The Polygon Zero zkEVM

DRAFT
November 14, 2023

Abstract

We describe the design of Polygon Zero's zkEVM, ...

Contents

1 Introduction

TODO

2 STARK framework

2.1 Cost model

Our zkEVM is designed for efficient verification by STARKs [1], particularly by an AIR with degree 3 constraints. In this model, the prover bottleneck is typically constructing Merkle trees, particularly constructing the tree containing low-degree extensions of witness polynomials.

2.2 Field selection

Our zkEVM is designed to have its execution traces encoded in a particular prime field $\mathbb{F}_p$, with $p = 2^{64} - 2^{32} + 1$. A nice property of this field is that it can represent the results of many common `u32` operations. For example, (widening) `u32` multiplication has a maximum value of $(2^{32} - 1)^2$, which is less than p. In fact a `u32` multiply-add has a maximum value of $p - 1$, so the result can be represented with a single field element, although if we were to add a carry in bit, this no longer holds.

This field also enables a very efficient reduction method. Observe that

$$2^{64} \equiv 2^{32} - 1 \pmod{p}$$

and consequently

$$\begin{aligned} 2^{96} &\equiv 2^{32}(2^{32} - 1) \pmod{p} \\ &\equiv 2^{64} - 2^{32} \pmod{p} \\ &\equiv -1 \pmod{p}. \end{aligned}$$

To reduce a 128-bit number n, we first rewrite n as $n_0 + 2^{64}n_1 + 2^{96}n_2$, where n_0 is 64 bits and n_1, n_2 are 32 bits each. Then

$$\begin{aligned} n &\equiv n_0 + 2^{64}n_1 + 2^{96}n_2 \pmod{p} \\ &\equiv n_0 + (2^{32} - 1)n_1 - n_2 \pmod{p} \end{aligned}$$

After computing $(2^{32} - 1)n_1$, which can be done with a shift and subtraction, we add the first two terms, subtracting p if overflow occurs. We then subtract n_2, adding p if underflow occurs.

At this point we have reduced n to a `u64`. This partial reduction is adequate for most purposes, but if we needed the result in canonical form, we would perform a final conditional subtraction.

2.3 Cross-table lookups

TODO

3 Tables

3.1 CPU

TODO

3.2 Arithmetic

TODO

3.3 Byte Packing

TODO

3.4 Logic

TODO

3.5 Memory

For simplicity, let's treat addresses and values as individual field elements. The generalization to multi-element addresses and values is straightforward.

Each row of the memory table corresponds to a single memory operation (a read or a write), and contains the following columns:

1. a, the target address
2. r, an "is read" flag, which should be 1 for a read or 0 for a write
3. v, the value being read or written
4. τ, the timestamp of the operation

The memory table should be ordered by (a, τ). Note that the correctness memory could be checked as follows:

1. Verify the ordering by checking that $(a_i, \tau_i) < (a_{i+1}, \tau_{i+1})$ for each consecutive pair.
2. Enumerate the purportedly-ordered log while tracking a "current" value c, which is initially zero.[1]
   (a) Upon observing an address which doesn't match that of the previous row, set $c \leftarrow 0$.
   (b) Upon observing a write, set $c \leftarrow v$.
   (c) Upon observing a read, check that $v = c$.

The ordering check is slightly involved since we are comparing multiple columns. To facilitate this, we add an additional column e, where the prover can indicate whether two consecutive addresses are equal. An honest prover will set

$$e_i \leftarrow \begin{cases} 1 & \text{if } a_i = a_{i+1}, \\ 0 & \text{otherwise.} \end{cases}$$

We then impose the following transition constraints:

1. $e_i(e_i - 1) = 0$,
2. $e_i(a_i - a_{i+1}) = 0$,
3. $e_i(\tau_{i+1} - \tau_i) + (1 - e_i)(a_{i+1} - a_i - 1) < 2^{32}$.

