A specification for a ZK-EVM

Olivier Bégassat [1], Alexandre Belling[1], Théodore Chapuis-Chkaiban[1], and Nicolas Liochon[1]

[1]ConsenSys, Applied R&D Team *

October 2021

Abstract

We describe a *zk-EVM* arithmetization supporting the three following design goals: (1) support for *all* EVM opcodes including internal smart contract calls, error management and gas management (2) ability to execute bytecode *as is* (3) minimal prover time. We strive to provide an arithmetization that respects the EVM specification as defined in the Ethereum yellow paper [1]. We provide an original and comprehensive approach of the zk-EVM problem which is technically realizable using existing zero-knowledge proving schemes.

1 Introduction

1.1 Context and previous results

Increasing blockchains capacity, and hence lowering the transaction cost is a major technical challenge.

Rollups are promising technologies that would allow to considerably increase the capacity of the Ethereum Blockchain. An introduction to Rollups, zk-EVMs and their role in improving Ethereum capacity can be respectively found in [2, 3]. Recently, multiple attempts at building scalable and practical rollup solutions have been positively received. zkSync [4], for instance, transpiles Yul into a zk-VM friendly bytecode. Cairo [5], on the other hand, uses a custom architecture adapted to an efficient STARK prover for smart contracts written in Cairo .

Other projects, such as Hermez [6] or Scroll Tech [7] aim to interpret directly the EVM bytecode, without any intermediary transpiler or extra compilation step. This is also the approach we took.

1.2 Relation with existing projects

Our work draws inspiration from the existing design of Cairo [5] from which we borrow the concepts of *virtual columns*, *execution traces*, *range-proof and memory integrity trace permutation trick*, and *provable read-only-memory*. We extend and adapt these concepts to the different parts of our EVM architecture. Most notably, we are able to adapt the EVM stack to a read-only architecture that leads to a significant reduction of the size of our execution traces: we only need six pointers to represent the stack, while a naive implementation of the EVM stack would require at least 1024 pointers.

Our zk-EVM architecture comprises different modules, each tasked with proving the execution of a specific part of the contract execution. We borrow this approach from Hermez [8].

*Email: firstName.lastName@consensys.net

1.3 Outline of this paper

The remainder of this paper is organized as follows:

1. **Global organisation of the zk-EVM**: provides a high level overview of the main components of our zk-EVM architecture.
2. **Tools and notations**: introduces the basic mathematical tools and methods used in our arithmetization,
3. **Constraint sets**: The low-level circuits for some of the components of our zk-EVM

To improve the readability of this paper, we have chosen to provide full constraint systems for a few (representative) modules only, which we strive to describe as comprehensively as possible. Other modules, like the main execution trace and the storage module, have been fully designed - however, given the complexity of these components, we chose to postpone slightly their publication. If interested, you may contact us directly for more information on these modules.

Given the complexity of our arithmetization, mistakes are to be expected.

Contents

2 Global organisation of the zk-ethereum virtual machine

Our model for a zk-EVM comprises various application specific subcomponents:

- A (block specific) ROM (Read-Only-Memory) contains the bytecode of the smart contracts to be executed.
- The *main execution trace* which processes the instructions and builds the stack memory.
- A small set of *Application specific modules* for RAM, storage, binary or arithmetic opcodes.
- An *instruction decoder* which decomposes the EVM opcodes in a series of instruction flags.

The ROM's (which one may imagine as the concatenation of the bytecodes of the smart contracts called in a block) main purpose is to translate the contracts bytecode into a succession of instructions that can be processed by the zk-EVM.

Every module is designed to deal only with a certain subset of the EVM instruction set. Modules select the instructions that concern them using specific flags from the flag decomposition provided by the instruction decoder. The correctness of these flags and parameters is guaranteed by means of an inclusion proof into the instruction decoder — an *immutable* piece of *public data*. Each module has its own internal execution trace and associated constraint system. Modules are linked to the global execution module by means of a bussing mechanism (in practice: plookup inclusion proofs).

This organization is represented in the figure 1.

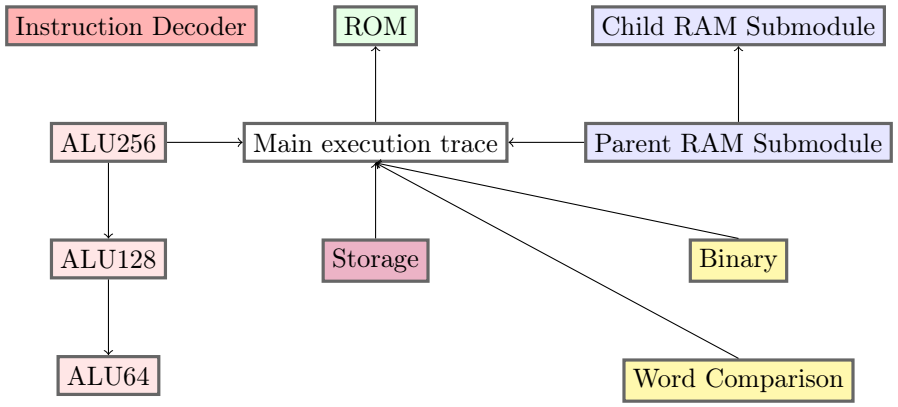


Figure 1: Modules of the zk-EVM. Arrows represent plookup inclusion proofs. N.B.: the Main execution trace, ALU, Storage, RAM, Binary and Word comparison modules also point to the instruction decoder.

2.1 A high level overview of the zk-EVM modules

Here, we provide more details on the zk-EVM modules and their roles in the actual architecture. The way these components interact and together constitute a zk-EVM is described on a high level in section 2.3.

About submodules: The modules in our zk-EVM split into submodules linked together by means of Plookup proofs. The submodules defining a given module satisfy a hierarchical relationship: a parent submodule decomposes (a subset of module relevant) EVM instructions into atomic operations. These operations are executed by its child submodule. This simplifies the mathematical formalization of complex EVM operations, by balancing complex logic between different submodules.

The parent/child submodule structure is currently implemented in the RAM and ALU module.

2.1.1 Main execution trace

The *main execution trace*, or *stack trace*, is the core of the program execution. Every instruction goes through the main execution trace and is either processed directly (e.g. $PUSH$, POP, DUP or $SWAP$ instructions) or sent to other modules via a bus system - which we will describe extensively later.

Stack memory: The main execution trace also reproduces the stack architecture of the EVM. We have chosen to rely on a read only memory model to reproduce the stack : the stack memory is a mapping of Read-only-memory addresses (Mem) to tuples (Val, Ptr) composed of a stack value and

a pointer to the memory cell containing the previous stack element. The figure 2 provides an example of such stack memory.

Mem	0	1	2	$\cdots$	$N-1$	N
Val	v_0	v_1	v_2	$\cdots$	v_{N-1}	v_N
Ptr	$\emptyset$	p_1	p_2	$\cdots$	p_{N-1}	p_N

Figure 2: Table representation of the stack memory

Most of the EVM opcodes involve 0, 1 or 2 inputs, but some EVM instructions take more: $CALL$, for instance, takes 7. Instructions like $SWAP$ or DUP, implicitly take up to 16 (these instructions may fetch the 16-th element of the stack and require access to all elements in between). Yet it proved sufficient to keep track of only the two upper stack elements (rather than all 16 top stack elements), thus our main execution trace only keeps track of the two upper stack elements. OPCODES that take 3 arguments or more (explicitly or implicitly) need to be processed in several steps (i.e. several lines in the execution trace), successively removing the top two elements of the stack.

It is comparatively easy to adapt our stack trace to deal with inter smart contract calls, and smart contract batches : the stack memory can be reinitialized by adding stack elements pointing to themselves which prevents contract calls from interacting with the stack cells that it shouldn't have access to. Dealing with inter smart contract calls is a recurrent issue among the existing models of zk-EVMs.

2.1.2 The RAM module

Emulating efficiently the EVM RAM is a challenge in itself. Indeed, several instructions acting on the RAM, such as $RETURNDATACOPY$, $CALLDATACOPY$, $RETURN$, $BYTECODECOPY$, involve copying words of an arbitrary size (likely to be greater than the EVM word size - 32 bytes), that may come from two different smart contract memories. On the other hand, instructions like $MSTORE$, $MLOAD$, or $MSTORE8$ act on at most 32 bytes of memory (1 byte for MSTORE8).

The EVM memory involves rather complex I/O relations due to it handling both bytes and words: one can $MSTORE$ elements in the memory at a byte granularity, but the $MLOAD$ instruction always retrieves EVM words (32 bytes) from memory. The EVM memory features a byte granularity, and 32-bytes word addresses - i.e., this memory model can be seen as an array of maximum size 2^{256} whose elements are bytes.

We have chosen to tweak slightly the representation and to store *EVM words at word multiple addresses*, instead of *bytes at any possible address*. To store/load words to/from non word multiple addresses, one would need to change at most two consecutive words in memory, the figure 3 provides a graphical example of such memory model, with words of 4 bytes.

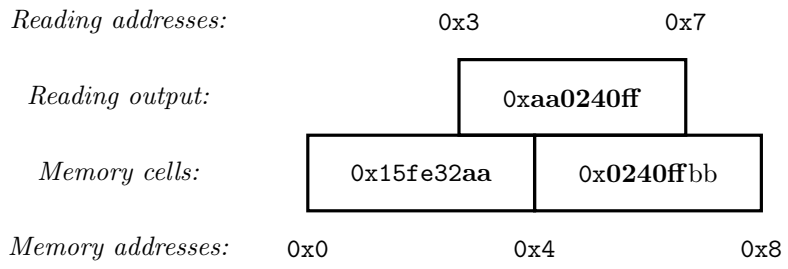


Figure 3: Overlapping reading process for words of 4 bytes.

To deal with the aforementioned technical challenges, we have decided to divide the RAM module into two submodules: the child RAM and the parent RAM submodules. The consistency between the values of the parent and the child RAM submodules is guaranteed by a Plookup inclusion proof of the parent RAM submodule into the child RAM submodule.

Parent RAM submodule: The parent RAM submodule processes the memory instructions coming from the main execution trace - it decomposes these complex EVM instructions into simple READ/WRITE memory instructions, that involve writing/reading at most two consecutive memory words. The parent RAM submodule sends these simple instructions to the child RAM submodule.

Child RAM submodule: The child RAM submodule executes the simple instructions transmitted by the parent RAM submodule, returns the result and verifies the consistency of the RAM memory which amounts to checking that the RAM addresses are continuous and that the reading operations are consistent with the previous values stored.

Worked out example. The figure 3 is an explanatory drawing of the interactions between the parent and the child RAM with 4-byte long words. This figure explains the interactions between the parent and the Child RAM for a RETURN operation, that stores the values of the current RAM located from the address `0x13a2` to `0x12ab`, to the caller RAM, at the address range [`0xaace`, `0xaac3`].

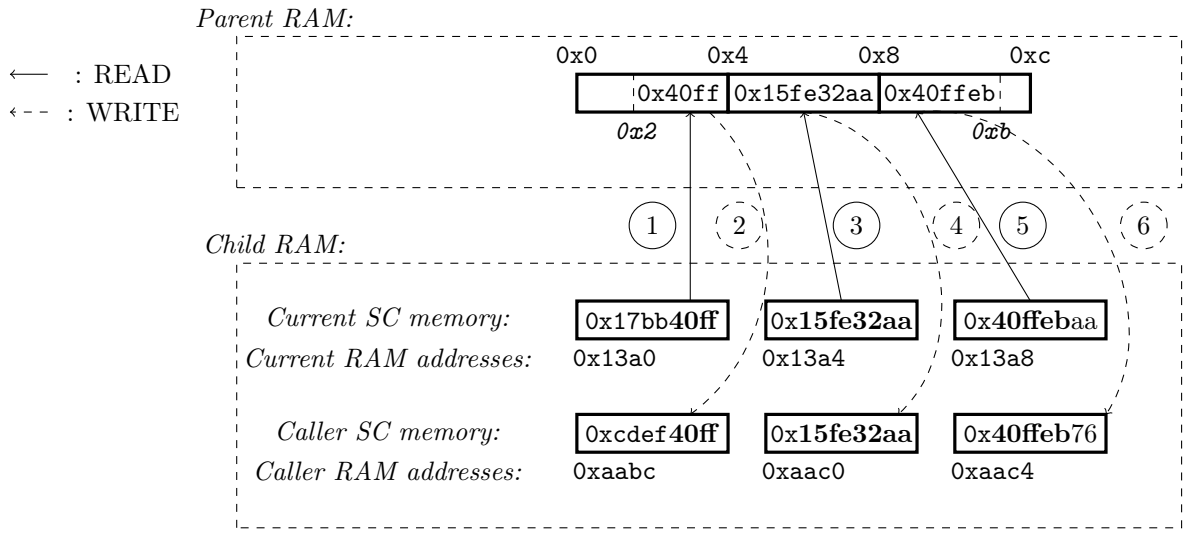


Figure 4: Illustration of the relations between the parent and the child RAM, for a RETURN instruction.

This operation may be divided into 6 steps, indicated in the figure 3:

1. Read the last two bytes from the current RAM address `0x13a0`
2. Replace the last two bytes of the caller RAM address `0xaabe` with the previously read bytes
3. Read the whole memory word of the current RAM starting at the address `0x13a4`
4. Store this memory word to the caller RAM at the address `0xaac0`
5. Read the first 3 bytes of the memory word at the address `0x13a8` of the current RAM.
6. Replace the first 3 bytes of the memory word at the address `0xaac4` of the caller RAM.

2.1.3 The arithmetic operations module

This module emulates the EVM 256-bit arithmetic. It is comprised of 3 submodules: the ALU256, the ALU128 and the ALU64 which respectively perform arithmetic operations on 256 bit long words, 128 bit long words and 64 bit long words. These three modules are tied together by Plookup inclusion proofs of the 256ALU into the 128ALU into the 64ALU. The figure 5 provides an example of such submodule communication to perform a 256-bit addition of two field elements.

