be range-checked. If one is mainly interested in supporting uints types, the whole felt based opcodes and memory could also be completely dropped.

On the other hand, given this maximum limb size, it is straightforward to derive any uint type with limb decomposition over this base limb size with no significant extra cost. Remember from the Section 2.3 section that a memory read is actually a memory slice read; one can read several limbs at once.

Eventually, since `u20` is not a regular base type in any software and this "20" is strongly dependent on some internal prover configuration (the largest available range-check component), it makes more sense to use `u16` or `u8` instead. The question of the most optimal base between the two depends on the context. Using `u8` would create more trace cells for `ADD` and `SUB` operations where 16-bit limbs are fine, but would save on `MUL` and `DIV` operations where numbers actually need to be written with 8-bit limbs since $\text{u16} \times \text{u16} \to \text{u32} > 2^{31} - 1$. In addition to these opcode considerations, it is worth noting that popular

instruction sets amongst the ZK community like WASM [21] or RISC-V [22] do use a byte-addressable memory; in this context, `u8` is a natural choice.

4.4.2 Built-in functions

Built-in functions, or precompiles, are functions with a dedicated circuit representation. Instead of compiling to the base instruction set, they directly map their inputs and outputs to a prover AIR.

They are a common way to optimize the prover's performance and are basically a flattened version of a given function over the native field arithmetic. The optimization comes from the fact that:

1. they reduce the number of required lookup arguments for a given operation;
2. they allow for the use of non-deterministic computation [23] by guessing directly during witness generation appropriate values.

The purpose of this design document is not to be a complete guide on how to define appropriate and efficient built-in functions. Building precompiles has historically been a complex handcrafted process. Recent results suggest though that they could be automatically generated [24]. Since AIR are just circuits, it is actually no surprise that they can just be derived from a compiler [25], associating each frame to a given column set. Leveraging non-determinism requires however a dedicated hint mechanism. Eventually, the coupling between witness generation and AIR generation is tight.

In any case, there are two ways to actually add precompiles: either *as opcodes* or as *preprocessed columns* with recursion. In the first case, they just get their own opcode id and are like any other opcode. This is the most straightforward approach, but requires adding more columns to the main AIR. On the other hand, when they are added as *preprocessed columns* with recursion, a single independent proof is first generated (in parallel), gathering in the public data the inputs-outputs final relationship. The main execution trace just consumes them in one lookup. The recursion step then ensures that they all work together.

5 Conclusion

The initial goal of Cairo M was to get the simplest, lightest and fastest possible general-purpose zkVM. It is, an updated version of the initial Cairo VM. Using Stwo's component system, it progressively became clear however that much more flexibility was currently possible, and we started to explore native types support, built-in functions, and more.

Eventually, the relevance of a general-purpose felt-based minimal zkVM is becoming questionable. If it may bring some improvements for specific optimized programs or contexts, most of the software we use is actually byte based. Consequently, building an efficient general-purpose proving system for regular software is more likely to be done with a byte-based (i.e. range-checked) memory segments

and a byte-based instruction set. In this context, reusing an existing reduced instruction set instead of building a new one looks like a very reasonable idea to be compatible with the existing software ecosystem and to avoid compiler complexity. RISC Zero pioneered this approach with RISC-V [26] and it's no surprise that it is currently the most successful one for general-purpose proving and especially proving the Ethereum State Transition function [27].

On the other hand, it also became clear that low level circuits (i.e. components' AIRs) would be best generated by compilers and as such, may benefit from a simple felt-based zkDSL. This minimal zkDSL could be used to either run in a VM or directly compile to a dedicated AIR for each program. The benefit of keeping a VM here is mainly for witness generation. Regardless the instruction set, the memory design with variable word-size and the registers management with secondary stacks design can be implemented. Future zkVM design will most likely focus less on the instruction set and more on the communication between different, almost independent, components while still producing, after recursion, a single unique proof.

