The Polygon Zero zkEVM

DRAFT
November 17, 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 `rem`:

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

. We then set the associated auxiliary columns to `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 `rem` and the auxiliary value is q.

Note on shifts: "shr" and "shl" are internally constrained as "div" and "mul" respectively with shifted operands. 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 since we do not need individual bits and can therefore save the remaining 248 columns. Moreover, the output 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 of the memory could be checked as follows:

1. Verify the ordering by checking that $(a_i, \tau_i) \leq (a_{i+1}, \tau_{i+1})$ for each consecutive pair.
2. Enumerate the purportedly-ordered log while tracking the "current" value of v, which is initially zero.[1]
   (a) Upon observing an address which doesn't match that of the previous row, if the operation is a read, check that $v = 0$.
   (b) Upon observing a write, don't constrain v.
   (c) Upon observing a read at timestamp τ_i which isn't the first operation at this address, check that $v_i = v_{i-1}$.

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 changed. An honest prover will set

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

We also introduce a range-check column c, which should hold:

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

The extra -1 ensures that the address actually changed if $e_i = 1$. We then impose the following transition constraints:

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

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.

[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

Memory operations are sorted by address a and timestamp τ. For a memory operation in the CPU, we have:

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

Since a memory channel can only hold at most one memory operation, every CPU memory operation's timestamp is unique.

Note that it doesn't mean that all memory operations have unique timestamps. There are two exceptions:

- Before bootstrapping, we write some global metadata in memory. These extra operations are done at timestamp $\tau = 0$.
- Some tables other than CPU can generate memory operations, like KeccakSponge. When this happens, these operations all have the timestamp of the CPU row of the instruction which invoked the table (for KeccakSponge, KECCAK_GENERAL).

3.6 Keccak-f

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

3.6.1 Keccak-f Permutation

To explain how this table is structured, we first need to detail how the permutation is computed. This page gives a pseudo-code for the permutation. Our implementation differs slightly – but remains equivalent – for optimization and constraint degree reasons.

Let:

- S be the sponge width ($S = 25$ in our case)
- `NUM_ROUNDS` be the number of Keccak rounds (`NUM_ROUNDS` $= 24$)
- RC a vector of round constants of size `NUM_ROUNDS`
- I be the input of the permutation, comprised of S 64-bit elements

The first step is to reshape I into a 5×5 matrix. We initialize the state A of the sponge with I:

$$A[x, y] := I[x, y] \ \forall x, y \in \{0..4\}$$

We store A in the table, and subdivide each 64-bit element into two 32-bit limbs. Then, for each round i, we proceed as follows:

1. First, we define $C[x] := \texttt{xor}_{i=0}^{4} A[x, i]$. We store C as bits in the table. This is because we need to apply a rotation on its elements' bits and carry out `xor` operations in the next step.
2. Then, we store a second vector C' in bits, such that:

$$C'[x, z] = C[x, z] \ \texttt{xor} \ C[x - 1, z] \ \texttt{xor} \ C[x + 1, z - 1]$$

.

3. We then need to store the updated value of A:

$$A'[x, y] = A[x, y] \ \texttt{xor} \ C[x, y] \ \texttt{xor} \ C'[x, y]$$

Note that this is equivalent to the equation in the official Keccak-f description:

$$A'[x, y] = A[x, y] \ \texttt{xor} \ C[x - 1, z] \ \texttt{xor} \ C[x + 1, z - 1]$$

.

4. The previous three points correspond to the θ step in Keccak-f. We can now move on to the ρ and π steps. These steps are written as:

$$B[y, 2 \times x + 3 \times y] := \texttt{rot}(A'[x, y], r[x, y])$$

where `rot(a, s)` is the bitwise cyclic shift operation, and r is the matrix of rotation offsets. We do not need to store B: B's bits are only a permutation of A''s bits.

5. The χ step updates the state once again, and we store the new values:

$$A''[x, y] := B[x, y] \texttt{ xor } (\texttt{not } B[x + 1, y] \texttt{ and } B[x + 2, y])$$

Because of the way we carry out constraints (as explained below), we do not need to store the individual bits for A'': we only need the 32-bit limbs.