256ALU: The 256-bit ALU decomposes 256-bit long input field elements into two 128-bit long field elements - the high 128 bits and the low 128 bits - and sends arithmetic operation requests to the 128 bit ALU. It further computes the final result by computing the high/low bit decomposition of it.

128ALU: The 128ALU follows the same pattern, except that, this time the 128-bit words are decomposed into 64-bit words and arithmetic operation requests are sent to the 64ALU.

64ALU: The 64ALU does the actual computations: it checks that the inputs are indeed 64 bit words by performing range proofs on the 16-bit decomposition of these 64-bit long words. The results are sent back to the 128ALU, which then sends back the results to the 256ALU.

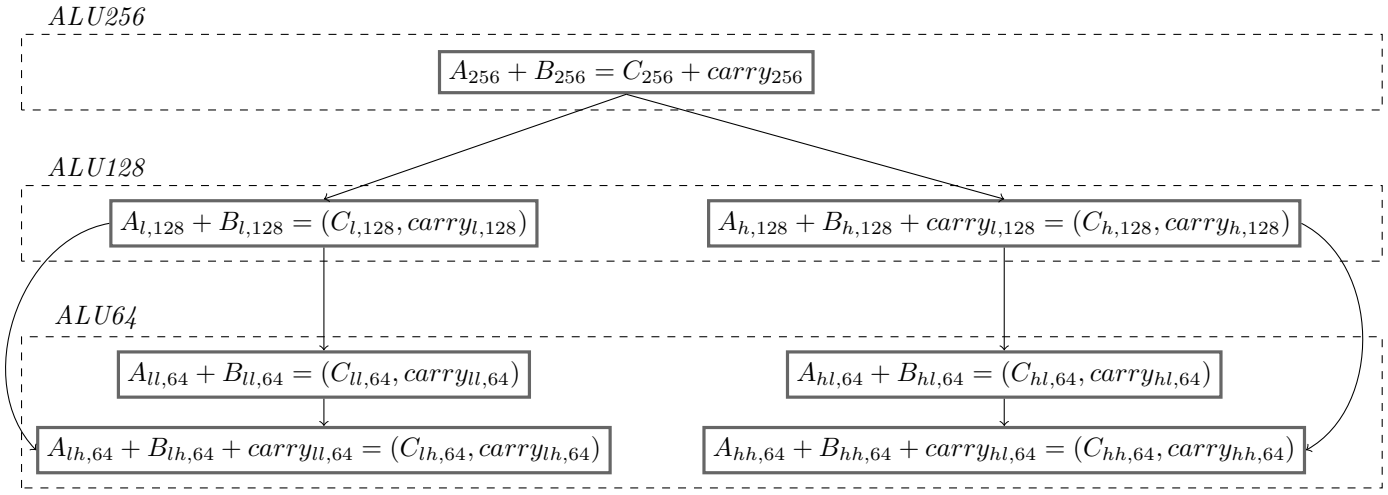


Figure 5: Field element decomposition and submodules communication inside the arithmetic module, addition example.

2.1.4 Binary, word comparison modules

The binary and word comparison modules are two separate modules that decompose field elements into 16-bit words and perform binary operations on these 16-bit word.

Binary module: The binary module relies on Plookup to perform binary operations - one has to prove the inclusion of quadruples $(Inst, Byte_1, Byte_2, Byte_{res})$ into a public logic table. Table 1 provides an illustration (with a byte decomposition, to simplify the representation) of this process - the timestamp column allows keeping track of the instruction being currently executed (instructions are executed in several steps, to decompose the inputs and perform bitwise operations).

Inst	Timestamp	$Input_1$	$Byte_1$	$Carry_1$	$Input_2$	$Byte_2$	$Carry_2$	Result	$Byte_{res}$	$Carry_{res}$	Length
AND	1	0x1ea1ff	0xff	0x1ea1	0xff00ff00	0x00	0xff00ff	0x00a100	0x00	0x00a1	3
AND	1	0x1ea1ff	0xa1	0x1e	0xff00ff00	0xff	0xff00	0x00a100	0xa1	0x00	2
AND	1	0x1ea1ff	0x1e	0x00	0xff00ff00	0x00	0xff	0x00a100	0x00	0x00	1
AND	1	0x1ea1ff	0x0	0x00	0xff00ff00	0xff	0x00	0x00a100	0x00	0x00	0
XOR	2	0x1001	0x01	0x10	0x1010	0x10	0x10	0x0011	0x11	0x00	1
XOR	2	0x1001	0x10	0x00	0x1010	0x10	0x10	0x0011	0x00	0x00	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 1: Binary module execution trace

Word Comparison module: The word comparison module computes the difference between the bytes of two memory words, starting from the most significant byte and returns a comparison boolean: for instance, when testing an inequality $a < b$, the word module returns 1 when this inequality is verified, 0 otherwise.
Table 2 represents the execution of this module in a similar way as the binary module execution. For the sake of the representation, we will use 8-bit (byte) decomposition instead of 16-bits word decomposition.

TS	*Sw*	*Eq*	*Inst*	*Input_1*	*Input_2*	*Res*	*Prefix_1*	*Prefix_2*	*Comp*	*B_1*	*B_2*	*BComp*
0	1	0	SGT	0xa3ff02	0xa3ffb7	0	0xa3	0xa3	1	a3	a3	0
0	1	0	SGT	0xa3ff02	0xa3ffb7	0	0xa3ff	0xa3ff	1	ff	ff	0
0	1	0	SGT	0xa3ff02	0xa3ffb7	0	0xa3ff02	0xa3ffb7	1	02	b7	1
1	0	1	LT	0x00a12c	0x00a12c	1	0x00	0x00	0	00	00	0
1	0	1	LT	0x00a12c	0x00a12c	1	0x00a1	0x00a1	0	a1	a1	0
1	0	1	LT	0x00a12c	0x00a12c	1	0x00a12c	0x00a12c	0	2c	2c	0
2	1	1	GT	0x0fabd9	0x0fc73d	0	0x0f	0x0f	1	0f	0f	0
2	1	1	GT	0x0fabd9	0x0fc73d	0	0x0fab	0x0fc7	1	ab	c7	1
2	1	1	GT	0x0fabd9	0x0fc73d	0	0x0fabd9	0x0fc73d	1	d9	3d	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 2: We abbreviate *WCTimeStamp*, *SwitchFlag* and *EqFlag* to *TS*, *Sw* and *Eq* respectively.

2.1.5 Storage Module

The storage module represents the EVM word addressable, word granular storage. This module is initialized with the previous storage of the smart contract (the storage initialization consistency is performed within the zk-EVM). The storage module outputs the new storage of the executed smart contracts.

This module performs the elementary operations *SSTORE* and *SLOAD*. The main complexity of it relies on its storage check part that allows to:

- Initialize the storage of the smart contracts loaded in the ROM
- Check the consistency of the storage values used within the execution
- Revert the storage state if the transaction has failed
- Compute the final storage hash.
- Compute the gas cost associated with the storage operations.

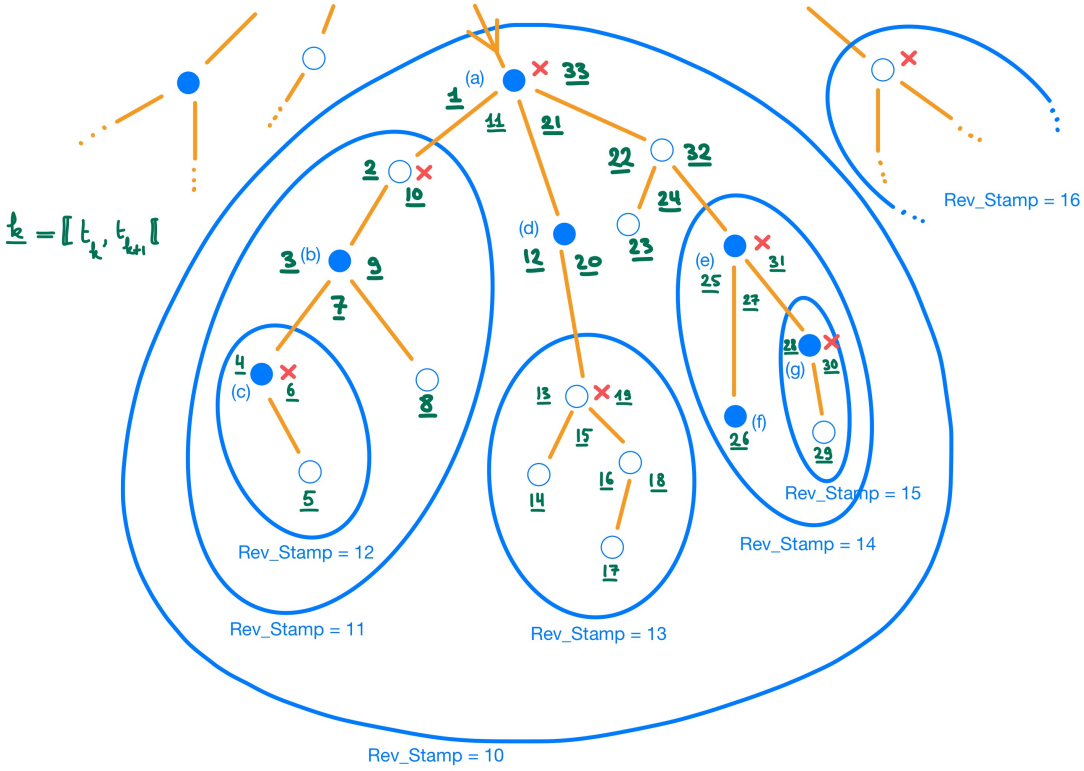


Figure 6: The above represents a portion of the tree of nested smart contract calls of a transaction. The solid blue dots represent calls to a given smart contract A, the hollow ones represent calls to other smart contracts. Red crosses represent reverts. The underlined integers $\underline{k} = [\![t_k, t_{k+1} [\![$ represent the *STORAGE_ TIMESTAMP* intervals wherein a given smart contract performs storage operations

The smart contract calls within a given transaction form a rooted tree with directed edges (contract calls) whose vertices (smart contracts) naturally carry labels: a smart contract address and a set of smart contract numbers ($1 + d_v$ where d_v is the outgoing degree of vertex v). For a given smart contract address and storage address we define extra labels: a time interval representing the (storage time stamp) validity interval of stored values, the revert stamp, the previous revert stamp (i.e. the revert stamp containing the currently valid storage values) and the parent revert stamp (i.e. the revert stamp to which to roll back in case of a revert). We stress that these labels depend not only on the smart contract address but also on the storage address. This allows us to only propagate auxiliary information only for the storage addresses that are effectively touched at a later point by the smart contract. For a more detailed explanation of the tools we use to process reverted transactions (like the revert stamps), please refer to the section 3.11.

The main properties of these labels are that, for a given smart contract address and storage address,

- the begin and end storage stamps labels with a given revert stamp form a partition by intervals of either the empty interval (if the values at that address aren't touched in the subtree) or of the interval with the "entry" and "exit" storage time stamps of the (nested) call
- the *Previous_ Revert_ Stamp* indicates the Revert Stamp of the last relevant modification to the data at the storage address
- the *Parent_ Revert_ Stamp* indicates the Revert Stamp to which to revert to when exiting the

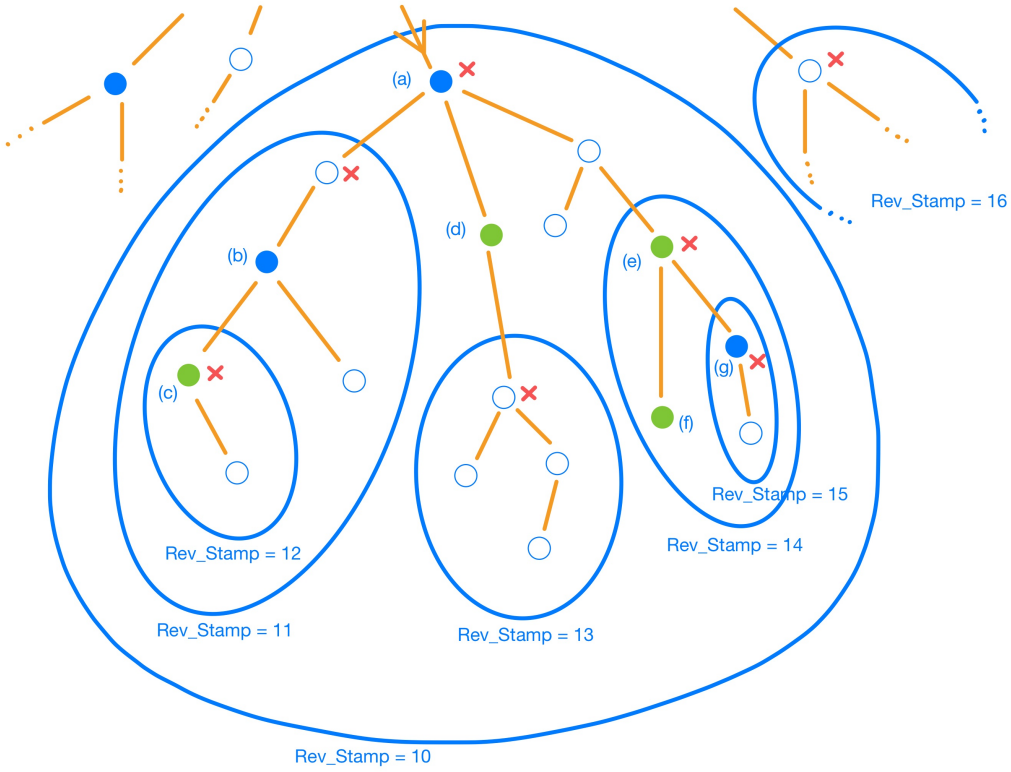


Figure 7: We work with the smart contract A and a given storage address *addr*. We color the calls to A in green if that call accesses (read or write) the address *addr*.