6 Appendix

6.1 Lookups

A lookup argument in zero-knowledge proofs is a cryptographic primitive that allows a prover to demonstrate that certain values in a computation trace exist in another table, without revealing the specific values or their positions. The prover commits to a "claimed sum" of lookup terms, and the verifier checks that all these sums equal to zero, ensuring all looked-up values are valid according to the specified relation constraints.

This entire document is drafted with Stwo's constraint framework in mind, which uses LogUp lookup arguments [28]; see the logup.rs [29] module for more details. Lookup and logup terms are used interchangeably in this document to denote a relation between two components.

6.1.1 Core Concept

LogUp lookup arguments form a global sum of fractions that must equal zero. Each component contributes fraction terms based on three elements:

1. **Relation**: Defines $(\alpha^i)_{i=0}^n$ coefficients and z value for tuple aggregation in the secure field;
2. **Denominator**: Aggregates the tuple values $(v_0, v_1, \dots, v_n)$ in the secure field;

$$\sum_{i=0}^{n} \alpha^i \cdot v_i - z$$

3. **Numerator (multiplicity)**: Usage count of the tuple: m. This can be a hard-coded value or a variable from the main trace.

Each logup term is then:

$$\frac{m}{\sum_{i=0}^{n} \alpha^i \cdot v_i - z}$$

Because each term is in the secure field, which is QM31, each lookup column requires 4 trace columns. The global logup commitment is then the sum of each such terms across all instances of all the relations used.

Throughout this document, we simply write:

$$\pm m \cdot \text{Relation}(v_0, ..., v_n)$$

to refer to the lookup of the tuple $(v_0, ..., v_n)$ for the relation `Relation` with multiplicity m. We refer to "emitted", "yielded", or "added" values when the multiplicity is positive, and "consumed" or "subtracted" values when the multiplicity is negative.

6.1.2 Constraint Formula

Each term is committed to and constraints are actually defined as:

$$\text{committed\_value} \cdot \left(\sum_{i=0}^{n} \alpha^i \cdot v_i - z \right) - m = 0$$

This means that the degree of such constraints depends on the degree of both the denominator and the numerator. Given the fact that the committed value is degree 1, the degree of such constraints is:

$$\max(\text{degree}(\text{denominator}) + 1, \text{degree}(\text{numerator}))$$

which needs to be less than the maximum constraint degree bound declared by the component. This also leaves a room for optimization by pre-summing terms: given the fact that the sum of two fractions writes as:

$$\frac{a}{b} + \frac{c}{d} = \frac{a \cdot d + c \cdot b}{b \cdot d}$$

degrees of numerators and denominators are summed up. If this sum remains lower than the maximum constraint degree bound, one can pre-sum the terms and actually save on committed values, i.e., on columns. Practically speaking, the max degree bound used in Cairo M is 3, which means that one can pre-sum the terms by up to two if all of them use variable of degree 1.

6.2 Opcodes components

This section describes in detail the minimal instruction set proposed in Section 4.3. It provides the detailed list of columns for each component. Not mentioned is the possible need for an enabler column, which distinguishes between the actual trace row and the padding required for the trace length to be a power of 2. Finally, for the sake of brevity, we usually don't add columns just for reducing the degree of the constraints in the lookups. This is left to the reader to optimize, remembering that saving one lookup actually removes 4 columns.

Each instruction is a variable-sized list of field elements, with the first one always being the opcode ID. The rest is context-dependent and denoted as off_i. The name op_i is used to refer to the i-th operand, which is a memory access at address $\text{fp} + \text{off}_i$, i.e., $\text{memory}[\text{fp} + \text{off}_i]$.

When several opcodes are proven with the same component, the opcode IDs are used to select the appropriate constraints. For simplicity, it is assumed that the opcode IDs are consecutive, so that the difference between opcode IDs directly yields a Boolean flag. This is not required, and one could alternatively use the constant `1 / (id_1 - id_0)` between the opcode IDs to compute the Boolean flag.

We use a python-like syntax to describe the opcodes and the components in pseudo-code. These snippet focus on the actual logic and are not complete: they don't make any explicit references to lookups, which are actually implicit operation of the target logic. Especially, it's worth remembering that every memory access requires a previous clock column to cancel the previous lookup term.