6. The final step, ι, consists in updating the first element of the state as follows:

$$A'''[0, 0] = A''[0, 0] \texttt{ xor } RC[i]$$

where

$$A'''[x, y] = A''[x, y] \forall (x, y) \neq (0, 0)$$

Since only the first element is updated, we only need to store $A'''[0, 0]$ of this updated state. The remaining elements are fetched from A''. However, because of the bitwise `xor` operation, we do need columns for the bits of $A''[0, 0]$.

Note that all permutation elements are 64-bit long. But they are stored as 32-bit limbs so that we do not overflow the field.

It is also important to note that all bitwise logic operations (`xor` , `not` and `and`) are checked in this table. This is why we need to store the bits of most elements. The logic table can only carry out eight 32-bit logic operations per row. Thus, leveraging it here would drastically increase the number of logic rows, and incur too much overhead in proving time.

3.6.2 Columns

Using the notations from the previous section, we can now list the columns in the table:

1. NUM_ROUND$S = 24$ columns c_i to determine which round is currently being computed. $c_i = 1$ when we are in the i-th round, and 0 otherwise. These columns' purpose is to ensure that the correct round constants are used at each round.

2. 1 column t which stores the timestamp at which the Keccak operation was called in the cpu. This column enables us to ensure that inputs and outputs are consistent between the cpu, keccak-sponge and keccak-f tables.

3. $5 \times 5 \times 2 = 50$columns to store the elements of A. As a reminder, each 64-bit element is divided into two 32-bit limbs, and A comprises $S = 25$ elements.

4. $5 \times 64 = 320$ columns to store the bits of the vector C.

5. $5 \times 64 = 320$ columns to store the bits of the vector C'.

6. $5 \times 5 \times 64 = 1600$ columns to store the bits of A'.

7. $5 \times 5 \times 2 = 50$ columns to store the 32-bit limbs of A''.

8. 64 columns to store the bits of $A''[0, 0]$.

9. 2 columns to store the two limbs of $A'''[0, 0]$.

In total, this table comprises 2,431 columns.

3.6.3 Constraints

Some constraints checking that the elements are computed correctly are not straightforward. Let us detail them here.

First, it is important to highlight the fact that a `xor` between two elements is of degree 2. Indeed, for x `xor` y, the constraint is $x + y - 2 \times x \times y$, which is of degree 2. This implies that a `xor` between 3 elements is of degree 3, which is the maximal constraint degree for our STARKs.

We can check that $C'[x, z] = C[x, z]$ `xor` $C[x - 1, z]$ `xor` $C[x + 1, z - 1]$. However, we cannot directly check that $C[x] = \texttt{xor}_{i=0}^{4} A[x, i]$, as it would be a degree 5 constraint. Instead, we use C' for this constraint. We see that:

$$\texttt{xor}_{i=0}^{4} A'[x, i, z] = C'[x, z]$$

This implies that the difference $d = \sum_{i=0}^{4} A'[x, i, z] - C'[x, z]$ is either 0, 2 or 4. We can therefore enforce the following degree 3 constraint instead:

$$d \times (d - 2) \times (d - 4) = 0$$

Additionally, we have to check that A' is well constructed. We know that A' should be such that $A'[x, y, z] = A[x, y, z]$ `xor` $C[x, z]$ `xor` $C'[x, z]$. Since we do not have the bits of A elements but the bits of A' elements, we check the equivalent degree 3 constraint:

$$A[x, y, z] = A'[x, y, z] \texttt{ xor } C[x, z] \texttt{ xor } C'[x, z]$$

Finally, the constraints for the remaining elements, A'' and A''' are straightforward: A'' is a three-element bitwise `xor` where all bits involved are already storedn and $A'''[0, 0]$ is the output of a simple bitwise `xor` with a round constant.

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 (`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 (`MPT_NODE_EXTENSION`, k, c), k represents the part of the key associated with this extension, and is encoded as a 2-tuple (`packed_nibbles`, `num_nibbles`). c is a pointer to a child node.
4. A leaf node is encoded as (`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 (`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 (`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 in a contract bytecode for instance would result in an unprovable state.

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, see below). Writing to memory an integer fitting 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.