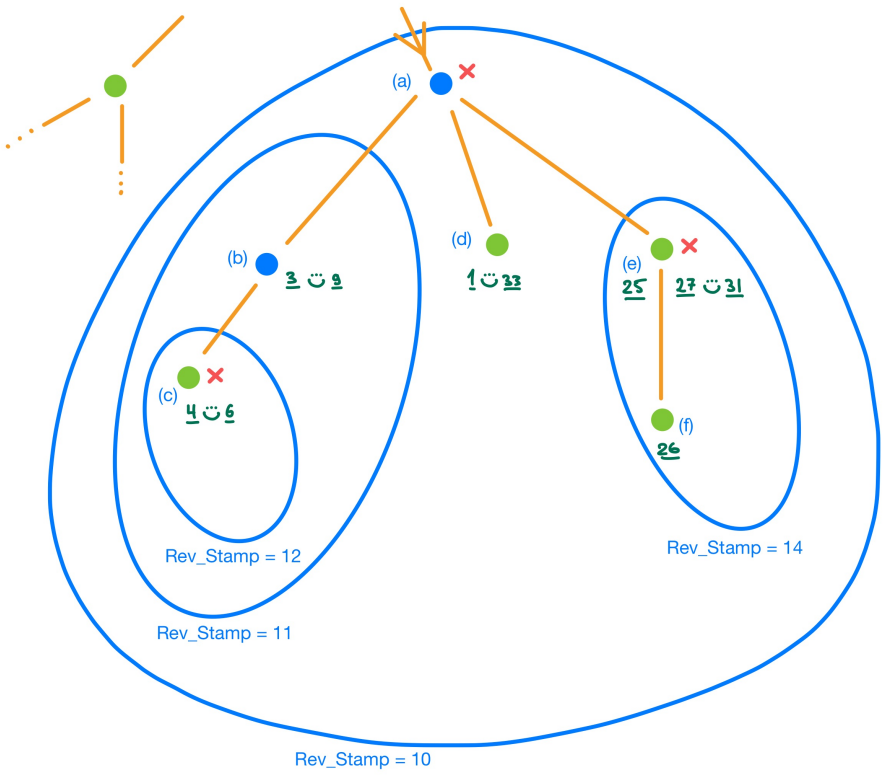


Figure 8: This simplified view of the nested calls to A discards all calls to other smart contracts as well as all calls to A that don't access the storage address *addr* and whose descendants also don't access it. N.B. We use the short hand $\underline{a} \smile \underline{b}$ to represent $\bigcup_{k=a,\dots,b} \underline{k} = [\![t_a, t_{b+1} [\![$ to represent the $STORAGE\_STAMP$ intervals of validity. Since the call (a) reverts, one must initialize the storage address *addr* of the contract A for the $REVERT\_STAMP = 10$, this initialized value remains valid until the next time *addr* is modified by a call of A within $REVERT\_STAMP = 10$. Only the call (d) to A accesses *addr* at a $REVERT\_STAMP = 10$. Its interval is thus $\underline{10} \smile \underline{33}$. The call (b) to A does't access *addr* but one of its descendants does. We must thus load the currently valid value at the storage address *addr* along with the validity period, $\underline{3} \smile \underline{9}$. Its descendant (call (c)) access *addr* and reverts and is the only call to A for that $REVERT\_STAMP = 12$, thus its $STORAGE\_STAMP$ validity interval is the whole time interval wherein (c) runs, i.e. $\underline{4} \smile \underline{6}$. There are two calls to A at $REVERT\_STAMP = 14$, one which reverts, one which doesn't revert on its own (but will be reverted nonetheless, having positive $REVERT\_STAMP$). There are thus 3 $STORAGE\_STAMP$ intervals partitioning the $STORAGE\_STAMP$ interval wherein $REVERT\_STAMP = 14$: $\underline{25}$, $\underline{26}$ and $\underline{27} \smile \underline{31}$

SC	*ADDR*	*REV*	*SS*	*Inst*	*Val*	*Beg*	*End*	*PREV*	*PARENT*
A	$addr$	10	τ_0	*INIT*	v_0	t_1	t_{10}	6	6
A	$addr$	10	τ_1	*SLOAD*	v_0	t_{10}	t_{34}	10	6
A	$addr$	10	τ_2	*SLOAD*	v_0	t_{10}	t_{34}	10	6
A	$addr$	10	τ_3	*SSTORE*	v_1	t_{10}	t_{34}	10	6
A	$addr$	10	τ_4	*SSTORE*	v_2	t_{10}	t_{34}	10	6
A	$addr$	10	τ_5	*SLOAD*	v_2	t_{10}	t_{34}	10	6
A	$addr$	11	τ_6	*INIT*	v_1	t_3	t_{10}	10	10
A	$addr$	12	τ_7	*INIT*	v_1	t_4	t_7	11	11
A	$addr$	12	τ_8	*SSTORE*	v_3	t_4	t_7	12	11
A	$addr$	12	τ_9	*SSTORE*	v_4	t_4	t_7	12	11
A	$addr$	12	τ_{10}	*SLOAD*	v_4	t_4	t_7	12	11
A	$addr$	14	τ_{11}	*INIT*	v_2	t_{25}	t_{26}	10	10
A	$addr$	14	τ_{12}	*SLOAD*	v_2	t_{25}	t_{26}	14	10
A	$addr$	14	τ_{13}	*SSTORE*	v_6	t_{25}	t_{26}	14	10
A	$addr$	14	τ_{14}	*SSTORE*	v_6	t_{25}	t_{26}	14	10
A	$addr$	14	τ_{15}	*SLOAD*	v_6	t_{26}	t_{27}	14	10
A	$addr$	14	τ_{16}	*SLOAD*	v_6	t_{26}	t_{27}	14	10
A	$addr$	14	τ_{17}	*SSTORE*	v_7	t_{26}	t_{27}	14	10
A	$addr$	14	τ_{18}	*SLOAD*	v_7	t_{27}	t_{32}	14	10
A	$addr$	14	τ_{19}	*SSTORE*	v_8	t_{27}	t_{32}	14	10
A	$addr$	14	τ_{20}	*SLOAD*	v_8	t_{27}	t_{32}	14	10
A	$addr$	14	τ_{21}	*SSTORE*	v_9	t_{27}	t_{32}	14	10
A	$addr$	14	τ_{22}	*SSTORE*	v_{10}	t_{27}	t_{32}	14	10

Figure 9: The part of the storage execution trace that concerns itself with the storage operations performed by a smart contract in a block. It is ordered first by smart contract address (here: A), then by storage address (here $addr$), then by *REVERT_STAMP*, then by *STORAGE_STAMP*. We use the shorthand *Inst* for *INSTRUCTION*, *SS* for *STORAGE_STAMP*, *Beg* for *Begin_STORAGE_STAMP*, *End* for *End_STORAGE_STAMP*, *REV* for *REVERT_STAMP*, *PREV* for *PREVIOUS_REVERT_STAMP*, *PARENT* for *PARENT_REVERT_STAMP*, *Val* for *VALUE*.

current revert stamp.

We provide a worked out example below. In the pictures below the blue dots represent calls to the same smart contract. We have given each node a label (a)-(g) to be distinguish between them.

2.2 Arithmetization main ideas

We explain the main concepts of the arithmetization of the zk-EVM. We provide more details on the modules, their structure structures and their interactions with each other.

2.2.1 Execution trace

Each module comes with an associated execution trace. Execution traces can be pictured as 2 dimensional arrays (high level representation). In applications they will be encoded using polynomial commitment (low level representation). The number of rows of each execution trace is proportional to the number of instructions executed in the module; the number of columns is module-dependent and independent of the number of instructions executed by the module in a particular execution of the zk-EVM. Execution traces have anywhere from 10 to 100 columns. We may assume, padding with appropriate default values as needed, that the number of rows of each execution trace is a power of two.

In practice, describing every column of the execution trace can quickly become tedious, so we have chosen to borrow the virtual column/subcolumn paradigm from Cairo - which is a rather convenient and powerful way to describe the arithmetization of the zk-EVM.

Virtual columns/subcolumns: Using our high-level representation of the execution trace, a virtual column is a power of two-periodic subset of rows of a given column of an execution trace. For instance, given an execution trace of size 2^i, and a column C of this execution trace, VC defined by:

$$VC_k = C_{3+2^4\times k}, \forall k \in [\![0, 2^{i-4} [\![$$

is the virtual column containing all rows of C whose row index is conguent to 3 mod 16.

A virtual subcolumn is *a set* of virtual columns obtained from another column or virtual column. Hence, the notion of *virtual subcolumn* supposes a relation dependance between the different subcolumns belonging to the same column.

The main advantage of the notion of virtual column and virtual subcolumn is the fact that one is not bound by the exact size of a given column: one column of an execution trace can be composed of several *virtual columns* stacked together.

2.2.2 Module architecture

The columns of a module's execution trace may be grouped into different subsets - each such subset having a specific role in the execution proof generation. Figure 10 represents such a module architecture:

- The instruction unpacking columns (green box): a group of columns that associates a given EVM opcode to a sequence of flags, subsets of these flags being meaningful to a given module's execution trace. The consistency of the instruction unpacking columns is ensured by a Plookup inclusion proof of the corresponding columns of the *Instruction decoder* module.
- The main execution consistency columns: a group of columns that matches the instruction currently executed by a given module to the corresponding instruction that appears in the main execution trace. The consistency of this set of columns is guaranteed by a Plookup inclusion proof of these module columns in the corresponding main execution trace columns.

- The module specific time execution columns (blue box): are the core columns of each module. The constraints that must be verified during the module execution are applied to these columns.
- The module specific space execution columns (black box): are the columns associated with an underlying memory model specific to the module. These columns are used to check the consistency of the memory model with the time execution part of the module. For instance, the main execution trace space execution columns ensure the consistency of the Read only stack memory; the ones of the child RAM submodule ensure the consistency of the RAM memory with the read/write operation performed on the RAM.
- The range proof columns (orange box): are used for range proof purposes using the *Cairo-style range proofs* that will be described in the next section

(Sub)Columns may belong to more than one group, and some modules may contain more than one group for each category: for instance, in the main execution trace, there are two groups of space execution columns - a first group ensures stack memory consistency; a second group ensures call stack memory consistency. Besides, some modules may not use some groups of columns - the binary module, for instance, doesn't use range proof columns.

To simplify the formalisation, we have added a layer of abstraction, dividing the above groups of columns into two bigger sets of columns: the *time ordered columns* and the *space ordered columns*.

The *time ordered columns* display an internal time coherence: two successive instructions appearing in the time ordered part of a module are executed one after the other. This is needed to maintain a global execution order coherence.

The *space ordered columns* often describe an underlying memory model - these columns are used to enforce spatial coherence (e.g. memory address coherence). Two successive rows of a virtual column belonging to the *space ordered columns* represent the same or two successive memory addresses. These columns are needed to ensure the memory consistencies of the zk-EVM, i.e. that the values read from a specific memory address match the ones previously stored at that same address.

2.2.3 Dealing with inter-contract calls and batches

The inter-contract call process of the EVM involves complex interactions between the two execution environments of the caller and called contracts. We have chosen to handle all the interactions directly in our zk-EVM model by creating a new execution environment every time a inter-contract call is performed. To be able to correctly emulate inter-contract calls, one should be careful that:

- The two execution environments are usually completely independant of one another (i.e. the called smart contract can't access to the memory of the caller contract) as in the case of a CALL instruction.
- Some interactions between the caller and the called contract are permitted in specific cases (e.g. storing the returned data to the memory of the caller).
- Other less common opcodes (e.g. DELEGATECALL) allow the called smart contract to access the caller smart contract's execution environment.
- Two call operations of the same contract involve creating two different execution environments.

We have decided to handle inter-contract calls using a CALL STACK structure, that keeps track of the execution environment of the caller contract - that structure is particularly useful in dealing with chained contract calls as it is needed to retrieve successively the previous execution environments.