The last constraint emulates a comparison between two addresses or timestamps by bounding their difference; this assumes that all addresses and timestamps fit in 32 bits and that the field is larger than that.

Finally, the iterative checks can be arithmetized by introducing a trace column for the current value c. We add a boundary constraint $c_0 = 0$, and the following transition constraints:

This is out of date, we don't actually need a c column.

1. $v_{\text{from},i} = c_i$,
2. $c_{i+1} = e_i v_{\text{to},i}$.

[1] EVM memory is zero-initialized.

3.5.1 Virtual memory

In the EVM, each contract call has its own address space. Within that address space, there are separate segments for code, main memory, stack memory, calldata, and returndata. Thus each address actually has three compoments:

1. an execution context, representing a contract call,
2. a segment ID, used to separate code, main memory, and so forth, and so on
3. a virtual address.

The comparisons now involve several columns, which requires some minor adaptations to the technique described above; we will leave these as an exercise to the reader.

3.5.2 Timestamps

TODO: Explain $\tau = \texttt{NUM\_CHANNELS} \times \texttt{cycle} + \texttt{channel}$.

3.6 Keccak-f

This table computes the Keccak-f[1600] permutation.

3.7 Keccak sponge

This table computes the Keccak256 hash, a sponge-based hash built on top of the Keccak-f[1600] permutation.

4 Merkle Patricia tries

4.1 Internal memory format

Withour our zkEVM's kernel memory,

1. An empty node is encoded as (`MPT_NODE_EMPTY`).
2. A branch node is encoded as $(\texttt{MPT\_NODE\_BRANCH}, c_1, \ldots, c_{16}, v)$, where each c_i is a pointer to a child node, and v is a pointer to a value. If a branch node has no associated value, then $v = 0$, i.e. the null pointer.

3. An extension node is encoded as $(\texttt{MPT\_NODE\_EXTENSION}, k, c)$, k represents the part of the key associated with this extension, and is encoded as a 2-tuple $(\texttt{packed\_nibbles}, \texttt{num\_nibbles})$. c is a pointer to a child node.

4. A leaf node is encoded as $(\texttt{MPT\_NODE\_LEAF}, k, v)$, where k is a 2-tuple as above, and v is a pointer to a value.

5. A digest node is encoded as $(\texttt{MPT\_NODE\_HASH}, d)$, where d is a Keccak256 digest.

4.2 Prover input format

The initial state of each trie is given by the prover as a nondeterministic input tape. This tape has a slightly different format:

1. An empty node is encoded as $(\texttt{MPT\_NODE\_EMPTY})$.

2. A branch node is encoded as $(\texttt{MPT\_NODE\_BRANCH}, v_?, c_1, \ldots, c_{16})$. Here $v_?$ consists of a flag indicating whether a value is present, followed by the actual value payload if one is present. Each c_i is the encoding of a child node.

3. An extension node is encoded as $(\texttt{MPT\_NODE\_EXTENSION}, k, c)$, k represents the part of the key associated with this extension, and is encoded as a 2-tuple $(\texttt{packed\_nibbles}, \texttt{num\_nibbles})$. c is a pointer to a child node.

4. A leaf node is encoded as $(\texttt{MPT\_NODE\_LEAF}, k, v)$, where k is a 2-tuple as above, and v is a value payload.

5. A digest node is encoded as $(\texttt{MPT\_NODE\_HASH}, d)$, where d is a Keccak256 digest.

In the current implementation, we use a length prefix rather than a is-present prefix, but we plan to change that.

Nodes are thus given in depth-first order, enabling natural recursive methods for encoding and decoding this format.

5 Privileged instructions

To ease and speed-up proving time, the zkEVM supports custom, privileged instructions that can only be executed by the kernel. Any appearance of those

privileged instructions associated opcode value in a contract bytecode would result in an invalid

In what follows, we denote by p_{BN} the characteristic of the BN254 curve base field, curve for which Ethereum supports the ecAdd, ecMul and ecPairing precompiles.