6.2.1 Subroutine control opcodes

`CallRel`

```
# Read instruction from memory
(CALL_REL_ID, off0, off1) = memory[pc]

# Write return fp and pc
memory[fp + off0] = fp
memory[fp + off0 + 1] = pc

# Update fp and pc
fp = fp + off0 + 2
pc = pc + off1
```

`Ret`

```
# Read instruction from memory
(RET_ID, ) = memory[pc]

# Update fp and pc
fp = memory[fp - 2]
pc = memory[fp - 1]
```

One can actually factorize both instructions into the same component by encoding the `Ret` instruction as `(RET_ID, -2, 0)`, i.e., setting `off0 = -2` and padding with 0 (which has no effect, see Section 2.3). The component's logic becomes:

```
# Read instruction from memory
(opcode_id, off0, off1) = memory[pc]

# Read operands from memory
op0_prev_val = memory[fp + off0]
op0_plus_one_prev_val = memory[fp + off0 + 1]

# Compute flag
is_ret = opcode_id - CALL_REL_ID
assert is_ret * (1 - is_ret) = 0

# Update operands
op0_val = op0_prev_val * is_ret + fp * (1 - is_ret)
op0_plus_one_val = (
  op0_plus_one_prev_val * is_ret +
  pc * (1 - is_ret)
)
memory[fp + off0] = op0_val
memory[fp + off0 + 1] = op0_plus_one_val

# Update registers
fp = op0_prev_val * is_ret + (fp + off0 + 2) * (1 - is_ret)
pc = op0_plus_one_prev_val * is_ret + (pc + off1) * (1 - is_ret)
```

Eventually, the final columns list for the main trace is:

- registers: 3
  - `pc`
  - `fp`
  - `clock`
- memory prev clocks: 2
  - `op0_prev_clock`

    - op0_plus_one_prev_clock
- instruction: 3
    - opcode_id
    - off0
    - off1
- operands prev values: 2
    - op0_prev_val
    - op0_plus_one_prev_val

The component also does the following lookups:

- update registers

```
-Registers(pc, fp, clock)
+Registers(pc_next, fp_next, clock + 1)
```

- read instruction from read-only memory

```
-ROM(pc, opcode_id, off0, off1)
```

- read/write operands from memory

```
-RAM(fp + off0, prev_clock, op0_prev_val)
+RAM(fp + off0, clock, op0_val)
-RAM(fp + off0 + 1, prev_clock, op0_plus_one_prev_val)
+RAM(fp + off0 + 1, clock, op0_plus_one_val)
```

- range-check clock difference

```
+RangeCheck20(clock - op0_prev_clock - 1)
+RangeCheck20(clock - op0_plus_one_prev_clock - 1)
```

This is a total of $10T + 9L$.

6.2.2 Branching opcodes

JmpRel

```
# Read instruction from memory
(JMP_REL_ID, off0) = memory[pc]
```

```
# Update pc only
pc = pc + off0
```

JnzRel

```
# Read instruction from memory
(JNZ_REL_ID, off0, off1) = memory[pc]

# Read operand from memory
op1_val = memory[fp + off1]

# Hint op1_val_inv as 1 / op1_val or 0
# Hint not_zero as op1_val != 0
assert (not_zero - 1) * op1_val = 0
assert not_zero * (op1_val * op1_val_inv - 1) = 0

# Update pc
pc = pc + 1 + (off0 - 1) * not_zero
```

Noticing that `JmpRel` is just a special case of `JnzRel` where the jump is always taken, one can factorize both instructions into the same component.

```
# Read instruction from memory
(opcode_id, off0, off1) = memory[pc]

# Compute flag
taken = opcode_id - JNZ_REL_ID
assert taken * (1 - taken) = 0

# Hint op1_val_inv as 1 / op1_val or 0
# Hint not_zero as op1_val != 0
op1_val = memory[fp + off1]
assert (not_zero - 1) * op1_val = 0
assert not_zero * (op1_val * op1_val_inv - 1) = 0

# Update pc
# Hint pc_next to reduce lookup degree
assert (
  pc + 1 +
  (off0 - 1) * not_zero * (1 - taken) +
  (off0 - 1) * taken
) - pc_next = 0
```

```
pc = pc_next
```