The consistency of the CALL STACK memory is ensured in the main execution trace, the same way as the main STACK memory. The CALL STACK memory is a continuous READ ONLY MEMORY that associates to a unique smart contract number, SC_Num, a n-tuple of associated information (such as $SC\_ADDRESS$, $Prev\_Top$, $Prev\_RSP$, $Prev\_PC$, Ret_Offset, Ret_Len, $Caller\_Num$,

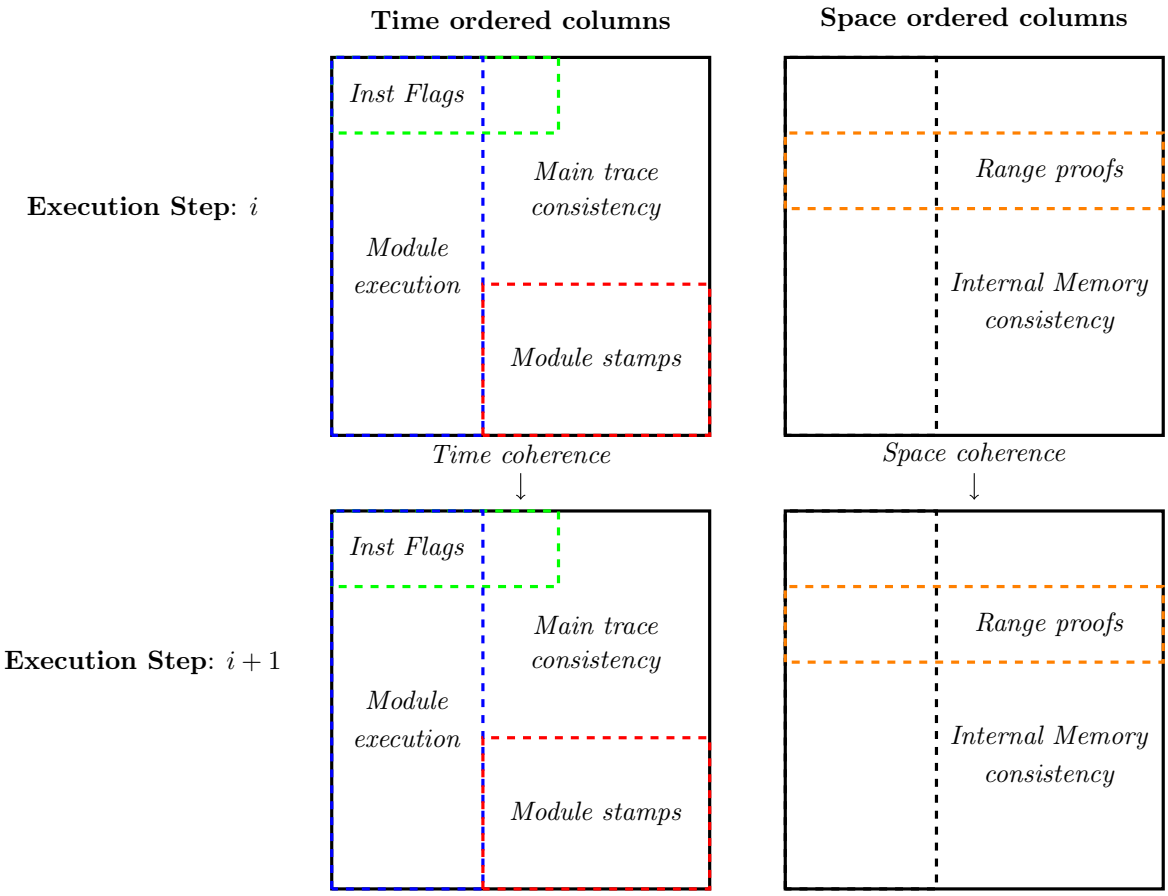
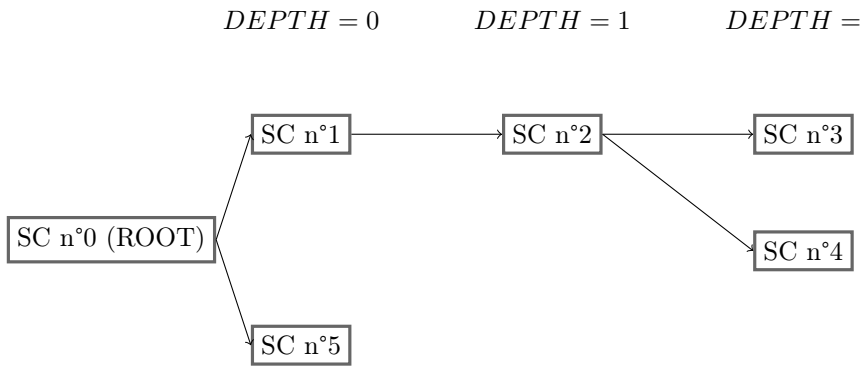


Figure 10: A module architecture illustration

Depth), specific to the set of contracts executed. In a sequence of smart contracts executions (each of which may call other smart contracts) the smart contract number is defined as starting with 0 for the first smart contract being executed and incremented by one every time another smart contract is executed (triggered either by an internal call or when exiting a sequence of internal calls and moving onto a new transaction). As such the smart contract number is distinct from the smart contract's address/identifier and a given contract may have several smart contract numbers if it is called several times in a batch of transactions. A pointer $Next\_SC\_Num$ keeps track of the next CALL STACK address that is not yet attributed (in that respect its role is similar to the WSP pointer for the main STACK memory).

The figure 11 provides a worked out example of the evolution of the call stack associated with the execution of a batch of smart contracts.



(a) A call stack graphical example

Step	Instruction	*SC_Num*	PC	Top	RSP	WSP	*Caller_Num*	*Prev_PC*	*Prev_Top*	*Prev_RSP*	DEPTH
0	*GLOB_INIT*	0	0	0	0	0	0	0	0	0	0
1	PUSH	1	0	2	1	3	0	0	0	0	0
2	PUSH	1	1	3	2	4	0	0	0	0	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
103	CALL	1	92	87	76	88	0	0	0	0	0
104	PUSH	2	0	89	88	90	1	93	87	76	1
105	JUMPI	2	1	90	89	91	1	93	87	76	1
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
208	CALL	2	750	169	157	173	1	93	87	76	1
209	PUSH	3	0	174	173	175	2	751	169	157	2
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
287	RETURN	3	75	254	232	263	2	751	169	157	2
288	PUSH	2	751	169	157	263	1	93	87	76	1
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
354	CALL	2	354	273	269	275	1	93	87	76	1
355	PUSH	4	0	276	275	277	2	354	273	269	2
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
360	RETURN	4	43	285	283	288	2	354	273	269	2
361	SWAP2	2	354	273	269	288	1	93	87	76	1
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
365	RETURN	2	250	299	296	305	1	93	87	76	1
365	MLOAD	1	93	87	76	305	0	0	1	0	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
387	RETURN	1	165	353	344	354	0	0	1	0	0
388	PUSH	5	0	355	354	356	0	0	0	0	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
453	RETURN	5	230	432	415	453	0	0	0	0	0
0	*GLOB_END*	0	1	0	0	0	0	0	0	0	0

(b) An example of a (partial) main execution trace call stack, associated with 11a

Figure 11: An example of the call stack evolution with inter-smart contract calls and batches.

2.3 Putting it all together

The architecture of our zk-EVM is rather complex - modules, instruction decoding, flags, bussing system ... This short section sums up the "gist" of how the modules and their constraint systems relate to the main execution trace and assemble into a zk-EVM. Suppose that we want to prove the execution of a smart contract whose bytecode and storage are stored in our zk-rollup. Here are the main steps of how that works:

- We first load the full contract (EVM) bytecode into the ROM. We do this by building a $Bytecode_{ROM}$ column and a PC_{ROM} column. To prove that the bytecode of the ROM is correctly built, we compute the root hash associated with the ROM bytecode and compare it with the codehash stored in the rollup.
- We then load the sequence instructions effectively carried out during the smart contract execution into the main execution trace. To verify that the opcodes executed are part of the ROM bytecode, we perform a Plookup inclusion proof (table version) of the pair of commitments $(PC_{exec\ trace}, Bytecode_{exec\ trace})$ into the table $(PC_{ROM}, Bytecode_{ROM})$.

Our zk-EVM (i.e. the main execution trace and the various modules) cannot directly interpreted these EVM instructions: we have to translate the $Bytecode_{exec\ trace}$ column into a sequence of module specific flags. This is the role of the instruction decoder, a public, immutable, data table, whose commitment to the values is publicly available for consistency checks - the instruction decoder allows for the flag decomposition of the EVM opcodes into a sequence of flags $(IDFlag_1, \dots, IDFlag_n)$ that can be interpreted by the (main) execution trace and other relevant modules. The problem is then to check the consistency between the instruction flags written into the execution traces (let's note these commitments $ETFlag_1, \dots, ETFlag_n$) and the corresponding instruction flags hard-coded into the instruction decoder. This is solved with a plookup proof of consistency.

- We perform a plookup inclusion proof between the n+1-tuple of commitments ($Bytecode_{exec\ trace}$, $ETFlag_1, \dots, ETFlag_n$) and the associated tuple of commitments from the Instruction decoder $(EVM_{opcode}, IDFlag_1, \dots, IDFlag_n)$

Note that this process (instruction decomposition) is done both in the main execution trace and in the module execution traces (using different sets of flags). The general idea is that the flags that are loaded into the Main Execution Trace are used for the inclusion proofs linking certain columns of the module execution traces to the main execution trace, while the flags loaded into a module are used for the internal constraints that must be satisfied by the columns of the module execution trace.

- Now that the EVM instructions can be interpreted by our zk-EVM, we have to execute the operations and verify that the computations have been well performed. Depending on the value of the instruction flags, a different part of the zk-circuit is going to be activated.
  - For instance, to ensure that the Memory instructions of the current contract (such as $MSTORE, MLOAD, RETURN, CALL, ...$) are transmitted to the RAM module, a flag column $RAM\_Instruction$ is set in the main execution trace. When this flag is set, some constraints of the main execution trace become active - for example, the value of the RAM timestamp has to increase during this clockcycle.

    In the RAM module, other flags will be set to select the internal module behaviour - to distinguish between the $MSTORE$ and $MSTORE8$ instructions, a flag $SIZE$, contained in the instruction decoder and loaded into the RAM, imposes the length of the word stored in the RAM, etc...

    The ram execution trace is augmented with a constant column (equal to 1) which will be included (along with other columns, such as the instruction column, the time stamp column, ...) into the main execution trace into the $RAM\_Instruction$ flag column.

  - Another example may be the JUMP instruction that implies an exceptional modification of the value of the PC - when the $JUMP\_FLAG$ is set, the circuit that performs the normal update the PC is deactivated and the special PC update circuit is activated.
  - Importantly, the constraint equations for the modules **is independent** of the precise sequence of module instruction being executed: there is a *fixed*, *module-dependent* set of constraints ensuring the internal coherence of each module execution trace. The *instruction-specific* behaviour is selected within that set of constraints through the instruction specific flags.

- Every module is linked to the main execution trace by a Plookup proof: the $MODULE\_INST$ flag (in the main execution trace) is set when an instruction that has to be transmitted to a given module, the $MODULE\_STAMP$ keeps track of the current instruction number being executed by the module, and the top stack elements acts as input values for the modules, hence every Plookup inclusion proof has to show that the ($MODULE\_INST$, $INST$, $MODULE\_STAMP$, $INPUT_1$, $INPUT_2$) module trace columns are included within the corresponding main execution trace columns.

3 Tools and notations

This section contains a high-level description of the main tools and techniques used in the arithmetization. The tools we describe below allow us (1) to prove claims about relationships between subcolumns within a given execution trace (2) to prove claims about relationships between subcolumns from different execution traces and thus to connect different sumodules of the zk-evm to each other (3) to express program logic in the arithmetization.

3.1 Intertwining operator $\odot$

Given two column vectors $A = (a_0, a_1, \ldots, a_{N-1})^T$ and $B = (b_0, b_1, \ldots, b_{N-1})^T$ of equal length N define $A \odot B$ to be the column vector of length $2N$ obtained by intertwining the coefficients of A and B like so: $A \odot B = (a_0, b_0, a_1, b_1, \ldots, a_{N-1}, b_{N-1})^T$. Both A and B can be recovered from $A \odot B$ as periodic subsets (i.e. A and B are subcolumns of $A \odot B$). When A and B are viewed as polynomials given by their evaluations (i.e. $a_k = P(k)$ and $b_k = Q(k)$ for polynomials P, Q) we can view $A \odot B$ as a polynomial R, too, via $R(k) = P(k/2) = a_{k/2}$ when k is even, $R(k) = Q((k-1)/2) = b_{(k-1)/2}$ when k is odd.

3.2 Permutation argument for column vectors

We recall a Plonk [9]/Cairo-type [5] permutation argument allowing one to prove that two column vectors A and B are, with overwhelming probability, row permutations of one another. Recall that A and B are row permutations of one another if there exists a permutation σ of the index set $\{0, 1, \ldots, N-1\}$ such that for every row index i, $B_i = A_{\sigma(i)}$. The argument requires the introduction of a single new column $\Pi^{\mathbf{B}}_{\mathbf{A}}$ of length N, the construction of which requires a public random field element $z \in \mathbb{F}$. The values in this column are defined by

$$\left[\Pi^B_A\right]_i = \prod_{0 \leq \ell < i} \frac{z - A_\ell}{z - B_\ell}.$$

3.3 (Row) permutation argument for matrices

We can extend the previous argument to a row permutation argument for matrices. Two matrices $\mathbf{A} = (A_{ij})$ and $\mathbf{B} = (B_{ij})$ (with r columns, N rows and coefficients in the field $\mathbb{F}$) are row-permutations

of one another if there exists a permutation σ of the index set of rows $\{0, 1, \ldots, N-1\}$ such that the i-th row of $\mathbf{B}$ equals the $\sigma(i)$-th row of $\mathbf{A}$ i.e. if

$$\forall i \in \{0, 1, \ldots, N-1\}\ \forall j \in \{1, 2, \ldots, r\}, \quad B_{i,j} = A_{\sigma(i),j}$$

Our arithmetization requires arguments proving that two matrices are row permutations of one another. We can use the row permutation argument for column vectors described above to prove the row permutation property. Indeed, it is enough to prove the row permutation property for a random linear combination $\sum_{j=1}^{r} \theta_j A_j$ and $\sum_{j=1}^{r} \theta_j B_j$ (where the $A_1, \ldots, A_r$ and $B_1, \ldots, B_r$ are the columns of $\mathbf{A}$ and $\mathbf{B}$ respectively.) This argument requires the introduction of a single new column vector $\left[\Pi^{\mathbf{B}}_{\mathbf{A}}\right]$ of length N constructed by accumulating quotients of the coefficients of the random linear combinations of values in A and B:

$$\left[\Pi^{\mathbf{B}}_{\mathbf{A}}\right]_i = \prod_{0 \leq \ell < i} \frac{z - \sum_{k=1}^{r} \theta_k A_{\ell,k}}{z - \sum_{k=1}^{r} \theta_k B_{\ell,k}}.$$

In our applications the matrices A and B are constructed by specifying r column vectors $A_1, A_2, \ldots, A_r$ for A and r column vectors $B_1, B_2, \ldots, B_r$ for B (all of equal length N). These column vectors are obtained as columns/subcolumns of some module's execution trace. We require such permutation arguments for two reasons:

Space-time reordering the stack, memory and storage modules must satisfy two largely independent sets of constraints: time consistency and space consistency. The stack, say, has to be consistent with the temporal execution of the EVM — thus popping the top of the stack for instance must modify the pointer to the top of the stack accordingly at the next clockcycle. On the other hand, performing several pops in a row (or one of the stack pointers being sent a ways back down below the top of the stack) will unearth values in the stack that may have been written there a while back – thus we need to make sure that values once written to the stack are indeed the ones recovered at a later stage through a pointer pointing to the corresponding stack address.

