The Polygon Zero zkEVM

DRAFT
November 16, 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

Each row of the arithmetic table corresponds to a binary or ternary arithmetic operation. Each of these operations has an associated flag f_{op} in the table, such that $f_{\texttt{op}} = 1$ whenever the operation is `op` and 0 otherwise. The full list of operations carried out by the table is as follows:

Binary operations:

- basic operations: "add", "mul", "sub" and "div",
- comparisons: "lt" and "gt",
- shifts: "shr" and "shl",
- "byte": given x_1, x_2, returns the x_1-th "byte" in x_2,
- modular operations: "mod", "AddFp254", "MulFp254" and "SubFp254",
- range-check: no operation is performed, as this is only used to range-check the input and output limbs in the range $[0, 2^{16} - 1]$.

For 'mod', the second input is the modulus. "AddFp254", "MulFp254" and "SubFp254" are modular operations modulo "Fp254'' – the prime for the BN curve's base field.

Ternary operations: There are three ternary operations: modular addition "AddMod", modular multiplication "MulMod" and modular subtraction "SubMod".

Besides the flags, the arithmetic table needs to store the inputs, output and some auxiliary values necessary to constraints. The input and output values are range-checked to ensure their canonical representation. Inputs are 256-bits words. To avoid having too large a range-check, inputs are therefore split into sixteen 16-bits limbs, and range-checked in the range $[0, 2^{16} - 1]$.

Overall, the table comprises the following columns:

- 17 columns for the operation flags f_{op},
- 1 column *op* containing the opcode,
- 16 columns for the 16-bit limbs $x_{0,i}$ of the first input x_0,
- 16 columns for the 16-bit limbs $x_{1,i}$ of the second input x_1,
- 16 columns for the 16-bit limbs $x_{2,i}$ of the third input x_2,
- 16 columns for the 16-bit limbs r_i of the output r,
- 32 columns for auxiliary values $\texttt{aux}_i$,
- 1 column `range_counter` containing values in the range $[0, 2^{16} - 1]$, for the range-check,
- 1 column storing the frequency of appearance of each value in the range $[0, 2^{16} - 1]$.

Note on *op*: The opcode column is only used for range-checks. For optimization purposes, we check all arithmetic operations against the cpu table together. To ensure correctness, we also check that the operation's opcode corresponds to its behavior. But range-check is not associated to a unique operation: any operation in the cpu table might require its values to be checked. Thus, the arithmetic table cannot know its opcode in advance: it needs to store the value provided by the cpu table.

3.2.1 Auxiliary columns

The way auxiliary values are leveraged to efficiently check correctness is not trivial, but it is explained in detail in each dedicated file. Overall, five files explain the implementations of the various checks. Refer to:

1. "mul.rs" for details on multiplications.
2. "addcy.rs" for details on addition, subtraction, "lt" and "gt".
3. "modular.rs" for details on how modular operations are checked. Note that even though "div" and "mod" are generated and checked in a separate file, they leverage the logic for modular operations described in "modular.rs".
4. "byte" for details on how "byte" is checked.
5. "shift.rs" for details on how shifts are checked.

Note on "lt" and "gt": For "lt" and "gt", auxiliary columns hold the difference d between the two inputs x_1, x_2. We can then treat them similarly to subtractions by ensuring that $x_1 - x_2 = d$ for "lt" and $x_2 - x_1 = d$ for "gt". An auxiliary column cy is used for the carry in additions and subtractions. In the comparisons case, it holds the overflow flag. Contrary to subtractions, the output of "lt" and "gt" operations is not d but cy.

Note on "div": It might be unclear why "div" and "mod" are dealt with in the same file.

Given numerator and denominator n, d, we compute, like for other modular operations, the quotient q and remainder $\texttt{rem}$:

$$div(x_1, x_2) = q * x_2 + \texttt{rem}$$

. We then set the associated auxiliary columns to $\texttt{rem}$ and the output to q.

This is why "div" is essentially a modulo operation, and can be addressed in almost the same way as "mod". The only difference is that in the "mod" case, the output is $\texttt{rem}$ and the auxiliary value is q.

Note on shifts: SHR and SHL leverage DIV and MUL respectively for the checks. Indeed, given inputs s, x, the output should be $x >> s$ for "shr" (resp. $x << s$ for "shl"). Since shifts are binary operations, we can use the third input columns to store $s_{\texttt{shifted}} = 1 << s$. Then, we can use the "div" logic (resp. "mul" logic) to ensure that the output is $\frac{x}{s_{\texttt{shifted}}}$ (resp. $x * s_{\texttt{shifted}}$).

3.3 Byte Packing

TODO

3.4 Logic

Each row of the logic table corresponds to one bitwise logic operation: either AND, OR or XOR. Each input for these operations is represented as 256 bits, while the output is stored as eight 32-bit limbs.

Each row therefore contains the following columns:

1. $f_{\texttt{and}}$, an "is and" flag, which should be 1 for an OR operation and 0 otherwise,
2. $f_{\texttt{or}}$, an "is or" flag, which should be 1 for an OR operation and 0 otherwise,
3. $f_{\texttt{xor}}$, an "is xor" flag, which should be 1 for a XOR operation and 0 otherwise,
4. 256 columns $x_{1,i}$ for the bits of the first input x_1,
5. 256 columns $x_{2,i}$ for the bits of the second input x_2,
6. 8 columns r_i for the 32-bit limbs of the output r.

Note that we need all three flags because we need to be able to distinguish between an operation row and a padding row – where all flags are set to 0.

The subdivision into bits is required for the two inputs as the table carries out bitwise operations. The result, on the other hand, is represented in 32-bit limbs because it is checked against the cpu, which stores values in the same way.

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

0xFB. `MLOAD_GENERAL`. Returns

0xFC. `MSTORE_GENERAL`. Returns

TODO. `STACK_SIZE`. Returns

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.