The final columns list for the main trace is:

- registers: 3
  - `pc`
  - `fp`
  - `clock`
- memory prev clocks: 1
  - `op1_prev_clock`
- instruction: 3
  - `opcode_id`
  - `off0`
  - `off1`
- operands values: 1
  - `op1_val`
- hinted values: 3
  - `op1_val_inv`
  - `not_zero`
  - `pc_next`

The component also does the following lookups:

- update registers

```
-Registers(pc, fp, clock)
+Registers(pc_next, fp, clock + 1)
```

- read instruction from read-only memory

```
-ROM(pc, opcode_id, off0, off1)
```

- read operand from memory

```
-RAM(fp + off1, op1_prev_clock, op1_val)
+RAM(fp + off1, clock, op1_val)
```

- range check clock difference

```
+RangeCheck20(clock - op1_prev_clock - 1)
```

This is a total of $11T + 6L$.

6.2.3 Arithmetic opcodes

We define the four arithmetic opcodes over the base field.

StoreAdd

```
# Read instruction from memory
(STORE_ADD_ID, off0, off1, off2) = memory[pc]

# Read operands from memory
op0_val = memory[fp + off0]
op1_val = memory[fp + off1]

# Update memory
memory[fp + off2] = op0_val + op1_val

# Update pc
pc = pc + 1
```

StoreSub

```
# Read instruction from memory
(STORE_SUB_ID, off0, off1, off2) = memory[pc]

# Read operands from memory
op0_val = memory[fp + off0]
op1_val = memory[fp + off1]

# Update memory
memory[fp + off2] = op0_val - op1_val

# Update pc
pc = pc + 1
```

StoreMul

```
# Read instruction from memory
(STORE_MUL_ID, off0, off1, off2) = memory[pc]

# Read operands from memory
op0_val = memory[fp + off0]
op1_val = memory[fp + off1]
```

```
# Update memory
memory[fp + off2] = op0_val * op1_val

# Update pc
pc = pc + 1
```

StoreDiv

```
# Read instruction from memory
(STORE_DIV_ID, off0, off1, off2) = memory[pc]

# Read operands from memory
op0_val = memory[fp + off0]
op1_val = memory[fp + off1]

# Hint op1_val_inv as 1 / op1_val
assert op1_val * op1_val_inv - 1 = 0

# Update memory
memory[fp + off2] = op0_val * op1_val_inv

# Update pc
pc = pc + 1
```

These four opcodes are actually factorized into the same component.

```
# Hind two opcode flags as 0 or 1
assert opcode_flag_0 * (1 - opcode_flag_0) = 0
assert opcode_flag_1 * (1 - opcode_flag_1) = 0
opcode_id = STORE_ADD_ID + opcode_flag_0 * 2 + opcode_flag_1

# Read instruction from memory
(_, off0, off1, off2) = memory[pc]

# Read operands from memory
op0_val = memory[fp + off0]
op1_val = memory[fp + off1]

# Hint op1_val_inv as 1 / op1_val or 0
assert op1_val_inv * (op1_val_inv * op1_val - 1) = 0
assert op1_val * (op1_val_inv * op1_val - 1) = 0
```

```
# Hint op2_val to reduce degree in lookups
is_add = (1 - opcode_flag_0) * (1 - opcode_flag_1)
is_sub = (1 - opcode_flag_0) * opcode_flag_1
is_mul = opcode_flag_0 * (1 - opcode_flag_1)
is_div = opcode_flag_0 * opcode_flag_1
assert (
  is_add * (op0_val + op1_val) +
  is_sub * (op0_val - op1_val) +
  is_mul * (op0_val * op1_val) +
  is_div * (op0_val * op1_val_inv)
) - op2_val = 0

# Update memory
memory[fp + off2] = op2_val

# Update pc
pc = pc + 1
```