The execution trace of these modules will thus be tasked with proving both types of constraints

Cairo-style range proofs explained below

We use the notation $B = \widetilde{A}$ (resp. $\mathbf{B} = \widetilde{\mathbf{A}}$) to signify that B is a row permutation of the column vector A (resp. $\mathbf{B}$ is a row permutation of the matrix $\mathbf{A}$). Whenever we use this notation it is tacitly assumed that the execution trace is augmented with a column Π witnessing the permutation.

3.4 Plookup

The overall architecture of our zk-evm is modular — separate subunits are specialized in certain subtasks: they obey independent constraint sets and have separate (but not unrelated) execution traces. Thus one module may be tasked with *256* bit arithmetic while another deals with binary operations and yet another deals with memory. Connections between these modules allow modules to delegate computations that fall outside of their specialization to the relevant submodules.

Plookup arguments [10] are the means by which these connections are realized:

Plookup inclusion proofs between subcolumns we can relate the contents of subcolumns of one execution trace to another by means of an inclusion proof. This serves two purposes: (1) it allows the contents of the first column to flow as inputs into the second module where a specialized computation might be verifiable or (2) to use the values of the first column to justify values of the second column when the second module can't verify the associated constraints (e.g. how hashes enter a non hash module).

Plookup inclusion proofs for tables arithmetic operations that are inefficient in field arithmetic (e.g. binary operations) are hardcoded in a plookup table and find their way into execution traces by means of plookup inclusion proofs. Note that column contents can be filtered using flag columns.

3.5 Small-range range proofs

Cairo-style range proofs (or short-range range proofs) apply to show that values in a column are within a relatively short interval $[\![a, b]\!]$ of consecutive integers. Doing this requires reordering the column in ascending order while filling in all missing gaps. We thus require a permutation argument (witnessed by an extra column Π) along with row-wise checks to the effect that the reordered values start at a, end with b, and from one row to the next the values can only remain the same or change by $+1$.

3.6 If-elseif-else logic

Arithmetization of the EVM needs to handle branching execution paths conditional to some condition being satisfied or not. In particular, we need to handle branching of the following sort: (1) given a boolean $b \in \{0, 1\}$, execute either path P_0 or P_1 according to whether $b = 0$ or $b = 1$ (1') more generally, given a constrained variable $c \in \{c_1, \ldots, c_k\}$ for pairwise distinct field elements $c_1, \ldots, c_k \in \mathbb{F}$, execute the corresponding path $P_1, P_2, \ldots, P_k$ depending on the value of c (2) given an arbitrary field element $a \in \mathbb{F}$, execute either path P_0 or $P_{\neq 0}$ according to whether $a = 0$ or $a \neq 0$.

Typically the variables conditioning the branching behaviour (b, c or a above) are taken direcly from a (sub)column of the execution trace. Booleanity is simply checked in a quadratic constraint ($b \in \{0, 1\} \iff b^2 = b$) and the branching behaviour (i.e. the associated constraint) can be simply expressed by means of a second quadratic constraint ($(1-b)P_0+bP_1$), similarly for constrained variables albeit at the cost of higher degree constraints ($\prod_i (c - c_i) = 0$ and $\sum_i L_i(c)P_i$ for adequate Lagrange interpolation polynomials L_i). Branching behaviour of the third kind (i.e. where we need to distinguish $a = 0$ from $a \neq 0$) requires the introduction of a new column which we label $\widehat{1/a}$. This column is constructed in such a way that $a \circ \widehat{1/a}$ is a boolean column (where $\circ$ represents componentwise product of columns) along the appropriate subdomain of $\mathbb{G}$.

3.7 GKR hashing

Our zk-evm uses an algebraic hash function compatible with efficient GKR arithmetization, for instance MiMC hashes. There is thus a separate GKR-verifier module for hashing linked to execution traces of the relevant submodules by means of plookup inclusion proofs. This decision was motivated by the recent results on GKR, published on EthResearch [11]

3.8 Modules timestamps

Timestamps are used internally in our arithmetization to link the different modules together. There are two kinds of timestamps per module: **instruction timestamps** increase when a new instruction starts sending requests to the module bus; **atomic timestamps** for each individual request that is sent to the module bus — a single instruction may send several requests to the bus (e.g. CALL removes the 7 elements on top of the stack in a sequence of 4 elementary steps popping the top two elements of the stack, which requires 4 requests to the Memory module). These timestamps are used to prove the inclusion of every instruction executed by each external module in the main execution trace; but also to prove that each request sent to a child submodule by an associated parent module has been processed.

The $GLOBAL\_INIT$ special opcode, executed before any other instruction, keeps track of the total number of instructions and atomic module calls and initializes the module timestamp values to zero for the next instruction. This number is compared to the sequentially built timestamp values of the $GLOBAL\_END$ special opcode, executed at the last step of the execution trace.

3.9 Constraint propagation

An important point in the sequel is that the conjunction of our read-only-memory model with the use of plookup permutation arguments implicitly imposes a number of constraints on execution traces. In

other words, there are many times where it is enough for us to describe a small number of constraints to implicitly force other constraints to hold for *a priori* unconstrained variables.

For instance, if the memory's Read Stack Pointer (RSP) and Pointer to the top of the stack (Top) are reassigned after the execution of an instruction to historical values of themselves, they will be reassigned coherently along with the corresponding values ($Val(RSP)$ and $Val(Top)$).

3.10 Flags and instruction decoder

Another major component of our zk-evm is the instruction decoder, which is basically a multiplexer that associates an opcode to a sequence of flags that fully determine the behaviour of our zk-evm. The instruction decoder data is a public data that explicitly states constraints between different sets of flags - for instance, when the CALL flag is set (ie its value is equal to one), then the RETURN flag have to remain turned off to kill the constraints associated with the RETURN instruction that aren't used by the CALL instruction. This technique is extremely useful as it allows to drastically remove a considerable number of execution paths that can't be explored because of native constraints imposed in the instruction decoder.

3.11 Dealing with reverted transactions

The REVERT opcode and exceptional halting in the EVM reverts the changes that have been performed in the current transaction - the storage of the currently executed contract has to be reinitialized to its former value, but also the ones that have been executed in nested calls that originated from the current contract.

In this section and the following ones, we'll introduce the two following definitions that are extremely useful for the formalization of REVERT opcodes:

1. *A reverted set* is the set of the children of a reverted transaction. Any reverted transaction belongs to a *reverted set*.

We introduce several tools and concepts to deal with the REVERT opcode and error flags:

- A $Curr\_REV$ virtual column that activates whenever the current transaction is reverted or a child of a REVERTED transaction.
- A REV_FLAG virtual column that activates when a REVERT opcode is executed. When the $REVERT\_FLAG$ is set, the interrupt flag has to be set too.
- An $INTERRUPT\_FLAG$ that is a virtual column that activates whenever an error flag is set or the REVERT opcode executed.
- A REV_STAMP that is unique to every *reverted set*. Note that if a given reverted set, $\mathcal{R}_C$, is included in another reverted set $\mathcal{R}_P$, then the reverted stamp associated with the transactions belonging to $\mathcal{R}_C$ is different from the ones belonging to $\mathcal{R}_P - \mathcal{R}_C$
- A $Parent\_REV\_STAMP$: tracks the revert stamp of the reverted set in which the current reverted set is included. By convention, if the current transaction isn't included in any reverted set, $Parent\_REV\_STAMP = 0$
- A $Next\_REVERT\_STAMP$: which tracks the number of the next revert stamp

To build coherently these flags, one has to detects when one enters or leave a *reverted set* and how the reverted transactions automatically transmit the $Curr\_REVERTED$ flag value in case of an inner-contract call. Basically, these three cases fully determine the $Curr\_REVERTED$ and $REVERT\_STAMP$ updates :

1. If the current instruction is $RETURN$ or $STOP$, the $Curr\_REVERTED$ is not set and the $Curr\_REVERTED$ flag is set, then the parent contract must be reverted too (ie, its $Curr\_REVERTED$ flag is set): because otherwise there is a contradiction between the correct execution of the contract (it has returned a value) and the fact it has been reverted.

2. If the current instruction is CALL and the $Curr\_REVERTED$ flag is set then set the called contract $Curr\_REVERTED$ flag.

3. If the current instruction is neither CALL, RETURN nor STOP, then verify that the $Curr\_REVERTED$ flag of the next instruction is the same as the current instruction.

4. If the $INTERRUPT\_FLAG$ is set, check that the $Curr\_REVERTED$ flag is set. Set the current $REVERT\_STAMP$ to the $Next\_REVERT\_STAMP$ and increment the $Next\_REVERT\_STAMP$

Having defined the tools for REVERTED updates, one can now start to use them to update the STORAGE in compliance with REVERT statements.

3.12 Error flags

According to the Ethereum yellowpaper, here are the cases that trigger an exceptional halt in the EVM:

- Insufficient gas.
- Invalid instruction.
- JUMP/JUMPI to an invalid destination.
- Insufficient stack items.
- Stack overflow (stack size exceeds 1024).
- Copying RETURNDATA from non existing positions
- State modification during a static call.

Most of these cases can be simply dealt with using the tools and techniques described above. For instance:

- To deal with the insufficient gas error flag, one can perform a word comparison between the gas value and its decrement for each step, raising an error flag when the gas value is insufficient for the execution to pursue.
- The stack related errors are easily dealt with using stack size trackers and verifying that its value is always comprised in [0,1023], setting an error flag if that's not the case anymore.
- RETURNDATA related flags are dealt with by tracking RETURNDATASIZE and setting the associated error flag in case of overflow.
- State modification during a static call is dealt by raising a flag when calling a forbidden instruction during a static call.
- To check the validity of the destination of JUMP/JUMPI, one may impose a range check for the maximum value of the PC in the current program, and add a $PUSH\_ARG$ flag to the ROM in the case that one jumps to a PUSH argument in the bytecode. This case is considered to be an illicit jump destination by the Ethereum yellowpaper.

- The instruction validity check is a bit more complex. Indeed, this amounts to verify that the instruction opcode does not belong to the set of 80 EVM opcodes (which is not a continuous set of values). Normally, this check is performed using Plookup, which can be hardly adapted for a try/catch logic. A solution to perform instruction validity checks is to pad the opcode gaps with INVALID instructions, and then impose an opcode maximum value. Then, for every instruction one has to perform a range proof to verify that the opcode falls in the authorized set of values. Inside that range of values, Plookup will distinguish between licit and INVALID opcodes inside the instruction decoder.

3.13 Fees/Gas Costs

After the error flags, dealing with the fees (or gas costs) is another tricky part of the zk-EVM, although necessary to allow for backward-compatibility in smart contract execution.

According to the Ethereum yellowpaper, there are three types of gas costs that may occur in a smart contract execution:

1. A constant fee intrinsic to the operation currently executed. This is the case for most of the common instructions of the EVM (ADD, SWAP, DUP, ...)
2. A memory extension fee that depends on the current size of the memory RAM. This extension fee depends on the number of *EVM words* one have stored in the RAM so far and is updated dynamically during the execution of the program. There is an equivalent of the RAM extension fee for the storage - the first access of a given address in storage will be way more expensive than the subsequent ones, and setting a storage value to zero induces a gas reimbursement.
3. A last category of gas fee is related to smart contract interactions. This fee is payed for the execution of instructions like $*CALL$ or $CREATE*$. These fees are called *dynamical fees* and are the most complex category of fees to adapt to the zk-EVM framework.

The current fee value will be tracked using a *gas* virtual column. The current size of memory is tracked in a MEM_SIZE variable that is updated in the child memory module as memory segments are accessed. This MEM_SIZE variable allows us to perform this dynamic fee update during the program execution.

4 Word comparison module

The word comparison module deals with the four word comparison instructions

- LT
- GT
- SLT
- SGT

which respectively return, for $a := Input\_1$, $b := Input\_1$, the boolean $a \leq b$, $a \geq b$, $a < b$ and $a > b$.

4.1 Trace Columns

This module is comprised of the following columns:

- *WCTimeStamp* increases by one with every instruction and remains constant during the execution of a given word comparison instruction.

- *Inst* column of instructions
- *SwitchFlag*
- *EqFlag*
- *Input_ 1* The first word, remains constant during the instruction execution.
- *Input_ 2* The second word, remains constant during the instruction execution.
- *Comp* This column is constant along a given time stamp and returns the result of the comparison *Input_ 1* < *Input_ 2*, i.e. *Comp* computes the value of *SLT*.
- *Prefix_ 1* constructs the sequence of prefixes of *Input_ 1*, Byte by Byte
- *Prefix_ 2* same for *Input_ 2*
- *B_ 1* the sequence of Bytes making up *Input_ 1* starting from its highest Byte to its lowest
- *B_ 2* the sequence of Bytes making up *Input_ 2* starting from its highest Byte to its lowest
- *ByteComp* this column contains the result of the comparison *B_ 1* < *B_ 2*.
- *Decided* this column may switch from 0 to 1 as soon as one can know the result of *Comp*

The *SwitchFlag* and *EqFlag* encode the four comparison operations:

	$SwitchFlag = 0$	$SwitchFlag = 1$
$EqFlag = 0$	SLT	SGT
$EqFlag = 1$	LT	GT

4.2 Trace constraints

The result column computes the result of the instruction using the equivalences

$$\begin{cases} LT: & a \leq b \iff (a < b) \text{ OR } (a = b) \\ SGT: & a > b \iff \text{NOT } \Big((a < b) \text{ OR } (a = b)\Big) \\ GT: & a \geq b \iff \text{NOT } (a < b) \end{cases}$$

Thus it is enough for us to compute the comparison $a < b$ and this is the purpose of the *Comp* column which always computes the boolean *Input_ 1* < *Input_ 2*

1. The column *Inst* along with its flag columns *SwitchFlag*, *EqFlag* are Plookup verified against the instruction decoder.
2. The columns *B_ 1*, *B_ 2* and *BComp* are Plookup verified against a lookup-table that stores all 256×256 Byte comparisons.
3. $WC\_\ TimeStamp_0$ is initialized to the total number of calls to the Word Count module. This condition is required to ensure that the inclusion proofs don't "miss out" on requests to the word count module.
4. for $i = 1$ $WC\_\ TimeStamp_1 = 0$
5. for $i \geq 1$, $WC\_\ TimeStamp_{i+1} = WC\_\ TimeStamp_i$ or $WC\_\ TimeStamp_{i+1} = WC\_\ TimeStamp_i + 1$, i.e. $(WC\_\ TimeStamp_{i+1} - WC\_\ TimeStamp_i)(WC\_\ TimeStamp_{i+1} - WC\_\ TimeStamp_i - 1) = 0$.