0x0C. `ADDFP254`. Pops 2 elements from the stack interpreted as BN254 base field elements, and pushes their addition modulo p_{BN} onto the stack.

0x0D. `MULFP254`. Pops 2 elements from the stack interpreted as BN254 base field elements, and pushes their product modulo p_{BN} onto the stack.

0x0E. `SUBFP254`. Pops 2 elements from the stack interpreted as BN254 base field elements, and pushes their difference modulo p_{BN} onto the stack. This instruction behaves similarly to the SUB (0x03) opcode, in that we subtract the second element of the stack from the initial (top) one.

0x0F. `SUBMOD`. Pops 3 elements from the stack, and pushes the modular difference of the first two elements of the stack by the third one. It is similar to the SUB instruction, with an extra pop for the custom modulus.

0x21. `KECCAK_GENERAL`. Pops 4 elements (successively the context, segment, and offset portions of a Memory address, followed by a length ℓ) and pushes the hash of the memory portion starting at the constructed address and of length ℓ. It is similar to KECCAK256 (0x20) instruction, but can be applied to any memory section (i.e. even privileged ones).

0x49. `PROVER_INPUT`. Pushes a single prover input onto the stack.

0xC0-0xDF. `MSTORE_32BYTES`. Pops 4 elements from the stack (successively the context, segment, and offset portions of a Memory address, and then a value), and pushes a new offset' onto the stack. The value is being decomposed into bytes and written to memory, starting from the reconstructed address. The new offset being pushed is computed as the initial address offset + the length of the byte sequence being written to memory. Note that similarly to PUSH (0x60-0x7F) instructions there are 31 MSTORE_32BYTES instructions, each corresponding to a target byte length (length 0 is ignored, for the same reasons as MLOAD_32BYTES). Writing to memory an integer fittin in n bytes with a length $\ell > n$ will result in the integer being truncated. On the other hand, specifying a length ℓ greater than the byte size of the value being written will result in padding with zeroes. This process is heavily used when resetting memory sections (by calling MSTORE_32BYTES_32 with the value 0).

0xF6. `GET_CONTEXT`. Pushes the current context onto the stack. The kernel always has context 0.

0xF7. `SET_CONTEXT`. Pops the top element of the stack and updates the current context to this value. It is usually used when calling another contract or precompile, to distinguish the caller from the callee.

0xF8. `MLOAD_32BYTES`. Pops 4 elements from the stack (successively the context, segment, and offset portions of a Memory address, and then a length ℓ), and pushes a value onto the stack. The pushed value corresponds to the U256 integer read from the big-endian sequence of length ℓ from the memory address being reconstructed. Note that an empty length is not valid, nor is a length greater than 32 (as a U256 consists in at most 32 bytes). Missing these conditions will result in an unverifiable proof.

0xF9. `EXIT_KERNEL`. Pops 1 element from the stack. This instruction is used at the end of a syscall, before proceeding to the rest of the execution logic. The popped element, *kexit_info*, contains several informations like the current program counter, current gas used, and if we are in kernel (i.e. privileged) mode.

0xFB. `MLOAD_GENERAL`. Pops 3 elements (successively the context, segment, and offset portions of a Memory address), and pushes the value stored at this memory address onto the stack. It can read any memory location, general (similarly to MLOAD (0x51) instruction) or privileged.

0xFC. `MSTORE_GENERAL`. Pops 4 elements (successively a value, then the context, segment, and offset portions of a Memory address), and writes the popped value from the stack at the reconstructed address. It can write to any memory location, general (similarly to MSTORE (0x52) / MSTORE8 (0x53) instructions) or privileged.

References

[1] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, "Scalable, transparent, and post-quantum secure computational integrity." Cryptology ePrint Archive, Report 2018/046, 2018. https://ia.cr/2018/046.