The final columns list for the main trace is:

- registers: 3
  - `pc`
  - `fp`
  - `clock`
- memory prev clocks: 3
  - `op0_prev_clock`
  - `op1_prev_clock`
  - `op2_prev_clock`
- instruction: 3
  - `off0`
  - `off1`
  - `off2`
- operands values: 3
  - `op0_val`
  - `op1_val`
  - `op2_prev_val`
- hinted values: 4
  - `opcode_flag_0`
  - `opcode_flag_1`
  - `op1_val_inv`
  - `op2_val`

The component also does the following lookups:

- update registers

```
-Registers(pc, fp, clock)
+Registers(pc + 1, fp, clock + 1)
```

- read instruction from read-only memory

```
-ROM(pc, opcode_id, off0, off1, off2)
```

- read/write operands from memory

```
-RAM(fp + off0, op0_prev_clock, op0_val)
+RAM(fp + off0, clock, op0_val)
-RAM(fp + off1, op1_prev_clock, op1_val)
+RAM(fp + off1, clock, op1_val)
-RAM(fp + off2, op2_prev_clock, op2_prev_val)
+RAM(fp + off2, clock, op2_val)
```

- range check clock difference

```
+RangeCheck20(clock - op0_prev_clock - 1)
+RangeCheck20(clock - op1_prev_clock - 1)
+RangeCheck20(clock - op2_prev_clock - 1)
```

This is a total of $16T + 12L$.

6.2.4 Memory opcodes

Remember from Section 2.3 that a memory access is actually a memory slice access. We use here the `*_` syntax to stress the fact that these memory read and write can actually be slices and not just single elements.

`MoveString`

```
# Read instruction from memory
(MOVE_STRING_ID, off0, off1) = memory[pc]

# Read operand from memory
(op0_val, *_) = memory[fp + off0]

# Write operand to memory
memory[fp + off1] = (op0_val, *_)
```

```
# Update pc
pc = pc + 1
```

The actual component has a maximum slice "capacity", i.e. a total number of columns for materializing the input slice and the output slice. Let us define this slice maximum size as `n`. The final columns list for the main trace is:

- registers: 3
  - `pc`
  - `fp`
  - `clock`
- memory prev clocks: 2
  - `op0_prev_clock`
  - `op1_prev_clock`
- instruction: 2
  - `off0`
  - `off1`
- limbs values: `2n`

The component also does the following lookups:

- update registers

```
-Registers(pc, fp, clock)
+Registers(pc + 1, fp, clock + 1)
```

- read instruction from read-only memory

```
-ROM(pc, MOVE_STRING_ID, off0, off1)
```

- read/write operands from memory

```
-RAM(fp + off0, op0_prev_clock, op0_val)
+RAM(fp + off0, clock, op0_val)
-RAM(fp + off1, op1_prev_clock, op1_prev_val)
+RAM(fp + off1, clock, op0_val)
```

- range check clock difference

```
+RangeCheck20(clock - op0_prev_clock - 1)
+RangeCheck20(clock - op1_prev_clock - 1)
```

This is a total of $(7 + 2n)T + 9L$.

`MoveStringIndirect`

This is similar to the `MoveString` opcode, but with an additional memory read to compute the source address.

```
# Read instruction from memory
(MOVE_STRING_INDIRECT_ID, off0, off1, off2) = memory[pc]

# Read base address from memory
op0_val = memory[fp + off0]

# Write operand to memory
memory[fp + off2] = memory[op0_val + off1]

# Update pc
pc = pc + 1
```

`MoveStringIndirectTo`

```
# Read instruction from memory
(MOVE_STRING_INDIRECT_TO_ID, off0, off1, off2) = memory[pc]

# Read base address from memory
op0_val = memory[fp + off0]

# Write operand to memory
memory[op0_val + off1] = memory[fp + off2]

# Update pc
pc = pc + 1
```