6. for $1 \leq i$, $WC\_\ TimeStamp_{i+32} = WC\_\ TimeStamp_i$ or $WC\_\ TimeStamp_{i+32} = WC\_\ TimeStamp_i + 1$ since words have a fixed length of 32 bytes and 32 lines are enough for word comparison.
7. *Res*, *Comp* and *Decided* are binary, thus for all i: $Res_i(Res_i - 1) = 0$, $Comp_i(Comp_i - 1) = 0$ and $Decided_i(Decided_i - 1) = 0$

The other variables are updated differently according to whether $WC\_\ TimeStamp_i = WC\_\ TimeStamp_{i-1}$ or not:

1. If - $WC\_\ TimeStamp_i = WC\_\ TimeStamp_{i-1} + 1$:
   (a) $(B\_\ 1)_i = (Prefix\_\ 1)_i$ and $(B\_\ 2)_i = (Prefix\_\ 2)_i$,
   (b) $(Prefix\_\ 1)_{i-1} = (Input\_\ 1)_{i-1}$ and $(Prefix\_\ 2)_{i-1} = (Input\_\ 2)_{i-1}$, i.e. we expect the prefixes to equal the inputs at the end of a word comparison
   (c) If - $(B\_1)_i = (B\_2)_i$ then $Decided_{i-1} = 0$
   (d) Else If - $(B\_1)_i \neq (B\_2)_i$ then $Decided_i = 1$ and $Comp_i = BComp_i$.
   (e) If - $(Input\_\ 1)_i = (Input\_\ 2)_i$ then $Comp_i = 0$
2. Else If - $WC\_\ TimeStamp_i = WC\_\ TimeStamp_{i-1}$ then
   (a) $(Prefix\_\ 1)_i = 256 \cdot (Prefix\_\ 1)_{i-1} + (B\_\ 1)_i$ and $(Prefix\_\ 2)_i = 256 \cdot (Prefix\_\ 2)_{i-1} + (B\_\ 2)_i$,
   (b) the inputs, the instruction and its flags, and the result of the comparison $(Input\_\ 1 < Input\_\ 2)$ remain constant throughout the execution of a single word comparison:

$$\begin{cases} (Input\_\ 1)_{i-1} = (Input\_\ 1)_i \\ (Input\_\ 2)_{i-1} = (Input\_\ 2)_i \\ Inst_{i-1} = Inst_i \\ SwitchFlag_{i-1} = SwitchFlag_i \\ EqFlag_{i-1} = EqFlag_i \\ Comp_{i-1} = Comp_i \\ Res_{i-1} = Res_i \end{cases}$$

   (c) i. If - $Decided_{i-1} = 0$:
      A. If - $(B\_1)_i = (B\_2)_i$ then $Decided_i = 0$
      B. Else If - $(B\_1)_i \neq (B\_2)_i$ then $Decided_i = 1$ and $Comp_i = BComp_i$
   ii. Else If - $Decided_{i-1} = 1$: then $Decided_i = Decided_{i-1}$.

We now specify the computation of *Res*. We have already specified that *Res* remains constant during the execution of an instruction.

1. If - $SwitchFlag_i = 0$ AND $EqFlag_i = 0$: then $Res_i = Comp_i$
2. If - $SwitchFlag_i = 0$ AND $EqFlag_i = 1$: then $Res_i = 1$ iff $Comp_i = 1$ or $(Input\_\ 1)_i = (Input\_\ 2)_i$,
3. If - $SwitchFlag_i = 1$ AND $EqFlag_i = 0$: then $Res_i = 0$ iff $Comp_i = 1$ or $(Input\_\ 1)_i = (Input\_\ 2)_i$.
4. If - $SwitchFlag_i = 1$ AND $EqFlag_i = 1$: then $Res_i = 1 - Comp_i$

5 Constraint set for the Parent Memory module

5.1 Instructions treated

- CALL
- RETURN
- MSTORE
- MSTORE8
- MLOAD
- CALLDATALOAD

Instructions not yet implemented:

- CALLDATACOPY
- RETURNDATACOPY
- RETURNDATASize
- CODECOPY
- EXTCODECOPY
- DELEGUATECALL
- STATICALL
- CALLCODE

5.2 Trace columns

5.2.1 Stack trace inclusion columns:

- $Stack\_Inst$: the instruction to be executed.
- $RAM\_TIMESTAMP$
- $Input_1$: first input of the instruction.
- $Input_2$: second input (if any)
- $Result$: result of the instruction (element to be added on the stack - if any)
- END_FLAG^0
- $INVALID\_INSTRUCTION$

5.2.2 Instruction unpacking

- $IOPC\_FLAG$: if set, selects interactive memory opcodes like $CALL$, $RETURN$. Otherwise, selects non interactive memory opcodes such as $MLOAD$, $MSTORE$, $MSTORE8$, $CALLDATALOAD$.
- $MOPC\_FLAG$: Bit selection flag for non interactive memory opcodes. If set, selects $MSTORE$, $MSTORE8$. Otherwise, selects $MLOAD$ and $CALLDATALOAD$ opcodes.
- $Size\_FLAG$: set to the Size of the word to READ/STORE in memory.
- RET_FLAG: special case of the RETURN instruction
- $CALL\_FLAG$: special case of the CALL instruction
- $CALLDATA\_FLAG$: special case that makes the instruction linked memory offsets to be written as a calldata (for the CALL instruction)
- $INIT\_FLAG$: initialize the RAM after a CALL instruction

5.2.3 Parent module instruction decomposition:

- $Inst\_BInt\_Offset$: instruction's internal begin offset
- $Inst\_BWd\_Offset$: instruction's word begin offset
- $Inst\_EInt\_Offset$: instruction's internal end offset
- $Inst\_EWd\_Offset$: instruction's word end offset
- $Curr\_BInt\_Offset$: current begin internal offset
- $Curr\_EInt\_Offset$: current end internal offset
- $Curr\_BWd\_Offset$: current beginning word offset
- $Curr\_EWd\_Offset$: current ending word offset (constrained to be either $Curr\_BWd\_Offset$ or $Curr\_BWd\_Offset + 1$).
- $Curr\_Wd\_Val$: value of the current word read/written
- Aux_BWd_Offset, Aux_BInt_Offset, Aux_EWd_Offset, Aux_EInt_Offset, $Curr\_Aux\_Wd\_Offset$, $Curr\_Aux\_BInt\_Offset$, $Curr\_Aux\_EInt\_Offset$: auxiliary parameters for the smart contract call.

5.2.4 Child module inclusion columns (word multiple memory):

- $CRAM\_Inst$: the instruction to be executed by the child module (can be LOAD ($CRAM\_Inst = 0$) or STORE ($CRAM\_Inst = 1$)).
- $CRAM\_STAMP$: timestamp of the child module.
- $CRAM\_BWd\_Offset$: the word multiple address to start writing to / reading from.
- $CRAM\_EWd\_Offset$: the word multiple address to end writing to / reading from.
- $CRAM\_BInt\_Offset$: contains the internal beginning address of the word currently read/written.
- $CRAM\_EInt\_Offset$: contains the internal ending address of the word currently read/written.
- $Val(CRAM\_Inst)$: value of the RAM instruction.

- $CRAM\_SC\_Num$: the SC memory to be loaded/written.
- $CRAM\_CALLDATA\_FLAG$: a flag that determines whether the instruction acts on the RAM or the CALLDATA

5.2.5 Call stack/SC batching:

- SC_Num: current smart contract number in the batch.
- $Caller\_Num$: caller smart contract number.
- $Ret\_Offset(SC\_Num), Ret\_Len(SC\_Num)$: execution stack inclusion

5.2.6 Memory size, gas costs

- $Curr\_Mem\_Size$: the current memory size in EVM words
- Max_Mem_Size: maximum memory size
- $Curr\_CALLDATA\_Size$: the current CALLDATA Size in EVM words
- $Mem\_INCREASE\_FLAG$: a flag that indicates whether the memory size is increased
- $\Delta Curr\_Mem\_Size$: the difference between the maximum and the current memory sizes.
- Gas_RAM: dynamic RAM gas cost paid by the instruction.
- $Cost\_RAM$: current value of the memory cost function
- $Carry\_Gas\_RAM$: used as a carry for the Gas_RAM calculation
- $Curr\_Mem^2/512$: the floor value of $Curr\_Mem^2$ divided by 512
- $*COPY\_Cost$: cost associated with external data copy operations.

5.2.7 Auxiliaries

- AUX_TRANS

5.3 Opcodes constraints

5.3.1 MSTORE/MLOAD specific initialization.

If - $\boldsymbol{(IOPC\_FLAG)_i = 0}$ (MSTORE, MSTORE8, MLOAD, CALLDATALOAD instructions) then

1. Unpack the first input (memory word offset) and set the instruction specific beginning word and internal offsets:

$$(Input_1)_i = 32 * Inst\_BWord\_Offset_i + Inst\_BInt\_Offset_i$$

2. Compute the memory word ending word/internal offset using the $Size$ variable (which is equal to 31 for MSTORE/MLOAD and is equal to 1 for MSTORE8)

$$(Input_1)_i + Size_i = 32 * Inst\_EWord\_Offset_i + Inst\_EInt\_Offset_i$$

3. If - $\boldsymbol{(MOPC\_FLAG)_i = 1}$ (MSTORE, MSTORE8 instruction) then:

(a) Unpack the second input (word to write to memory) and set the instruction specific ending word and internal offsets:

$$(Input_2)_i = Val(CRAM\_Inst)_i$$

(b) Set the Child RAM instruction to store the value of the second input

$$CRAM\_Inst_i = 1$$

4. Else If - $\boldsymbol{(MOPC\_FLAG)_i = 0}$ (MLOAD, CALLDATALOAD instruction) then:

(a) Unpack the result input (word to read from the memory) and set the instruction specific ending word and internal offsets:

$$(Result)_i = Val(CRAM\_Inst)_i$$

(b) Set the Child RAM instruction to read the value of the result input

$$CRAM\_Inst_i = 0$$

5.3.2 INIT instruction initialization

- If - $INIT\_FLAG_i = 1$, then:

1. Initialize the begin word and interior offsets:

$$32 * Inst\_BWord\_Offset_i + Inst\_BInt\_Offset_i = 0$$

2. Initialize the end word and interior offsets:

$$32 * Inst\_EWord\_Offset_i + Inst\_EInt\_Offset_i = 32 * Max\_Mem\_Size_i$$

3. Write to the memory:

$$CRAM\_Inst_i = 1$$

4. Set the initial value to zero:

$$Val(CRAM\_Inst)_i = 0$$

5.3.3 RETURN specific initialization

If - $\boldsymbol{RET\_FLAG = 1}$:

1. Unpack the first input (instruction beginning offset)

$$(Input_1)_i = 32 * Inst\_BWord\_Offset_i + Inst\_BInt\_Offset_i$$

2. Unpack the second input (length of the word RETURNED)

$$(Input_1)_i + (Input_2)_i = 32 * Inst\_EWord\_Offset_i + Inst\_EInt\_Offset_i$$

3. Initialize the auxiliary begin internal and word offsets with the return offset:

$$Ret\_Offset(SC\_Num)_{i+1} = 32 * Aux\_BWd\_Offset_{i+1} + Aux\_BInt\_Offset_{i+1}$$

4. Initialize the auxiliary end internal and word offsets with the return offset:

$$Ret\_Offset(SC\_Num)_{i+1} + Ret\_Len(SC\_Num)_{i+1} = 32 * Aux\_EWd\_Offset_{i+1} + Aux\_EInt\_Offset_{i+1}$$

5. If - $\boldsymbol{RAM\_STAMP_i = RAM\_STAMP_{i-1} + 1}$, Start by a reading operation

$$CRAM\_Inst = 0$$

5.3.4 CALL specific initialization

If - $\boldsymbol{CALL\_FLAG_i = 1}$

1. If - it is the first CALL step: $\boldsymbol{RAM\_STAMP_i = RAM\_STAMP_{i-1} + 1}$: initialize the CALLDATA parameters:

   (a) Unpack the memory data offset and length as instruction decomposition:

   $$\begin{cases} (Input_2)_{i+2} = 32 * Inst\_BWord\_Offset_i + Inst\_BInt\_Offset_i \\ (Input_2)_{i+2} + (Input_1)_{i+3} = 32 * Inst\_EWord\_Offset_i + Inst\_EInt\_Offset_i \end{cases}$$

   (b) Initialize the return data offset as the auxiliary memory offsets:

   $$\begin{cases} Aux\_BWd\_Offset_i = 0 \\ Aux\_BInt\_Offset_i = 0 \\ (Input_1)_{i+3} = 32 * Aux\_EWord\_Offset_i + Aux\_EInt\_Offset_i \end{cases}$$

2. If - it is the last CALL step: $\boldsymbol{RAM\_STAMP_{i+1} = RAM\_STAMP_i + 1}$: initialize the new CALLDATASize (done in the main execution trace constraints).

5.3.5 General value constraints.

1. If - $\boldsymbol{RAM\_STAMP_i = RAM\_STAMP_{i-1} + 1}$:

   (a) Initialize the current word and begin interior offset

   $$\begin{cases} Curr\_BWd\_Offset_i = Inst\_BWd\_Offset_i \\ Curr\_BInt\_Offset_i = Inst\_BInt\_Offset_i \end{cases}$$

5.3.6 Memory size update

The memory size is updated when an instruction starts to be executed. Then, the memory size is kept constant throughout the execution of the instruction.

1. If - $\boldsymbol{RAM\_STAMP_i = RAM\_STAMP_{i-1} + 1}$ AND $\boldsymbol{END\_FLAG^0 = 0}$:

   (a) If - $\boldsymbol{Mem\_Increase_i = 0}$:

   i. Verify that the memory size is greater than the current max offset:

   $$\Delta Curr\_Mem\_Size_i = Curr\_Mem\_Size_i - Inst\_EWd\_Offset_i$$

   ii. Do not increase the memory size:

   $$Curr\_Mem\_Size_i = Curr\_Mem\_Size_{i-1}$$

   (b) Else If - $\boldsymbol{Mem\_Increase_i = 1}$:

   i. Verify that the memory size is lower than the current max offset:

   $$\Delta Curr\_Mem\_Size_i = Inst\_EWd\_Offset_i - Curr\_Mem\_Size_i$$

   ii. Increase the memory size:

   $$Curr\_Mem\_Size_i = Inst\_EWd\_Offset_i$$

2. Else If - $\boldsymbol{RAM\_STAMP_i = RAM\_STAMP_{i-1}}$ AND $\boldsymbol{END\_FLAG^0 = 0}$ then keep the values constant:

   $$\begin{cases} \Delta Curr\_Mem\_Size_i = \Delta Curr\_Mem\_Size_{i-1} \\ Curr\_Mem\_Size_i = Curr\_Mem\_Size_{i-1} \end{cases}$$

3. Else If - $\boldsymbol{END\_FLAG^0 = 1}$: dealt with in the main execution trace.

5.3.7 RAM gas cost computation:

- General cost computation:

$$\begin{cases} Cost\_RAM_i = *COPY\_Cost_i + Curr\_Mem^2/512_i + 3 * Curr\_Mem\_Size_i \\ Gas\_RAM_i = Cost\_RAM_i - Cost\_RAM_{i-1} \\ Curr\_Mem\_Size_i^2 = Curr\_Mem\_Size^2/512_i * 512 + Carry\_Gas\_RAM \end{cases}$$

- If - $CALLDATA\_COPY\_FLAG = 1$:

$$*COPY\_Cost_i = 3 * CALLDATA\_Size_i$$

- Else If - $CALLDATA\_COPY\_FLAG = 0$:

$$*COPY\_Cost_i = 0$$

5.3.8 Child RAM interior offsets

These conditions allows to adjust the child RAM interior offsets depending on the current word offset.

1. Always increase the Child RAM stamp

$$CRAM\_STAMP_{i+1} = CRAM\_STAMP_i + 1$$

2. Impose the values of $Curr\_[\emptyset, B, E]\_[Wd, Int]\_Offset$
   (a) Always impose that $Curr\_BWd\_Offset_i = Curr\_EWd\_Offset_i$
   (b) If - $\boldsymbol{Curr\_EWd\_Offset_i = Inst\_EWd\_Offset_i}$, then

$$Curr\_EInt\_Offset_i = Inst\_EInt\_Offset_i$$

   (c) Else If - $\boldsymbol{Curr\_EWd\_Offset_i \neq Inst\_EWd\_Offset_i}$, then

$$Curr\_EInt\_Offset_i = 31$$

   (d) If - $\boldsymbol{Curr\_BWd\_Offset_i = Inst\_BWd\_Offset_i}$, then

$$Curr\_BInt\_Offset_i = Inst\_BInt\_Offset_i$$

   (e) Else If - $\boldsymbol{Curr\_BWd\_Offset_i \neq Inst\_BWd\_Offset_i}$ then

$$Curr\_BInt\_Offset_i = 0$$

3. Impose the values of $Curr\_Aux\_[\emptyset, B, E]\_[Wd, Int]\_Offset$. The values of Aux_Offset are constrained depending on the values of $Curr\_Offset$: we can't impose the exact values of $Curr\_Aux$, the same way as $Curr\_Inst$, because the respective internal offsets may be different.
   (a) If - $\boldsymbol{RAM\_STAMP_i = RAM\_STAMP_{i-1} + 1}$ (initialization performed in the general value constraints) then
      i. Initialize the current auxiliary offsets

$$\begin{cases} Curr\_Aux\_BWd\_Offset_i = Aux\_BWd\_Offset_i \\ Curr\_Aux\_BInt\_Offset_i = Aux\_BInt\_Offset_i \end{cases}$$

ii. Initialize the current auxiliary ending offsets

$$32 * Curr\_Aux\_EWd\_Offset_i + Curr\_Aux\_EInt\_Offset_i = \\ 32 * Curr\_Aux\_BWd\_Offset_i + Curr\_Aux\_BInt\_Offset_i + \\ (Curr\_EInt\_Offset_i - Curr\_BInt\_Offset_i)$$

(b) If - $\boldsymbol{Curr\_BWd\_Offset_i \neq Curr\_BWd\_Offset_{i-1}}$ AND $\boldsymbol{RAM\_STAMP_{i-1} = RAM\_STAMP_i}$ (the current instruction offsets changed)

i. Setting the beginning offset

$$32 * Curr\_Aux\_BWd\_Offset_i + Curr\_Aux\_BInt\_Offset_i = \\ 32 * Curr\_Aux\_EWd\_Offset_{i-1} + Curr\_Aux\_BInt\_Offset_{i-1} + 1$$

ii. Setting the ending offset

$$32 * Curr\_Aux\_EWd\_Offset_i + Curr\_Aux\_EInt\_Offset_i = \\ 32 * Curr\_Aux\_BWd\_Offset_i + Curr\_Aux\_BInt\_Offset_i + \\ (Curr\_EInt\_Offset_i - Curr\_BInt\_Offset_i)$$

(c) Else If - $\boldsymbol{Curr\_BWd\_Offset_i = Curr\_BWd\_Offset_{i-1}}$ AND $\boldsymbol{RAM\_STAMP_{i-1} = RAM\_STAMP_i}$ (the current instruction offsets did not change):

i. Keep every offset constant:

$$\begin{cases} Curr\_Aux\_BWd\_Offset_i = Curr\_Aux\_BWd\_Offset_{i-1} \\ Curr\_Aux\_EWd\_Offset_i = Curr\_Aux\_EWd\_Offset_{i-1} \\ Curr\_Aux\_BInt\_Offset_i = Curr\_Aux\_BInt\_Offset_{i-1} \\ Curr\_Aux\_EInt\_Offset_i = Curr\_Aux\_EInt\_Offset_{i-1} \end{cases}$$

5.3.9 Value consistency/constraints:

1. If - $\boldsymbol{CALLDATA\_FLAG = 0}$: do not set the CALLDATA flag:

$$CRAM\_CALLDATA_i = 0$$

2. Else If - $\boldsymbol{CALLDATA\_FLAG = 1}$

(a) If - $\boldsymbol{IOPC\_FLAG = 1}$

i. If - $\boldsymbol{CRAM\_Inst = 0}$: $CRAM\_CALLDATA\_FLAG_i = 0$

ii. If - $\boldsymbol{CRAM\_Inst = 1}$: $CRAM\_CALLDATA\_FLAG_i = 1$

(b) Else If - $\boldsymbol{IOPC\_FLAG = 0}$:

$$CRAM\_CALLDATA\_FLAG_i = 1$$

3. If - $\boldsymbol{MOPC\_FLAG_0 = 1}$ (MSTORE/MLOAD),

(a) Set the current Child RAM smart contract number to the executed SC number

$$CRAM\_SC\_Num_i = SC\_Num_i$$

(b) Set the current child RAM begin/end interior offset to the current begin/end interior offset

$$CRAM\_BInt\_Offset_i = Curr\_BInt\_Offset_i$$

$$CRAM\_EInt\_Offset_i = Curr\_EInt\_Offset_i$$

(c) Same thing for the word offsets

4. Else If - $\boldsymbol{IOPC\_FLAG_i = 1}$

   (a) If - $\boldsymbol{CRAM\_Inst = 0}$ (READ from the current smart contract) then

      i. If - $\boldsymbol{RETURN\_FLAG = 1}$ OR $\boldsymbol{CALL\_FLAG = 1}$

         A. Set the child RAM smart contract number to be the current smart contract number

$$CRAM\_SC\_Num_i = SC\_Num_i$$

      ii. Set the child RAM interior offsets to be the current interior offsets.

$$CRAM\_BInt\_Offset_i = Curr\_BInt\_Offset_i$$
$$CRAM\_EInt\_Offset_i = Curr\_EInt\_Offset_i$$

      iii. Same thing for the word offsets

   (b) If - $\boldsymbol{CRAM\_Inst = 1}$ (WRITE to the caller contract) then

      i. If - $\boldsymbol{RETURN\_FLAG = 1}$

         A. Set the child RAM smart contract number to be the caller smart contract number (return the data to the memory of the called contract)

$$CRAM\_SC\_Num_i = Caller\_Num_i$$

      ii. If - $\boldsymbol{CALL\_FLAG = 1}$

         A. Set the child RAM smart contract number to be the next smart contract number (write to the calldata of the next SC)

$$CRAM\_SC\_Num_i = Next\_SC\_Num_i$$

      iii. Set the child RAM interior offsets to be the return interior offsets.

$$CRAM\_BInt\_Offset_i = Curr\_Aux\_BInt\_Offset_i$$
$$CRAM\_EInt\_Offset_i = Curr\_Aux\_EInt\_Offset_i$$

      iv. Same thing for the word offsets

      v. Impose the previous value READ to be the new value stored:

$$Val(CRAM\_Inst)_i = Val(CRAM\_Inst)_{i-1}$$

5.3.10 Transition constraints:

1. If - $\boldsymbol{RET\_FLAG_i = 1}$ AND $\boldsymbol{RAM\_STAMP_{i+2} = RAM\_STAMP_i}$ AND $\boldsymbol{CRAM\_Inst_i = 0}$ (READ and RETURN), then:

   (a) The next instruction is WRITE back to the caller memory and the second next is READ again.

$$CRAM\_Inst_{i+1} = 1$$
$$CRAM\_Inst_{i+2} = 0$$

   (b) One has to increase the word offset once the writing and reading operations are done:

$$Curr\_Wd\_Offset_{i+2} = Curr\_Wd\_Offset_i + 1$$

2. If - $\boldsymbol{MOPC\_FLAG_i = 1}$, AND $\boldsymbol{RAM\_STAMP_{i+1} = RAM\_STAMP_i}$ increase the word offset.

$$Curr\_Word\_Offset_{i+1} = Curr\_Word\_Offset_i + 1$$

5.3.11 End instruction constraints:

If - $\boldsymbol{End\_Wd\_Offset_i = Curr\_Wd\_Offset_i}$, increase the RAM STAMP

$$RAM\_STAMP_{i+1} = RAM\_STAMP_i + 1$$

Else If - $\boldsymbol{End\_Wd\_Offset_i \neq Curr\_Wd\_Offset_i}$, keep the same RAM STAMP

$$RAM\_STAMP_{i+1} = RAM\_STAMP_i$$

6 Constraint set for the Child Memory module.

6.1 Role in the architecture

The child memory module has been introduced to take a part of the complexity of the parent memory module to simplify the constraint system of this complex part of the architecture. The child memory module is designed to perform the following operations:

- Initialize all the memory cells that are going to be read/written from.
- Store to/load from the memory at most 32 bytes at two consecutive word addresses, in/from any RAM/calldata that are loaded in the ROM
- Verify the integrity of the RAM/calldata of the smart contracts loaded in the ROM

We will then decompose this section into two parts: on the one hand we will describe the constraint set associated with he STORE/LOAD operations of the child RAM, on the other hand we will describe how to check the memory integrity inside the RAM module.

6.2 Constraint columns for child RAM STORE/LOAD operations

6.2.1 Parent module inclusion columns:

These trace columns are used for the communication between the child and the parent RAM module.

- $CRAM\_Inst$: the instruction to be executed by the child module (can be LOAD ($CRAM\_Inst = 0$) or STORE ($CRAM\_Inst = 1$)).
- $INIT\_OPERATION$: special operation that initializes the RAM of a given SC.
- $CRAM\_STAMP$: timestamp of the child module.
- $CRAM\_BWd\_Offset$: the word multiple address to start writing to / reading from.
- $CRAM\_EWd\_Offset$: the word multiple address to end writing to / reading from.
- $CRAM\_BInt\_Offset$: contains the internal beginning address of the word currently read/written.
- $CRAM\_EInt\_Offset$: contains the internal ending address of the word currently read/written.
- $Val(CRAM\_Inst)$: value of the RAM instruction.
- $CRAM\_SC\_Num$: the SC memory to be loaded/written.
- $CRAM\_CALLDATA\_FLAG$: a flag that determines whether the instruction acts on the RAM or the CALLDATA
- $INVALID\_INSTRUCTION$: the invalid instruction flag, turned on when a RAM INIT operation is performed at an invalid position. .

To simplify slightly the notations in the following subsections describing the constraint set of the child RAM memory module, we will drop the $CRAM$ prefix in the virtual columns names.