Given the fact that each read or write is actually a read of the previous value and a write of the new value, possibly equal, the two snippets above are easily merged into the same component.

```
# Read instruction from memory
(opcode_id, off0, off1, off2) = memory[pc]

# Read base address from memory
op0_val = memory[fp + off0]
```

```
# Read operands to memory
op1_val = memory[op0_val + off1]
op2_val = memory[fp + off2]

# Write result to memory
indirect_to = opcode_id - MOVE_STRING_INDIRECT_ID
assert indirect_to * (1 - indirect_to) = 0
dst_val = op2_val * indirect_to + op1_val * (1 - indirect_to)
memory[op0_val + off1] = dst_val
memory[fp + off2] = dst_val

# Update pc
pc = pc + 1
```

The final columns list for the main trace is:

- registers: 3
  - pc
  - fp
  - clock
- memory prev clocks: 3
  - op0_prev_clock
  - op1_prev_clock
  - op2_prev_clock
- instruction: 4
  - opcode_id
  - off0
  - off1
  - off2
- operands values: 3
  - op0_val
  - op1_val
  - op2_val
- limbs values: 2n

The component also does the following lookups:

- update registers

```
-Registers(pc, fp, clock)
+Registers(pc + 1, fp, clock + 1)
```

- read instruction from read-only memory

```
-ROM(pc, MOVE_STRING_INDIRECT_ID, off0, off1, off2)
```

- read/write operands from memory

```
-RAM(fp + off0, op0_prev_clock, op0_val)
+RAM(fp + off0, clock, op0_val)
-RAM(op0_val + off1, op1_prev_clock, op1_val)
+RAM(op0_val + off1, clock, op1_val)
-RAM(fp + off2, op2_prev_clock, op2_prev_val)
+RAM(fp + off2, clock, op1_val)
```

- range check clock difference

```
+RangeCheck20(clock - op0_prev_clock - 1)
+RangeCheck20(clock - op1_prev_clock - 1)
+RangeCheck20(clock - op2_prev_clock - 1)
```

This is a total of $(13 + 2n)T + 12L$.

7 References

1. https://github.com/kkrt-labs/keth
2. https://eprint.iacr.org/2021/1063.pdf
3. https://starknet.io
4. https://github.com/starkware-libs/stwo-cairo
5. https://docs.starknet.io/learn/protocol/cryptography
6. https://github.com/starkware-libs/stwo-cairo/blob/main/stwo_cairo_prover/crates/common/src/memory.rs#L1
7. https://risczero.com/blog/continuations
8. https://github.com/kkrt-labs/cairo-m
9. https://github.com/kkrt-labs/cairo-m/releases/tag/v0.1.0
10. https://openvm.dev/whitepaper.pdf
11. https://github.com/starkware-libs/stwo
12. https://eprint.iacr.org/2023/323
13. https://github.com/starkware-libs/stwo/blob/dev/crates/stwo/src/core/fields/qm31.rs#L21

14. https://lita.gitbook.io/lita-documentation/architecture/valida-zk-vm/technical-design-vm
15. https://docs.starknet.io/learn/s-two/air-development/components
16. https://x.com/ClementWalter/status/1896131941109506309
17. https://x.com/ClementWalter/status/1964997331612488085
18. https://github.com/starkware-libs/cairo-lang/blob/v0.14.0.1/src/starkware/cairo/common/uint256.cairo
19. https://doc.rust-lang.org/reference/expressions/operator-expr.html#overflow
20. https://x.com/ClementWalter/status/1927617083967234483
21. https://webassembly.github.io/spec/core/exec/runtime.html#memory-instances
22. https://en.wikipedia.org/wiki/RISC-V#Memory_access
23. https://eprint.iacr.org/2021/1063.pdf#page=55.16
24. https://www.powdr.org/blog/accelerating-ethereum-with-autoprecompiles
25. https://github.com/kkrt-labs/keth/blob/main/python/cairo-ec/src/cairo_ec/compiler.py
26. https://risczero.com/blog/announce
27. https://ethproofs.org/zkvms
28. https://eprint.iacr.org/2022/1530.pdf
29. https://github.com/starkware-libs/stwo/blob/dev/crates/constraint-framework/src/prover/logup.rs