6.2.2 CRAM execution specific columns

These columns are used within the child RAM module to perform the CRAM specific STORE/LOAD operations :

- $Curr\_Wd\_Offset$: the current word offset to STORE to / READ from
- $Curr\_Int\_Offset$: the current interior offset being READ/STORED
- $Curr\_Val$: the current value to store/read in the RAM
- $Curr\_Byte$: the current byte being stored/read in the RAM
- $Curr\_Carry$: the remaining bytes to store/read in the RAM
- $Curr\_Byte\_Num$: auxiliary variable that keeps track of the current byte number.
- $Inst\_Range\_Flag$: a flag that is set if

$$2^8 Curr\_Wd\_Offset + Curr\_Int\_Offset \in [2^8 * BWd\_Offset + BInt\_Offset, 2^8 * EWd\_Offset + EInt\_Offset]$$

- $Inst\_Byte$: the instruction byte that is currently being read
- $Inst\_Carry$: the remaining instruction bytes to read/store.
- $Prev\_Val$: the previous value that was stored in the RAM at $(Curr\_Wd\_Offset, Curr\_Int\_Offset)$
- $Prev\_Carry$: carry for the previous value that was stored in the RAM at $(Curr\_Wd\_Offset, Curr\_Int\_Offset)$
- $Prev\_Byte$: the previous byte being stored in the RAM at $(Curr\_Wd\_Offset, Curr\_Int\_Offset)$

6.3 Constraint set for the execution

One has first to distinguish between two cases: the current instruction is an INIT operation, the current instruction is not an INIT operation.

The INIT operation is quite tricky as it must initialize all the memory cells that are going to be accessed by the current transaction. A way to do that is to store a zero value for every word multiple cell between zero and MAX_Mem_Size.

- If - $\boldsymbol{CRAM\_STAMP_i = CRAM\_STAMP_{i-1} + 1}$:
  1. Initialise the $Curr\_Wd\_Offset$:
  $$Curr\_Wd\_Offset_i = BWd\_Offset_i$$
  2. Initialise the $Curr\_Int\_Offset$:
  $$Curr\_Int\_Offset_i = BInt\_Offset_i$$
  3. Initialise the $Curr\_Byte\_Num$:
  $$Curr\_Byte\_Num_i = 31$$
  4. Initialise the $Curr\_Carry$:
  $$Curr\_Carry_i = 0$$

5. Initialise the $Prev\_Carry$:
$$Prev\_Carry_i = 0$$
6. Initialise the $Inst\_Carry$:
$$Inst\_Carry_i = 0$$

- If - $\boldsymbol{INIT\_OPERATION_i = 1}$:
  1. Store a value:
  $$CRAM\_Inst_i = 1$$
  2. Impose to store zero:
  $$Val(CRAM\_Inst)_i = 0$$
  3. If - $\boldsymbol{Curr\_Wd\_Offset_i = CRAM\_EWd\_Offset}$:
     (a) Increase the CRAM stamp
     $$CRAM\_STAMP_{i+1} = CRAM\_STAMP_i + 1$$
  4. Else If - $\boldsymbol{Curr\_Wd\_Offset_i \neq CRAM\_EWd\_Offset}$:
     (a) Keep the CRAM stamp constant
     $$CRAM\_STAMP_{i+1} = CRAM\_STAMP_i$$
     (b) Increase the $Curr\_Wd\_Offset$
     $$Curr\_Wd\_Offset_{i+1} = Curr\_Wd\_Offset_i + 1$$

- Else If - $\boldsymbol{INIT\_OPERATION_i = 0}$:
  1. If - $\boldsymbol{BInt\_Offset_i = 0}$ AND $\boldsymbol{EInt\_Offset_i = 31}$ AND $\boldsymbol{CRAM\_BWd\_Offset_i = CRAM\_EWd\_Offset_i}$: fast READING/WRITING operation:
     (a) Set the current RAM value to the READ/STORED value :
     $$Curr\_Val_i = Val(CRAM\_Inst)$$
     (b) Increase the RAM stamp :
     $$CRAM\_STAMP_{i+1} = CRAM\_STAMP_i + 1$$
  2. Else If - $\boldsymbol{BInt\_Offset_i \neq 0}$ OR $\boldsymbol{EInt\_Offset_i \neq 31}$ OR $\boldsymbol{CRAM\_BWd\_Offset_i \neq CRAM\_EWd\_Offset_i}$: slow READING/WRITING operation
     (a) If - $\boldsymbol{Curr\_Byte\_Num \neq 0}$:
        i. Propagate the carry decomposition :
        $$\begin{cases} Curr\_Carry_{i+1} = Curr\_Carry_i + 2^8 * Curr\_Byte_i \\ Prev\_Carry_{i+1} = Prev\_Carry_i + 2^8 * Prev\_Byte_i \end{cases}$$
        ii. Propagate the current and previous values:
        $$\begin{cases} Curr\_Val_{i+1} = Curr\_Val_i \\ Prev\_Val_{i+1} = Prev\_Val_i \end{cases}$$
        iii. Decrease the $Curr\_Byte\_Num$:
        $$Curr\_Byte\_Num_{i+1} = Curr\_Byte\_Num_i - 1$$

(b) If - $\boldsymbol{Curr\_Byte\_Num = 0}$:

i. Check that the $Curr\_Carry$ and the $Prev\_Carry$ matches the $Curr\_Val$ and the $Prev\_Val$:

$$\begin{cases} Curr\_Carry_i = Curr\_Val_i \\ Prev\_Carry_i = Prev\_Val_i \end{cases}$$

ii. If - $\boldsymbol{Curr\_Wd\_Offset_i = EWd\_Offset_i}$:

A. Move to the next instruction: $CRAM\_STAMP_{i+1} = CRAM\_STAMP_i + 1$

iii. Else If - $\boldsymbol{Curr\_Wd\_Offset_i = EWd\_Offset_i - 1}$:

A. Move to the next word offset:

$$Curr\_Wd\_Offset_{i+1} = Curr\_Wd\_Offset_i + 1$$

B. Reset the $Curr\_Byte\_Num$:

$$Curr\_Byte\_Num_{i+1} = 31$$

C. Reset the $Curr\_Carry$ and the $Prev\_Carry$:

$$\begin{cases} Curr\_Carry_i = 0 \\ Prev\_Carry_i = 0 \end{cases}$$

(c) If - $\boldsymbol{32 * Curr\_Wd\_Offset_i + Curr\_Int\_Offset_i = 32 * BWd\_Offset_i + BInt\_Offset_i}$: set the $Inst\_Range\_Flag$:

$$Inst\_Range\_Flag_i = 1$$

(d) Else If - $\boldsymbol{32*Curr\_Wd\_Offset_i+Curr\_Int\_Offset_i = 32*EWd\_Offset_i+EInt\_Offset_i}$:

i. Turn off the $Inst\_Range\_Flag$:

$$Inst\_Range\_Flag_{i+1} = 0$$

ii. Check that the $Inst\_Carry$ is the $Val(CRAM\_Inst)$:

$$Inst\_Carry = Val(CRAM\_Inst)$$

(e) Else If - $\boldsymbol{32*Curr\_Wd\_Offset_i+Curr\_Int\_Offset_i \neq 32*BWd\_Offset_i+BInt\_Offset_i}$ AND $\boldsymbol{32 * Curr\_Wd\_Offset_i + Curr\_Int\_Offset_i \neq 32 * EWd\_Offset_i + EInt\_Offset_i}$ keep the $Inst\_Range\_Flag$ constant:

$$Inst\_Range\_Flag_{i+1} = Inst\_Range\_Flag_i$$

(f) If - $\boldsymbol{Inst\_Range\_Flag_i = 1}$:

i. The current byte is the instruction byte:

$$Curr\_Byte_i = Inst\_Byte_i$$

ii. Shift the instruction carry:

$$Inst\_Carry_{i+1} = Inst\_Carry_i + 2^8 * Inst\_Byte_i$$

(g) Else If - $\boldsymbol{Inst\_Range\_Flag_i = 0}$:

i. The current byte is the previous byte:

$$Curr\_Byte_i = Prev\_Byte_i$$

ii. Don't shift the instruction carry:

$$Inst\_Carry_{i+1} = Inst\_Carry_i$$

6.4 Memory consistency columns

These columns are used to check the consistency of the RAM/Calldata of every smart contract loaded in the ROM. These columns are the space reordered version of the constraint columns from the child RAM (section 6.2). More precisely, the columns from 6.2 are lexicographically ordered by $(SC\_Num, CALLDATA\_FLAG, Curr\_Wd\_Offset, CRAM\_STAMP)$.

6.4.1 Permutation columns:

- $\widetilde{SC\_Num}$
- $\widetilde{CALLDATA\_FLAG}$
- $\widetilde{Curr\_Wd\_Offset}$
- $\widetilde{CRAM\_STAMP}$

6.4.2 Force sorted columns:

These columns are sorted by force, following the lexicographic order defined above:

- $\widetilde{CRAM\_Inst}$: the child RAM instruction
- $\widetilde{INIT\_OPERATION}$: the initialization flag.
- $\widetilde{INVALID\_INSTRUCTION}$: is turned on when an $INIT\_OPERATION$ is performed at a wrong position.
- $\widetilde{Curr\_Val}$
- $\widetilde{Prev\_Val}$

6.4.3 Range checks:

To verify that the virtual column $\widetilde{CRAM\_STAMP}$ increases well, one has to perform range checks on the difference between $\widetilde{CRAM\_STAMP}_i$ and $\widetilde{CRAM\_STAMP}_{i+1}$. This leads us to introduce the following virtual columns

- $\Delta\widetilde{CRAM\_STAMP}$
- $\Delta\widetilde{CRAM\_STAMP}^{i}$: the 16-bit decomposition of $\Delta\widetilde{CRAM\_STAMP}$ to check that $\Delta\widetilde{CRAM\_STAMP} \in [0, 2^{128}]$ (the difference has not overflown)

6.5 Constraint set for the memory consistency

1. If - $\boldsymbol{\widetilde{SC\_Num}_i = \widetilde{SC\_Num}_{i+1}}$
   (a) If - $\boldsymbol{\widetilde{CALLDATA\_FLAG}_i = \widetilde{CALLDATA\_FLAG}_{i+1}}$
      i. If - $\boldsymbol{\widetilde{Curr\_Wd\_Offset}_i = \widetilde{Curr\_Wd\_Offset}_{i+1}}$
         A. If - $\boldsymbol{\widetilde{INIT\_FLAG}_{i+1} = 1}$: turn on the $INVALID\_INSTRUCTION$ flag - the user has tried to initialize the RAM cell at an invalid timestamp.

$$\widetilde{INVALID\_INSTRUCTION}_{i+1} = 1$$

B. Compute the $\Delta\widetilde{CRAM\_STAMP}$:

$$\Delta\widetilde{CRAM\_STAMP}_i = \widetilde{CRAM\_STAMP}_{i+1} - \widetilde{CRAM\_STAMP}_i$$

C. Impose that the current value is the same as next previous value:

$$\widetilde{Prev\_Val}_{i+1} = \widetilde{Curr\_Val}_i$$

D. If - $\boldsymbol{\Delta\widetilde{CRAM\_STAMP}_i = \Delta\widetilde{CRAM\_STAMP}_{i+1}}$: impose the values of $Curr\_Val$ and $Prev\_Val$ to be constant:

$$\begin{cases} \widetilde{Curr\_Val}_i = \widetilde{Curr\_Val}_{i+1} \\ \widetilde{Prev\_Val}_i = \widetilde{Prev\_Val}_{i+1} \end{cases}$$

E. Else If - $\boldsymbol{\Delta\widetilde{CRAM\_STAMP}_i \neq \Delta\widetilde{CRAM\_STAMP}_{i+1}}$ AND $\boldsymbol{\widetilde{CRAM\_Inst} = 0}$ (reading operation at two successive timestamps); impose the value consistency:

$$\widetilde{Curr\_Val}_i = \widetilde{Curr\_Val}_{i+1}$$

ii. Else If - $\boldsymbol{\widetilde{Cur\_Wd\_Offset}_i \neq \widetilde{Curr\_Wd\_Offset}_{i+1}}$

A. Impose the next operation to be the initialization of the memory cell:

$$\widetilde{INIT\_OPERATION}_{i+1} = 1$$

(b) Else If - $\boldsymbol{\widetilde{CALLDATA\_FLAG}_i \neq \widetilde{CALLDATA\_FLAG}_{i+1}}$

i. Impose the next operation to be the initialization of the memory cell:

$$\widetilde{INIT\_OPERATION}_{i+1} = 1$$

2. Else If - $\boldsymbol{\widetilde{SC\_Num}_i \neq \widetilde{SC\_Num}_{i+1}}$: impose the next operation to be the initialization of the memory cell:

$$\widetilde{INIT\_OPERATION}_{i+1} = 1$$

References

[1] DR. Gavin Wood. "Ethereum : A secure decentralised generalised transaction ledger". In: (2021). https://ethereum.github.io/yellowpaper/paper.pdf.

[2] Vitalik Buterin. *An Incomplete Guide to Rollups*. Jan. 2021. URL: https://vitalik.ca/general/2021/01/05/rollup.html.

[3] DeGate Team. *An article to understand zkEVM, the key to Ethereum scaling.* Sept. 2021. URL: https://medium.com/degate/an-article-to-understand-zkevm-the-key-to-ethereum-scaling-ff0d83c417cc.

[4] *ZK-sync official website.* URL: https://zksync.io/.

[5] Lior Goldberg, Shahar Papini, and Michael Riabzev. *Cairo – a Turing-complete STARK-friendly CPU architecture.* Cryptology ePrint Archive, Report 2021/1063. https://ia.cr/2021/1063. 2021.

[6] *Hermez official website.* URL: https://hermez.io/.

[7] *Scroll tech github repository.* URL: https://github.com/scroll-tech/.

[8] *Hermez presentation at the EthCC4.* URL: https://youtu.be/17d5DG6L2nw.

[9] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. *PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge*. Cryptology ePrint Archive, Report 2019/953. https://ia.cr/2019/953. 2019.

[10] Ariel Gabizon and Zachary J. Williamson. *plookup: A simplified polynomial protocol for lookup tables*. Cryptology ePrint Archive, Report 2020/315. https://ia.cr/2020/315. 2020.

[11] Olivier Begassat Alexandre Belling. *Using GKR inside a SNARK to reduce the cost of hash verification down to 3 constraints*. June 2020. URL: https://ethresear.ch/t/using-gkr-inside-a-snark-to-reduce-the-cost-of-hash-verification-down-to-3-constraints/7550.